What Cannot Be Implemented on Weak Memory?

Armando Castañeda
Instituto de Matemáticas, Universidad Nacional Autónoma de México

Gregory Chockler
Department of Computer Science, University of Surrey

Brijesh Dongol
Department of Computer Science, University of Surrey

Ori Lahav
School of Computer Science, Tel Aviv University

Abstract

We present a general methodology for establishing the impossibility of implementing certain concurrent objects on different (weak) memory models. The key idea behind our approach lies in characterizing memory models by their *mergeability properties*, identifying restrictions under which independent memory traces can be merged into a single valid memory trace. In turn, we show that the mergeability properties of the underlying memory model entail similar mergeability requirements on the specifications of objects that can be implemented on that memory model. We demonstrate the applicability of our approach to establish the impossibility of implementing standard distributed objects with different restrictions on memory traces on three memory models: strictly consistent memory, total store order, and release-acquire. These impossibility results allow us to identify tight and almost tight bounds for some objects, as well as new separation results between weak memory models, and between well-studied objects based on their implementability on weak memory models.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrent algorithms

Keywords and phrases Impossibility, Weak Memory Models, Total-Store Order, Release-Acquire

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.6

Related Version *Full Version*: `https://arxiv.org/abs/2405.16611`

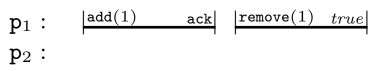
Funding *Armando Castañeda*: Supported by DGAPA PAPIIT project IN108723 and Royal Society grant IES\R1\221226.
Gregory Chockler: Supported by Royal Society grant: IES\R1\221226; CHIST-ERA project REDONDA EP/Y036425/1; and EPSRC grants: EP/X037142/1 and EP/X015149/1.
Brijesh Dongol: Supported by VeTSS; Royal Society grant: IES\R1\221226; CHIST-ERA project REDONDA EP/Y036425/1; and EPSRC grants: EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2.
Ori Lahav: Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant number 814/22).

1 Introduction

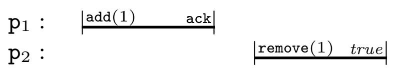
Weak memory models have become standard in modern hardware architectures and programming languages. Unlike traditional strictly consistent memory (SCM), which provides *atomic* read/write instructions, memories achieve efficiency by multiple optimizations, which, in particular, delay propagation of writes instead of making them immediately visible to subsequent reads in other threads. Two well-studied models, which we consider in this paper, are *total store order* model (TSO), as implemented in SPARC [36, 23] and x86 multiproces-

Example 1 Linearizable Obstruction-Free Set

Consider a set object that provides the high-level operations `add`(v) and `remove`(v), where `remove` returns *true* iff the element v is in the abstract set and in this case removes v from the set. Consider the following histories assuming two processes:



history h_1



history h_2

Let σ_0 be the trace of a set implementation I generated by $\mathtt{p}_1$ executing `add`(1) until completion from the initial state, and for $i \in \{1, 2\}$, let σ_i be the trace generated by $\mathtt{p}_i$ after σ_0 to induce history h_i. Such traces must exist assuming I is obstruction-free. If σ_1 and σ_2 can be merged into a trace σ such that $\sigma_0 \cdot \sigma$ is a valid trace of a memory model M, then we reach a contradiction because $\mathtt{p}_1$ (resp., $\mathtt{p}_2$) cannot distinguish between σ_1 (resp., σ_2) and σ, and thus both `remove` operations of $\mathtt{p}_1$ and $\mathtt{p}_2$ in σ return *true*, contradicting linearizability of I. In other words, since the two `remove` invocations cannot be merged into a single linearizable object history, it must be that the corresponding memory traces cannot be merged. In particular, if σ_1 and σ_2 have neither RAW nor RMW, then they can always be merged on SCM, which gives us the impossibility result of [6] for this object.

sors [31], and the weaker *release-acquire* model (RA), a fragment of C/C++11 [8, 26], which guarantees causal consistency together with per-location strict consistency (a.k.a. coherence).

The standard memory model for the design and analysis of asynchronous shared memory algorithms is SCM. These algorithms however, are not guaranteed to work correctly on weaker memory models (such as TSO and RA) due to the lack of atomicity of reads and writes. To ensure atomicity, one can use *fence* or atomic *read-modify-write (RMW)* instructions provided by the weak memory models. However, since fences and RMWs disable hardware optimizations and enforce synchronization between threads, they incur substantial performance overheads. Thus, one would like to understand when fences and RMWs are necessary and when they can be avoided, in order to correctly and efficiently implement the large body of existing shared memory algorithms on weak memory architectures.

In this paper, we set out to tackle this important and challenging question. The crux of our approach is based on *mergeability* of traces and object histories. Roughly speaking, two memory traces (sequences of memory accesses) of some memory model M are strongly (resp., weakly) mergeable if every (resp., some) interleaving of these traces forms a valid trace of M. Likewise, two object histories (sequences of invocations and responses) of some object O are strongly (resp., weakly mergeable) if every (resp., some) interleaving of these histories forms a valid history of O. Then, our key result is the *Merge Theorem*, which, roughly speaking, states that strongly (resp., weakly) mergeable memory traces can only be used to implement strongly (resp., weakly) mergeable object histories. Contrapositively, when operations of a certain concurrent object are not strongly (resp., weakly) mergeable, then the memory traces implementing these operations on a memory model M cannot be strongly (resp., weakly) mergeable in M. The correctness and progress conditions in the Merge Theorem are weaker versions of *linearizability* [22] and *obstruction-freedom* [21].

A prerequisite for applying our Merge Theorem for a particular memory model is to identify useful mergeability properties of the model. For SCM, TSO, and RA, we develop a set of properties (see Table 1) that describe conditions under which traces of the models can be (weakly/strongly) merged. These results provide key insights into the synchronization power of these memory models, and together with the Merge Theorem allow us to derive multiple impossibility results, and identify optimal implementations.

For instance, consider the read-after-write pattern (RAW), which is often used by shared memory algorithms under SCM (such as classical mutual exclusion [17, 28]) as a synchronization mechanism. In RAW, a process first writes to a shared variable and then reads from a different shared variable, and under SCM, this ensures that at least one of the two processes writing to two different variables has to observe the value written by the other process (see the SB program in §2). This means that solo traces that use RAW are not mergeable into a single trace. In turn, it is straightforward to establish that any two RAW-free read-write traces (by distinct processes) are weakly mergeable under SCM (§3).

With this observation, we easily re-establish (and generalize) the "Laws of Order" results from [6], showing that mutual exclusion protocols, as well as concurrent objects with *strongly non-commutative* methods, cannot be implemented on SCM with neither RAW nor RMW. We do so by simple mergeability-based arguments (see, e.g., Example 1), instead of rather complex and ad-hoc application of the covering technique used in [6]. Intuitively, two methods are strongly non-commutative if executing one of them first affects the response of the other, and vice versa. Moreover, by using mergeability properties for TSO and RA we directly obtain similar impossibility results for these models, whereas the argument in [6] for weak memory models is only implicit, based on the fact that enforcing a write to be executed before a read (i.e., implementing RAW) on a weak model requires a fence.

