
Proceedings Paper:
Doherty, S. and Derrick, J. (2016) Linearizability and Causality. In: Software Engineering and Formal Methods. 14th International Conference, SEFM 2016, 04-08 Jul 2016, Vienna, Austria. Lecture Notes in Computer Science, 9763. Springer, pp. 45-60. ISBN: 978-3-319-41590-1. ISSN: 0302-9743.

https://doi.org/10.1007/978-3-319-41591-8_4

Linearizability and Causality

Simon Doherty and John Derrick

Department of Computing, University of Sheffield, Sheffield, UK

Abstract. Most work on the verification of concurrent objects for shared memory assumes sequential consistency, but most multicore processors support only *weak memory models* that do not provide sequential consistency. Furthermore, most verification efforts focus on the *linearizability* of concurrent objects, but there are existing implementations optimized to run on weak memory models that are not linearizable.
In this paper, we address these problems by introducing *causal linearizability*, a correctness condition for concurrent objects running on weak memory models. Like linearizability itself, causal linearizability enables concurrent objects to be composed, under weak constraints on the client's behaviour. We specify these constraints by introducing a notion of *operation-race freedom*, where programs that satisfy this property are guaranteed to behave as if their shared objects were in fact linearizable. We apply these ideas to objects from the Linux kernel, optimized to run on TSO, the memory model of the x86 processor family.

1 Introduction

The past decade has seen a great deal of interest in the verification of highly optimized, shared-memory concurrent objects. This interest is partly motivated by the increasing importance of multicore systems. Much of this verification work has assumed that these concurrent implementations run on the sequentially consistent memory model. However, contemporary multicore architectures do not implement this strong model. Rather, they implement *weak memory models*, which allow reorderings of memory operations, relative to what would be legal under sequential consistency. Examples of such models include TSO (implemented on the x86) [10], POWER and ARM [2]. These models create significant challenges for verifying that an implementation satisfies a particular correctness condition [5].

Furthermore it is not always clear *what* correctness conditions are appropriate for an implementation running on a weak memory model. Specifically, the standard correctness condition for concurrent objects is *linearizabilty* [8]. However, as described in Section 1.1, there are implementations of concurrent objects optimized to run on weak memory models that are not linearizable. Nevertheless, these implementations are used in important contexts, including the Linux kernel. This is possible because when these objects are used in a stereotypical fashion, their nonlinearizable behaviours are not observable to their clients.

Our goal in this paper is to define a correctness condition appropriate for these nonlinearizable objects. We introduce a correctness condition called *causal*

linearizablilty. Roughly speaking, an object is causally linearizable if all its executions can be transformed into linearizable executions, in a way that is not observable to any thread. As we shall see, causal linearizability is stronger than sequential consistency, and therefore programmers can reason about causally linearizable systems using established intuitions and verification techniques. Furthermore, unlike some competing proposals, causal linearizability places no constraints on the algorithmic techniques used in the implementation of concurrent objects.

Causal linearizability enables concurrent objects to be composed, under certain constraints on the client's behaviour. We specify these constraints by introducing a notion of *operation-race freedom*, where programs that satisfy this property are guaranteed to behave as if their shared objects were linearizable.

In the remainder of the introduction we motivate our work by describing a nonlinearizable data structure designed for a particular weak memory model (in this case, TSO). The structure of the rest of the paper is as follows. Section 2 outlines our contribution, and compares it to related work. Section 3 defines the formal framework and notation, and Section 4 defines independence and causal ordering, which are key concepts in our definition of causal linearizability. Section 5 defines causal linearizability itself. Section 6 then defines operation-race freedom and outlines a proof method for proving causal linearizability. Section 7 applies our ideas to the TSO memory model. Section 8 concludes.

1.1 Motivation - Nonlinearizable Objects on TSO

The Total Store Order (TSO) memory model optimizes write operations by first *buffering* a write to a local *write buffer*, and later *flushing* the write to shared memory. The effect of the write is immediately visible to the core that issues it, but is only visible to other cores after the write has been flushed. The x86 instruction set provides primitives for ensuring that the effect of a write is visible to other threads on other cores. The *barrier* operation flushes all writes of the executing core that have not previously been flushed. In addition, memory operations that both read and modify shared memory may be *locked*. A locked operation appears to execute atomically, and the locking mechanism causes the executing core's write buffer to be emptied both before and after the execution of the locked operation. We formalize this memory model in Section 7.

