Show No Weakness: Sequentially Consistent Specifications of TSO Libraries

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

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

Abstract. Modern programming languages, such as C++ and Java, provide a sequentially consistent (SC) memory model for well-behaved programs that follow a certain synchronisation discipline, e.g., for those that are data-race free (DRF). However, performance-critical libraries often violate the discipline by using low-level hardware primitives, which have a weaker semantics. In such scenarios, it is important for these libraries to protect their otherwise well-behaved clients from the weaker memory model.
In this paper, we demonstrate that a variant of linearizability can be used to reason formally about the interoperability between a high-level DRF client and a low-level library written for the Total Store Order (TSO) memory model, which is implemented by x86 processors. Namely, we present a notion of linearizability that relates a concrete library implementation running on TSO to an abstract specification running on an SC machine. A client of this library is said to be DRF if its SC executions calling the abstract library specification do not contain data races. We then show how to compile a DRF client to TSO such that it only exhibits SC behaviours, despite calling into a racy library.

1 Introduction

Modern programming languages, such as C++ [3, 2] and Java [10], provide memory consistency models that are weaker than the classical *sequential consistency (SC)* [9]. Doing so enables these languages to support common compiler optimisations and to compile efficiently to modern architectures, which themselves do not guarantee SC. However, programming on such *weak memory models* can be subtle and error-prone. As a compromise between programmability and performance, C++ and Java provide *data-race free (DRF)* memory models, which guarantee SC for programs without data races, i.e., those that protect data accesses with an appropriate use of high-level synchronisation primitives defined in the language, such as locks and semaphores[4].

While DRF memory models protect most programmers from the counter-intuitive effects of weak memory models, performance-minded programmers often violate the DRF discipline by using low-level hardware primitives. For instance, it is common for a systems-level C++ program, such as an operating system kernel, to call into highly-optimised libraries written in assembly code. Moreover, the very synchronisation prim-

[4] C++ [3, 2] also includes special *weak atomic* operations that have a weak semantics. Thus, a C++ program is guaranteed to be SC only if it is DRF and avoids the use of weak atomics.

itives of the high-level language that programmers use to ensure DRF are usually implemented in its run-time system in an architecture-specific way. Thus, it becomes necessary to reason about the interoperability between low-level libraries *native* to a particular hardware architecture and their clients written in a high-level language. While it is acceptable for expert library designers to deal with weak memory models, high-level language programmers need to be protected from the weak semantics.

In this paper, we consider this problem for libraries written for the *Total Store Order (TSO)* memory model, used by x86 processors (and described in Section 2). TSO allows for the *store buffer* optimisation implemented by most modern processors: writes performed by a processor are buffered in a processor-local store buffer and are flushed into the memory at some later time. This complicates the interoperability between a client and a TSO library. For instance, the client cannot assume that the effects of a library call have taken place by the time the call returns. Our main contributions are:

- a notion of specification of native TSO libraries in terms of the concepts of a high-level DRF model, allowing the model to be extended to accommodate such libraries, while preserving the SC semantics; and
- conditions that a compiler has to satisfy in order to implement the extended memory model correctly.

Our notion of library specification is based on *linearizability* [8], which fixes a correspondence between a *concrete* library and an *abstract* one, the latter usually implemented atomically and serving as a specification for the former. To reason formally about the interoperability between a high-level DRF client and a low-level TSO library, we propose a variant of linearizability called *TSO-to-SC linearizability* (Section 3). It relates a concrete library implementation running on the TSO memory model to its abstract specification running on SC. As such, the abstract specification describes the behaviour of the library in a way compatible with a DRF memory model. Instead of referring to hardware concepts, it fakes the effects of the concrete library implementation executing on TSO by adding extra non-determinism into SC executions. TSO-to-SC linearizability is compositional and allows soundly replacing a library by its SC specification in reasoning about its clients.

TSO-to-SC linearizability allows extending DRF models of high-level languages to programs using TSO libraries by defining the semantics of library calls using their SC specifications. In particular, this allows generalising the notion of data-race freedom to such programs: a client using a TSO library is DRF if so is every SC execution of the same client using the SC library specification. Building on this, we propose requirements that a compiler should satisfy in order to compile such a client onto a TSO machine correctly (Section 5), and establish the *Simulation Theorem* (Theorem 13, Section 5), which guarantees that a correctly compiled DRF client produces only SC behaviours, despite calling into a native TSO library. The key benefit of our framework is that both checking the DRF property of the client and checking the compiler correctness does not require TSO reasoning. Reasoning about weak memory is only needed to establish the TSO-to-SC linearizability of the library implementation. However, this also makes the proof of the Simulation Theorem challenging.

Our results make no difference between custom-made TSO libraries and TSO implementations of synchronisation primitives built into the run-time system of the high-

level language. Hence, TSO-to-SC linearizability and the Simulation Theorem provide conditions ensuring that a given TSO implementation of the run-time system for a DRF language has the desired semantics and interacts correctly with its compilation.

Recently, a lot of attention has been devoted to criteria for checking whether a TSO program produces only sequentially consistent behaviours [11, 4, 1]. Such criteria are less flexible than TSO-to-SC linearizability, as they do not allow a program to have internal non-SC behaviours; however, they are easier to check. We therefore also analyse which of the criteria can be used for establishing the conditions required by our framework (Sections 4 and 6).

Proofs of all the theorems stated in the paper are given in Appendix C.

2 TSO Semantics

Due to space constraints, we present the TSO memory model only informally; a formal semantics is given in Appendix A. The most intuitive way to explain TSO is using an abstract machine [12]. Namely, consider a multiprocessor with n CPUs, indexed by $\mathsf{CPUid} = \{1, \ldots, \mathsf{NCPUs}\}$, and a shared memory. The state of the memory is described by an element of $\mathsf{Heap} = \mathsf{Loc} \to \mathsf{Val}$, where Loc and Val are unspecified sets of locations and values, such that $\mathsf{Loc} \subseteq \mathsf{Val}$. Each CPU has a set of general-purpose registers $\mathsf{Reg} = \{\mathtt{r}_1, \ldots, \mathtt{r}_m\}$ storing values from Val. In TSO, processors do not write to memory directly. Instead, every CPU has a ***store buffer***, which holds write requests that were issued by the CPU, but have not yet been ***flushed*** into the shared memory. The state of a buffer is described by a sequence of location-value pairs.

The machine executes programs of the following form:

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

A program $C(L)$ consists of a declaration of a library L, implementing methods $m \in M \subseteq \mathsf{Method}$ by commands C_m, 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 $\mathsf{sig}(L) = M$. We assume that the program is stored separately from the memory. The particular syntax of commands C_t and C_m is of no concern for understanding the main results of this paper and is deferred to Appendix A. We consider programs using a single library for simplicity only; we discuss the treatment of multiple libraries in Section 3.

The abstract machine can perform the following transitions:

- A CPU wishing to write a value to a memory location adds an appropriate entry to the *tail* of its store buffer.
- The entry at the *head* of the store buffer of a CPU is flushed into the memory at a non-deterministically chosen time. Store buffers thus have the FIFO ordering.
- A CPU wishing to read from a memory location first looks at the pending writes in its store buffer. If there are entries for this location, it reads the value from the newest one; otherwise, it reads the value directly from the memory.
- Modern multiprocessors provide commands that can access several memory locations atomically, such as compare-and-swap (CAS). To model this in our machine, a CPU can execute a special lock command, which makes it the only CPU able to

execute commands until it executes an unlock command. The unlock command has a built-in ***memory barrier***, forcing the store buffer of the CPU executing it to be flushed completely. This can be used by the programmer to recover SC when needed.

– Finally, a CPU can execute a command affecting only its registers. In particular, it can call a library method or return from it (we disallow nested method calls).

The behaviour of programs running on TSO can sometimes be counter-intuitive. For example, consider two memory locations x and y initially holding 0. On TSO, 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:

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

Here a and b are local variables of the corresponding threads, stored in CPU registers. The outcome shown cannot happen on an SC 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. Note that executing the writes to x and y in the above program within lock..unlock blocks (which on x86 corresponds to adding memory barriers after them) would make it produce only SC behaviours.

We describe computations of the machine using ***traces***, which are finite sequences of ***actions*** of the form

$$\begin{array}{rcl} \varphi & ::= & (t, \mathsf{read}(x, u)) \mid (t, \mathsf{write}(x, u)) \mid (t, \mathsf{flush}(x, u)) \mid \\ & & (t, \mathsf{lock}) \mid (t, \mathsf{unlock}) \mid (t, \mathsf{call}\ m(r)) \mid (t, \mathsf{ret}\ m(r)) \end{array}$$

where $t \in \mathsf{CPUid}$, $x \in \mathsf{Loc}$, $u \in \mathsf{Val}$, $m \in \mathsf{Method}$ and $r \in \mathsf{Reg} \to \mathsf{Val}$. 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. The rest of the actions have the expected meaning. Of transitions by a CPU affecting solely its registers, only calls and returns are recorded in traces. We assume that parameters and return values of library methods are passed via CPU registers, and thus record their values in call and return actions. We use the standard notation for traces: $\tau(i)$ is the i-th action in the trace τ, $|\tau|$ is its length, and $\tau|_t$ its projection to actions by CPU t. We denote the concatenation of two traces τ_1 and τ_2 with $\tau_1\tau_2$.

Given a suitable formalisation of the abstract machine transitions, we can define the set of traces $[\![C(L)]\!]_{\mathsf{TSO}}$ generated by executions of the program $C(L)$ on TSO (Appendix A). For simplicity, we do not consider traces that have a (t, lock) action without a matching (t, unlock) action.

To give the semantics of a program on the SC memory model, we do not define another abstract machine; instead, we identify the SC executions of a program with those of the TSO machine that flush all writes immediately. Namely, we let $[\![C(L)]\!]_{\mathsf{SC}}$ be the set of sequentially consistent traces from $[\![C(L)]\!]_{\mathsf{TSO}}$, defined as follows.

DEFINITION 1. *A trace is **sequentially consistent (SC)**, if every action $(t, \mathsf{write}(x, u))$ in it is immediately followed by $(t, \mathsf{flush}(x, u))$.*

We assume that the set of memory locations Loc is partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc}$. The client C and the library L are ***non-interfering*** in $C(L)$, if in every computation from $[\![C(L)]\!]_{\mathsf{TSO}}$, commands performed by the client (library) code access only locations from CLoc (LLoc). In the following, we consider only programs where the client and the library are non-interfering. We provide pointers to lifting this restriction in Section 7.

3 TSO-to-SC Linearizability

We start by presenting our notion of library specification, discussing its properties and giving example specifications. The notion of specification forms the basis for interoperability conditions presented in Section 5.

TSO-to-SC Linearizability. When defining library specifications, we are not interested in internal library actions recorded in traces, but only in interactions of the library with its client. We record such interactions using ***histories***, which are traces including only ***interface actions*** of the form $(t, \mathsf{call}\ m(r))$ or $(t, \mathsf{ret}\ m(r))$, where $t \in \mathsf{CPUid}$, $m \in \mathsf{Method}$, $r \in \mathsf{Reg} \to \mathsf{Val}$. Recall that r records the values of registers of the CPU that calls the library method or returns from it, which serve as parameters or return values. We define the history $\mathsf{history}(\tau)$ of a trace τ as its projection to interface actions and lift history to sets T of traces pointwise: $\mathsf{history}(T) = \{\mathsf{history}(\tau) \mid \tau \in T\}$. In the following, we write $\_$ for an expression whose value is irrelevant.

DEFINITION 2. *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 $\forall i, j.\, i < j \wedge H(i) = (\_, \mathsf{ret}\ \_) \wedge H(j) = (\_, \mathsf{call}\ \_) \Rightarrow \pi(i) < \pi(j)$.*

That is, H' linearizes H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations.

To generate the set of all histories of a given library L, we consider its ***most general client***, whose hardware threads on every CPU repeatedly invoke library methods in any order and with any parameters possible. Its formal definition is given in Appendix B. Informally, assume $\mathsf{sig}(L) = \{m_1, \ldots, m_l\}$. Then $\mathsf{MGC}(L) = (\mathsf{let}\ L\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_{\mathsf{NCPUs}}^{\mathsf{mgc}})$, where for all t, the command C_t^{mgc} behaves as

```
while (true) { havoc; if (*) m_1; else if (*) m_2; ... else m_l; }
```

Here `*` denotes non-deterministic choice, and `havoc` sets all registers storing method parameters to arbitrary values. The set of traces $[\![\mathsf{MGC}(L)]\!]_{\mathsf{TSO}}$ includes all library behaviours under any possible client. We write $[\![L]\!]_{\mathsf{TSO}}$ for $[\![\mathsf{MGC}(L)]\!]_{\mathsf{TSO}}$ and $[\![L]\!]_{\mathsf{SC}}$ for $[\![\mathsf{MGC}(L)]\!]_{\mathsf{SC}}$. We can now define what it means for a library executing on SC to be a specification for another library executing on TSO.

DEFINITION 3. *For libraries L_1 and L_2 such that $\mathsf{sig}(L_1) = \mathsf{sig}(L_2)$, we say that L_2 **TSO-to-SC linearizes** L_1, written $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_2$, if $\forall H_1 \in \mathsf{history}([\![L_1]\!]_{\mathsf{TSO}}).\ \exists H_2 \in \mathsf{history}([\![L_2]\!]_{\mathsf{SC}}).\ H_1 \sqsubseteq H_2$.*

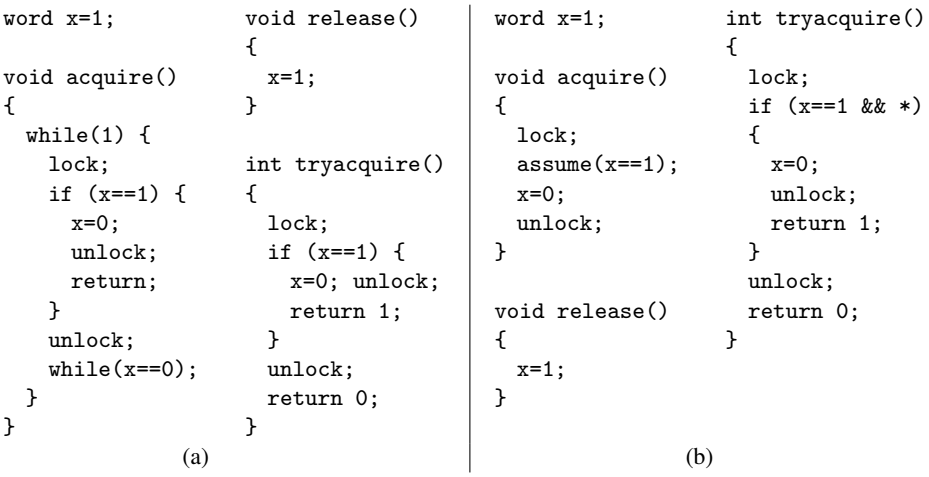

Fig. 1. (a) L_{spinlock}: a test-and-test-and-set spinlock implementation on TSO; (b) $L^{\sharp}_{\mathsf{spinlock}}$: its SC specification. Here $*$ denotes non-deterministic choice. The $\texttt{assume}(E)$ command acts as a filter on states, choosing only those where E evaluates to non-zero values (see Appendix A).