A benefit of our generic methodology is that we can also reason about implementability of methods that are not strongly non-commutative, hence not covered by [6]:

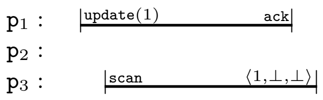
One-Sided Non-Commutative Operations. Some objects such as register, max-register, snapshot and monotone counter have pairs of methods that do not strongly non-commute. To support them, we consider *one-sided non-commutativity* of pairs of methods, which, roughly speaking, means that executing one of them first affects the response of the other, but not necessarily vice versa. We then apply the Merge Theorem to show that any linearizable obstruction-free implementations of these objects must use fences or RMWs in TSO and RA.

Then, for max-register, a useful building block in several implementations, e.g., [3, 7, 15], we obtain *fence-optimal* implementations in TSO and RA. The TSO implementation is obtained through a more general *fence-insertion strategy*: a transformation that takes any read/write linearizable implementation in SCM and adds fences between every write followed by a read or a return of an operation, provably resulting in a linearizable implementation in TSO. Combined with a wait-free read/write max-register implementation in SCM (with uses neither RAW nor RMW), the transformation gives a fence-optimal wait-free read/write max-register implementation in TSO. For RA, we develop a similar linearizable implementation by placing a fence in the beginning and the end of every operation, which leads to a fence-optimal implementation of max-register in RA.

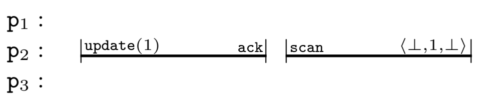
Snapshot and Counter. We also reason about snapshot and (non-monotone) counter, which fall beyond the scope of non-commutativity. These two objects are of particular interests: snapshot is *universal* for a family of objects whose pairs of operations either commute or one overwrites the other [5], and counter is a useful building block for randomized consensus [2, 4]. For TSO, the fence-insertion transformation above once again provides a wait-free fence-optimal snapshot (resp., counter) implementation where every update operation ends with a fence. However, we use our Merge Theorem to show that, in sharp contrast to max-register, there is no obstruction-free read/write snapshot (resp., counter) implementation in RA, whose operations start with a fence and end with a fence (see outline in Example 2). To the best of our knowledge, this is the first sharp separation between max-register on the one hand and snapshot and counter on the other in terms of their implementability under RA using only reads, writes and fences.

Example 2 Linearizable Obstruction-Free Snapshot

Mergeability can justify a novel impossibility result for RA, showing that a shared (single-writer multi-reader) snapshot object cannot be implemented with only reads, writes and fences under the restriction that all fences are only placed at the beginning and end of a method invocation. Consider the following histories assuming three processes:



history h_1



history h_2

An obstruction-free implementation should generate both histories. A merge-based argument implies that the memory traces σ_1 and σ_2 induced by the implementation when it generates h_1 and h_2 must not be mergable in the underlying memory model. Otherwise, the same algorithm will also allow some interleaving h of h_1 and h_2, but it is easy to observe that no such interleaving is linearizable: no valid single history h with only two updates, $\mathtt{update}(1)$ by $\mathtt{p}_1$ and $\mathtt{update}(1)$ by $\mathtt{p}_2$, can have *both* scan results $\langle 1, \perp, \perp\rangle$ and $\langle \perp, 1, \perp\rangle$. The RA memory model allows any two RMW-free traces σ_1 and σ_2 by disjoint sets of processes to be merged, provided that fences are not used in the middle of these traces. Roughly speaking, following [26, 24], the semantics of RA is based on point-to-point communication, making it is possible for $\mathtt{p}_1$ and $\mathtt{p}_3$ to communicate directly, without affecting $\mathtt{p}_2$. Thus, every implementation of snapshot on RA uses RMWs or fences in the middle of operations.

Outline. The rest of this paper is structured as follows. In §2 we define the notion of a memory model. In §3 we establish multiple mergeability properties for these memory models. In §4 we present the general impossibility result. In §5 we discuss applications of the theorem for well known objects, and tightness of the obtained lower bounds. We conclude and discuss related work in §6. The full version of that paper [14] contains more details and full proofs.

2 Weak Memory Models

In this paper, we consider three memory models:

Strictly Consistent Memory (SCM): In this model every write is propagated to all threads immediately after being executed. In the weak memory literature, this memory model is often referred to as sequential consistency, but it essentially corresponds to a collection of *linearizable* (a.k.a. atomic) register objects [22].
Total Store Order (TSO): Each process has a local FIFO store buffer. Writes are first enqueued in the buffer of the writing process, and later propagate from the buffer to main memory in an *internal* step that occurs non-deterministically as part of the system's execution. A read of a variable returns the latest write to the variable in the reading process' buffer or the value in main memory if there is no pending write to that variable in the buffer.
Release/Acquire (RA): This model employs a notion of synchronization between processes through acquiring instructions (read or RMW) which synchronize with previously executed releasing instructions (write or RMW) when the acquiring instruction reads its value from the releasing instruction. Such synchronization transfers "happens-before" knowledge from the releasing instruction to the acquiring instruction. Following a release-acquire synchronization, instructions that follow (in "happens-before" order) the acquire instruction must be consistent with the happens-before knowledge received through the synchronization.

The classic examples used to explain these memory models are the *store buffering* (SB), *independent reads of independent writes* (IRIW), and *message passing* (MP) programs, given below. We assume shared variables x and y initialized with the value 0 and process-local variables $a, b, \ldots$. The possible final values of $a, b, \ldots$ depend on the memory model.

```
Proc p1 || Proc p2      Proc p1 || Proc p2 || Proc p3 || Proc p4      Proc p1 || Proc p2
x := 1; || y := 1;      x := 1;  || a := x; || c := y; || y := 1;      x := 1; || a := y;
a := y; || b := x;               || b := y; || d := x; ||              y := 1; || b := x;
      (SB)                                 (IRIW)                              (MP)
```

Under SCM, no execution of SB ends with $a = b = 0$, while this outcome is possible under both TSO and RA. Under both SCM and TSO, no execution of IRIW ends with $a = c = 1$ and $b = d = 0$, while this outcome is possible under RA, indicating that under RA, processes p_2 and p_3 observe the writes to x and y in a different order. In particular, under RA, suppose that both p_1 and p_4 execute their writes. It is possible for p_2 (resp., p_3) to read the new value for x (resp., y) then read the old value for y (resp., x). Although RA is weaker than both SCM and TSO, like TSO, RA maintains causal consistency as demonstrated MP. Under all three memory models, when MP terminates, if $a = 1$, then $b = 1$, indicating that if p_2 is aware of the write to y by p_1, then it must also be aware of the prior write to x.

Non-SCM-outcomes (a.k.a. *weak behaviors*) can be avoided in weak memory models by using *fence* instructions. In TSO fences drain the store buffer of the process that executes the fence. In RA fences synchronize *in pairs*, transferring happens-before knowledge from one process to another. We formally include fences also in SCM (with "no-op" semantics).

2.1 Formalizing Weak Memory Models

For the formal definitions of the models, we find it most convenient to follow an operational presentation, where memory models are specified by labeled transition systems.