Locked operations and barriers are typically costly, relative to simple reads and writes. For this reason, optimized datastructures often avoid such synchronization primitives where possible. Here we describe a simple example of such an algorithm: a *spinlock* algorithm for x86 processors that is adapted from an implementation in the Linux kernel. Figure 1 presents pseudocode for the algorithm, which uses a simple boolean flag (`F` below) to record whether the lock is currently held by some thread. A thread acquires the lock using the `try_acquire` procedure, which fails if the lock is currently held (an unconditional `acquire` procedure can be implemented by repeatedly invoking `try_acquire` until successful). The `try_acquire` procedure uses a locked operation to atomically determine whether the lock is held and to set the flag to true. (This operation

```
bool F := false;

void release() {
R1  F := false;
}
```

```
bool try_acquire() {
T1  locked {
T2    held := F;
T3    F := true;
T4  }
T5  return !held;
}
```

Fig. 1. Nonlinearizable spinlock implementation

is called an atomic *test-and-set*). Note that this operation has no effect if the flag is already true. Thus, if the lock is not held, then `try_acquire` successfully acquires the lock and returns `true`. Otherwise, the acquisition attempt fails, and `try_acquire` returns `false`. The optimised `release` operation simply sets the flag to `false`, without using any locked or barrier operations.

The spinlock implementation is *not* linearizable. Intuitively, linearizability requires that each operation on the lock appears to take effect at some point between its invocation and response. To see how this fails, consider the execution in Figure 2. In this example, two threads, t_1 and t_2 attempt to acquire the lock `L`, using the `try_acquire` operation. The first acquisition attempt (of t_1) succeeds, because the lock is free; t_1 then releases the lock (presumably after accessing some shared state protected by the lock), but the write that changes the lock's state is not yet flushed to shared memory. Now t_2's lock acquisition fails, despite being invoked after the end of t_1's `release` operation. This is because, in the example, the releasing write is not flushed until after the completion of t_2's `try_acquire`. Thus, t_2's `try_acquire` appears to take effect between t_1's acquisition and release operations. Linearizability requires that the `try_acquire` appear to take effect *after* t_1's release.

T1	T2
`L.try_acquire()`	
`returns true`	
`L.release()`	
`return`	
	`L.try_acquire()`
	`returns false`

Fig. 2. Nonlinearizable spinlock history.

Despite the fact that it is not linearizable, there are important circumstances in which this spinlock implementation can be correctly used. Indeed, a spinlock essentially identical to this has been used extensively in the Linux kernel. Fundamentally, the goal of this paper is to investigate and formalize conditions under which objects like this spinlock may be used safely on weak memory models.

2 Our Contribution

In this paper, we describe a weakening of linearizability which we call *causal linearizability*. Note that in the execution of Figure 2, no thread can observe that the invocation of thread t_2's `try_acquire` occurred after the response of t_1's release, and therefore no thread can observe the failure of linearizability.

Causal linearizability allows operations to be linearized *out of order*, in cases where no thread can observe this reordering.

However, the lock `L` is the only object in execution in Figure 2. In general, clients of the lock may have other means of communicating, apart from operations on `L`. Therefore, under some circumstances, clients may observe that the execution is not linearizable. Our second contribution is to define a condition called *operation-race freedom* (ORF) on the *clients* of nonlinearizable objects like the TSO spinlock, such that clients satisfying ORF cannot observe a failure of linearizability. ORF is inspired by *data-race freedom* (DRF) approaches, which we discuss below. However, unlike DRF, ORF is defined in terms of high-level invocations and responses, rather than low-level reads and writes.

Finally, we provide a proof method for verifying causal linearizability. We define a correctness condition called *response-synchronized linearizability* (RS-linearizability). In the context of TSO, proving that a data structure satisfies RS-linearizability amounts to proving that the structure is linearizable in all executions where every write executed during an operation are flushed by the time any operation completes. Because of this, we can prove RS-linearizability using essentially standard techniques used for proving linearizability. In Secton 6, we show that any set of RS-linearizable objects is causally linearizable, when the objects' clients satisfy ORF.

Related Work One way to address the issues raised by weak memory models is based on the observation that locks and other synchronization primitives are typically used in certain stereotypical ways. For example, a lock is never released unless it has been first acquired; and the shared state that a lock protects is not normally accessed without holding the lock. As shown in [9], these circumstances mean that the spinlock's nonlinearizable behaviour can never be observed by any participating thread.

The analysis given in [9] belongs to a class of approaches that define conditions under which a program running on a weak memory model will behave as if it were running on a sequentially-consistent memory. These conditions are often phrased in terms *data-races*: a data-race is a pair of operations executed by different threads, one of which is a write, such that the two operations can be adjacent in some execution of the (multithreaded) program. Data-race free (DRF) programs are those whose executions never contain data races, and the executions of DRF programs are always sequentially consistent.

Data-race free algorithms that are linearizable on the sequentially-consistent model will appear to be linearizable on the appropriate weak memory model. Thus the problem of verifying a DRF algorithm on weak-memory is reduced to that of verifying it under the standard assumption of sequential consistency. However, DRF-based approaches have the drawback that algorithms that are not DRF cannot be verified. Our approach does not suffer from this limitation: implementations are free to use any algorithmic techniques, regardless of DRF. Furthermore, the ORF property only constrains the ordering of high-level invocations and responses, rather than low level reads and writes.