Thus, L_2 linearizes L_1 if every history of the latter on TSO may be reproduced in a linearized form by the former on SC. When the library L_2 is implemented atomically, and so histories in $\mathsf{history}([\![L_2]\!]_{\mathsf{SC}})$ are sequential, Definition 3 becomes identical to the standard linearizability [8], except the libraries run on different memory models.

Example: Spinlock. Figure 1a shows a simple implementation L_{spinlock} of a spinlock on TSO. We consider only well-behaved clients of the spinlock, which, e.g., do not call `release` without having previously called `acquire` (this can be easily taken into account by restricting the most general client appropriately). The `tryacquire` method tries to acquire the lock, but, unlike `acquire`, does not wait for it to be released if it is busy; it just returns 0 in this case. For efficiency, `release` writes 1 to `x` without executing a memory barrier. This optimisation is used, e.g., by implementations of spinlocks in the Linux kernel [5]. On TSO this can result in an additional delay before the write releasing the lock becomes visible to another CPU trying to acquire it. As a consequence, `tryacquire` can return 0 even after the lock has actually been released. For example, the following is a valid history of the spinlock implementation on TSO, which cannot be produced on an SC memory model:

$$(1, \mathsf{call}\ \texttt{acquire})\,(1, \mathsf{ret}\ \texttt{acquire})\,(1, \mathsf{call}\ \texttt{release})\,(1, \mathsf{ret}\ \texttt{release})\\ (2, \mathsf{call}\ \texttt{tryacquire})\,(2, \mathsf{ret}\ \texttt{tryacquire}(0)). \quad (1)$$

Figure 1b shows an abstract SC implementation $L^{\sharp}_{\mathsf{spinlock}}$ of the spinlock capturing the behaviours of its concrete TSO implementation, such as the one given by the above history. Here `release` writes 1 to `x` immediately. To capture the effects of the concrete library implementation running on TSO, the SC specification is weaker than might be expected: `tryacquire` in Figure 1b can spuriously return 0 even when `x` contains 1.

PROPOSITION 4. $L_{\mathsf{spinlock}} \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^{\sharp}_{\mathsf{spinlock}}$.

The same specification is also suitable for more complicated spinlock implementations (Appendix B). We note that the weak specification of `tryacquire` has been adopted by the C++ memory model [3] to allow certain compiler optimisations. As we show in Section 5, linearizability with respect to an SC specification ensures the correctness of implementations of `tryacquire` and other synchronisation primitives comprising the run-time system of a DRF language. Our example thus shows that the specification used in C++ is also needed to capture the behaviour of common spinlock implementations.

Correctness of TSO-to-SC Linearizability. A good notion of library specification has to allow replacing a library implementation with its specification in reasoning about a client. We now show that the notion of TSO-to-SC linearizability proposed above satisfies a variant of this property. To reason about clients of TSO libraries with respect to SC specifications of the latter, we consider a mixed ***TSO/SC semantics*** of programs, which executes the client on TSO and the library on SC. That is, read and write commands by the library code bypass the store buffer and access the memory directly (the formal semantics is given in Appendix B). We denote the set of traces of a program $C(L)$ in this semantics with $[\![C(L)]\!]_{\mathsf{TSO/SC}}$.

To express properties of a client preserved by replacing the implementation of the library it uses with its specification, we introduce the following operation. For a trace τ of $C(L)$, let $\mathsf{client}(\tau)$ be its projection 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 $\varphi = (t, \_)$ such that $\tau = \tau'\varphi\tau''$ into the projection if:

- φ is an interface action, i.e., a call or a return; or
- φ is not a flush or an interface action, and it is not the case that $\tau|_t = \tau_1\,(t, \mathsf{call}\ \_)\,\tau_2\varphi\tau_3$, where τ_2 does not contain a $(t, \mathsf{ret}\ \_)$ action; or
- $\varphi = (\_, \mathsf{flush}(x, \_))$ for some $x \in \mathsf{CLoc}$.

We lift client to sets T of traces pointwise: $\mathsf{client}(T) = \{\mathsf{client}(\tau) \mid \tau \in T\}$.

THEOREM 5 (Abstraction to SC). *If* $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_2$, *then* $\mathsf{client}([\![C(L_1)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}([\![C(L_2)]\!]_{\mathsf{TSO/SC}})$.

According to Theorem 5, while reasoning about a client $C(L_1)$ of a TSO library L_1, we can soundly replace L_1 with its SC version L_2 linearizing L_1: if a trace property over client actions holds of $C(L_2)$, it will also hold of $C(L_1)$. The theorem can thus be used to simplify reasoning about TSO programs. Although Theorem 5 is not the main contribution of this paper, it serves as a sanity check for our definition of linearizability, and is useful for discussing our main technical result in Section 5.

Compositionality of TSO-to-SC Linearizability. The following corollary of Theorem 5 states that, like the classical notion of linearizability [8], ours is compositional: if several non-interacting libraries are linearizable, so is their composition. This allows extending the results presented in the rest of the paper to programs with multiple libraries. Formally, consider libraries $L_1, \ldots, L_k$ with disjoint sets of declared methods and assume that the set of library locations LLoc is partitioned into locations belonging to every library: $\mathsf{LLoc} = \mathsf{LLoc}_1 \uplus \ldots \uplus \mathsf{LLoc}_k$. We assume that, in any program, a library L_j accesses only locations from LLoc_j. We let L, respectively, $L^\sharp$ be the library implementing all of the methods from $L_1, \ldots, L_k$, respectively, $L_1^\sharp, \ldots, L_k^\sharp$.

COROLLARY 6 (Compositionality). *If* $\forall j.\, L_j \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_j^\sharp$, *then* $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$.

Comparison with TSO-to-TSO Linearizability. As the abstract library implementation in TSO-to-SC linearizability executes on SC, it does not describe how the concrete library implementation uses store buffers. TSO libraries can also be specified by abstract implementations running on TSO, which do describe this usage. In [6], we proposed the notion of TSO-to-TSO linearizability $\sqsubseteq_{\mathsf{TSO}\to\mathsf{TSO}}$ between two TSO libraries, which validates the following version of the Abstraction Theorem.

THEOREM 7 (Abstraction to TSO). *If* $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{TSO}} L_2$, *then* $\mathsf{client}(\llbracket C(L_1) \rrbracket_{\mathsf{TSO}}) \subseteq \mathsf{client}(\llbracket C(L_2) \rrbracket_{\mathsf{TSO}})$.

The particularities of TSO-to-TSO linearizability are not relevant here; suffice it to say that the definition requires that the two libraries use store buffers in similar ways, and to this end, enriches histories with extra actions. The spinlock from Figure 1a has the abstract TSO implementation with `acquire` and `release` implemented as in Figure 1b, and `tryacquire`, as in Figure 1a (the implementation and the specification of `tryacquire` are identical in this case because the spinlock considered is very simple; see Appendix B for more complicated cases). Since the specification executes on TSO, the write to `x` in `release` can be delayed in the store buffer. In exchange, the specification of `tryacquire` does not include spurious failures.

Both TSO-to-SC and TSO-to-TSO linearizability validate versions of the Abstraction Theorem (Theorems 5 and 7). The theorem validated by TSO-to-SC is weaker than the one validated by TSO-to-TSO: a property of a client of a library may be provable after replacing the latter with its TSO specification using Theorem 7, but not after replacing it with its SC specification using Theorem 5. Indeed, consider the following client of the spinlock in Figure 1a, where `a` and `b` are local to the second thread:

```
                                  u = 0;
acquire(); release(); u = 1;  ||  a = u; b = tryacquire();
```
$$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$$

The postcondition shown holds of the program: since store buffers in TSO are FIFO, if the write to `u` has been flushed, so has been the write to `x` in `release`, and `tryacquire` has to succeed. However, it cannot be established after we apply Theorem 5 with the spinlock specification in Figure 1b, as the abstract implementation of `tryacquire` returns an arbitrary result when the lock is free. The postcondition can still be established after we apply Theorem 7 with the TSO specification of the spinlock given in Section 3, since the specification allows us to reason about the correlations in the use of store buffers by the library and the client. To summarise, SC specifications of TSO libraries trade the weakness of the memory model for the weakness of the specification.