Sequences. For a sequence $s = \langle x_1, \ldots, x_n \rangle$, $s[i]$ denotes the ith element of s (i.e., x_i), and $|s|$ denotes the length of s (i.e., n). We write $x \in s$ when $s[i] = x$ for some $1 \leq i \leq n$. We denote by ε the empty sequence, write $s_1 \cdot s_2$ for concatenation of s_1 and s_2 and denote by X^* the set of all sequences over elements of a set X. The restriction of a sequence s w.r.t. a set Y, denoted $s|_Y$, is the longest subsequence of s that consists only of elements in Y. These notations are lifted to sets in the obvious way (e.g., $S \cdot s' \triangleq \{s \cdot s' \mid s \in S\}$ and $S|_Y \triangleq \{s|_Y \mid s \in S\}$). We use the suffix '-set' to lift a function f from some set X to a function form sequences over X, formally defined by: $f\text{-}\mathsf{set}(s) \triangleq \{f(s[i]) \mid 1 \leq i \leq |s|\}$.

Labeled Transition Systems (LTSs). An LTS L consists of a set of states, $\mathsf{states}(L)$; an initial state, $\mathsf{init}(L) \in \mathsf{states}(L)$; a set of transition labels, $\mathsf{labels}(L)$; and a set of transitions, $\mathsf{trans}(L) \subseteq \mathsf{states}(L) \times \mathsf{labels}(L) \times \mathsf{states}(L)$. We write $q \xrightarrow{l}_L q'$ for $\langle q, l, q' \rangle \in \mathsf{trans}(L)$, and given $\pi = \langle l_1, \ldots, l_n \rangle \in \mathsf{labels}(L)^*$, we write $q \xrightarrow{\pi}_L q'$ for $\exists q_2, \ldots, q_n.\, q \xrightarrow{l_1}_L q_2 \xrightarrow{l_2}_L \ldots q_n \xrightarrow{l_n}_L q'$. An *execution fragment* of L is a sequence $\alpha = \langle q_0, l_1, q_1, l_2, \ldots, l_n, q_n \rangle$ of alternating states and transition labels such that $q_i \xrightarrow{l_{i+1}}_L q_{i+1}$ for every $0 \leq i \leq n-1$. The *trace* of α, denoted $\mathsf{trace}(\alpha)$, is the restriction of α w.r.t. $\mathsf{labels}(L)$. We denote by $\mathsf{traces}(L, q)$ the set of all sequences that are traces of some execution fragment α of L that starts from $q \in \mathsf{states}(L)$. An execution fragment α of L is an *execution* of L if it starts from $\mathsf{init}(L)$. A sequence π of transition labels is a *trace* of L if it is a trace of some execution of L. We denote by $\mathsf{traces}(L)$ the set of all traces of L (so we have $\mathsf{traces}(L) = \mathsf{traces}(L, \mathsf{init}(L))$).

Domains. We assume sets Var of *shared variables* and Val of *values* with a distinguished *initial* value $0 \in \mathsf{Val}$. We let $\mathsf{P} \triangleq \{\mathsf{p}_1, \ldots, \mathsf{p}_N\}$ be the set of process identifiers.

Memory Actions. Memory operations execute atomically using *memory actions*, which include both argument and return values. Formally, a *memory action* $a \in \mathsf{MemActs}$ is one the following (where $x \in \mathsf{Var}$ and $v, v_{\mathsf{old}}, v_{\mathsf{new}} \in \mathsf{Val}$): (i) write action of the form $\mathtt{W}(x, v)$; (ii) read action of the form $\mathtt{R}(x, v)$; (iii) RMW action of the form $\mathtt{RMW}(x, v_{\mathsf{old}}, v_{\mathsf{new}})$; and (iv) fence action of the form $\mathtt{F}$. We denote by $\mathsf{typ}(a)$ the type of the memory action a ($\mathtt{W}$, $\mathtt{R}$, $\mathtt{RMW}$, or $\mathtt{F}$) and by $\mathsf{var}(a)$ the variable accessed by action a (when applicable).

Memory Events. A *memory event* $e \in \mathsf{MemEvs}$ is a pair $e = p{:}a$ where $p \in \mathsf{P}$ and $a \in \mathsf{MemActs}$. We use $\mathsf{proc}(e)$ and $\mathsf{act}(e)$ to retrieve the components of e (p and a, respectively). The functions $\mathsf{typ}(\cdot)$ and $\mathsf{var}(\cdot)$ are lifted to events in the obvious way.

Memory Models. The semantics of the memory operations is given by an LTS, called a *memory model*. The transition labels of a memory model M, $\mathsf{labels}(M) \triangleq \mathsf{MemEvs} \cup \{\tau\}$, consist of memory events, as well as τ, which represents a silent memory internal step.

We demonstrate the formulation of TSO as an LTS. The formal models for SCM and RA can be found in [14].

▶ **Definition 2.1.** TSO's states are pairs $\langle m, b\rangle$, where $m \in \mathsf{Var} \to \mathsf{Val}$ is the main memory and $b \in \mathsf{P} \to (\mathsf{Var} \times \mathsf{Val})^*$ assigns a store buffer to every process; the initial state is $\mathsf{init}(\mathrm{TSO}) \triangleq \langle \lambda x.\, 0, \lambda p.\, \varepsilon\rangle$ (i.e., all variables in memory are zeroed and all store buffers are empty); and the transitions are as follows, where $\beta|_x$ denotes the restriction of a store buffer β to pairs of the form $\langle x, \_\rangle$:

$$\textsc{write}\ \frac{\begin{array}{c} e = p{:}\mathtt{W}(x,v) \\ b' = b[p \mapsto b(p)\cdot\langle x,v\rangle] \end{array}}{\langle m,b\rangle \xrightarrow{e} \langle m,b'\rangle} \qquad \textsc{read-from-buffer}\ \frac{\begin{array}{c} e = p{:}\mathtt{R}(x,v) \\ b(p)|_x = \_ \cdot \langle\langle x,v\rangle\rangle \end{array}}{\langle m,b\rangle \xrightarrow{e} \langle m,b\rangle} \qquad \textsc{read-from-memory}\ \frac{\begin{array}{c} e = p{:}\mathtt{R}(x,v) \\ b(p)|_x = \varepsilon \qquad m(x) = v \end{array}}{\langle m,b\rangle \xrightarrow{e} \langle m,b\rangle}$$

$$\textsc{rmw}\ \frac{\begin{array}{c} e = p{:}\mathtt{RMW}(x, v_{\mathsf{old}}, v_{\mathsf{new}}) \\ b(p) = \varepsilon \qquad m(x) = v_{\mathsf{exp}} \end{array}}{\langle m,b\rangle \xrightarrow{e} \langle m[x \mapsto v_{\mathsf{new}}], b\rangle} \qquad \textsc{fence}\ \frac{\begin{array}{c} e = p{:}\mathtt{F} \\ b(p) = \varepsilon \end{array}}{\langle m,b\rangle \xrightarrow{e} \langle m,b\rangle} \qquad \textsc{propagate}\ \frac{\begin{array}{c} b(p) = \langle\langle x,v\rangle\rangle \cdot \beta \\ m' = m[x \mapsto v] \qquad b' = b[p \mapsto \beta] \end{array}}{\langle m,b\rangle \xrightarrow{\tau} \langle m',b'\rangle}$$

Memory Sequences. We refer to sequences $\rho \in (\mathsf{MemEvs} \cup \{\tau\})^*$ as *memory sequences* and to sequences $\sigma \in \mathsf{MemEvs}^*$ as *observable memory sequences*. We use the following notations:

- $\sigma|_p$ denotes the restriction of σ w.r.t. $\{e \in \mathsf{MemEvs} \mid \mathsf{proc}(e) = p\}$.
- $\mathsf{otraces}(M, q)$ denotes the set of all observable memory sequences obtained by restricting traces of M from a state q to non-τ steps, i.e., $\mathsf{otraces}(M, q) \triangleq \mathsf{traces}(M, q)|_{\mathsf{MemEvs}}$.
- $\mathsf{otraces}(M) \triangleq \mathsf{traces}(M)|_{\mathsf{MemEvs}}$ is the set of all observable memory sequences of M.