[3, 7] define correctness conditions for TSO by weakening linearizability. [3] introduces abstract specifications that manipulate TSO-style write-buffers such that the abstract effect of an operation can be delayed until after the operation's response. [7] proposes adding nondeterminism to concurrent objects' specifications to account for possible delay in the effect of an operation becoming visible. Neither work systematically addresses how to reason about the behaviour of clients that use these weakened abstract objects. In our work, the abstract specifications underlying linearizability are unchanged, and programs satsifying the ORF constraint are guaranteed to behave as if their shared objects were linearizable.

RS-linearizability is a generalisation of *TSO-linearizability*, described in [5]. That work shows that TSO-linearizability can be verified using more-or-less standard techniques for proving linearizability. However, [5] does not address how to reason about the behaviour of clients that use TSO-linearizable objects, as we do with the ORF constraint.

3 Modelling Threads, Histories and Objects

As is standard, we assume a set of invocations I and responses R, which are used to represent operations on a set X of *objects*. The invocations and responses of an object define its interactons with the external environment so we define $Ext = I \cup R$ to be the set of *external actions*. We denote by $obj(a)$ the object associated with $a \in Ext$. We also assume a set of *memory actions* Mem, which typically includes reads, writes and other standard actions on shared-memory. Operational definitions of weak memory models typically involve *hidden actions* that are used to model the memory system's propagation of information between threads, so we assume a set $Hidden \subseteq Mem$ (for example, in TSO the hidden actions are the flushes). Let $Act = Ext \cup Mem$ be the set of *actions*.

In our model, each action is executed either on behalf of a thread (e.g., invocations, or read operations), or on behalf of some memory system (these are the hidden actions). To represent this, we assume a set of threads T, a function $thr : Act \to T \cup \{\bot\} = T_\bot$, such that $thr(a) = \bot$ iff $a \in Hidden$.

Executions are modelled as *histories*, which are sequences of actions. We denote by gh the concatenation of two histories g and h. When h is a history and A a set of actions, we denote by $h \downharpoonright A$ the sequence of actions $a \in A$ occurring in h. For a history h, the *thread history of* $t \in T$, denoted $h \downharpoonright t$, is $h \downharpoonright \{a : thr(a) = t\}$. Two histories h and h' are *thread equivalent* if $h \downharpoonright t = h' \downharpoonright t$, for all threads $t \in T$. (Note that two histories may be thread equivalent while having different hidden actions.)

For example, the behaviour shown in Figure 2 is represented by the history:

$$L.try\_acq_{t_1}, locked_{t_1}(TAS, F, false), resp_{t_1}(L, true), L.release_{t_1}, write_{t_1}(F, false),$$
$$resp_{t_1}(L), L.try\_acq_{t_2}, locked_{t_2}(TAS, F, true), resp_{t_2}(L, false), flush(F, false) \quad (1)$$

Let $a = L.try\_acq_{t_1}$. Then a is an invocation of the `try_acquire` operation, $thr(a) = t_1$ and $obj(a) = L$. The action $resp_{t_1}(L, true)$ is a response from object

L, of the thread t_1, returning the value *true*. $\mathit{locked}_{t_1}(\mathit{TAS}, F, \mathit{false})$ is a locked invocation of the test-and-set operation on the location F, again by thread t_1 that returns the value *false*. $\mathit{flush}(F, \mathit{false})$ is a flush action of the memory subsystem, that sets the value of F to false in the shared store. This history is thread equivalent to the following:

$$\begin{aligned} &L.\mathit{try\_acq}_{t_1}, \mathit{locked}_{t_1}(\mathit{TAS}, F, \mathit{false}), \mathit{resp}_{t_1}(L, \mathit{true}), \\ &\qquad L.\mathit{try\_acq}_{t_2}, \mathit{locked}_{t_2}(\mathit{TAS}, F, \mathit{true}), \mathit{resp}_{t_2}(L, \mathit{false}), \\ &\qquad\qquad L.\mathit{release}_{t_1}, \mathit{write}_{t_1}(F, \mathit{false}), \mathit{flush}(F, \mathit{false}), \mathit{resp}_{t_1}(L) \end{aligned} \quad (2)$$

A history is well-formed if for all $t \in T$, $h \lfloor t \lfloor \mathit{Ext}$ is an alternating sequence of invocations and responses, beginning with an invocation. Note that well-formedness only constrains invocations and responses. Memory operations may be freely interleaved with the external actions. From now on, we assume that all histories are well-formed. A history is *complete* if every thread history is empty or ends in a response.

An *object system* is a prefix-closed set of well-formed histories. A *sequential object system* is an object system where every invocation is followed immediately by a response, in every history. If O is an object system then $\mathit{acts}(O)$ is the set of actions appearing in any history of O.