Example: Seqlock. We now consider an example of a TSO library whose SC specification is more subtle than that of a spinlock. Figure 2 presents a simplified version L_{seqlock} of a seqlock [5]—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

```
word x1 = 0, x2 = 0, 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. 2. L_{seqlock}: a TSO seqlock implementation

threads can read from. A version number is stored at `c`. The writing thread maintains the invariant that the version 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. After reading, it checks that the version has not changed, thereby ensuring that no write has overlapped the read. Neither `write` nor `read` includes a barrier, so that writes to `x1`, `x2` and `c` may not be visible to readers immediately.

An SC specification for the seqlock is better given not by the source code of an abstract implementation, like in the case of a spinlock, but by explicitly describing the set of its histories $\mathsf{history}([\![L_2]\!]_{\mathsf{SC}})$ to be used in Definition 2 (an operational specification also exists, but is more complicated; see Appendix B). We now adjust the definition of TSO-to-SC linearizability to accept a library specification defined in this way.

Specifying Libraries by Sets of Histories. For a TSO library L and a set of histories T, we let $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} T$, if $\forall H_1 \in \mathsf{history}([\![L]\!]_{\mathsf{TSO}}).\, \exists H_2 \in T.\, H_1 \sqsubseteq H_2$. The formulation of Theorem 5 can be easily adjusted to accommodate this notion of linearizability.

We now give a specification to the seqlock as a set of histories T_{seqlock}. First of all, methods of a seqlock should appear to take effect atomically. Thus, in histories from T_{seqlock}, if call action has a matching return, then the latter has to follow it immediately. Consider a history H_0 satisfying this property. Let $\mathsf{writes}(H_0)$ be the sequence of pairs (d_1, d_2) from actions of the form $(\_, \mathsf{call}\ \mathtt{write}(d_1, d_2))$ in H_0, and $\mathsf{reads}(H_0)$, the sequence of (d_1, d_2) from actions of the form $(\_, \mathsf{ret}\ \mathtt{read}(d_1, d_2))$. For a sequence α, let $\alpha^\dagger$ be its stutter-closure, i.e., the set of sequences obtained from α by repeating some of its elements. We lift the stutter-closure operation to sets of sequences pointwise. Given the above definitions, a history H belongs to T_{seqlock} if for every prefix H_0 of H, $\mathsf{reads}(H_0)$ is a subsequence of a sequence from $((0,0)\,\mathsf{writes}(H_0))^\dagger$. Recall that a seqlock allows only a single thread to call `write`. This specification thus ensures that readers see the writes in the order it issues them, but possibly with a delay.

PROPOSITION 8. $L_{\mathsf{seqlock}} \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} T_{\mathsf{seqlock}}$.

4 TSO-to-SC Linearizability and Robustness

One way to simplify reasoning about a TSO program is by checking that it is ***robust***, meaning that it produces only those externally visible behaviours that could also be obtained by running it on an SC machine. Its properties can then be proved by considering only its SC executions. Several criteria for checking robustness of TSO programs have been proposed recently [11, 4, 1]. TSO-to-SC linearizability is more flexible than such criteria: since an abstract library implementation can have different source code than

its concrete implementation, it allows the latter to have non-SC behaviours. However, checking the requirements of a robustness criterion is usually easier than proving linearizability. We therefore show how one such criterion, data-race freedom, can be used to simplify establishing TSO-to-SC linearizability when it is applicable. On the way, we introduce some of the technical ingredients necessary for our main result in Section 5.

We first define the notion of DRF for the low-level machine of Section 2. Our intention is that the DRF of a program must ensure that it produces only SC behaviours (see Theorem 10 below). All robustness criteria proposed so far have assumed a closed program P consisting of a client that does not use a library. We define the robustness of libraries using the most general client of Section 3. For a trace fragment τ with all actions by a thread t, we denote with $\mathsf{block}(\tau)$ a trace of one of the following two forms: τ or $(t, \mathsf{lock})\,\tau_1\,\tau\,\tau_2\,(t, \mathsf{unlock})$, where τ_1, τ_2 do not contain (t, unlock).

DEFINITION 9. *A **data race** is a fragment of an SC trace of the form* $\mathsf{block}(\tau)\,(t', \mathsf{write}(x, \_))\,(t', \mathsf{flush}(x, \_))$*, where* $\tau \in \{(t, \mathsf{write}(x, \_))\,(t, \mathsf{flush}(x, \_)), (t, \mathsf{read}(x, \_))\}$ *and* $t \neq t'$*. A program P is **data-race free (DRF)**, if so are traces in* $[\![P]\!]_{\mathsf{SC}}$*; a library L is DRF, if so are traces in* $[\![L]\!]_{\mathsf{SC}}$*.*

Thus, a race is a memory access followed by a write to the same location, where the former, but not the latter, can be in a lock..unlock block. This is a standard notion of a data race with one difference: even though $(t, \mathsf{read}(x))\,(t', \mathsf{write}(x))\,(t', \mathsf{flush}(x))$ is a race, $(t, \mathsf{read}(x))\,(t', \mathsf{lock})\,(t', \mathsf{write}(x))\,(t', \mathsf{flush}(x))\,(t', \mathsf{unlock})$ is not. We do not consider conflicting accesses of the latter kind (e.g., with the write inside a CAS, which includes a memory barrier) as a race, since they do not lead to a non-SC behaviour.

We adopt the following formalisation of externally visible program behaviours. Assume a set $\mathsf{VLoc} \subseteq \mathsf{CLoc}$ of client locations whose values in memory can be observed during a program execution by its environment. ***Visible actions*** are those of the form $(t, \mathsf{read}(x, u))$ or $(t, \mathsf{flush}(x, u))$, where $x \in \mathsf{VLoc}$. We let $\mathsf{visible}(\tau)$ be the projection of τ to visible actions, and lift visible to sets T of traces pointwise: $\mathsf{visible}(T) = \{\mathsf{visible}(\tau) \mid \tau \in T\}$. Visible locations are ***protected*** in $C(L)$, if every visible action in a trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ occurs within a lock..unlock block. On x86, this requires a memory barrier after every output action, thus ensuring that it becomes visible immediately. The following is a folklore robustness result.

THEOREM 10 (Robustness via DRF). *If P is DRF and visible locations are protected in it, then* $\mathsf{visible}([\![P]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![P]\!]_{\mathsf{SC}})$.

Note that here DRF is checked on the SC semantics and at the same time implies that the program behaves SC. This circularity is crucial for using results such as Theorem 10 to simplify reasoning, as it allows not considering TSO executions at all.

THEOREM 11 (Linearizability via DRF). *If L is DRF, then* $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L$.

This allows using classical linearizability [8] to establish TSO-to-SC one by linearizing the library L running on SC to its SC specification $L^\sharp$, thus yielding $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$. This can then be used for modular reasoning by applying Theorem 5.

Many concurrent algorithms on TSO (e.g., the classical Treiber's stack) are DRF, as they modify the data structure using only CAS operations, which include a memory

barrier. Hence, their linearizability with respect to SC specifications can be established using Theorem 11. However, the DRF criterion may sometimes be too strong: e.g., in the spinlock implementation from Figure 1a, the read from `x` in `acquire` and the write to it in `release` race. We consider more flexible robustness criteria in Section 6.

5 Conditions for Correct Compilation

Our goal in this section is to extend DRF memory models of high-level languages to the case of programs using native TSO libraries, and to identify conditions under which the compiler implements the models correctly. We start by presenting the main technical result of the paper that enables this—the Simulation Theorem.

Simulation Theorem. Consider a program C, meant to be compiled from a high-level language with a DRF model, which uses a native TSO library L. We wish to determine the conditions under which the program produces only SC behaviours, despite possible races inside L. To this end, we first generalise DRF on TSO (Definition 9) to such programs. We define DRF with respect to an SC specification $L^\sharp$ of L.

DEFINITION 12. *$C(L^\sharp)$ is **DRF** if so is any trace from* $\mathsf{client}(\llbracket C(L^\sharp) \rrbracket_{\mathsf{SC}})$.

This allows races inside the library code, as its internal behaviour is of no concern to the client. Note that checking DRF does not require reasoning about weak memory.

THEOREM 13 (Simulation). *If $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$, $C(L^\sharp)$ is DRF, and visible locations are protected in $C(L)$, then* $\mathsf{visible}(\llbracket C(L) \rrbracket_{\mathsf{TSO}}) \subseteq \mathsf{visible}(\llbracket C(L^\sharp) \rrbracket_{\mathsf{SC}})$.

Thus, the behaviour of a DRF client of a TSO library can be reproduced when the client executes on the SC memory model and uses a TSO-to-SC linearization of the library implementation. Note that the DRF of the client is defined with respect to the SC specification $L^\sharp$ of the TSO library L. Replacing L by $L^\sharp$ allows hiding non-SC behaviours internal to the library, which are of no concern to the client. Corollary 6 allows applying the theorem to clients using multiple libraries.

Extending Memory Models of High-Level Languages. We describe a method for extending a high-level memory model to programs with native TSO libraries in general terms, without tying ourselves to its formalisation. We give an instantiation for the case of the C++ memory model (excluding weak atomics) in Appendix B. Consider a high-level language with a DRF memory model. That is, we assume an SC semantics for the language, and a notion of DRF on this semantics. For a program $\mathcal{P}$ in this language, let $\llbracket \mathcal{P} \rrbracket$ be the set of its externally visible behaviours resulting from its executions in the semantics of the high-level language. At this point, we do not need to define what these behaviours are; they might include, e.g., input/output information.

Let $\mathcal{C}(L)$ be a program in a high-level language using a TSO library L with an SC specification $L^\sharp$, i.e., $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$. The specification $L^\sharp$ allows us to extend the semantics of the language to describe the intended behaviour of $\mathcal{C}(L)$. Informally, we let the semantics of calling a method of L be the effect of the corresponding method of $L^\sharp$. As both $\mathcal{C}$ and $L^\sharp$ are meant to have an SC semantics, the effect of $L^\sharp$ can be described within the memory model of the high-level language.

To define this extension more formally, it is convenient for us to use the specification of $L^\sharp$ given by its set of histories $\mathsf{history}([\![L^\sharp]\!]_{\mathsf{SC}})$, rather than by its source code, as this sidesteps the issues arising when composing the sources of programs in a low-level language and a high-level one. Namely, we define the semantics of $\mathcal{C}(L)$ in two stages. First, we consider the set of executions of $\mathcal{C}(L)$ in the semantics of the high-level language where a call to a method of L is interpreted in the same way as a call to a method of the high-level language returning arbitrary values. Since the high-level language has an SC semantics, every program execution in it is a trace obtained by interleaving actions of different threads, which has a single history of calls to and returns from L. We then define the intended behaviour of $\mathcal{C}(L)$ by the set $[\![\mathcal{C}(L^\sharp)]\!]$ of externally visible behaviours resulting from the executions that have a history from $\mathsf{history}([\![L^\sharp]\!]_{\mathsf{SC}})$[5]. This semantics also generalises the notion of DRF to the extended language: programs are DRF when the executions of $\mathcal{C}(L)$ selected above have no races between client actions as defined for high-level programs without TSO libraries. In particular, DRF is defined with respect to SC specifications of libraries that the client uses, not their TSO implementations.

From the point of view of the extended memory model, the run-time system of the high-level language, implementing built-in synchronisation primitives, is no different from external TSO libraries. The extension thus allows deriving a memory model consistent with the implementation of synchronisation primitives on TSO (e.g., spinlocks or seqlocks from Section 3) from the memory model of the base language excluding the primitives. Below, we use this fact to separate the reasoning about the correctness of a compiler for the high-level language from that about the correctness of its run-time system. This approach to deriving the memory model does not result in imprecise specifications: e.g., the SC specification of a TSO spinlock implementation in Section 3 corresponds to the one in the C++ standard.

Conditions for Correct Compilation. Theorem 13 allows us to formulate conditions under which a compiler from a high-level DRF language correctly implements the extended memory model defined above. Let $\langle\mathcal{C}\rangle(L)$ be the compilation of a program $\mathcal{C}$ in the high-level language to the TSO machine from Section 2, linked with a native TSO library L. Assume an SC specification $L^\sharp$ of L:

(i) $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$.

Then the extended memory model defines the intended semantics $[\![\mathcal{C}(L^\sharp)]\!]$ of the program. Let us denote the compiled code linked with $L^\sharp$, instead of L, as $\langle\mathcal{C}\rangle(L^\sharp)$. We place the following constraints on the compiler:

(ii) $\mathcal{C}$ is correctly compiled to an SC machine: $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}}) \subseteq [\![\mathcal{C}(L^\sharp)]\!]$.

(iii) $\langle\mathcal{C}\rangle(L^\sharp)$ is DRF, i.e., so are all traces from $\mathsf{client}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}})$.

(iv) Visible locations are protected in $\langle\mathcal{C}\rangle(L)$.

From Theorem 13 and (i), (iii) and (iv), we obtain $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}})$, which, together with (ii), implies $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L)]\!]_{\mathsf{TSO}}) \subseteq$

[5] Here we assume that language-level threads correspond directly to hardware-level ones. This assumption is sound even when the actual language implementation multiplexes several threads onto fewer CPUs using a scheduler, provided the latter executes a memory barrier at every context switch; see Appendix B for discussion.

$[\![\mathcal{C}(L^\sharp)]\!]$. Hence, any observable behaviour of the compiled code using the TSO library implementation is included into the intended semantics of the program defined by the extended memory model. Therefore, our conditions entail the compiler correctness.

The conditions allow for a separate consideration of the hardware memory model and the run-time system implementation when reasoning about the correctness of a compiler from a DRF language to a TSO machine. Namely, (ii) checks the correctness of the compiler while ignoring the fact that the target machine has a weak memory model and assuming that the run-time system is implemented correctly. Conditions (iii) and (iv) then ensure the correctness of the compiled code on TSO, and condition (i), the correctness of the run-time system.

Establishing (iii) requires ensuring the DRF of the compiled code given the DRF of the source program in the high-level language. In practice, this might require the compiler to insert additional memory barriers. For example, the SC fragment of C++ [3, 2] includes so-called ***strong atomic*** operations, whose concurrent accesses to the same location are not considered a race. The DRF of the high-level program thus ensures that, in the compiled code, we cannot have a race in the sense of Definition 9, except between instructions resulting from strong atomic operations. To prevent the latter, existing barrier placement schemes for C++ compilation on TSO [2] include a memory barrier when translating a strong atomic write. As this prevents a race in the sense of Definition 9, these compilation schemes satisfy our conditions.

Discussion. Theorem 13 is more subtle than might seem at first sight. The crux of the matter is that, like Theorem 10, it allows checking DRF on the SC semantics of the program. This makes the theorem powerful in practice, but requires its proof to show that a trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ with a visible non-SC behaviour can be converted into one from $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$ exhibiting a race. Proving this is non-trivial. A naive attempt to prove the theorem might first replace L with its linearization $L^\sharp$ using Theorem 5 and then try to apply a variant of Theorem 10 to show that the resulting program is SC:

$$\mathsf{visible}([\![C(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}) \subseteq \mathsf{visible}([\![C(L^\sharp)]\!]_{\mathsf{SC}}).$$

However, the second inclusion does not hold even if $C(L^\sharp)$ is DRF, as Theorem 10 does not generalise to the TSO/SC semantics. Indeed, take the spinlock implementation and specification from Figure 1 as L and $L^\sharp$ and consider the following client C:

```
                          x = y = 0;
acquire(); x = 1; release(); || lock; y = 1; unlock;
b = y;                       || acquire(); a = x; release();
```

$$\{\mathtt{a} = \mathtt{b} = 0\}$$

The outcome shown is allowed by $[\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}$, but disallowed by $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$, even though the latter is DRF. It is also disallowed by $[\![C(L)]\!]_{\mathsf{TSO}}$: in this case, the first thread can only read 0 from `y` if the second thread has not yet executed `y = 1;` but when the second thread later acquires the lock, the write of 1 to `x` by the first thread is guaranteed to have been flushed into the memory, and so the second thread has to read 1 from `x`. The trouble is that $[\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}$ loses such correlations between the store buffer usage by the client and the library, which are important for mapping a non-SC trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ into a racy trace from $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$. The need for maintaining

the correlations leads to a subtle proof that uses a non-standard variant of TSO to first make the client part of the trace SC and only then replace the library L with its SC specification $L^\sharp$. See Appendix C for a more detailed discussion.

6 Using Robustness Criteria More Flexible than DRF

Assume that code inside a lock..unlock block accesses at most one memory location.

DEFINITION 14. *A **quadrangular race** is a fragment of an SC trace of the form:*

$$(t, \mathsf{write}(x, \_))\ \tau_1\ (t, \mathsf{read}(y, \_))\ \mathsf{block}((t', \mathsf{write}(y, \_))\ (t', \mathsf{flush}(y, \_)))\ \tau_2\ \mathsf{block}(\varphi),$$

where $\varphi \in \{(t'', \mathsf{write}(x, \_))\ (t'', \mathsf{flush}(x, \_)), (t'', \mathsf{read}(x, \_))\}$, $t \neq t'$, $t \neq t''$, $x \neq y$, τ_1 *contains only actions by* t, *and* τ_1, τ_2 *do not contain* (t, unlock). *A program* P *is **quadrangular-race free (QRF)**, if so are traces in* $[\![P]\!]_{\mathsf{SC}}$.

THEOREM 15 (Robustness via QRF). *If* P *is QRF and visible locations are protected in it, then* $\mathsf{visible}([\![P]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![P]\!]_{\mathsf{SC}})$.

This improves on a criterion by Owens [11], which does not require the last access to x, and thus falsely signals a possible non-SC behaviour when x is local to thread t.

Unfortunately, QRF cannot be used to simplify establishing TSO-to-SC linearizability, because Theorem 11 does not hold if we assume only that L is QRF. Intuitively, transforming a TSO trace satisfying QRF into an SC one can rearrange calls and returns in ways that break linearizability. Formally, the spinlock L_{spinlock} in Figure 1a is QRF, and its history (1) has a single linearization—itself. However, it cannot be reproduced when executing L_{spinlock} on an SC memory model. Moreover, the QRF of a library does not imply Theorem 13 for $L^\sharp = L$ (Appendix B). We now show that Theorem 13 can be recovered for QRF libraries under a stronger assumption on the client.

DEFINITION 16. *A program* $C(L)$ *is **strongly DRF** if it is DRF and traces in* $\mathsf{client}([\![C(L)]\!]_{\mathsf{SC}})$ *do not contain fragments of the form* $(t, \mathsf{read}(x, \_))\ \mathsf{block}((t', \mathsf{write}(x, \_))\ (t', \mathsf{flush}(x, \_)))$, *where* $t \neq t'$.

THEOREM 17. *If* L *is QRF,* $C(L)$ *is strongly DRF, and visible locations are protected in it, then* $\mathsf{visible}([\![C(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![C(L)]\!]_{\mathsf{SC}})$.

When C is compiled from C++, the requirement that C be strongly DRF prohibits the C++ program from using strong atomic operations, which is restrictive.

7 Related Work

To the best of our knowledge, there has been no research on modularly checking the interoperability between components written for different language and hardware memory models. For example, the existing proof of correctness of C++ compilation to x86 [2] does not consider the possibility of a C++ program using arbitrary native components and assume fixed implementations of C++ synchronisation primitives in the run-time system. In particular, the correctness proofs would no longer be valid if

we changed the run-time system implementation. As we discuss in Section 5, this paper provides conditions for an *arbitrary* run-time system implementation of a DRF language ensuring the correctness of the compilation.

We have previously proposed a generalisation of linearizability to the TSO memory model [6] (TSO-to-TSO linearizability in Section 3). Unlike TSO-to-SC linearizability, it requires specifications to be formulated in terms of low-level hardware concepts, and thus cannot be used for interfacing with high-level languages. Furthermore, the technical focus of [6] was on establishing Theorem 7, not Theorem 13.

To concentrate on the core issues of handling interoperability between TSO and DRF models, we assumed that the data structures of the client and its libraries are completely disjoint. Recently, we have proposed a generalisation of classical linearizability that allows the client to communicate with the libraries via data structures [7]. We hope that the results from the two papers can be combined to lift the above restriction.

Acknowledgements. We thank Matthew Parkinson and Serdar Tasiran for comments that helped to improve the paper. Yang was supported by EPSRC.

References

1. J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, 2011.
2. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.
3. H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.
4. A. Bouajjani, R. Meyer, and E. Mohlmann. Deciding robustness against total store ordering. In *ICALP*, 2011.
5. D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd ed.* O'Reilly, 2005.
6. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.
7. A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.
8. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.
9. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, 1979.
10. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.
11. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, 2010.
12. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

A Formal Definition of the TSO Semantics

In this section, we give a formal definition of the abstract TSO machine described informally in Section 2.

Notation. We write A^* for the sets of all (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 $\alpha_1\alpha_2$. 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 denote the powerset of a set X with $\mathcal{P}(X)$, and the disjoint union of sets with $\uplus$.

Control-Flow Graphs. We represent thread and method bodies in the language of Section 2 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 $(N, F, \mathsf{start}, \mathsf{end})$, consisting of the set of program positions N, the control-flow relation $F \subseteq N \times \mathsf{PComm} \times N$, 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 $(N_t, F_t, \mathsf{start}_t, \mathsf{end}_t)$, and the body C_m of a method m by $(N_m, F_m, \mathsf{start}_m, \mathsf{end}_m)$. 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 $F = \biguplus_{t=1}^{n} F_t \uplus \biguplus_{m \in \mathsf{sig}(L)} F_m$.

Machine Configurations. The set of configurations Config our machine can be in is formally defined in Figure 3. 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 we disallow nested method calls). 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 function from 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 lock and unlock. 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 the newest one coming first.

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

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

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 $\mathtt{r}_1$;

$$\mathsf{Heap} = \mathsf{Loc} \to \mathsf{Val} \qquad \mathsf{Buff} = (\mathsf{Loc} \times \mathsf{Val})^* \qquad \mathsf{Pos} = N \uplus (N \times N)$$
$$\mathsf{Reg} = \{\mathtt{r}_1, \ldots, \mathtt{r}_m\} \qquad \mathsf{RegBank} = \mathsf{Reg} \to \mathsf{Val}$$
$$\mathsf{Config} = (\mathsf{CPUid} \to \mathsf{Pos}) \times (\mathsf{CPUid} \to \mathsf{RegBank}) \times (\mathsf{CPUid} \to \mathsf{Buff}) \times \mathsf{Heap} \times \mathcal{P}(\mathsf{CPUid})$$

Fig. 3. The set of machine configurations

– 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. As we noted in Section 2, unlock has a built-in memory barrier, flushing the store buffer of the CPU executing it. We call a sequence of commands bracketed by lock and unlock 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, lock and unlock commands alternate correctly. We also assume that every method called in the program is defined, and 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 $\mathsf{assume}(E)$ command acts as a filter on states, choosing only those where E evaluates to non-zero values. Using $\mathsf{assume}(E)$, 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.

Operational 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 4. We remind the

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

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

$$\frac{}{\mathsf{pc}, \theta, b[t:\alpha\,(x,u)], h, K \xrightarrow{(t, \mathsf{flush}(x,u))}_{C(L)} \mathsf{pc}, \theta, b[t:\alpha], h[x:u], K} \text{ FLUSH}$$

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

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

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

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

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

Fig. 4. Operational TSO semantics

reader that F 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:

$$\hat{F} = F \cup \{((v, v_0), c, (v', v_0)) \mid (v, c, v') \in F \wedge v_0 \in N\}.$$

The rules of the semantics formalise the transitions of the machine explained informally in Section 2 as follows:

- 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 in the store buffer is the leftmost one.
- A CPU may at any point decide to flush the entry at the head of the store buffer into memory (FLUSH).
- 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:

$$\mathsf{lookup}(\alpha, h, x) = \begin{cases} u, & \text{if } \alpha = \alpha_1\,(x,u)\,\alpha_2 \text{ and } \alpha_1 \text{ does not contain entries for } x; \\ h(x), & \text{if } \alpha \text{ does not contain entries for } x. \end{cases}$$

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. According to READ, the value read is stored in the register $\mathtt{r}_1$.

- A CPU executing lock makes itself the only active CPU, preventing the others from executing commands(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. The unlock command can only be executed when the store buffer is empty and thus forces the CPU to flush its store buffer beforehand using FLUSH.
- 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, 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. Note that configurations in CALL and RET rules have CPUid as the set of active CPUs, since we prohibit method calls inside atomic blocks.

A ***computation*** of $C(L)$ is a finite sequence of transitions using $\longrightarrow_{C(L)}$. For a computation λ, we let $\mathsf{trace}(\lambda)$ be the trace obtained by concatenating all the annotations of transitions in λ. We denote with $\xrightarrow{\tau}{}^*_{C(L)}$ the reflexive and transitive closure of $\longrightarrow_{C(L)}$, where τ is obtained by concatenating the transition annotations. Let the set of initial configurations of $C(L)$ be

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

For simplicity, Σ_0 allows the program to execute from an arbitrary heap. We define the semantics $[\![C(L)]\!]_{\mathsf{TSO}}$ of $C(L)$ as the set of traces of its computations with initial configurations from Σ_0 that do not have a LOCK transition unmatched by UNLOCK.

B Additional Definitions and Examples

Formal Definition of the Most General Client. The command C_t^{mgc} used in the most general client has the CFG $(\{v^t_{\mathsf{mgc}}\}, \{(v^t_{\mathsf{mgc}}, \mathsf{havoc}, v^t_{\mathsf{mgc}}), (v^t_{\mathsf{mgc}}, m, v^t_{\mathsf{mgc}}) \mid m \in \mathsf{sig}(L)\}, v^t_{\mathsf{mgc}}, v^t_{\mathsf{mgc}})$.

Formal Definition of the TSO/SC Semantics. The TSO/SC semantics is defined like the semantics in Appendix A, but where the WRITE rule in the case of a command c belonging to the library code ($\rho, \rho' \in N \times N$) is replaced by the following one:

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