Stable States. A state $q \in \mathsf{states}(M)$ is *stable* if $q \overset{\tau}{\not\to}_M q'$ for any $q' \in \mathsf{states}(M)$. Every state of SCM is stable, a state of TSO is stable iff all store buffers are empty, and a state of RA is stable iff all processes are aware of all writes.

Well-Behaved Memory Models. TSO is strictly weaker than SCM and RA is strictly weaker than TSO, which formally means that $\mathsf{otraces}(\mathrm{SCM}) \subsetneq \mathsf{otraces}(\mathrm{TSO}) \subsetneq \mathsf{otraces}(\mathrm{RA})$. In the sequel we will need the following assumption on memory models:

▶ **Definition 2.2.** A memory model M is *well-behaved* if there exists a simulation R from SCM to M whose codomain consists solely of stable states. That is, there should exist a relation $R \subseteq \mathsf{states}(\mathrm{SCM}) \times \{q \in \mathsf{states}(M) \mid q \text{ is stable}\}$ such that (i) $\langle \mathsf{init}(\mathrm{SCM}), \mathsf{init}(M)\rangle \in R$; and (ii) if $\langle m, q\rangle \in R$ and $m \xrightarrow{l}_{\mathrm{SCM}} m'$, then $q \xrightarrow{l}_M \xrightarrow{\tau}{}^*_M q'$ and $\langle m', q'\rangle \in R$ for some stable $q' \in \mathsf{states}(M)$.

#Name	Memory model	Restrictions on σ_1 process	Restrictions on σ_1 events	Restrictions on σ_1 pattern	Restrictions on σ_2 process	Restrictions on σ_2 events	Restrictions on σ_2 pattern	Merge property
1 TSO^s	TSO	solo	RW	—	solo	RW	—	Strong
2 RA$^s_1$	RA	—	RW	—	—	—	—	Strong
3 RA$^s_2$	RA	—	RWF	PPTF	—	—	PPTF	Strong
4 RA$^s_3$	RA	—	RWF	PPLF	—	—	PPLF	Strong
5 SCM^w	SCM	—	RW	RBW	—	—	—	Weak
6 TSO^w	TSO	solo	RWF	LTF	—	—	—	Weak
7 RA^w	RA	—	RWF	LTF	—	—	—	Weak

Table 1 Merging observable memory sequences σ_1 and σ_2 such that $\mathsf{proc\text{-}set}(\sigma_1) \cap \mathsf{proc\text{-}set}(\sigma_2) = \emptyset$

Note that if M is well-behaved, then $\sigma_0 \cdot \sigma \in \mathsf{otraces}(\text{SCM})$ implies that there exist a stable state $q \in \mathsf{states}(M)$ and a memory trace ρ_0 such that $\mathsf{init}(M) \xrightarrow{\rho_0}_M q$, $\rho_0|_{\mathsf{MemEvs}} = \sigma_0$, and $\sigma \in \mathsf{otraces}(M, q)$. The following lemma is proven in [14].

▶ **Lemma 2.3.** *Each $M \in \{\text{SCM}, \text{TSO}, \text{RA}\}$ is well-behaved.*

3 Mergeability Results for Memory Models

We consider two notions of mergeability of observable memory traces, *weak* mergeability, which means that *some* interleaving of the given traces is admitted, and *strong* mergeability, which requires that *all* interleavings are admitted. We denote by $s_1 \sqcup\!\sqcup s_2$ the the set of all interleavings of s_1 and s_2.

For our impossibility result to handle a non-empty base object history (as in Example 1), it does not suffice to merge memory traces from the initial state. Instead, we require the traces to be mergeable from every *stable* state:

▶ **Definition 3.1.** Two observable memory traces σ_1, σ_2 with $\mathsf{proc\text{-}set}(\sigma_1) \cap \mathsf{proc\text{-}set}(\sigma_2) = \emptyset$ are *weakly (resp., strongly) mergeable* in a memory model M if for every stable state $q_0 \in \mathsf{states}(M)$ such that $\sigma_1, \sigma_2 \in \mathsf{otraces}(M, q_0)$, we have $\sigma \in \mathsf{otraces}(M, q_0)$ for some (resp., every) $\sigma \in \sigma_1 \sqcup\!\sqcup \sigma_2$.

Table 1 presents the merge properties established for the memory models we consider (see [14] for the proofs). To specify restrictions on the mergeable traces, we say that an observable memory sequence σ is:

solo if $|\mathsf{proc\text{-}set}(\sigma)| = 1$;
read-write (RW) if $\mathsf{typ\text{-}set}(\sigma) \subseteq \{\mathtt{R}, \mathtt{W}\}$;
read-write-fence (RWF) if $\mathsf{typ\text{-}set}(\sigma) \subseteq \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$;
read-before-write (RBW) if for every $k_1 < k_2$, if $\mathsf{typ}(\sigma[k_1]) = \mathtt{W}$, $\mathsf{typ}(\sigma[k_2]) = \mathtt{R}$, and $\mathsf{var}(\sigma[k_1]) \neq \mathsf{var}(\sigma[k_2])$, then $\mathsf{typ}(\sigma[k]) = \mathtt{W}$ and $\mathsf{var}(\sigma[k]) = \mathsf{var}(\sigma[k_2])$ for some $k_1 < k < k_2$;[1]
trailing-fence (TF) if there is no k such that $\mathsf{typ}(\sigma[k]) = \mathtt{F}$ but $\mathsf{typ}(\sigma[k+1]) \neq \mathtt{F}$;
leading-fence (LF) if there is no k such that $\mathsf{typ}(\sigma[k]) = \mathtt{F}$ but $\mathsf{typ}(\sigma[k-1]) \neq \mathtt{F}$;
per-process trailing fence (PPTF) if $\sigma|_p$ is TF for all processes p;
per-process leading fence (PPLF) if $\sigma|_p$ is LF for all processes p; and
leading-and-trailing-fence (LTF) if $\sigma = \sigma_1 \cdot \sigma_2$ for some LF σ_1 and TF σ_2.

[1] RBW is equivalent to the absence of the read-after-write (RAW) pattern as defined in [6].

We have three types of restrictions, namely: (i) a restriction on the processes (solo); (ii) restrictions on the types of events (RW and RWF); and (iii) restrictions on the access pattern (all others). The restrictions on types and access patterns correspond to synchronization mechanisms that are expensive performance wise. RMWs and non-RBW were identified as such in [6], and since we explicitly deal with weak memory models, we add fences to this list. To motivate our focus on leading/trailing fence placement, we note that the trivial linearizable implementation of an atomic register using a write/read instruction requires fences: at the end of every write operation on TSO, and at the beginning and the end of every (write/read) operation on RA. We aim to investigate whether other objects admit similar implementations.