We wish to reason about orders on the actions appearing in histories. In general, each action may appear several times in a history. Strictly speaking, to define appropriate orders on the actions, we would need to tag actions with some identifying information, to obtain an *event* which is guaranteed to be unique in the history. However, for the sake of simplicity, we assume that each action only appears at most once in each history. For example, each thread may only execute at most one write for each location-value pair. This restriction can be lifted straightforwardly, at the cost of some notational complexity. [1]

Given a history h, the *real-time order* of h, denoted $\rightarrow_h$ is the strict total order on actions such that $a \rightarrow_h b$ if a occurs before b in h. The *program order*, denoted $\xrightarrow{p}_h$, is the strict partial order on the actions of h such that $a \xrightarrow{p}_h b$ if $\mathit{thr}(a) = \mathit{thr}(b)$ and $a \rightarrow_h b$. For example, in History 1 above, $L.\mathit{release}_{t_1} \xrightarrow{p}_h \mathit{write}_{t_1}(F, \mathit{false})$ and $\mathit{write}_{t_1}(F, \mathit{false}) \rightarrow_h \mathit{flush}(F, \mathit{false})$.

4 Independence and Causal Ordering

In this section, we develop a notion of causal ordering. Roughly speaking, an action a is causally prior to an action b in a history h if $a \rightarrow_h b$ and some thread can observe that a and b occurred in that order. Therefore, we can safely reorder events that are not causally ordered. Causal order itself is expressed in terms of an independence relation between actions, which we now define. The

[1] The full version of the paper, which can be found at `arxiv.org/abs/1604.06734`, presents a model of histories in which events are unique.

notion of independence, and the idea of using independence to construct a causal order has a long history. See [6] for a discussion in a related context.

Given an object system S, two actions a and b are *S-independent* if $thr(a) \neq thr(b)$ and for all histories g and h,

$$g\langle a, b\rangle h \in S \Leftrightarrow g\langle b, a\rangle h \in S \tag{3}$$

(Here, $\langle a, b\rangle$ denotes the sequence of length two containing a and then b.) According to this definition, TSO flushes are independent iff they are to distinct locations. Again in TSO, read and write actions in different threads are always independent, but two actions of the same thread never are. (Inter-thread communication only occurs during flush or locked actions.)

We define the causal order over a history in terms of this independence relation. We say that h is *S-causally equivalent* to h' if h' is obtained from h by zero or more transpositions of adjacent, S-independent actions. Note that causal equivalence is an equivalence relation. Actions a and b are S-causally ordered in h, denoted $a \hookrightarrow_h^S b$ if for all causally equivalent histories h', $a \rightarrow_{h'} b$. This is a transitive and acyclic relation, and therefore $\hookrightarrow_h^S$ is a strict partial order.

For example, because the release operation does not contain any locked actions, Histories 1 and 2 on page 6 are causally equivalent. On the other hand, the actions $\mathit{locked}_{t_2}(TAS, F, true)$ and $\mathit{flush}(F, false)$ are not independent, and therefore $\mathit{locked}_{t_2}(TAS, F, true) \hookrightarrow_h \mathit{flush}(F, false)$.

Note that independence, causal equivalence, and causal order are all defined relative to a specific object system. However, we often elide the object system parameter when it is obvious from context.

One key idea of this work is that a history is "correct" if it can be transformed into a linearizable history in a way that is not observable to any thread. The following lemma is our main tool for effecting this transformation. It says that a history can be reordered to be consistent with any partial order that contains the history's causal ordering. The thrust of our compositionality condition, presented in Section 6, is to provide sufficent conditions for the existence of a strict partial order satisfying the hypotheses of this lemma. [2]

Lemma 1. *Let S be an object system, let $h \in S$ be a history, and let $<$ be a strict partial order on the events of h such that $\hookrightarrow_h^S \subseteq <$. Then there exists an h' causally equivalent to h such that for all events a, b in h (equivalently in h') $a < b$ implies $a \rightarrow_{h'} b$.* □

We are now in a position to formally define causal linearizability. Essentially, an object system is causally linearizable if all its histories have causally equivalent linearizable histories. The key idea behind linearizablity is that each operation should appear to take effect atomically, at some point between the operation's invocation an response. See [8] or [4] for a formal definition.

[2] For reasons of space, this paper does not contain proofs of Lemma 1 or the other results presented in this paper. The full version of the paper contains the proofs, and can be found at `arxiv.org/abs/1604.06734`.

Definition 1 (Causal Linearizability). *An object system S is* causally linearizable *to a sequential object system T if for all $h \in S$, h is S-causally equivalent to some history h' such that $h' \downarrow acts(T) \cap Ext$ is linearizable to T.*

Note that causal linearizability is defined in terms of histories that contain both external and internal actions. Typically linearizability and related correctness conditions are defined purely in terms of external actions. Here, we preserve the internal actions of the object, because those internal actions carry the causal order.