```
word x = 1;

void acquire() {
  while (1) {
    lock;
    x--;
    if (x >= 0) {
      unlock;
      return;
    }
    unlock;
    while (x <= 0) ;
  }
}

void release() { x = 1; }

int tryacquire() {
  lock;
  x--;
  unlock;
  if (x >= 0) {
    unlock;
    return 1;
  }
  unlock;
  return 0;
}
```

(a)

```
struct Spinlock {
  short served = 1;
  short next = 1;
};
Spinlock lk;

void acquire() {
  short ticket;
  lock;
  ticket = lk.next++;
  unlock;
  while (lk.served != ticket) ;
}

release () { lk.served++; }

int tryacquire() {
  Spinlock old, new;
  new = old = lk;
  if (old.served != old.next)
    return 0;
  new.next++;
  lock;
  if (lk == old) {
    lk = new;
    unlock;
    return 1;
  }
  unlock;
  return 0;
}
```

(b)

Fig. 5. Two spinlock implementations used in different versions of the Linux kernel

Other Spinlock Implementations. Figure 5 gives two spinlock implementations used in different versions of the Linux kernel, 2.6.24.7 and 3.2.4, respectively. The first one (Figure 5a) implements `acquire` using an atomic decrement, instead of a CAS. The second one (Figure 5b) ensures fairness using a variant of the Bakery algorithm. In the algorithms, we assume that integers are unbounded (or, equivalently, consider only executions where an overflow does not occur). In the second algorithm we also assume that the structure storing the lock state can be read atomically. Both algorithms are linearized by the abstract implementation in Figure 1b.

An Operational Seqlock Specification. In Section 3, we gave a specification to the seqlock library as a set of histories. Figure 6 gives an operational SC specification $L^{\sharp}_{\mathsf{seqlock}}$ of a seqlock, such that $L_{\mathsf{seqlock}} \sqsubseteq L^{\sharp}_{\mathsf{seqlock}}$. The specification 'virtualises' the TSO store

```
Queue<pair<word,word>> q = {(0, 0)};

write(in word d1, in word d2) {
  q.enqueue(d1,d2);
}
```

```
read(out word d1, out word d2) {
  while (*)
    q.dequeue();
  (d1,d2) = q.top();
}
```

Fig. 6. An operational SC specification $L^{\sharp}_{\mathsf{seqlock}}$ of a seqlock. `Queue` is an abstract data type of a FIFO queue storing pairs of words, whose operations are executed atomically; `dequeue` does nothing when the queue is empty; and $*$ denotes non-deterministic choice.

buffer using an abstract queue data type.

Establishing the Correspondence between Language-Level and Hardware-Level Threads. In Section 5, we defined the extension of a high-level language memory model with a low-level library using histories of the latter: the set of executions in the client-local semantics of the high-level program is projected to those with valid library histories. This composition has a mismatch: actions in library histories are indexed by physical CPU identifiers, whereas language-level memory models are defined in terms of threads of the high-level language. In practice, a scheduler multiplexes several threads over fewer physical CPUs. We now justify why our way of composing the semantics of the client and the library is sound in this situation.

In the following, we do not consider dynamic thread creation so as not to obfuscate discussion. Let $\mathsf{NThreads}$ be the number of threads. Assume that the scheduler executes a memory barrier at every context-switch, which is true of schedulers in common operating system kernels [5]. Let $C(L)$ be a TSO program compiled from the high-level language, which runs under the management of a scheduler. Consider a semantics of this program, in which we do not have the scheduler, but instead the machine has the number of CPUs equal to the number $\mathsf{NThreads}$ of threads. In particular, every thread has its own store buffer. Such a semantics is a sound approximation of the TSO semantics with the scheduler, in the sense that any visible behaviour that $C(L)$ can produce on the latter can be reproduced on the former. Intuitively, this is because a correct scheduler establishes the illusion that every thread owns a dedicated CPU, and flushing the store buffer at every context switch does not let a thread notice that the actual store buffers it uses change. This can be formalised using existing techniques[6]. Hence, we can restrict ourselves to reasoning in the setting where the number of CPUs is equal to the number of threads, for which the composition of client and library semantics we used is valid.

Extension of the C++ Memory Model with Native TSO Libraries. Assume a TSO library L with an SC specification given by a set of histories T. We consider the C++ memory model as defined in [2], but excluding weak atomics, where every method of L implemented by a C++ stub that has an empty body and returns an arbitrary value. We then leave only those executions in this semantics, whose happens-before relations hb are consistent with T:

$$\forall H.\, (a \xrightarrow{\mathsf{hb}} b \Rightarrow (a \text{ precedes } b \text{ in } H)) \Rightarrow H \in T,$$

[6] A. Gotsman and H. Yang. Modular verification of preemptive OS kernels. In *ICFP*, 2011.

where a, b range over call and return events. That is, for any history consistent with the happens-before relation there exists a history from T linearizing it.

Example showing that the QRF of a library does not imply Theorem 13 for $L^\sharp = L$. Consider the spinlock library in Figure 1a, which is QRF, but not DRF. Take the following client:

$$\begin{array}{c} \texttt{u = 0;} \\ \texttt{acquire(); release(); a = u;} \;\|\; \texttt{lock; u = 1; unlock; b = tryacquire();} \\ \{\texttt{a} = 0 \wedge \texttt{b} = 0\} \end{array}$$

The client is DRF, as the write to `u` is protected by a memory barrier. On TSO, it is possible to get the outcome shown above when the write to `x` representing the state of the lock (Figure 1) is delayed in the store buffer of the first thread. This is not possible on SC: if `tryacquire` returns 0, then it has to happen before `release`, but then the write to `u` happens before the read from it, so this read has to return 1.

C Proofs

C.1 Proof Sketch for Proposition 4

We prove the linearizability relation $L_{\mathsf{spinlock}} \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp_{\mathsf{spinlock}}$ using the method of linearization points. Consider a computation of $\mathsf{MGC}(L_{\mathsf{spinlock}})$ on TSO with a history H. For every method invocation in this computation, we identify a single transition that is the linearization point of this method—the point where the method, informally, 'takes effect'. Given this, the required computation of $\mathsf{MGC}(L^\sharp_{\mathsf{spinlock}})$ on SC with a history linearizing H is constructed as follows. We generate a call or a return transition by $\mathsf{MGC}(L^\sharp_{\mathsf{spinlock}})$ at every call or return transition in the computation of $\mathsf{MGC}(L_{\mathsf{spinlock}})$, and we execute the body of the abstract method at every linearization point.

The linearization points of the spinlock methods are defined as follows:

- `acquire`: the UNLOCK transition before the `return` statement.
- `release`: the WRITE transition;
- `trylock`: the UNLOCK transition;

It is easy to check that methods in the resulting computation of the abstract library implementation return the same values as in the original one. In particular, when a `tryacquire` in the TSO computation returns 0 due to a write to `x` not flushed into the memory, in the corresponding SC computation we already have $\texttt{x} = 1$. In this case, `tryacquire` returns 0 in the SC computation using the non-deterministic branch in its SC implementation. □

C.2 Proof Sketch for Proposition 8

To ease the exposition, Figure 7 shows the implementation of the `write` method of a seqlock including code labels, which correspond to nodes in its CFG. In the following, we assume that CPU 1 runs the code of the writer. Let v_{wc} be the node of the corresponding thread body of the most general client.

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

write(in word d1, in word d2) {
v0: c++;
v1: x1 = d1;
v2: x2 = d2;
v3: c++;
v4:
}
```

Fig. 7. The implementation of seqlock's `write` method with code labels

Consider a computation of $\mathsf{MGC}(L_{\mathsf{seqlock}})$ on TSO with a history H. We again construct the history H' linearizing H using linearization points, associated with method invocations as follows:

- `read`: the last read of `x2` before returning;
- `write`: the second write to `c`.

If a method invocation terminates, then we associate a pair of the corresponding call and return actions with its linearization point. If a `write` method does not terminate, but executes the linearization point, then we associate the call action with the point. Otherwise, we associate a call action for `write` with the final point in the computation. If a `read` method does not terminate, we associate a call action with its call. We obtain the desired history H' by concatenating these actions associated with every element of the sequence of the code points. It is easy to see that $H \sqsubseteq H'$. We now show that $H' \in T_{\mathsf{seqlock}}$.

Consider an arbitrary prefix λ of the computation of $\mathsf{MGC}(L_{\mathsf{seqlock}})$ and the corresponding prefix H'_0 of the linearizing history H'. We prove that $\mathsf{reads}(H_0)$ is a subsequence of a sequence from $((0,0)\,\mathsf{writes}(H_0))^\dagger$ by induction on the length of λ. The induction hypothesis includes a supporting invariant relating $\mathsf{reads}(H'_0)$ and $\mathsf{writes}(H'_0)$ with the final configuration of λ, which is formulated in Figure 8. The auxiliary function g converts the store buffers of the writer CPU in $\mathsf{MGC}(L_{\mathsf{seqlock}})$ to the sequences of values to be written. Note that the last clause of the invariant implies that, at a linearization point of `read`, the reader will read values correctly positioned in the sequence of writes. The induction step amounts to a case analysis on all possible transition of $\mathsf{MGC}(L_{\mathsf{seqlock}})$. □

C.3 Proof of Theorem 5

In the following, we use the ***client-local semantics*** of a client C, which defines its set of traces $[\![C]\!]_{\mathsf{TSO}}$ assuming any behaviour of the library methods it calls. The set $[\![C]\!]_{\mathsf{TSO}}$ is defined as the set of traces of the program

$$C(\cdot) = (\mathsf{let}\ \{m = C_m^{\mathsf{stub}} \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_{\mathsf{NCPUs}}),$$

$$g : \mathsf{RegBank} \times \mathsf{Pos} \times \mathsf{Heap} \times \mathsf{Buff} \rightharpoonup (\mathsf{Val} \times \mathsf{Val})^*$$

$$\begin{aligned}
g(r,\rho,h,\varepsilon) &= \varepsilon, \\
g(r,\rho,h,(\mathtt{c},c)\,\alpha) &= g(r,\rho,h,\alpha), && c \text{ is odd}, \rho = (\mathtt{v1}, v_{\mathsf{wc}}) \\
g(r,\rho,h,(\mathtt{x1},r(\mathtt{d1}))(\mathtt{c},c)\,\alpha) &= g(r,\rho,h,\alpha), && c \text{ is odd}, \rho = (\mathtt{v2}, v_{\mathsf{wc}}) \\
g(r,\rho,h,(\mathtt{x2},r(\mathtt{d2}))(\mathtt{x1},r(\mathtt{d1}))(\mathtt{c},c)\,\alpha) &= g(r,\rho,h,\alpha), && c \text{ is odd}, \rho = (\mathtt{v3}, v_{\mathsf{wc}}) \\
g(r,\rho,h,(\mathtt{c},c+1)(\mathtt{x2},x_2)(\mathtt{x1},x_1)(\mathtt{c},c)\,\alpha) &= (x_1,x_2)\,g(r,\rho,h,\alpha), && c \text{ is odd} \\
g(r,\rho,h,(\mathtt{c},c')(\mathtt{x2},x_2)(\mathtt{x1},x_1)) &= (x_1,x_2), && c' \text{ is even} \\
g(r,\rho,h,(\mathtt{c},c')(\mathtt{x2},x_2)) &= (h(\mathtt{x1}),x_2), && c' \text{ is even} \\
g(r,\rho,h,(\mathtt{x2},r(\mathtt{d2}))(\mathtt{x1},r(\mathtt{d1}))) &= \varepsilon, \\
& && h(\mathtt{c}) \text{ is odd}, \rho = (\mathtt{v3}, v_{\mathsf{wc}}) \\
g(r,\rho,h,(\mathtt{c},c')) &= (h(\mathtt{x1}),h(\mathtt{x2})), && c' \text{ is even} \\
g(r,\rho,h,(\mathtt{x2},r(\mathtt{d2}))) &= \varepsilon, \\
& && h(\mathtt{c}) \text{ is odd}, h(\mathtt{x1}) = r(\mathtt{d1}), \rho = (\mathtt{v3}, v_{\mathsf{wc}}) \\
g(r,\rho,h,(\mathtt{x1},r(\mathtt{d1}))) &= \varepsilon, \\
& && h(\mathtt{c}) \text{ is odd}, \rho = (\mathtt{v2}, v_{\mathsf{wc}})
\end{aligned}$$