Next, we briefly discuss the results in the table:

SCM. In SCM, if σ_1 is RW-RBW, then it can be weakly merged with any other observable memory trace. Indeed, being RW-RBW, σ_1 must be of the form $\sigma_1^{\mathsf{r}} \cdot \sigma_1^{\mathsf{w}}$ where σ_1^{r} is a sequence of reads and σ_1^{w} is a sequence of writes and reads, starting with a write, where the reads in σ_1^{w} read from the writes in σ_1^{w}. Then, it is straightforward to see that σ_1 and any observable memory sequence σ_2 can be merged to form the trace $\sigma = \sigma_1^{\mathsf{r}} \cdot \sigma_2 \cdot \sigma_1^{\mathsf{w}}$, which is valid trace under SCM. We note that the RBW restriction is necessary here, as $\langle \mathtt{p}_1{:}\mathtt{W}(x,1), \mathtt{p}_1{:}\mathtt{R}(y,0)\rangle$ and $\langle \mathtt{p}_2{:}\mathtt{W}(y,1), \mathtt{p}_2{:}\mathtt{R}(x,0)\rangle$ (which may arise from the SB example) are not weakly mergeable. Also note that there is no useful *strong* merge property for SCM. Even $\langle \mathtt{p}_1{:}\mathtt{W}(x,1)\rangle$ and $\langle \mathtt{p}_2{:}\mathtt{R}(x,0)\rangle$ cannot be strongly merged.

TSO. In TSO, σ_1 and σ_2 can be *strongly* merged when they are both solo-RW traces. This holds because with only writes and reads, there is always an observable trace where *all* the writes of both σ_1 and σ_2 remain in the local store buffers, allowing the events of σ_1 and σ_2 to be arbitrarily interleaved. TSO also satisfies a weak merge property if σ_1 is solo-RWF-LTF and σ_2 is arbitrary. To do so, we let $\sigma_1 = \sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_1^{\mathsf{tf}}$ where $\mathsf{typ\text{-}set}(\sigma_1^{\mathsf{lf}}) \cup \mathsf{typ\text{-}set}(\sigma_1^{\mathsf{tf}}) \subseteq \{\mathtt{F}\}$ and σ_1' is RW. Then, $\sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_2 \cdot \sigma_1^{\mathsf{tf}}$ is a valid TSO observable trace since no instruction in σ_1' forces writes to propagate. We note that the solo restriction is essential. For example, $\langle \mathtt{p}_1{:}\mathtt{W}(x,1), \mathtt{p}_2{:}\mathtt{R}(x,1), \mathtt{p}_2{:}\mathtt{R}(y,0)\rangle$ and $\langle \mathtt{p}_4{:}\mathtt{W}(y,1), \mathtt{p}_3{:}\mathtt{R}(y,1), \mathtt{p}_3{:}\mathtt{R}(x,0)\rangle$ (which may arise from the IRIW example) are not weakly mergeable.

RA. We prove three strong merge properties for RA: ($\mathrm{RA}_1^{\mathsf{s}}$) If σ_1 is RW, then it can be strongly merged with σ_2 even when σ_1 is non-solo. Indeed, in the absence of RMWs and fences in σ_1, the writes in σ_1 can be propagated to other processes of σ_1, but never propagate to the processes of σ_2, and vice-versa. ($\mathrm{RA}_2^{\mathsf{s}}$) If σ_1 is RWF-PPTF and σ_2 is PPTF, the strong merge argument is as follows. First, we remove all the fences in σ_1, which results in an RW trace. From $\mathrm{RA}_1^{\mathsf{s}}$, this trace can be strongly merged with σ_2. In the resulting trace, we reintroduce the fences removed from σ_1 arbitrarily after the last read or write of the corresponding process. Regardless of whether this fence is before or after a fence of σ_2, the resulting fence synchronization has no effect since σ_2 is also PPTF. ($\mathrm{RA}_3^{\mathsf{s}}$) If σ_1 is RWF-PPLF and σ_2 is PPLF the argument is symmetric to $\mathrm{RA}_2^{\mathsf{s}}$. Finally, RA satisfies a weak merge property if σ_1 is RWF-LTF. As in the TSO weak merge property, we split $\sigma_1 = \sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_1^{\mathsf{tf}}$. By $\mathrm{RA}_1^{\mathsf{s}}$, $\sigma_1' \cdot \sigma_2$ is an RA observable trace. Then, $\sigma_1^{\mathsf{lf}} \cdot \sigma_1' \cdot \sigma_2 \cdot \sigma_1^{\mathsf{tf}}$ is an RA observable trace since the leading/trailing fences have no bearing on the execution.

4 A Recipe for Merge-Based Impossibility Results

We introduce objects, implementations, and histories (§4.1), and our main theorem (§4.2).

$$\frac{e = p{:}\mathtt{inv}(o) \quad q = \mathsf{init}(\mathcal{I}(o,p))}{\bot \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \langle o, q\rangle} \qquad \frac{e \in \mathsf{MemEvs} \quad q \xrightarrow{e}_{\mathcal{I}(o,p)} q'}{\langle o, q\rangle \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \langle o, q'\rangle} \qquad \frac{e = p{:}\mathtt{res}(u) \quad q \xrightarrow{e}_{\mathcal{I}(o,p)} -}{\langle o, q\rangle \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} \bot}$$

Figure 1 Transitions of $\mathsf{S}^p_{\mathcal{I}}$

$$\frac{\bar{q}(p) \xrightarrow{e}_{\mathsf{S}^p_{\mathcal{I}}} q'}{\bar{q} \xrightarrow{e}_{\mathsf{S}_{\mathcal{I}}} \bar{q}[p \mapsto q']}$$

Figure 2 Transitions of $\mathsf{S}_{\mathcal{I}}$

4.1 Objects and Their Implementations

We consider systems implementing of a high-level object O using the low-level atomic shared-memory operations provided by the memory model M.

Objects. An *object* O is a pair $O = \langle ops, rets\rangle$, where ops is a set of *operation names* (each of which may include argument values) and $rets$ is a set of *response values*. We use $\mathsf{ops}(O)$ and $\mathsf{rets}(O)$ to retrieve the components of an object O (ops and $rets$, respectively). We use $\mathtt{ack}$ for a default response value for operations that do not return any value.

Object Actions. To delimit executions of operations of O, we use *object actions* that can be either *invocation actions* of the form $\mathtt{inv}(o)$ with $o \in \mathsf{ops}(O)$, or *response actions* of the form $\mathtt{res}(u)$ with $u \in \mathsf{rets}(O)$. We let $\mathsf{acts}(O)$ denote the set of all object actions of O.

Object Events. Like memory events defined in §2, *object events* are pairs $e = p{:}a$ where $p \in \mathsf{P}$ and $a \in \mathsf{acts}(O)$. We apply the same notations used for memory events to object events, and let $\mathsf{Evs}(O)$ denote the set of all object events. By *event* we collectively refer to either a memory event or an object event. Given a sequence π of events, we define the following notations:

- $\pi|_p$ denotes the restriction of π w.r.t. the set of events e with $\mathsf{proc}(e) = p$.
- $\pi|_{\mathsf{M}}$ denotes the restriction of π w.r.t. the set MemEvs of memory events.
- $\pi|_O$ denotes the restriction of π w.r.t. the set $\mathsf{Evs}(O)$ of object events.