5 Observational Refinement and Causal Linearizability

In this section, we introduce a notion of *client* and a notion of composition of a client with an object system (Definition 5). We then define a notion of *observational refinement* for object systems. One object system S observationally refines another object system T for a client C if the external behaviour of C composed with S is included in the external behaviour of C composed with T. These notions have a twofold purpose. First, they provide a framework in which to show that causal linearizability is a reasonable correctness condition: the composition of a client with a causally linearizable object system has only the behaviours of the client composed with a corresponding linearizable object system (Theorem 1). Second, these notions allow us to specify a constraint on the behaviour of a client, such that the client can safely use a composition of nonlinearizable objects.

A *client* is a prefix-closed set of histories, where each history contains only one thread, and all actions are thread actions (so that the client contains no hidden actions). Each client history represents a possible interaction of a client thread with an object system. While each client history contains only one thread, the client itself may contain histories of several threads. For example, consider the histories that might be generated by a thread t_1 repeatedly executing spinlock's `try_acquire` operation (Figure 1) until the lock is successfully acquired. The set of histories generated in this way for every thread is a client. One such history is $L.try\_acq_{t_1}, locked_{t_1}(TAS, F, false), resp_{t_1}(L, true)$, where t_1 successfully acquires the lock on the first attempt. A history where the thread acquires the lock after two attempts is

$$L.try\_acq_{t_1}, locked_{t_1}(TAS, F, true), resp_{t_1}(L, false), \\ L.try\_acq_{t_1}, locked_{t_1}(TAS, F, false), resp_{t_1}(L, true) \quad (4)$$

Thus, the client histories contain the memory operations determined by the implementations of the shared objects.

The composition of an object system O and client program C, denoted $C[O]$ is the object system defined as follows:

$$C[O] = \{h : h \downarrow acts(O) \in O \wedge \forall\, t \in T.\ h \downarrow t \in C\} \quad (5)$$

So for all $h \in C[O]$, h is an interleaving of actions of the threads in C, and every thread history of h is allowed by both the object system and the client program.

We need a notion of observational refinement relative to a given client.

Definition 2 (Observational Refinement). *An object system S* observationaly refines *an object system T for a client C if for every $h \in C[S]$, there exists some $h' \in C[T]$ where $h \downarrow Ext$ and $h' \downarrow Ext$ are thread equivalent.*

The following theorem shows that causal linearizability is sound with respect to observational refinement. Because of this, a causally linearizable object can be used instead of a linearizable object, while preserving correctness of the client's behaviour.

Theorem 1 (Causal Linearizability Implies Observational Refinement). *Let T be a sequential object system, and let T' be its set of linearizable histories. Let S be an object system such that $acts(T) \cap Ext = acts(S) \cap Ext$. If $C[S]$ is causally linearizable to T, then S observationally refines T' for C.*

6 Flush-based Memory and Operation-race Freedom

Causal linearizability is a general correctness condition, potentially applicable in a range of contexts. Our goal is to apply it to objects running on weak memory models. To this end, we formally define a notion of *flush-based memory*. Flush-based memory is a generalisation of TSO and some other memory models, including *partial store order* [1]. This section develops a proof technique for causal linearizability of an object system running on flush-based memory, and hence for observational refinement.

Our proof technique can be encapsulated in the following formula: *Operation-race freedom + Response-synchronized linearizabilty $\Rightarrow$ Causal linearizability.* Response-synchronized linearizability, a weakening of linearizability, is a correctness property specialised for flush-based memory, and is adapted from *TSO linearizability* studied in [5]. That work presents techniques for verifying TSO linearizability and proofs that spinlock and seqlock are TSO linearizable. Theorem 2 below shows that a multi-object system composed of response-synchronized linearizable objects is causally linearizable, under a constraint on the multi-object system's clients. This constraint is called *operation-race freedom*, given in Definition 6.

A *flush-based memory* is an object system whose histories do not contain invocations or responses (so its only actions are memory actions), together with a *thread-action* function $thr\_act_h : Hidden \to Act$, for each history h in the memory model. Hidden actions model the propagation of writes and other operations that modify shared memory. We use the thr_act function to record the operation that each hidden action propagates. Therefore, for each $f \in Hidden$, we require that $thr\_act_h(f) \notin Hidden$. ($f$ is short for *flush*.) For example, in TSO, the hidden actions are the flushes, and thr_act_h associates with each flush the write that created the buffer entry which is being flushed.

Flush based memories must satisfy a technical constraint. We require that the effect of a flush be invisible to the thread on whose behalf the flush is being performed. This captures the idea that flushes are responsible for propagating the effect of operations from one thread to another, rather than affecting the behaviour of the invoking thread.

Definition 3 (Local Flush Invisible). *A memory model M is* local flush invisible *if for all histories $h \in M$, actions a, b, f in h such that $a = thr\_act(f)$ and $a \xrightarrow{p}_h b \rightarrow_h f$, b and f are M-independent.*

For the rest of this section, fix a memory model M with thread action function thr_act. Furthermore, fix an object system S, such that for all $h \in S$, $h \lfloor Mem \in M$. Thus, S is an object system that may contain both external and internal actions.