Let $(\mathsf{pc}[1:\rho], \theta[1:r], b[1:\alpha], h, K)$ be the final configuration of λ. The invariant is as follows:

$$\begin{aligned}
&(\alpha = \varepsilon \Rightarrow ((h(\mathtt{c}) \text{ is even}) \wedge \rho \in \{(\mathtt{v4}, v_{\mathsf{wc}}), (\mathtt{v0}, v_{\mathsf{wc}}), v_{\mathsf{wc}}\}) \vee \\
&\quad ((h(\mathtt{c}) \text{ is odd}) \wedge \rho \in \{(\mathtt{v1}, v_{\mathsf{wc}}), (\mathtt{v2}, v_{\mathsf{wc}}), (\mathtt{v3}, v_{\mathsf{wc}})\} \wedge \\
&\quad (\rho \in \{(\mathtt{v2}, v_{\mathsf{wc}}), (\mathtt{v3}, v_{\mathsf{wc}})\} \Rightarrow h(\mathtt{x1}) = r(\mathtt{d1})) \wedge (\rho = (\mathtt{v3}, v_{\mathsf{wc}}) \Rightarrow h(\mathtt{x2}) = r(\mathtt{d2})))) \wedge \\
&((\exists \beta, c'.\, \alpha = (\mathtt{c}, c')\,\beta \wedge (c' \text{ is even})) \Rightarrow \rho \in \{v_{\mathsf{wc}}, (\mathtt{v4}, v_{\mathsf{wc}}), (\mathtt{v0}, v_{\mathsf{wc}})\}) \wedge \\
&(\exists \beta, \gamma, c.\, \alpha = \beta\,(\mathtt{c}, c)\,\gamma \wedge (\gamma \text{ does not contain entries for } \mathtt{c}) \Rightarrow h(\mathtt{c}) = c - 1) \wedge \\
&(\exists \beta.\, \mathsf{writes}(H_0') = \beta\, g(r,\rho,h,\alpha) \wedge (\mathsf{reads}(H_0') \text{ is a subsequence of a sequence in } ((0,0)\,\beta)^\dagger) \wedge \\
&\quad (h(\mathtt{c}) \text{ is even } \Rightarrow ((h(\mathtt{x1}), h(\mathtt{x2})) \text{ is the last element of } (0,0)\,\beta))).
\end{aligned}$$

Fig. 8. The supporting invariant for the proof of the linearizability of a seqlock

where M is the set of methods C may call and the command C_m^{stub} has the CFG

$$(\{v_{\mathsf{start}}^m, v_{\mathsf{end}}^m\}, \{(v_{\mathsf{start}}^m, \mathsf{havoc}, v_{\mathsf{end}}^m)\}, v_{\mathsf{start}}^m, v_{\mathsf{end}}^m).$$

The proof of the theorem relies on the following lemmas. The first one states that a TSO trace of $C(L)$ generates two traces in the client-local and library-local semantics with the same history.

LEMMA 18 (Decomposition on TSO).

$$\begin{aligned}
&\forall \tau \in [\![C(L)]\!]_{\mathsf{TSO}}.\, \exists \eta \in [\![C]\!]_{\mathsf{TSO}}.\, \exists \xi \in [\![L]\!]_{\mathsf{TSO}}. \\
&\qquad \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{client}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{lib}(\xi).
\end{aligned}$$

The following lemma shows that an SC trace of a most general client of a library can be transformed into another one of its traces with a given history linearized by the history of the original one.

LEMMA 19 (Rearrangement on SC).

$$\begin{aligned}
&\forall H, H'.\, H \sqsubseteq H' \Rightarrow (\forall \tau' \in [\![L]\!]_{\mathsf{SC}}.\, \mathsf{history}(\tau') = H' \Rightarrow \\
&\qquad \exists \tau \in [\![L]\!]_{\mathsf{SC}}.\, \mathsf{history}(\tau) = H).
\end{aligned}$$

Finally, the following lemma states that any pair of a client-local TSO trace and an SC library-local trace agreeing on the history can be combined into a trace of $C(L)$ on the mixed TSO/SC semantics.

LEMMA 20 (Composition on TSO/SC).

$$\forall \eta \in [\![C]\!]_{\mathsf{TSO}}.\, \forall \xi \in [\![L]\!]_{\mathsf{SC}}.\, \mathsf{history}(\eta) = \mathsf{history}(\xi) \Rightarrow \\ \exists \tau \in [\![C(L)]\!]_{\mathsf{TSO/SC}}.\, \mathsf{client}(\tau) = \mathsf{client}(\eta).$$

Lemmas 18 and 19 are variants of theorems proved in [6]. Lemma 20 is proved below.

Proof of Theorem 5. Consider $\tau_1 \in [\![C(L_1)]\!]_{\mathsf{TSO}}$. By Lemma 18, τ_1 generates two traces—a library-local trace $\xi_1 \in [\![L_1]\!]_{\mathsf{TSO}}$ and a client-local one $\eta \in [\![C]\!]_{\mathsf{TSO}}$—such that $\mathsf{client}(\tau_1) = \mathsf{client}(\eta)$ and $\mathsf{history}(\eta) = \mathsf{history}(\xi_1)$. Since $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_2$, for some trace $\xi_2 \in [\![L_2]\!]_{\mathsf{SC}}$, we have $\mathsf{history}(\xi_1) \sqsubseteq \mathsf{history}(\xi_2)$. By Lemma 19, ξ_2 can be transformed into a trace $\xi_2' \in [\![L_2]\!]_{\mathsf{SC}}$ such that $\mathsf{history}(\xi_2') = \mathsf{history}(\xi_1) = \mathsf{history}(\eta)$. We then use Lemma 20 to compose the library-local SC trace ξ_2' with the client-local TSO trace η into a trace $\tau_2 \in [\![C(L_2)]\!]_{\mathsf{TSO/SC}}$ such that $\mathsf{client}(\tau_2) = \mathsf{client}(\eta) = \mathsf{client}(\tau_1)$. □

We now proceed to prove Lemma 20. We first define an auxiliary partial operation $\circ : \mathsf{Config} \times \mathsf{Config} \rightharpoonup \mathsf{Config}$ that combines configurations in the local semantics of C and L to yield a configuration of $C(L)$. We first define $\circ$ on the components of a configuration:

- We define $\circ$ on CPU positions as follows: $v \circ v^t_{\mathsf{mgc}} = v$ for $v \in \biguplus_{t=1}^{n} N_t$; $(v^m_k, v) \circ (v', v^t_{\mathsf{mgc}}) = (v', v)$ for $k \in \{\mathsf{start}, \mathsf{end}\}$, $v \in \biguplus_{t=1}^{n} N_t$ and $v' \in N_m$; undefined in all other cases. We lift $\circ$ to program counters pointwise.
- For $r_1, r_2 \in \mathsf{RegBank}$ and $\rho \in \mathsf{Pos}$ we let

$$r_1 \circ_\rho r_2 = \begin{cases} r_1, & \text{if } \rho \in N; \\ r_2, & \text{if } \rho \in N \times N. \end{cases}$$

  For $\theta_1, \theta_2 \in \mathsf{CPUid} \to \mathsf{RegBank}$ and $\mathsf{pc} \in \mathsf{CPUid} \to \mathsf{Pos}$ we then let

$$\forall t \in \mathsf{CPUid}.\, (\theta_1 \circ_{\mathsf{pc}} \theta_2)(t) = \theta_1(t) \circ_{\mathsf{pc}(t)} \theta_2(t).$$

- We define $\circ$ on store buffers as follows: $\alpha \circ \varepsilon = \alpha$ for all $\alpha \in \mathsf{Buff}$; undefined in all other cases. We lift $\circ$ to vectors of store buffers pointwise.
- For $h_1, h_2 \in \mathsf{Heap}$ we define $h_1 \circ h_2$ as follows:

$$\forall x \in \mathsf{Loc}.\, (h_1 \circ h_2)(x) = \begin{cases} h_1(x), & \text{if } x \in \mathsf{CLoc}; \\ h_2(x), & \text{if } x \in \mathsf{LLoc}. \end{cases}$$

- We define $\circ$ on sets of active CPUs as follows: $\mathsf{CPUid} \circ \mathsf{CPUid} = \mathsf{CPUid}$, $\{t\} \circ \mathsf{CPUid} = \mathsf{CPUid} \circ \{t\} = \{t\}$; undefined in all other cases.

Finally, we lift $\circ$ to configurations:

$$(\mathsf{pc}_1, \theta_1, b_1, h_1, K_1) \circ (\mathsf{pc}_2, \theta_2, b_2, h_2, K_2) = \\ (\mathsf{pc}_1 \circ \mathsf{pc}_2, \theta_1 \circ_{(\mathsf{pc}_1 \circ \mathsf{pc}_2)} \theta_2, b_1 \circ b_2, h_1 \circ h_2, K_1 \circ K_2).$$

Proof of Lemma 20. Assume $\eta' \in [\![C]\!]_{\mathsf{TSO}}$ and $\xi' \in [\![L]\!]_{\mathsf{SC}}$ such that $\mathsf{history}(\eta') = \mathsf{history}(\xi')$. Let η be the shortest prefix of η' such that $\mathsf{client}(\eta) = \mathsf{client}(\eta')$, and ξ, the shortest prefix of ξ' such that $\mathsf{history}(\xi) = \mathsf{history}(\xi')$; let λ^c and λ^l be the shortest prefixes of the computations corresponding to η' and ξ' that produce η and ξ. Then λ^c does not end with a havoc transition in a method stub, and the last transition of λ^l, when it exists, is a call or a return. Without loss of generality we can assume that λ^l is generated by the TSO/SC semantics. We now construct a computation λ of $C(L)$ with the desired trace τ from these preprocessed computations λ^c and λ^l. The properties of λ^c and λ^l noted above are used during the construction.

Let σ_0^1 and σ_0^2 be the initial configurations of the computations λ^c and λ^l; then $\sigma_0^1 \circ \sigma_0^2$ is defined. We first build a series of finite computations $\lambda_0, \lambda_1, \lambda_2, \ldots$, such that for $i < j$, λ_i is a prefix of λ_j. Thus, the series has the limit, which is the desired computation λ. In the following, we consider prefixes λ_i^c and λ_i^l of computations λ^c and λ^l. We let $\tau_i = \mathsf{trace}(\lambda_i)$, $\eta_i = \mathsf{trace}(\lambda_i^c)$ and $\xi_i = \mathsf{trace}(\lambda_i^l)$.

The first element in the series is the empty computation consisting of the initial configuration $\sigma_0^1 \circ \sigma_0^2$ only. For the $(i+1)$-st element with $i \geq 0$, we assume that the i-th element λ_i has been constructed and satisfies the following property:

For some finite prefixes λ_i^c of λ^c and λ_i^l of λ^l of the form

$$\sigma_0^1 \xrightarrow{\mathsf{client}(\tau_i)}{}^*_{C(\cdot)} \sigma_i^1 \ \wedge \ \sigma_0^2 \xrightarrow{\mathsf{lib}(\tau_i)}{}^*_{\mathsf{MGC}(L)} \sigma_i^2,$$

we have

$$\mathsf{history}(\eta_i) = \mathsf{history}(\xi_i) \ \wedge \ \mathsf{client}(\tau_i) = \mathsf{client}(\eta_i) \ \wedge \ \mathsf{lib}(\tau_i) = \mathsf{lib}(\xi_i).$$

Furthermore, $\sigma_i^1 \circ \sigma_i^2$ is defined and λ_i is:

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{\tau_i}{}^*_{C(L)} \sigma_i^1 \circ \sigma_i^2.$$

We now define the $(i+1)$-th element λ_{i+1} that maintains this property. As we explained above, the computation λ_{i+1} is an extension of λ_i by one or more steps.

Let the following be the next transitions in the computations λ^c and λ^l (we consider the case when one of the computations has no next transition later):

$$\sigma_i^1 \xrightarrow{\varepsilon}{}^*_{C(\cdot)} \sigma_1 \xrightarrow{\tau'}{}_{C(\cdot)} \sigma_1' \ \wedge \ \sigma_i^2 \xrightarrow{\varepsilon}{}^*_{\mathsf{MGC}(L)} \sigma_2 \xrightarrow{\tau''}{}_{\mathsf{MGC}(L)} \sigma_2', \tag{2}$$

where the computation transforming σ_i^1 into σ_1 is the maximal prefix of the first computation consisting only of havoc transitions in method stubs, and the computation transforming σ_i^2 into σ_2 is the maximal prefix of the second computation consisting only of havoc transitions in the client code of $\mathsf{MGC}(L)$. The non-havoc transitions exist in

both cases, because of the preprocessing step described above. It is easy to show that $\sigma_i^1 \circ \sigma_i^2 = \sigma_1 \circ \sigma_2$, so that

$$\sigma_0^1 \circ \sigma_0^2 \xrightarrow{\tau_i}{}^*_{C(L)} \sigma_1 \circ \sigma_2.$$

To construct λ_{i+1}, we make a case-split on the rules of the operational semantics used to obtain the non-havoc transitions τ' and τ''. We use one of these two transitions to extend λ_i to λ_{i+1}. The construction described below defines λ_{i+1} with a degree of non-determinism: the computation of $C(L)$ can sometimes be extended either using the transition from $C(\cdot)$ or the one from $\mathsf{MGC}(L)$. All possible results produce a valid computation of $C(L)$, and, as we show below, $\mathsf{client}(\tau) = \mathsf{client}(\eta)$.

Below we use symbols K_1 and K_2 to denote the last components of σ_1 and σ_2, respectively.