Histories. A *history* of an object O is a sequence of events in $\mathsf{Evs}(O)$. We denote by $(p\colon \lfloor o \quad u \rfloor)$ the history consisting of a single operation by process $p \in \mathsf{P}$ invoking $o \in \mathsf{ops}(O)$ with response value $u \in \mathsf{rets}(O)$ (and omit the response value if it is $\mathtt{ack}$), i.e., $(p\colon \lfloor o \quad u \rfloor) \triangleq \langle p{:}\mathtt{inv}(o), p{:}\mathtt{res}(u)\rangle$ and $(p\colon \lfloor o \quad \rfloor) \triangleq \langle p{:}\mathtt{inv}(o), p{:}\mathtt{res}(\mathtt{ack})\rangle$. A history h is:

- *sequential* if it is a prefix of a history of the form $(p_1\colon \lfloor o_1 u_1 \rfloor) \cdot (p_2\colon \lfloor o_2 u_2 \rfloor) \cdots (p_n\colon \lfloor o_n u_n \rfloor)$;
- *well-formed* if $h|_p$ is sequential for every $p \in \mathsf{P}$; and
- *complete* if it is well-formed and each $h|_p$ ends with a response event.

We let $\mathsf{H}(O)$, $\mathsf{ComH}(O)$, and $\mathsf{ComSeqH}(O)$ denote the sets of all well-formed histories of O, all complete histories of O, and all complete sequential histories of O (respectively).

Specifications. We assume that every object O is associated with a *specification*, denoted $\mathsf{spec}(O)$, that is a subset of $\mathsf{ComSeqH}(O)$ that is prefix-closed (in the sense that $h' \in \mathsf{spec}(O)$ for every $h' \in \mathsf{ComSeqH}(O)$ that is a prefix of some $h \in \mathsf{spec}(O)$). An object O is *deterministic* if no two histories in $\mathsf{spec}(O)$ have longest common prefix that ends with an invocation.

Implementations. An *implementation I of an operation o for a process p* is an LTS whose set of transition labels are events with process identifier p. We assume that a response event is always the last transition of executions of I (i.e., if $q \xrightarrow{p{:}\mathtt{res}(u)}_I q'$, then no transition is enabled in q'). An *implementation $\mathcal{I}$ of an object O* is a function assigning an implementation $\mathcal{I}(o,p)$ of o for p to every $o \in \mathsf{ops}(O)$ and $p \in \mathsf{P}$.

An implementation $\mathcal{I}$ of an object O induces an LTS, denoted $\mathsf{S}_{\mathcal{I}}$, that repeatedly and concurrently executes the operations of O as $\mathcal{I}$ prescribes. To formally define $\mathsf{S}_{\mathcal{I}}$, we first define the "per-process" LTS induced by $\mathcal{I}$, denoted $\mathsf{S}^p_{\mathcal{I}}$. This LTS is given by: $\mathsf{states}(\mathsf{S}^p_{\mathcal{I}}) \triangleq$

$\{\bot\} \cup \{\langle o, q\rangle \mid o \in \mathsf{ops}(O), q \in \mathsf{states}(\mathcal{I}(o,p))\}$; $\mathsf{init}(\mathsf{S}^p_{\mathcal{I}}) \triangleq \bot$; $\mathsf{labels}(\mathsf{S}^p_{\mathcal{I}}) \triangleq \mathsf{Evs}(O) \cup \mathsf{MemEvs}$; and the transitions are given in Fig. 1. The state $\bot$ means that the process is not currently executing any operation, whereas $\langle o, q\rangle$ means that process p is currently executing o and it is in state q of the implementation of o for p.

In turn, $\mathsf{S}_{\mathcal{I}}$ is given by: $\mathsf{states}(\mathsf{S}_{\mathcal{I}})$ is the set of all mappings assigning a state in $\mathsf{states}(\mathsf{S}^p_{\mathcal{I}})$ to every $p \in \mathsf{P}$; $\mathsf{init}(\mathsf{S}_{\mathcal{I}}) \triangleq \lambda p.\, \bot$; $\mathsf{labels}(\mathsf{S}_{\mathcal{I}}) \triangleq \mathsf{Evs}(O) \cup \mathsf{MemEvs}$; and the transition relation in Fig. 2. This transition simply interleaves the transitions of the different processes. In the sequel, we let $\mathsf{traces}(\mathcal{I}) \triangleq \mathsf{traces}(\mathsf{S}_{\mathcal{I}})$.

Histories of Implementations. Let $\mathcal{I}$ be an implementation of an object O, π_0 be a sequence of events, and M be a memory model. A history h of O is:

- *generated by* $\mathcal{I}$ *after* π_0 if $h = \pi|_O$ for some π such that $\pi_0 \cdot \pi \in \mathsf{traces}(\mathcal{I})$.
- *generated by* $\mathcal{I}$ *after* π_0 *under* M if $h = \pi|_O$ for some π such that $\pi_0 \cdot \pi \in \mathsf{traces}(\mathcal{I})$ and $(\pi_0 \cdot \pi)|_{\mathsf{M}} \in \mathsf{otraces}(M)$.

We denote by $\mathsf{H}(\pi_0, \mathcal{I})$ the set of all histories that are generated by $\mathcal{I}$ after π_0, and by $\mathsf{H}(\pi_0, \mathcal{I}, M)$ the set of all histories generated by $\mathcal{I}$ after π_0 under M. We also write $\mathsf{H}(\mathcal{I})$ instead of $\mathsf{H}(\varepsilon, \mathcal{I})$ and $\mathsf{H}(\mathcal{I}, M)$ instead of $\mathsf{H}(\varepsilon, \mathcal{I}, M)$.

4.2 The Merge Theorem

Our main result relates mergeability properties of memory models and objects implemented in those models, assuming that the implementation provides minimal safety and liveness guarantees. This result can be also seen as a *CAP Theorem* for weak memory models [19], where partition tolerance of CAP corresponds to mergeability, as it allows two traces of distinct set of processes to run concurrently without interaction. Our results are more fine grained, as we show the correspondence between mergeability of certain traces in a memory model, and the (in)ability of these traces to implement non-mergeable object histories.

For the formal treatment, we first present the following lemma (proven in [14]). The lemma describes the key shape of our results, namely that given two traces of an implementation over a memory model, the merge property over these traces carries over to a merge property over the histories induced by the traces.

▶ **Lemma 4.1.** *Let $\mathcal{I}$ be an implementation of O. Suppose that there exist sequences π_0, π_1, π_2 of events such that the following hold:*

(a) $\mathsf{proc\text{-}set}(\pi_1) \cap \mathsf{proc\text{-}set}(\pi_2) = \emptyset$*;* $\pi_0 \cdot \pi_1, \pi_0 \cdot \pi_2 \in \mathsf{traces}(\mathcal{I})$*;* $\pi_0|_O \in \mathsf{ComH}(O)$*; and*
(b) $\pi_0|_{\mathsf{M}} \cdot \sigma \in \mathsf{otraces}(M)$ *for some (resp., every)* $\sigma \in \pi_1|_{\mathsf{M}} \sqcup \pi_2|_{\mathsf{M}}$.

Then, $h \in \mathsf{H}(\pi_0, \mathcal{I}, M)$ *for some (resp., every)* $h \in \pi_1|_O \sqcup \pi_2|_O$.

The Merge Theorem, which we obtain using this lemma, makes several assumptions on implementations. First, the safety condition, which we call *consistency*, is restriction of linearizability to complete histories. For its definition, we first define reorderings of sequences.