Definition 4 (Response Synchronization). *Given a history h, the* response-synchronization relation *of h is*

$$\xrightarrow{RS}_h = \hookrightarrow_h^S \cup \{(f, resp_h(thr\_act_h(f))) : f \in Hidden\} \tag{6}$$

A *response-synchronized history* is one where each flush appears before its associated response. That is, $h \in S$ is response-synchronized if $\xrightarrow{RS}_h \subseteq \rightarrow_h$. An object system is *response-synchronized linearizable* (or RS-linearizable) if all its response synchronized histories are linearizable.

It is relatively easy to verify RS-linearizability. The idea is to construct a model of the system such that response actions are not enabled until the operation's writes have been flushed, and then to prove that the implementation is linearizable on this stronger model. See [5] for a careful development of the technique.

Operation-race freedom requires that clients provide sufficent synchronization to prevent any thread from observing that a flush has taken place after its corresponding response action. Definition 5 formalizes which actions count as synchronizing actions, for the purposes of operation-race freedom. Operation-race freedom has one key property not shared by standard notions of data-race freedom: invocations and responses can count as synchronizing actions. This has two advantages. First, we can reason about the absence of races based on the presence of synchronizing invocations and responses, rather than being based on low-level memory operations that have synchronization properties. Second, implementations of concurrent objects are free to employ racey techniques within each operation.

Definition 5 (Synchronization Point). *An action b is a* synchronization point *in $h \in S$, if for all actions a such that $a \xrightarrow{p}_h b$ or $a = b$, all actions c such that $thr(c) \neq thr(a)$ and $b \hookrightarrow_h^S c$, and all hidden actions f such that $thr\_act(f) = a$, not $c \hookrightarrow_h^S f$.*

For example, in TSO, barrier operations are synchronization points. This is because such operations ensure that the issuing thread's write buffer is empty before the barrier is executed. Therefore, any write before the barrier in program order is flushed before the barrier executes, and so the write's flush cannot be after the barrier in causal order. For the same reason, locked operations are also synchronization points in TSO.

Under this definition, invocations and responses may also be synchronization points. An invocation is a synchronization point if its first memory action is a synchronization point, and a response is a synchronization point if its last memory action is a synchronization point. This is because any external action is independent of any hidden action.

Definition 6 (Operation Race). *An* operation race *(or* o-race*) in a history h is a triple r_0, i, r_1, where r_0, r_1 are responses, i is an invocation such that $r_0 \xrightarrow{p}_h i$, $i \hookrightarrow^S_h r_1$, $thr(r_0) \neq thr(r_1)$, $obj(r_0) = obj(r_1)$, there is some hidden action f such that $r_0 = resp_h(thr\_act_h(f))$, and there is no synchronization point between r_0 and i (inclusive) in program order.*

We say that an object system is *o-race free* (ORF) if no history has an o-race.

Below we provide an example of an execution containing an o-race. This example and the next use a datastructure called a *seqlock*, another concurrent object optimised for use on TSO, and adapted from an implementation in the Linux kernel [9]. Seqlock is an object providing *read* and *write* operations with the usual semantics, except that several values can be read or written in one operation. Seqlock has the restriction that there may only be one active write operation at a time, but there may be any number of concurrent read operations and reads may execute concurrently with a write. Seqlock does not use any locking mechanism internally, instead relying on a counter to ensure that read operations observe a consistent set of values. Seqlock does not use any locked or barrier operations, and the read operation never writes to any location in memory. Other details of the algorithm do not matter for our purposes. See [3] for a complete description.

Consider the behaviour presented in Figure 3, adapted from [9]. Here, three threads interact using an instance `L` of the spinlock object, and an instance `S` of seqlock. In this example execution, the flush correponding to the write of t_2's release operation is delayed until the end of the execution, but the flushes associated with the writes of t_1's seqlock write operation occur immediately (note that because the seqlock does not use any barrier or locked operations, this flush could have occurred at any point after the write to memory). This history is not sequentially consistent. If it were sequentially consistent, thread t_2's release would need to take effect before thread t_1's write, which in turn would take effect before thread t_3's read. However, this is inconsistent with the fact that t_3's try-acquire apears to take effect before thread t_2's release. Because it is not sequentially consistent, this execution would be impossible if the spinlock and seqlock were both linearizable objects. Therefore, the composition of spinlock and seqlock do not observationally refine a composition of linearizable objects,