- CALL in λ^c and CALL in λ^l such that $\tau' = \tau'' = (t, \mathsf{call}\ m(r))$. Then for some v, v', pc_1, pc_2, θ_1, θ_2, b_1, b_2, h_1 and h_2, we have $(v, m, v') \in F$, $\mathsf{pc}_1(t)$, $\mathsf{pc}_2(t)$, $\theta_1(t)$, $\theta_2(t)$ are undefined and

  $$\begin{array}{l}\sigma_1 = (\mathsf{pc}_1[t : v], \theta_1[t : r], b_1, h_1, \mathsf{CPUid}) \ \wedge \\ \sigma_2 = (\mathsf{pc}_2[t : v^t_{\mathsf{mgc}}], \theta_2[t : r], b_2, h_2, \mathsf{CPUid}) \ \wedge \\ \sigma_1' = (\mathsf{pc}_1[t : (v^m_{\mathsf{start}}, v')], \theta_1[t : r], b_1, h_1, \mathsf{CPUid}) \ \wedge \\ \sigma_2' = (\mathsf{pc}_2[t : (\mathsf{start}_m, v^t_{\mathsf{mgc}})], \theta_2[t : r], b_2, h_2, \mathsf{CPUid}) \ \wedge \\ \sigma_1 \circ \sigma_2 = ((\mathsf{pc}_1 \circ \mathsf{pc}_2)[t : v], (\theta_1 \circ_{(\mathsf{pc}_1 \circ \mathsf{pc}_2)} \theta_2)[t : r], b_1 \circ b_2, h_1 \circ h_2, \mathsf{CPUid}).\end{array}$$

  Hence,

  $$\sigma_1' \circ \sigma_2' = ((\mathsf{pc}_1 \circ \mathsf{pc}_2)[t : (\mathsf{start}_m, v')], (\theta_1 \circ_{(\mathsf{pc}_1 \circ \mathsf{pc}_2)} \theta_2)[t : r], b_1 \circ b_2, h_1 \circ h_2, \mathsf{CPUid}).$$

  Then, $\sigma_1 \circ \sigma_2 \xrightarrow{\tau'}_{C(L)} \sigma_1' \circ \sigma_2'$. Thus, in this case the desired λ_{i+1} is obtained by extending λ_i with this transition.
- RET in λ^c and RET in λ^l such that $\tau' = \tau''$. This case is handled similarly to the previous one.
- The transition in λ^l is generated by LOCAL, WRITE-SC, READ, LOCK or UNLOCK, and we have $K_2 \subseteq K_1$. Note that by the definition of $\circ$ on K, at least one of K_1 and K_2 is CPUid. Our condition thus ensures that $K_1 = \mathsf{CPUid}$. We consider only the case of the WRITE-SC rule; the others are analogous. In this case, for some t, x, u, $k \in \{\mathsf{start}, \mathsf{end}\}$, v, v', v_1, θ_1, θ_2, r, r', pc_1, pc_2, b_1, b_2, h_1 and h_2, we have $(v, c, v') \in F$, $c \in \mathsf{Write}$, $(x, u) \in f_c(r)$, $\mathsf{pc}_1(t)$, $\mathsf{pc}_2(t)$, $\theta_1(t)$, $\theta_2(t)$ are undefined and

  $$\begin{array}{l}\sigma_1 = (\mathsf{pc}_1[t : (v^m_k, v_1)], \theta_1[t : r'], b_1, h_1, \mathsf{CPUid}) \ \wedge \\ \sigma_2 = (\mathsf{pc}_2[t : (v, v^t_{\mathsf{mgc}})], \theta_2[t : r], b_2, h_2, K_2) \ \wedge \\ \sigma_2' = (\mathsf{pc}_2[t : (v', v^t_{\mathsf{mgc}})], \theta_2[t : r], b_2, h_2[x : u], K_2) \ \wedge \\ \sigma_1 \circ \sigma_2 = ((\mathsf{pc}_1 \circ \mathsf{pc}_2)[t : (v, v_1)], (\theta_1 \circ_{(\mathsf{pc}_1 \circ \mathsf{pc}_2)} \theta_2)[t : r], b_1 \circ b_2, h_1 \circ h_2, K_2).\end{array}$$

  Since we only consider non-interfering programs, $x \in \mathsf{LLoc}$. Hence, $h_1 \circ (h_2[x : u]) = (h_1 \circ h_2)[x : u]$ and

  $$\sigma_1 \circ \sigma_2' = ((\mathsf{pc}_1 \circ \mathsf{pc}_2)[t : (v', v_1)], (\theta_1 \circ_{(\mathsf{pc}_1 \circ \mathsf{pc}_2)} \theta_2)[t : r], b_1 \circ b_2, (h_1 \circ h_2)[x : u], K_2).$$

Then, $\sigma_1 \circ \sigma_2 \xrightarrow{\tau''}_{C(L)} \sigma_1 \circ \sigma_2'$. The desired λ_{i+1} is obtained by extending λ_i with this transition.

– The transition in λ^c is generated by LOCAL, WRITE, FLUSH, READ, LOCK, UNLOCK, and we have $K_1 \subseteq K_2$. This case is similar to the previous one.

One of the above cases is always applicable. When both K_1 and K_2 are CPUid, the uncovered cases are those when both transitions in λ^c and λ^l are obtained using CALL or RET and the actions produced are different. However, this case is impossible, since $\mathsf{history}(\eta) = \mathsf{history}(\xi)$ and $\mathsf{history}(\eta_i) = \mathsf{history}(\xi_i)$. When one of K_1 and K_2 is not CPUid, our construction covers all the cases: the transition of the local computation whose current configuration has the machine locked is always enabled in the corresponding configuration of the global computation.

Consider now the case when only one transition in (2) exists. Due to our preprocessing step, λ^l is either empty or ends with a transition producing an interface action. Besides, the above construction consumes the latter together with the corresponding transition in λ^c. Hence, the only transition in (2) has to be one from λ^c, and the last configuration in λ^l does not have the machine locked. This implies that the transition in λ^c is enabled in the global configuration, and we can obtain λ^c_{i+1} and λ_{i+1} by extending λ^c_i and λ_i with this transition.

Thus, we have shown how to construct λ_i for all possible cases, sometimes nondeterministically. Besides, our construction consumes both computations completely, so that $\mathsf{client}(\tau) = \mathsf{client}(\eta)$. □

C.4 Proof of Corollary 6

To simplify presentation, in our development we have assumed that any method called in a program by the client belongs to the library. The proof of Theorem 5 can be simply generalised to omit this requirement. Namely, we assume a setting where the client is allowed to have its private methods (as before, nested method calls are disallowed). We correspondingly generalise the client operation on traces to client_M: the new operation interprets the given set of methods M as constituting a library. Theorem 5 still holds in this setting. It also generalises to the case when some of the client-private libraries L execute on the SC semantics. We denote by $\mathsf{TSO/SC}(L)$ the semantics where the methods from L access the memory directly. We now prove Corollary 6 using this generalisation.

The idea of the proof is simple: to show $L \sqsubseteq L^\sharp$ we linearize libraries $L_1, \ldots, L_k$ one by one using Theorem 5. The program $\mathsf{MGC}(L) = \mathsf{MGC}(L_1, \ldots, L_k)$ can be viewed as consisting of the library L_1 and its client including the implementations of methods in $L_2, \ldots, L_k$, with LLoc_1 and $\mathsf{LLoc}_2 \uplus \ldots \uplus \mathsf{LLoc}_k \uplus \mathsf{CLoc}$ as the address spaces of the library and the client, respectively. Since $L_1 \sqsubseteq L_1^\sharp$, by Theorem 5, we can linearize L_1, obtaining

$$\mathsf{client}_{\mathsf{sig}(L_1)}([\![\mathsf{MGC}(L_1, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}_{\mathsf{sig}(L_1)}([\![\mathsf{MGC}(L_1^\sharp, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO/SC}(L_1^\sharp)}). \quad (3)$$

The resulting program $\mathsf{MGC}(L_1^\sharp, L_2, L_3, \ldots, L_k)$ can be viewed as consisting of the library implementing the methods from L_2 and the client including the methods from

$L_1^\sharp, L_3, \ldots, L_k$. Since $L_2 \sqsubseteq L_2^\sharp$, by Theorem 5, we get

$$\mathsf{client}_{\mathsf{sig}(L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp)}) \subseteq \mathsf{client}_{\mathsf{sig}(L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2^\sharp, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp, L_2^\sharp)}). \quad (4)$$

From (3), we get

$$\mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp)}).$$

From (4), we get

$$\mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp)}) \subseteq \mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2^\sharp, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp, L_2^\sharp)}).$$

The last two inclusions entail

$$\mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}_{\mathsf{sig}(L_1,L_2)}([\![\mathsf{MGC}(L_1^\sharp, L_2^\sharp, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}/\mathsf{SC}(L_1^\sharp, L_2^\sharp)}).$$

Repeatedly applying Theorem 5 as above to linearize $L_3, \ldots, L_k$, we get

$$\mathsf{client}_{\mathsf{sig}(L_1,L_2,L_3,\ldots,L_k)}([\![\mathsf{MGC}(L_1, L_2, L_3, \ldots, L_k)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}_{\mathsf{sig}(L_1,L_2,L_3,\ldots,L_k)}([\![\mathsf{MGC}(L_1^\sharp, L_2^\sharp, L_3^\sharp, \ldots, L_k^\sharp)]\!]_{\mathsf{SC}}),$$

which implies $L \sqsubseteq L^\sharp$.

C.5 Proof of Theorem 10

In the following, we omit values from write, read and flush actions when they are not relevant. We write access to mean either read or flush. Consider the k-th write $(t, \mathsf{write}(x))$ to x by thread t in a trace τ. We call the k-th action $(t, \mathsf{flush}(x))$ in τ the flush action ***corresponding*** to the write action $(t, \mathsf{write}(x))$. We also use the easy generalisation of this notion to fragments of computations, which takes into account the initial contents of store buffers. We call a write action in a trace ***immediate***, if it is immediately followed by the corresponding flush. A write action is ***pending*** in a trace, if there is no corresponding flush for it. We call a trace ***complete*** if it has no pending write actions.

To avoid an explosion in the number of different variables during the proof, the scope of all variables ξ_i below is limited to the formula they occur in. References to variables in the text pertain to their last occurrence in a formula.

Consider a DRF program P with visible locations protected. Let λ_0 be a computation of P with a trace τ_0. We preprocess this computation as follows. For every thread t, consider the leftmost WRITE transition by t in λ_0 such that its write action is pending in τ_0. We remove all non-flush transitions by every thread t starting from the WRITE

transition selected for this thread, which makes the trace of the computation complete. Since visible locations are protected in P, this does not change the projection of the trace to visible actions. Besides, for every FLUSH transition within an atomic block, if it corresponds to a write action in the same block, we move it to the position directly after the WRITE transition; otherwise we move it to the position right before the LOCK transition starting the atomic block. Let $\tau = \mathsf{trace}(\lambda)$. Given that visible locations are only accessed in atomic blocks, it is easy to check that λ is a valid computation and $\mathsf{visible}(\tau) = \mathsf{visible}(\tau_0)$.

We now transform the computation λ into a computation with an SC trace that either has the same projections to visible actions as τ, or contains a data race in the sense of Definition 9. To this end, we attempt to construct a sequence of computations λ_i, $i = 1..k$ with prefixes ν_i. Let $\tau_i = \mathsf{trace}(\lambda_i)$ and $\eta_i = \mathsf{trace}(\nu_i)$, $i = 1..k$. The sequence of computations we construct satisfies the following conditions:

- $\lambda_1 = \lambda$ and ν_1 is the empty computation consisting of the initial configuration of λ_1 only;
- η_i is SC, $i = 1..k$;
- ν_{i+1} extends ν_i by one or more transitions, $i = 1..(k-1)$;
- the end configuration of λ_{i+1} is the same as that of λ_i, $i = 1..(k-1)$;
- $\mathsf{visible}(\tau_{i+1}) = \mathsf{visible}(\tau_i)$, $i = 1..(k-1)$;
- $\lambda_k = \nu_k$.

Hence, if the construction is successful, then the final computation λ_k and its trace τ_k are the desired ones.

Assume λ_i and ν_i for some $i \in \{1, \ldots, (k-1)\}$ satisfying the above conditions. The following proposition describes the transformations on computations we use to construct λ_{i+1} and ν_{i+1}.

PROPOSITION 21. *1. Assume*

$$\sigma_1 \xrightarrow{\tau}{}^*_P \sigma_2 \xrightarrow{(t,\mathsf{flush}(x,u))}_P \sigma_3,$$

where

- *τ does not contain an* UNLOCK *transition without a matching* LOCK *transition by a thread other that t;*
- *any transition by thread t in τ is not a* FLUSH *transition or the* WRITE *transition corresponding to $(t, \mathsf{flush}(x, u))$; and*
- *any transition by a thread other than t in τ is not* FLUSH *or* READ *accessing x.*

Then for some σ'_2 we have

$$\sigma_1 \xrightarrow{(t,\mathsf{flush}(x,u))}_P \sigma'_2 \xrightarrow{\tau}{}^*_P \sigma_3.$$

2. Assume

$$\sigma_1 \xrightarrow{(t,\mathsf{write}(x,u))}_P \sigma_2 \xrightarrow{\tau}{}^*_P \sigma_3.$$

where

– *τ does not contain a* LOCK *transition without a matching* UNLOCK *transition by a thread other that t; and*
– *τ does not contain transitions by t other than* FLUSH *transitions that do not correspond to $(t, \mathsf{write}(x, u))$.*

Then for some σ_2' we have

$$\sigma_1 \xrightarrow{\tau}{}^*_P \sigma_2' \xrightarrow{(t,\mathsf{write}(x,u))}_P \sigma_3,$$

Consider the transition in λ_i following ν_i. Since η_i is SC, the buffer of every CPU in the final configuration of ν_i is empty. Hence, the transition cannot be obtained using FLUSH. If it is obtained using LOCAL, READ, LOCK or UNLOCK, then we obtain ν_{i+1} by extending ν_i with the transition; in this case $\lambda_{i+1} = \lambda_i$. Consider the remaining case when the transition is obtained using WRITE, so that its label is $(t, \mathsf{write}(x))$. Then there exists the corresponding flush action later in the trace τ_i. If the FLUSH transition producing it immediately follows the WRITE transition producing $(t, \mathsf{write}(x))$, then we can obtain ν_{i+1} by extending ν_i with these two transitions; again $\lambda_{i+1} = \lambda_i$. Otherwise, we have a computation with the following trace:

$$\tau_i = \eta_i\, (t, \mathsf{write}(x))\, \xi_1\, (t, \mathsf{flush}(x))\, \xi_2,$$

where $(t, \mathsf{flush}(x))$ corresponds to $(t, \mathsf{write}(x))$. Note that ξ_1 cannot contain a flush action by t. Indeed, since η_i is SC, such an action would correspond to a write action in ξ_1. But then this write would be flushed earlier than a preceding write $(t, \mathsf{write}(x))$, contradicting the FIFO ordering of store buffers.

If ξ_1 does not contain any actions of threads other than t reading or flushing x, then by Proposition 21, we can move the transition producing $(t, \mathsf{flush}(x))$ to the position right after the one producing $(t, \mathsf{write}(x))$, obtaining a computation with the following trace:

$$\eta_i\, (t, \mathsf{write}(x))\, (t, \mathsf{flush}(x))\, \xi_1\, \xi_2.$$

We then take this computation as λ_{i+1}, and obtain ν_{i+1} by extending ν_i with the transitions producing $(t, \mathsf{write}(x))\, (t, \mathsf{flush}(x))$.

All the above transformations preserve the final configuration of the computation. Furthermore, since visible locations are protected in P, they also preserve the projection of the trace of the computation to visible actions. Hence, λ_{i+1} and ν_{i+1} thus constructed satisfy the required conditions.