▶ **Definition 4.2.** Let $R \subseteq X \times X$. A sequence $s' \in X^*$ is an *R-reordering* of a sequence $s \in X^*$ if there exists a bijection $f : \{1, \dots, |s|\} \to \{1, \dots, |s'|\}$ such that $s[i] = s'[f(i)]$ for every $1 \le i \le |s|$, and $f(i) < f(j)$ whenever $i < j$ and $\langle s[i], s[j]\rangle \in R$. We denote by $\mathsf{reorder}_R(s)$ the set of all R-reorderings of s, and lift this notation to sets by letting $\mathsf{reorder}_R(S) \triangleq \bigcup_{s \in S} \mathsf{reorder}_R(s)$.

We define sproc and lin relations on events:

$$\mathsf{sproc} \triangleq \{\langle e_1, e_2\rangle \mid \mathsf{proc}(e_1) = \mathsf{proc}(e_2)\}$$
$$\mathsf{lin} \triangleq \mathsf{sproc} \cup (\{e \mid e \text{ is a response event}\} \times \{e \mid e \text{ is a invocation event}\})$$

▶ **Definition 4.3.** A history $h' \in \mathsf{H}(O)$ *linearizes* a history $h \in \mathsf{H}(O)$, denoted $h \sqsubseteq h'$, if $h' \in \mathsf{reorder}_{\mathsf{lin}}(h)$. For a set $H' \subseteq \mathsf{H}(O)$, we write $h \sqsubseteq H'$ if $h \sqsubseteq h'$ for some $h' \in H'$.

▶ **Definition 4.4.** An implementation $\mathcal{I}$ of an object O is *consistent* under a memory model M if $h \sqsubseteq \mathsf{spec}(O)$ for every complete history $h \in \mathsf{H}(\mathcal{I}, M)$.

Consistency follows from linearizability [22], and it is equivalent to linearizability for implementations in which every history can be extended to a complete history.

Next, the liveness condition, which we call *availability*, requires progress for the specific histories under consideration.

▶ **Definition 4.5.** An implementation $\mathcal{I}$ of O is *available after a history* $h_0 \in \mathsf{ComSeqH}(O)$ w.r.t. a history $h \in \mathsf{H}(O)$ if $h \in \mathsf{H}(\pi_0, \mathcal{I}, \mathrm{SCM})$ for every $\pi_0 \in \mathsf{traces}(\mathcal{I})$ such that $\pi_0|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ and $\pi_0|_O = h_0$. We say $\mathcal{I}$ is *available w.r.t. h*, if it is available after ε w.r.t. h (i.e., $h \in \mathsf{H}(\mathcal{I}, \mathrm{SCM})$). We call $\mathcal{I}$ *spec-available* if for every $h_0, h \in \mathsf{ComSeqH}(O)$ such that $h_0 \cdot h \in \mathsf{spec}(O)$, $\mathcal{I}$ is available after h_0 w.r.t. h.

Availability w.r.t. h after h_0 only guarantees that the implementation under SCM is able to generate the history h when it starts executing after generating h_0. For deterministic implementations, availability w.r.t. h after h_0 follows from availability w.r.t. $h_0 \cdot h$ (after ϵ). Note that availability considers SCM rather than a general memory model M, but when M is well-behaved (Def. 2.2), $h \in \mathsf{H}(\pi_0, \mathcal{I}, \mathrm{SCM})$ ensures that $h \in \mathsf{H}(\pi_0, \mathcal{I}, M)$. Spec-availability essentially means that the implementation can generate all (sequential) specification histories and for deterministic objects and implementations, it follows from obstruction-freedom [21].

The next lemma (proven in [14]) is used in the sequel to derive availability w.r.t. a history h from the fact that availability holds w.r.t. a sequential history that linearizes h.

▶ **Lemma 4.6.** *Suppose that $\mathcal{I}$ is available after h_0 w.r.t. a history $h' \in \mathsf{H}(O)$. Then, $\mathcal{I}$ is available after h_0 w.r.t. every $h \in \mathsf{H}(O)$ such that $h \sqsubseteq h'$.*

Next, we define mergeability for objects, akin to mergeability for memory models (Def. 3.1):

▶ **Definition 4.7.** Two histories $h_1, h_2 \in \mathsf{ComH}(O)$ with $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$ are *weakly (resp., strongly) mergeable in* $\mathsf{spec}(O)$ *after a history* $h_0 \in \mathsf{ComSeqH}(O)$ if $h_0 \cdot h_1 \sqsubseteq \mathsf{spec}(O)$ and $h_0 \cdot h_2 \sqsubseteq \mathsf{spec}(O)$ imply that $h_0 \cdot h \sqsubseteq \mathsf{spec}(O)$ for some (resp., every) $h \in h_1 \sqcup h_2$.

We now have all prerequisites to state our Merge Theorem (see [14] for the proof).

▶ **Theorem 4.8.** *Let $\mathcal{I}$ be an implementation of an object O that is consistent under a well-behaved memory model M. Suppose that there exist $\pi_0 \in \mathsf{traces}(\mathcal{I})$, $h_1, h_2 \in \mathsf{ComH}(O)$ such that the following hold, where $h_0 = \pi_0|_O$ and $\sigma_0 = \pi_0|_{\mathsf{M}}$:*

(i) $h_0 \in \mathsf{spec}(O)$, $\sigma_0 \in \mathsf{traces}(\mathrm{SCM})$, *and* $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$,
(ii) $\mathcal{I}$ *is available after* h_0 *w.r.t. some* $h^i_{\mathsf{seq}} \in \mathsf{ComSeqH}(O)$ *such that* $h_i \sqsubseteq h^i_{\mathsf{seq}}$ *for* $i \in \{1, 2\}$,
(iii) h_1 *and* h_2 *are not weakly (resp., strongly) mergeable in* $\mathsf{spec}(O)$ *after* h_0.

Then, there exist π_1 and π_2 such that all of the following hold:

(a) For $i \in \{1,2\}$*, we have* $\pi_0 \cdot \pi_i \in \mathsf{traces}(\mathcal{I})$*;* $\pi_i|_O = h_i$*;* $\sigma_0 \cdot \pi_i|_\mathsf{M} \in \mathsf{traces}(\mathrm{SCM})$*; and* $\mathsf{proc\text{-}set}(\pi_i) = \mathsf{proc\text{-}set}(h_i)$*.*

(b) For every $\pi'_1 \in \mathsf{reorder}_\mathsf{sproc}(\pi_1)$ *and* $\pi'_2 \in \mathsf{reorder}_\mathsf{sproc}(\pi_2)$ *such that* $\pi'_1|_O = h_1$*,* $\pi'_2|_O = h_2$*, and* $\sigma_0 \cdot \pi'_1|_\mathsf{M}, \sigma_0 \cdot \pi'_2|_\mathsf{M} \in \mathsf{traces}(\mathrm{SCM})$*, we have that* $\pi'_1|_\mathsf{M}$ *and* $\pi'_2|_\mathsf{M}$ *are not weakly (resp., strongly) mergeable in* M*. In particular,* $\pi_1|_\mathsf{M}$ *and* $\pi_2|_\mathsf{M}$ *are not weakly (resp., strongly) mergeable in* M*.*