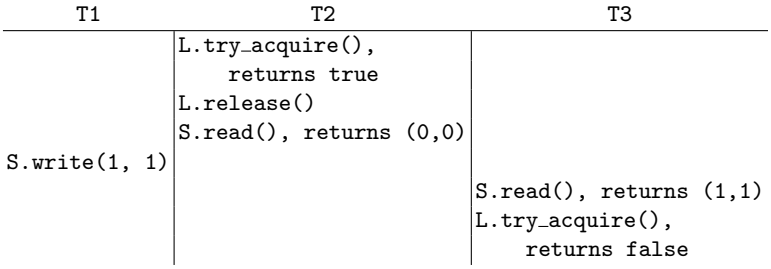

Fig. 3. A racey execution of a spinlock `L` and a seqlock `S`. Operation-race freedom prohibits the race between t_2's release and read, and t_3's try-acquire.

for any client capable of producing this behaviour. There is a race between the response of t_2's release operation, the invocation of t_2's subsequent read, and the release of t_3's try-acquire.

Theorem 2 below shows that an ORF multi-object system composed of response-synchronized linearizable objects is causally linearizable, under one technical assumption. We require that the objects themselves must not *interfere*. That is, each action of each object must be independent of all potentially adjacent actions of other objects. This constraint is implicit in the standard composition result for linearizability, and is satisfied by any multi-object system where each object uses regions of shared-memory disjoint from all the other objects. If a multi-object system does not satsify this property, then one object can affect the behaviour of another object by modifying its representation. Therefore, a composition of individually linearizable objects may not be linearizable itself.

Definition 7 (Noninterfering Object System). *An object system S is* noninterfering *if for all histories $h \in S$, and actions a, b adjacent in h, if $thr(a) \neq thr(b)$ and $obj(a) \neq obj(b)$ then a and b are independent.*

The following lemma shows that the response-synchronization relation is acyclic for an operation-race free object system. This allows us to prove Theorem 2 by applying Lemma 1.

Lemma 2 (Acyclicity of the Response-synchronization Relation). *If M is a memory model and $C[M]$ is a noninterfering, ORF object system, then for all $h \in C[M]$, the response-synchronisation relation is acyclic.*

We can now state our compositionality result. This result says that any client composed with a set of RS-linearizable objects observationally refines the client when composed with linearizable objects, so long as the client is ORF when composed with the RS-linearizable objects. In this case, RS-linearizable objects can be used instead of linearizable objects, while preserving correctness of the client.

Theorem 2 (Composition). *Let X be a set of objects and for each $x \in X$, let T_x be a sequential object system. If M is a flush-based memory and $C[M]$ is*

an ORF noninterfering multi-object system such that for each $x \in X$, $C[M] \downharpoonleft x$ is response-synchronized linearizable to T_x, then $C[M]$ is causally linearizable to $T = h : \forall\, x \in X.\ h \downharpoonleft acts(T_x) \in T_x$, and thus $C[M]$ observationally refines $C[T]$.

7 Operation-race Freedom on TSO

We apply our technique to the well-known *total store order* (TSO) memory model, a version of which is implemented by the ubiqitous x86 processor family. Indeed, we closely follow the formalization of TSO for x86 given in [10]. We then argue that TSO has the properties required of a flush-based memory, including the local flush invisibility property. Finally, we demonstrate how to determine whether a client is operation-race free.

We model TSO as a labelled transition system (LTS) $T = \langle S_T, A_T, I_T, R_T \rangle$. Each state $s \in S_T$ has the form $\langle M, B \rangle$ where

- M is the contents of shared memory, $M : Loc \to \mathbb{Z}$, where Loc is the set of *locations*.
- B records for each thread the contents of its buffer, which is a sequence of location/value pairs. Thus, $B : T \to (Loc \times \mathbb{Z})^*$.

The initial state predicate I_T says only that every buffer is empty (formally, $\forall\, t \in T.\ B(s) = \langle\rangle$). The transition relation R_T is given in Figure 4. The labels (or *actions*) in the set A_T are as follows. For each thread $t \in T$, location $x \in Loc$ and value $v, r \in \mathbb{Z}$, there is a write action $write_t(x, v)$, a read action $read_t(x, r)$, a flush action $flush(t, x, v)$ and a barrier action $barrier_t$. Further, there is a locked action $lock_t(f, x, v, r)$, for each $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ taken from an appropriate list of *read-modify-write* (RMW) operations. Locked actions model the atomic application of an RMW operation to shared memory. For example, $lock_t(+, x, 1, r)$ models the atomic increment of the value at x, and r is the value in location x immediately before the increment. The x86 instruction set supports a range of other RMW operations, such as *add* and *test-and-set*.

The set of traces of this TSO LTS is prefix-closed and thus forms an object system, which we denote by *TSO*. The system actions of *TSO* are just the flush actions, so the *TSO thr* function returns $\perp$ for flush actions, and the thread index of all other actions. The thr_act_h function associates with each flush f the write that is being flushed. *TSO* has the flush invisibility property, of Definition 3, because a flush is independent of any action of the issuing thread, except for the write that is being flushed (as proved in the full version of the paper).