Now, assume that ξ_1 contains actions by a thread other than t reading or flushing x. We show that the computation λ_i can be transformed into another computation with an SC trace containing a data race. Let $(t', \mathsf{access}(x))$ be the last action in ξ_1 by a thread other than t reading or flushing x. Proposition 21 allows us to move the transition producing $(t, \mathsf{flush}(x))$ right after the one producing $(t', \mathsf{access}(x))$, if the latter is not inside an atomic block, or right after the corresponding UNLOCK transition, if it is. We thus obtain a computation with the trace:

$$\eta_i\, (t, \mathsf{write}(x))\, \xi_1\, \mathsf{block}(t', \mathsf{access}(x))\, (t, \mathsf{flush}(x))\, \xi_2,$$

where $t \neq t'$. Let us drop the suffix ξ_2 and the corresponding fragment of the computation:

$$\eta_i\,(t, \mathsf{write}(x))\,\xi_1\,\mathsf{block}(t', \mathsf{access}(x))\,(t, \mathsf{flush}(x)).$$

As before, ξ_1 does not contain flush actions by t. Hence, by removing all transitions by t from the computation fragment generating ξ_1, we obtain a valid computation with the above trace, but where the computation fragment generating ξ_1 does not contain transitions by t. By Proposition 21, we can then move the transition producing $(t, \mathsf{write}(x))$ to the position right before the one producing $(t, \mathsf{flush}(x))$, obtaining a computation with the trace:

$$\eta_i\,\xi_1\,\mathsf{block}(t', \mathsf{access}(x))\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x)).$$

If $(t', \mathsf{access}(x))$ is inside an atomic block, or access is read, then the trace has the form

$$\eta_i\,\xi_1\,\xi, \tag{5}$$

where ξ is a data race.

Consider the case when access above is flush outside an atomic block. Since η_i is SC, the write $(t', \mathsf{write}(x))$ corresponding to $(t', \mathsf{access}(x))$ is in ξ_1:

$$\eta_i\,\xi_1\,(t', \mathsf{write}(x))\,\xi_2\,(t', \mathsf{flush}(x))\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x)), \tag{6}$$

Since store buffers are FIFO, all flushes by t' in ξ_2 correspond to write actions from ξ_1. Hence, by removing all transitions of thread t' except these flushes from ξ_2, we obtain a valid computation with a trace of the above form, but where ξ_2 does not contain transitions by t' except flushes of writes in ξ_1. By Proposition 21, we can now move the transition producing the action $(t', \mathsf{write}(x))$ to the position right before the one producing $(t', \mathsf{flush}(x))$, making the write immediate:

$$\eta_i\,\xi_1\,(t', \mathsf{write}(x))\,(t', \mathsf{flush}(x))\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x)). \tag{7}$$

Thus, in any case, we obtain a computation with a trace of the form (5). If there are pending writes in ξ_1, we can add the corresponding FLUSH transitions to the end of the computation producing ξ_1, thus ensuring that the trace is complete (this may lead to a read in ξ reading a different value, but does not affect the form of the trace).

Let us apply all the transformations described above to the prefix of the computation producing $\eta_i\,\xi_1$, which is of a smaller length than the original computation λ. If this converts it to a computation with an SC trace, then, since the transformations preserve the final configuration of the computation, the resulting computation can be extended with transitions producing the data race ξ. Otherwise, we obtain a computation with the trace of the form (5), but of a smaller length and recurse again. The latter case cannot happen infinitely often, so in the end we will construct a computation with a data race. □

C.6 Proof of Theorem 11

The proof is virtually identical to that of Theorem 10. Namely, consider $\tau \in [\![L]\!]_{\mathsf{TSO}}$ and the corresponding computation λ of $\mathsf{MGC}(L)$. We apply the same transformations

to λ as in the proof of Theorem 10, except that, instead of removing writes without corresponding flushes in the preprocessing step, we append the necessary FLUSH transitions at the end of the computation. This ensures that this step preserves the history of the computation. If the transformations produce an SC trace, then its history is equal to the original one, which implies the required. Otherwise, we find an SC computation of $\mathsf{MGC}(L)$ with a data race, contradicting the DRF of L. □

C.7 Proof of Theorem 13

We first state auxiliary lemmas. We use the counterpart of the TSO/SC semantics defined in Section 3 and Appendix B—an SC/TSO semantics in which client commands access the memory directly, while library ones go via store buffers. The following lemma shows that any TSO trace of $C(L)$ can be converted into either a trace on SC/TSO with the same visible behaviour or a trace exhibiting a data race.

LEMMA 22 (Client robustness). *Assume that visible locations are protected in* $C(L)$. *Then*

$$\forall \tau \in [\![C(L)]\!]_{\mathsf{TSO}}.\, \exists \tau' \in [\![C(L)]\!]_{\mathsf{SC/TSO}}.\, \mathsf{visible}(\tau') = \mathsf{visible}(\tau) \vee (\mathsf{client}(\tau') \textit{ contains a data race}).$$

This lemma is proved below. We use the following variant of Lemma 18 for the SC/TSO semantics. Its proof is an easy variation on the proof of the latter.

LEMMA 23 (Decomposition on SC/TSO).

$$\forall \tau \in [\![C(L)]\!]_{\mathsf{SC/TSO}}.\, \exists \eta \in [\![C]\!]_{\mathsf{SC}}.\, \exists \xi \in [\![L]\!]_{\mathsf{TSO}}.\, \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{client}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{lib}(\xi).$$

Finally, we use the following variant of Lemma 20, which is analogous to a theorem proved in [6]. It states that any pair of client-local and library-local SC computations agreeing on the history can be combined into a valid SC computation of $C(L)$.

LEMMA 24 (Composition on SC).

$$\forall \eta \in [\![C]\!]_{\mathsf{SC}}.\, \forall \xi \in [\![L]\!]_{\mathsf{SC}}.\, \mathsf{history}(\eta) = \mathsf{history}(\xi) \Rightarrow \exists \tau \in [\![C(L)]\!]_{\mathsf{SC}}.\, \mathsf{client}(\tau) = \mathsf{client}(\eta).$$

Proof of Theorem 13. Consider $\tau_1 \in [\![C(L)]\!]_{\mathsf{TSO}}$. By Lemma 22, for some $\tau_2 \in [\![C(L)]\!]_{\mathsf{SC/TSO}}$, either $\mathsf{visible}(\tau_2) = \mathsf{visible}(\tau_1)$ or $\mathsf{client}(\tau_2)$ contains a data race. By Lemma 23, τ_2 generates two traces—a library-local trace $\xi_1 \in [\![L]\!]_{\mathsf{TSO}}$ and a client-local one $\eta \in [\![C]\!]_{\mathsf{SC}}$—such that $\mathsf{client}(\tau_2) = \mathsf{client}(\eta)$ and $\mathsf{history}(\xi_1) = \mathsf{history}(\eta)$. Since $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$, for some trace $\xi_2 \in [\![L^\sharp]\!]_{\mathsf{SC}}$, we have $\mathsf{history}(\xi_1) \sqsubseteq \mathsf{history}(\xi_2)$. By Lemma 19, ξ_2 can be transformed into a trace $\xi_2' \in [\![L^\sharp]\!]_{\mathsf{SC}}$ such that $\mathsf{history}(\xi_2') = \mathsf{history}(\xi_1) = \mathsf{history}(\eta)$. We then use Lemma 24 to compose the library-local computation ξ_2' with the client-local one η into a computation $\tau_3 \in [\![C(L^\sharp)]\!]_{\mathsf{SC}}$ such that $\mathsf{client}(\tau_3) = \mathsf{client}(\eta) = \mathsf{client}(\tau_2)$. Hence, either $\mathsf{visible}(\tau_3) = \mathsf{visible}(\tau_1)$ or

$\mathsf{client}(\tau_3)$ contains a data race. Since $C(L^\sharp)$ is DRF, the latter case is impossible, which implies the required. □

Proof sketch for Lemma 22. Consider a trace $\tau \in [\![C(L)]\!]_{\mathsf{TSO}}$ and the corresponding computation λ of $C(L)$. We construct a computation with the required trace τ' by running the transformations from the proof of Theorem 10 to make *only the client part of the trace* SC. That is, we move FLUSH transitions to follow immediately the corresponding WRITE transitions only for client locations. When every client write is immediately followed by a flush, we get a computation in the SC/TSO semantics. However, intermediate computations arising during the transformations are not valid in either TSO or SC/TSO semantics. Thus, we consider yet another auxiliary semantics, ***client-first TSO (C-TSO)***, in which *we allow entries for client locations in store buffers to be flushed ahead of entries for library ones*, but not vice versa. A computation in the TSO semantics also belongs to C-TSO, which gives us the initial computation λ_1 for the construction of Theorem 10. The transformations from Theorem 10 are then run without changes, except on every step we now have that $\mathsf{client}(\eta_i)$, rather than η_i, is SC: if the transition following the computation prefix producing η_i is a library transition, a call or a return, we just obtain η_{i+1} by extending η_i with this transition and do not perform any transformations. The transformations used in the proof of Theorem 10 can be easily adjusted to the C-TSO semantics. For example, in several places during the proof of Theorem 10, we considered a WRITE transition by t and a corresponding later FLUSH transition, and argued that all FLUSH transitions by t between these two have to be for writes preceding the WRITE transition. This is still true in the C-TSO semantics when the WRITE transition is by the client. This justifies the step in the proof removing all transitions by t between the WRITE and the FLUSH except FLUSH transitions (such as converting (6) into (7)). Note that this transformation can change the history of the computation. By performing it in the C-TSO semantics, we preserve the knowledge that, despite this, the computation can still be obtained by executing the client with the given library. □

Discussion. As we noted in Section 5, a proof of Theorem 13 cannot be obtained by straightforwardly combining Theorems 5 and 10. Its proof relies crucially on the fact that Lemma 22 guarantees that, in both outcomes, the client behaviour can be reproduced while using the TSO implementation of the library. The proof of the lemma uses correlations in the use of store buffers by the client and the library to establish this. Because of our reliance on this property, the proof of Theorem 13 would not go through even if we used the strategy from the proof of Theorem 5 (instead of the end result), i.e., first split τ_1 into a client-local and a library-local trace and then applied Theorem 22 to make the former one SC. In the case when the client has a race, we would not be able to guarantee that the transformation converting (6) into (7) in Theorem 22 preserves the history of the library, and hence, would not be able to reproduce data race in $C(L^\sharp)$.

C.8 Proof of Theorem 15

In the following, η ranges over SC traces. Traces subscripted with $\neg t$ contain only actions by threads other than t, and those with t, only actions by t. Like in the proof of Theorem 10, the scope of all trace variables is limited to the formula they occur in.

Consider a trace $\tau \in [\![C]\!]_{\mathsf{TSO}}$ and the corresponding computation λ. Like in the proof of Theorem 10, we first preprocess the computation by removing transitions following pending writes and ensuring that all writes inside atomic blocks are immediate. We can thus assume that τ is complete.

We now define a sequence of steps that transforms λ into a computation with an SC trace that either has the same projection to visible actions or contains a quadrangular race. In the former case, on every step, we obtain a computation with the trace of the form $\eta\,\tau'$, where η is SC. We then either extend the SC prefix η of the trace without decreasing the number of immediate writes in it, or increase the number of immediate writes in τ'. The transformations preserve the projection of the trace to visible actions. For brevity, in the following we describe how our transformations affect traces only, without referring to computations producing them. We use a number of transformations on computations and traces similar to those in Proposition 21; we omit their formal statements. Whenever we move actions in a trace, we move atomic blocks indivisibly. To ease the exposition, we mark different stages in the transformation.

(A) Consider a trace $\eta\,\varphi\,\tau'$, where η is SC. Note that since η is SC the store buffers are empty at the end of its computation, and so φ cannot be a flush action. If φ is a read action or an action within an atomic block, then $\eta\,\varphi$ is SC and we can extend the SC prefix of the trace and continue with **(A)**. Assume now that $\varphi = (t, \mathsf{write}(x))$, which is not inside an atomic block. Then there exists a corresponding flush action later in the trace, which is outside an atomic block due to the preprocessing phase:

$$\eta\,(t, \mathsf{write}(x))\,\tau_1\,(t, \mathsf{flush}(x))\,\tau_f.$$

Note that τ_1 cannot contain a flush action by t. Indeed, since the projection of η to actions of t is SC, such an action would correspond to a write action in τ_1. But then this write would be flushed earlier than a preceding write $(t, \mathsf{write}(x))$, contradicting the FIFO ordering of store buffers.

If τ_1 does not contain any actions of threads other than t reading or flushing x, then we can move $(t, \mathsf{flush}(x))$ to the position right after $(t, \mathsf{write}(x))$:

$$\eta\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x))\,\tau_1\,\tau_f,$$

thus extending the SC prefix of the trace to $\eta\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x))$. We can then continue with the transformation **(A)**.

Assume now that τ_1 does contain actions of a thread other than t reading or flushing x. Then we can move $(t, \mathsf{flush}(x))$ to the position right after the last such action in τ_1 or the atomic block that contains it:

$$\eta\,(t, \mathsf{write}(x))\,\tau_1\,\mathsf{block}(t_1, \mathsf{access}(x))\,(t, \mathsf{flush}(x))\,\tau_f,$$

where $t_1 \neq t$.

(B) We now try to move all actions by threads other than t in $\tau_1\,\mathsf{block}(t_1, \mathsf{access}(x))$ leftwards, so that they directly follow η. If successful, this would result in a trace of the form

$$\eta\,\tau_{\neg t}\,(t, \mathsf{write}(x))\,\tau_t\,(t, \mathsf{flush}(x))\,\tau_f.$$

We can then move $(t, \mathsf{flush}(x))$ leftwards in this trace to make $(t, \mathsf{write}(x))$ immediate:

$$\eta\, \tau_{\neg t}\, (t, \mathsf{write}(x))\, (t, \mathsf{flush}(x))\, \tau_t\, \tau_f$$

and then proceed with **(A)**.

We move the actions or atomic blocks by threads other than t in τ_1 leftwards one by one, starting with the leftmost. To ensure that these transformations succeed, we need to check that the trace fragments being moved commute with actions by thread t in τ_1. Note that τ_1 cannot contain atomic blocks by t, since otherwise the action $(t, \mathsf{flush}(x))$ corresponding to $(t, \mathsf{write}(x))$ could not follow τ_1. Since τ_1 does not contain flush actions by t, the only non-commuting pair of trace fragments we might have is $(t, \mathsf{read}(y))$ and $\mathsf{block}(t_2, \mathsf{flush}(y))$, where $t \neq t_2$ and the read transition reads from the memory. Since every read from x by t in τ_1 reads from the store buffer, we have $x \neq y$ and $(t_1, \mathsf{access}(x))$ commutes with any action of t in τ_1. Besides, atomic blocks access at most one location, so that $(t_2, \mathsf{flush}(y))$ and $(t_1, \mathsf{access}(x))$ cannot be inside the same atomic block. Thus, if the transformation **(B)** gets stuck, the stuck trace will have the form

$$\eta\, \tau_1\, (t, \mathsf{write}(x))\, \tau_t\, (t, \mathsf{read}(y))\, \mathsf{block}(t_2, \mathsf{flush}(y)) \\ \tau_2\, \mathsf{block}(t_1, \mathsf{access}(x))\, (t, \mathsf{flush}(x))\, \tau_f, \quad (8)$$

where $x \neq y$, $t \neq t_1$, $t \neq t_2$ and $(t, \mathsf{flush}(x))$ corresponds to $(t, \mathsf{write}(x))$. In the current context, we know that τ_1 does not contain actions by t. However, to handle recursive invocations of transformations, in the following, we consider a more general case where this requirement is omitted. Let us drop all the actions from the suffix τ_f other than flushes for writes pending in the rest of the trace. We can thus assume that τ_f consists of only flush actions. We now show that (8) can be transformed into an SC trace exhibiting a quadrangular race.

