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 [11], provide memory consistency models that are weaker than the classical *sequential consistency (SC)* [10]. 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* [9], 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 [12, 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 [7, Appendix C].

2 TSO Semantics

Due to space constraints, we present the TSO memory model only informally; a formal semantics is given in [7, Appendix A]. The most intuitive way to explain TSO is using an abstract machine [13]. 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 [7, 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 [7, 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 $\forall i, j.\, i < j \wedge H(i) = (\_, \mathsf{ret}\ \_) \wedge H(j) = (\_, \mathsf{call}\ \_) \Rightarrow \pi(i) < \pi(j)$.*

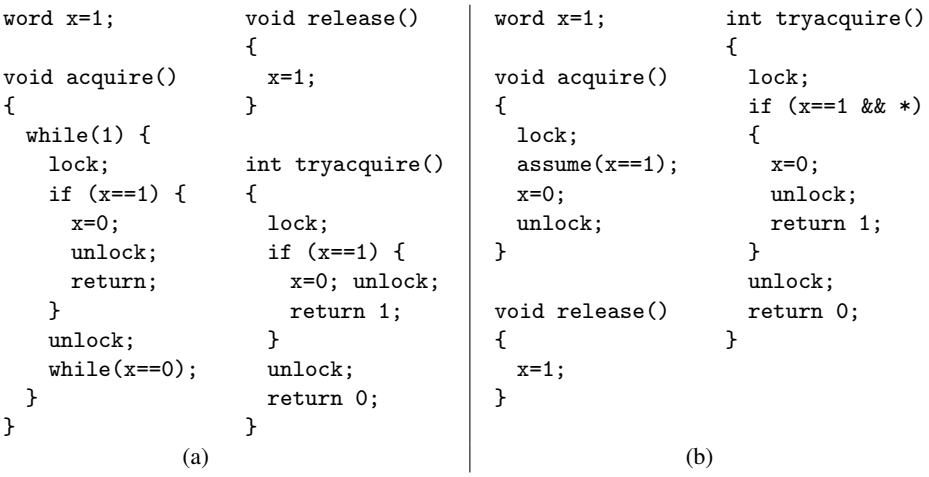
That is, H' linearizes H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations.

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 [7, 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 $\forall H_1 \in \mathsf{history}([\![L_1]\!]_{\mathsf{TSO}}).\ \exists H_2 \in \mathsf{history}([\![L_2]\!]_{\mathsf{SC}}).\ H_1 \sqsubseteq H_2$.*



```
word x=1;

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

```
void release()
{
  x=1;
}

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

(a)

```
word x=1;

void acquire()
{
  lock;
  assume(x==1);
  x=0;
  unlock;
}

void release()
{
  x=1;
}
```

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

(b)

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 [7, 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 [9], 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 [7, 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 [7, 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 [9], 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 [7, 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 [7, 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 [12, 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 [9] 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 [7, 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 [7, 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 [7, 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 [12], 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 $L^\sharp = L$ [7, 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 [8]. 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, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries (extended version). Available from `www.software.imdea.org/~gotsman`, 2012.
8. A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.
9. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.
10. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, 1979.
11. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.
12. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, 2010.
13. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.