We now explain by example how to check that a client is ORF. Our example is the double-checked locking implementation presented in Figure 5. Double-checked locking is a pattern for lazily initializing a shared object at most once in any execution. The `ensure_init` procedure implements this pattern. Here, the shared object is represented using a seqlock `X`. The `ensure_init` procedure first reads the values in `X`, and completes immediately if `X` has already been initialised. Otherwise, `ensure_init` acquires a spinlock `L` and then checks again whether `X`

$$\frac{(last\_write(B(t), x) = \bot \wedge M(x) = r) \vee last\_write(B(t), x) = r}{(M, B) \xrightarrow{read_t(x,r)} (M, B)}\ Read$$

$$\frac{b' = B(t)\langle (l, v)\rangle}{(M, B) \xrightarrow{write_t(x,v)} (M, B \oplus \{t \mapsto b'\})}\ Write$$

$$\frac{B(t) = \langle (l, v)\rangle b'}{(M, B) \xrightarrow{flush(t,x,v)} (M \oplus \{l \mapsto v\}, B \oplus \{t \mapsto b'\})}\ Flush$$

$$\frac{B(t) = \langle\rangle}{(M, B) \xrightarrow{barrier_t} (M, B)}\ Barrier$$

$$\frac{B(t) = \langle\rangle \quad r = M(x)}{(M, B) \xrightarrow{lock_t(f,x,v,r)} (M \oplus \{x \mapsto f(c, v)\}, B)}\ Locked\text{-}RMW$$

Fig. 4. Transition relation of the TSO memory model. If b is a write buffer, $latest\_write(b, x)$ returns the value of the last write to x in b, if it exists, or $\bot$ otherwise.

```
val ensure_init() {
1.  (v0, v1) = X.read();
2.  if (v0 == null) {
3.    L.acquire();
4.    (v0, v1) = X.read();
5.    if (v0 == null) {
6.      (v0, v1) = initial_value;
7.      X.write(v0, v1);
8.      Barrier();
9.    }
10.   L.release();
11. }
12. return (v0, v1);
}
```

Fig. 5. Pseudocode for a client executing a double checked locking protocol.

has already been initialised (by some concurrent thread), again completing if the initialisation has alread occurred. Otherwise, `ensure_init` initialises the object, executes a barrier, releases the lock and returns.

To show that this code is ORF, we must employ knowledge about which invocations and responses of our objects are synchronization points, and which operations do not execute write actions. As we described in the discussion after Definition 5, in TSO all barriers and locked operations are synchronization points. Furthermore, because the try-acquire's only memory operation is a locked operation, both the invocation and response of try-acquire are synchronization points. Finally, the read operation of seqlock can never execute a write action.

To show that `ensure_init` has no o-races, we must consider the relationship between each operation, and the next operation in program order. For each case, we must show that no o-race is possible.

- The read on Line 1 never executes a write, so its response cannot form an o-race with the subsequent invocation.
- The response of the acquire on Line 3 is a synchronization point, so it cannot form an o-race with the subsequent read.
- As with the read on Line 1, the read on Line 4 never executes a write operation, and so its response cannot form an o-race.
- The write on Line 7 is followed by the barrier on Line 8, so this cannot form an o-race.

Note that during this argument, we only need to consider whether or not the invocation or response of each operation is a synchronization point, or whether the operation never executes write actions. We do not require any further information about the operation's implementation. Again, this means that operations may themselves be racey.

8 Concluding Remarks

Although the details of the paper are fairly technical the essence of the contribution is simple: how can we use non-linearizable algorithms safely. The context that we work in here is that of weak memory models, where TSO provides an important example. This work should also be applicable to other flush-based memory models. Such an extension is work for the future.

To enable our multi-object systems to be composed safely we introduced a notion of operation-race freedom. However, what about non-operation-race free programs? Our formulation provides no composability guaranteess for a family of objects where even *one* of those objects is not response-synchronized. As indicated in Section 6, this is a less severe restriction than other proposals based on some notion of data race freedom (because of its modularity). However, it seems reasonable to expect that some compositionality result would hold for the subset of response-synchronized objects. Again this is left as future work.

References

1. Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, Dec 1996.
2. Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In L. Petersen and M.M.T. Chakravarty, editors, *DAMP '09*, pages 13–24. ACM, 2008.
3. Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 87–107. Springer Berlin Heidelberg, 2012.
4. John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, January 2011.
5. John Derrick, Graeme Smith, and Brijesh Dongol. Verifying Linearizability on TSO Architectures. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, volume 8739 of *Lecture Notes in Computer Science*, pages 341–356. Springer International Publishing, 2014.
6. Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379 – 4398, 2010.
7. Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In *Distributed Computing*, pages 31–45. Springer, 2012.
8. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
9. Scott Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In Theo D'Hondt, editor, *ECOOP 2010*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer Berlin Heidelberg, 2010.
10. Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg, 2009.