(C) For every thread t_3, let us partition its write actions in τ_1 into those for which the corresponding flush is in τ_1 and those for which it is later in the trace. Then in a projection of τ_1 to actions of t_3, the former precede the latter. Let us apply the transformations **(A)**–**(B)** to the former kind of write actions in τ_1: we move every such flush action leftwards to make the corresponding write immediate, processing flushes left to right. The only difference with the previous situation is that, while processing a write by a given thread, the prefix η may not be SC: it might have pending writes by other threads. However, in the transformations **(A)**–**(B)** we actually rely only on the fact that the projection on η to actions of the thread whose write we are trying to make immediate is SC. If the transformations fail, then we find a trace of the form (8) of a strictly smaller length. We can then apply the transformation **(C)** again.

If the transformations **(A)**–**(B)** succeed, then we convert the trace into one of the form (8), but such that $(\tau_1)|_{t_3} = \tau'_{t_3}\tau''_{t_3}$, where τ'_{t_3} is SC, and τ''_{t_3} contains only reads and writes pending in τ_1, with at least one pending write (in particular, it does not contain atomic blocks).

(D) Let us now move all actions in suffixes τ''_{t_3} to the end of τ_1. For this, we first process the actions or atomic blocks left to right in their order in τ_1 until we gather them in a

single block, and after this, we move the whole block together.

If this process succeeds, we get a trace of the form (8), but where τ_1 contains only reads and pending writes. Every pair of actions by different threads in τ_1 is commuting. We now ensure that the $(t_2, \mathsf{write}(y))$ action corresponding to $(t_2, \mathsf{flush}(y))$ is at the position right before the latter. If $(t_2, \mathsf{flush}(y))$ is inside an atomic block, then so is $(t_2, \mathsf{write}(y))$. Otherwise we try to move all actions by t_2 in τ_1 following $(t_2, \mathsf{write}(y))$ to the position after $(t_2, \mathsf{flush}(y))$, and then move $(t_2, \mathsf{write}(y))$ to the position before it. If one of these actions (a read) conflicts with a flush in τ_t, then we can drop the suffix of the trace following $(t_2, \mathsf{flush}(y))$, except some of the flushes, to keep the trace complete. This again yields a trace of the form (8), but of a strictly smaller length. We can then procede with **(C)**.

Otherwise, we obtain:

$$\eta\, \tau_1\, (t, \mathsf{write}(x))\, \tau_t\, (t, \mathsf{read}(y))\, \mathsf{block}((t_2, \mathsf{write}(y))\, (t_2, \mathsf{flush}(y)))\\ \tau_2\, \mathsf{block}(t_1, \mathsf{access}(x)))\, (t, \mathsf{flush}(x))\, \tau_f, \quad (9)$$

where τ_1 contains only reads and pending writes. We can then proceed with the transformation **(I)**.

(E) Assume the transformation **(D)** gets stuck at a trace of the form

$$\eta\, \tau_3\, \mathsf{block}((t_3, \mathsf{write}(z))\, (t_3, \mathsf{flush}(z)))\, \tau_f, \quad (10)$$

where τ_3 contains only reads and pending writes, and the last action in it does not commute with $\mathsf{block}((t_3, \mathsf{write}(z))\, (t_3, \mathsf{flush}(z)))$. Any pair of actions by different threads in τ_3 commute. Hence, let us move all actions in this subtrace that commute with $\mathsf{block}((t_3, \mathsf{write}(z))\, (t_3, \mathsf{flush}(z)))$ to τ_f. We know that an atomic block can access at most one memory location. Thus, after this, the last action in the subtrace by any thread different from t_3 is a $\mathsf{read}(z)$. This implies that no thread other than t_3 wrote to z in the subtrace, for otherwise such read actions would read from the corresponding store buffers and thus commute with the block. Also, we know that t_3 did not execute write actions in the subtrace, as the writes in $\mathsf{block}((t_3, \mathsf{write}(z))\, (t_3, \mathsf{flush}(z)))$ are the first ones by this thread to be flushed after the SC prefix η, and store buffers are FIFO. Hence, we can move all read actions by t_3 in τ_3 to η. We can thus assume that τ_3 does not contain actions by t_3.

(F) Every pending write $\mathsf{write}(u)$ in τ_3 has a corresponding flush $\mathsf{flush}(u)$ after

$$\mathsf{block}((t_3, \mathsf{write}(z))\, (t_3, \mathsf{flush}(z))).$$

Let us try to move all such flushes to the position right after this block, processing them left to right. If some flush cannot be moved to this position due to a race, we just move it as far leftwards as possible. The flush actions that can be moved to the above position can also be moved right before the block, since no thread wrote to z in τ_3, and atomic blocks access at most one location. For every thread with actions in τ_3, consider the maximal prefix of its actions there for which all corresponding flushes have been moved to this position. Let us sort actions in τ_3 so that it is split into two subtraces: τ_3',

containing the above prefixes for all threads, followed by τ_3'', containing the rest of the actions. Let us also sort actions in τ_3'' by thread identifier.

(G) Let us try to move the flushes that are currently right before $\mathsf{block}((t_3, \mathsf{write}(z))\,(t_3, \mathsf{flush}(z)))$ to the position in between τ_3' and τ_3'', processing them left to right. If this process gets stuck for some flush action $(t_4, \mathsf{flush}(u))$, this is because we cannot commute it over some action $(t_5, \mathsf{read}(u))$ in τ_3''. Since no thread wrote to z in τ_3, all flush actions to be moved commute with final $\mathsf{read}(z)$ actions in τ_3''. By the definition of τ_3'', there exists a write action to the left of $(t_5, \mathsf{read}(u))$ whose corresponding flush further down the trace participates in a race. Hence, we get a trace of the form:

$$\eta\,\tau_3'\,\tau_4\,(t_5, \mathsf{write}(w))\,\tau_{t_5}\,(t_5, \mathsf{read}(u))\,(t_4, \mathsf{flush}(u))\\ \tau_2\,\mathsf{block}(t_6, \mathsf{access}(w))\,(t_5, \mathsf{flush}(w))\,\tau_f,$$

where $t_5 \neq t_4$, $t_5 \neq t_6$. By dropping all actions from τ_f except flush actions for writes pending in the rest of the trace, we obtain a trace of the form (8). Let us also drop all transitions by thread t_5 between $(t_5, \mathsf{write}(w))$ and $(t_5, \mathsf{flush}(w))$ except FLUSH ones. This erases at least the final reads from z, so the resulting trace is smaller than the original one. We can thus apply the transformations starting from **(C)**.

If the transformation **(G)** succeeds, then τ_3', together with the flushes moved, is a complete trace smaller than the original one. We can thus apply the transformations starting from **(A)** to either make it SC, or to find a quadrangular race. In the former case, we obtain a trace of the form (10), but where for every pending write in τ_3, the corresponding flush has a race in τ_f.

(H) Consider the write in τ_3 with the earliest flush. By commuting actions in τ_3, we can ensure that this write is the first action in this trace. Since t_3 does not execute actions in τ_3, this write is not by t_3. Let us erase the suffix of the trace following the flush and the actions of all the threads with writes in τ_3 following η, except the one that performs the flush. Since the other writes in τ_3 are flushed later than the flush action we are considering, the resulting trace is still valid. It is of the form (9), but with $\tau_1 = \varepsilon$. We can then continue with the transformation **(J)**.

(I) We are left with the case when the transformation **(D)** succeeded, so that we have a trace of the form (9). However, this trace is also of the form (10), so we can again apply the transformations **(E)**–**(H)**, obtaining a trace of the form (9) with $\tau_1 = \varepsilon$.

(J) We thus have a trace

$$\eta\,(t, \mathsf{write}(x))\,\tau_t\,(t, \mathsf{read}(y))\,\mathsf{block}((t_2, \mathsf{write}(y))\,(t_2, \mathsf{flush}(y)))\\ \tau_2\,\mathsf{block}(t_1, \mathsf{access}(x))\,(t, \mathsf{flush}(x))\,\tau_f,$$

where $x \neq y$, $t \neq t_1$, $t \neq t_2$ and $(t, \mathsf{flush}(x))$ corresponds to $(t, \mathsf{write}(x))$. We now drop non-FLUSH transitions from τ_f.

Let us complete $(t, \mathsf{write}(x))$ with an immediate flush and discard $(t, \mathsf{flush}(x))$ later in the trace. This may result in some actions in τ_2 changing. However, this can only be

the case if there is a read or flush of x in τ_2, so in any case we get a trace of the form

$$\eta\,(t, \mathsf{write}(x))\,(t, \mathsf{flush}(x))\,\tau_t\,(t, \mathsf{read}(y)) \\ \mathsf{block}((t_2, \mathsf{write}(y))\,(t_2, \mathsf{flush}(y)))\,\tau_2\,\mathsf{block}(t_1, \mathsf{access}(x))\,\tau_f.$$

We then continue the same process with all writes in τ_t: again, if something changes in τ_2, this is due to an access to the location the write to which being completed. In the end, we obtain a trace of the above form, but where τ_t is SC. We can then drop all non-FLUSH transitions from τ_f.

(K) Since the trace $\tau_2\,\mathsf{block}(t_1, \mathsf{access}(x))$ is complete, we can apply the transformations to it starting from **(A)**. This will either make it SC, in which case we will have reproduced a quadrangular race, or find a quadrangular race inside the trace.

We now prove that the above transformation strategy indeed produces either an equivalent SC trace, or an SC trace with a quadrangular race.

Assume that the transformation of τ encountered a trace of the form (8). Since after this, every jump to an earlier stage in the strategy considers a trace strictly smaller than the original one, the transformation will eventually terminate with a quadrangular race.

Assume now that the transformation of τ never encounters a trace of the form (8), so that we only apply the transformations **(A)–(B)**. In this case, on every step of the transformation strategy where the SC prefix of the trace is not extended, the number of non-immediate writes following the prefix decreases. This means that eventually the trace will become SC. □

C.9 Proof of Theorem 17

We use yet another variant of Lemma 18[7]. Let lib be an operation on traces analogous to client, but selecting actions relevant to the library.

LEMMA 25 (Decomposition on SC).

$$\forall \tau \in [\![C(L)]\!]_{\mathsf{SC}}.\ \exists \eta \in [\![C]\!]_{\mathsf{SC}}.\ \exists \xi \in [\![L]\!]_{\mathsf{SC}}. \\ \mathsf{history}(\eta) = \mathsf{history}(\xi) \wedge \mathsf{client}(\tau) = \mathsf{client}(\eta) \wedge \mathsf{lib}(\tau) = \mathsf{lib}(\eta).$$

Assume L is QRF, $C(L)$ is strongly DRF, and visible locations are protected in it. Consider a computation λ of $C(L)$ on TSO. Let us apply the transformations from the proof of Theorem 15 that try to make the computation SC. If they succeed, we get a computation of $C(L)$ on SC with the same sequence of visible actions, as required. If they fail, we get a computation with the trace of the form (8), where τ_1 does not contain actions by t. We now show that this trace can be transformed into a trace violating the race-freedom assumptions in the theorem.

We consider several cases depending on whether the locations x and y involved in the problematic trace fragment belong to the client or the library.

[7] Proved in: A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.

– $x, y \in \mathsf{LLoc}$. By Lemma 18, there exists a trace from $[\![L]\!]_{\mathsf{TSO}}$ containing the quadrangular race on the locations x and y. Applying the transformations from the proof of Theorem 15 further, we can convert this trace to an SC trace in $[\![L]\!]_{\mathsf{SC}}$ containing a quadrangular race, which contradicts the QRF of L.
– $x \in \mathsf{CLoc}$. In this case we proceed like in the proof of Theorem 10. Since store buffers are FIFO, there are no flushes by t in the trace fragment from $(t, \mathsf{write}(x))$ up to, but not including $(t, \mathsf{flush}(x))$. Hence, by removing all transitions of thread t from this computation fragment and dropping the suffix of the computation corresponding to τ_f, we obtain a valid computation with the trace:

$$\eta\, \tau_1\, (t, \mathsf{write}(x))\, \tau_2\, \mathsf{block}(t_1, \mathsf{access}(x))\, (t, \mathsf{flush}(x)),$$

where τ_2 does not contain flushes by t. Thus, we can move the transition producing the action $(t, \mathsf{write}(x))$ to the position right before the one producing $(t, \mathsf{flush}(x))$:

$$\eta\, \tau_1\, \mathsf{block}(t_1, \mathsf{access}(x))\, (t, \mathsf{write}(x))\, (t, \mathsf{flush}(x)),$$

In the case when access is flush, we can ensure that the write action $(t_1, \mathsf{write}(x))$ corresponding to $(t_1, \mathsf{flush}(x))$ precedes it directly as in the proof of Theorem 10. Thus, we get a data race at the end of the trace. By adding FLUSH transitions for pending writes before the race, we can make sure that the trace is complete. We then apply the transformations from Theorem 15 to the computation prefix producing the part of the trace preceding the race. If this makes the computation SC, then we can still execute the racing commands at its end. By projecting the trace of the resulting computation to client actions using Lemma 25, we get a contradiction with the DRF of $C(L)$. If the transformations fail, we start from the beginning of this proof, but with a smaller trace.
– $y \in \mathsf{CLoc}$. Consider the transition producing the write action $(t_2, \mathsf{write}(y))$ corresponding to $(t_2, \mathsf{flush}(y))$. If this transition is within the same atomic block as the one producing $(t_2, \mathsf{flush}(y))$, then we can assume that it directly precedes the latter. Otherwise, like in the previous case, we can remove all non-flush transitions by thread t_2 from the fragment of computation starting with the transition producing $(t_2, \mathsf{write}(y))$ and ending with the one producing $(t_2, \mathsf{flush}(y))$ and drop the suffix of the trace following $(t_2, \mathsf{flush}(y))$. This allows moving $(t_2, \mathsf{write}(y))$ so that it is immediate. In both cases we obtain a computation with the trace of the form:

$$\eta\, \tau_1\, (t, \mathsf{read}(y))\, \mathsf{block}((t_2, \mathsf{write}(y)\, (t_2, \mathsf{flush}(y))),$$

As before, we can ensure that this trace is complete. We can then apply the transformations from the beginning, trying to make the prefix of the computation SC. In the end, we either reproduce a race in the sense of Definition 16, or end up with a smaller trace to be transformed.

Every recursive call of the above transformation strategy is on a smaller trace. Hence, it eventually terminates, with a QRF, a DRF or a strong DRF violation, contradicting the conditions of the theorem. □