For simplicity, we explain Thm. 4.8 for $\pi_0 = \varepsilon$ (and hence $h_0 = \sigma_0 = \varepsilon$). The theorem assumes that we start with an implementation $\mathcal{I}$ that is consistent under the memory model M under consideration. Moreover, we assume that we have two complete histories h_1 and h_2 of the object such that the processes of h_1 and h_2 are distinct (condition (i)), $\mathcal{I}$ is available w.r.t. some linearization of h_1 and h_2 (condition (ii)), and that h_1 and h_2 are ***not*** weakly (strongly) mergeable (condition (iii)). Then, for $i \in \{1,2\}$ there must be a trace π_i of $\mathcal{I}$, corresponding to h_i, whose memory events are allowed by SCM, and processes are only included in π_i if they call some operation of the object (condition (a)), such that π_1 and π_2 restricted to memory events are ***not*** weakly (strongly) mergeable in M (second clause of condition (b)). In fact, weak (strong) ***non***-mergeability extends to any process-preserving reordering of π_1 and π_2 whose corresponding histories are h_1 and h_2 and corresponding memory traces are SCM traces (first clause of condition (b)).

5 Implementability of Objects on Weak Memory Models

We demonstrate the power of the Merge Theorem by using it along with the mergeability results in Table 1 to characterize implementability of objects under weak memory models.

5.1 One-Sided Non-Commutative Operations

We start by analyzing implementability of pair of operations o_1 and o_2 such that o_1 is *one-sided non-commutative* w.r.t. o_2. Roughly, this means that the execution order of o_1 and o_2 affects the response of o_1. Formally:

▶ **Definition 5.1.** An operation $o_1 \in \mathsf{ops}(O)$ is *one-sided non-commutative w.r.t. an operation* $o_2 \in \mathsf{ops}(O)$ *in* $\mathsf{spec}(O)$ if there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1, v_1, u_2 \in \mathsf{rets}(O)$ such that: (i) $u_1 \neq v_1$; (ii) $h_0 \cdot (p_1\colon \underline{|\,o_1\,u_1\,|}\,) \in \mathsf{spec}(O)$; and (iii) $h_0 \cdot (p_2\colon \underline{|\,o_2\,u_2\,|}\,) \cdot (p_1\colon \underline{|\,o_1\,v_1\,|}\,) \in \mathsf{spec}(O)$.

▶ **Example 5.2.** Consider a standard register object Reg with initial value 0, and operations `write`(v), where $v \in V$ for some set of values V, and `read`. Then, `read` is one-sided non-commutative w.r.t. `write` in $\mathsf{spec}(\mathsf{Reg})$. Indeed, for $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have both $(p_1\colon \underline{|\,\texttt{read}\;\;0\,|}\,) \in \mathsf{spec}(\mathsf{Reg})$ and $(p_2\colon \underline{|\,\texttt{write(1)}\quad|}\,) \cdot (p_1\colon \underline{|\,\texttt{read}\;\;1\,|}\,) \in \mathsf{spec}(\mathsf{Reg})$. The same holds for *max-register* [3], denoted MaxReg, that stores integers with the initial value 0. We note that all pairs of specification histories of Reg and MaxReg with disjoint sets of processes are weakly mergeable.

▶ **Example 5.3.** Consider a monotone counter object MC with initial value 0, and operations `inc` and `read`. Then, `read` is one-sided non-commutative w.r.t. `inc` in $\mathsf{spec}(\mathsf{MC})$ as for $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have $(p_1\colon \underline{|\,\texttt{read}\;\;0\,|}\,) \in \mathsf{spec}(\mathsf{MC})$ and $(p_2\colon \underline{|\,\texttt{inc}\quad|}\,) \cdot (p_1\colon \underline{|\,\texttt{read}\;\;1\,|}\,) \in \mathsf{spec}(\mathsf{MC})$.

The next lemma (proven in [14]) shows that for deterministic objects, the existence of a pair of operations one of which is one-sided non-commutative w.r.t. to the other implies that their corresponding histories are not strongly mergeable:

▶ **Lemma 5.4.** *Let O be a deterministic object and suppose that $o_1 \in \mathsf{ops}(O)$ is one-sided non-commutative w.r.t. $o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Then, there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1, u_2 \in \mathsf{rets}(O)$ such that $(p_1\colon |\underline{o_1 u_1}|)$ and $(p_2\colon |\underline{o_2 u_2}|)$ are not strongly mergeable in $\mathsf{spec}(O)$ after h_0.*

Then, the following theorem (proven in [14]) follows from Thm. 4.8 and properties TSO$^{\mathsf{s}}$ and RA$^{\mathsf{s}}_1$ in Table 1.

▶ **Theorem 5.5.** *Let O be a deterministic object and suppose that $o_1 \in \mathsf{ops}(O)$ is one-sided non-commutative w.r.t. $o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Let $\mathcal{I}$ be a spec-available implementation of O that is consistent under $M \in \{\mathrm{TSO}, \mathrm{RA}\}$. Then, there exist $p_1, p_2 \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(o_1, p_1))$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(o_2, p_2))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) *if $M = \mathrm{TSO}$, then either σ_1 or σ_2 has a fence or a RMW event; and*
(b) *if $M = \mathrm{RA}$, then neither σ_1 nor σ_2 is RW, and one of the following holds: (i) either σ_1 or σ_2 has a RMW event; (ii) either σ_1 or σ_2 is not LTF (i.e., has a fence in the middle); (iii) σ_1 is LF and σ_2 is TF; or (iv) σ_1 is TF and σ_2 is LF.*

Since `read` is one-sided non-commutative w.r.t. `write` in both $\mathsf{spec}(\mathsf{Reg})$ and $\mathsf{spec}(\mathsf{MaxReg})$, their respective implementations under TSO and RA are subject to the constraints given in Thm. 5.5. The same holds for the implementations of the `read` and `inc` operations of MC.

To establish the tightness of these lower bounds, we present linearizable wait-free implementations of Reg and MaxReg that are optimal w.r.t. the above bounds: for TSO, it uses only reads, writes, and a single fence at the end of `write`; and for RA, it uses only reads, writes, and a pair of fences at both the beginning and the end of both `write` and `read`.

A Reg object is trivial to implement under SCM and there are MaxReg implementations under SCM [3] with every operation being RBW. We use these implementations as a basis for implementations under TSO and RA as follows:

TSO. For TSO, we utilize a *fence-insertion strategy*, which derives a linearizable TSO implementation of an object from its SCM counterpart by inserting a fence in-between every consecutive pair of write and read, as well as between a final write of an operation (if it exists) and the operation's response. We give full details, prove correctness, and present more examples of applications of this transformation in [14]. Using this strategy, we obtain a TSO implementation of Reg as follows: `write` first writes to a memory location, and then executes a fence, and `read` reads the same memory location and returns the value read. Likewise, to implement MaxReg under TSO, we add a fence at the end of the `write` implementations of [3], and leave their `read` implementation as is.

RA. We augment the TSO implementations above by adding another fence at the beginning of `write` as well as fences at the beginning and the end of `read`. The pseudocode of the MaxReg algorithm appears in §A and its correctness proof can be found in [14]. Further details of the register implementation and its correctness proof appear in [14]. For conciseness, our MaxReg implementation under RA is derived from a simplified version of the algorithm in [3] (with linear step complexity instead of logarithmic as in [3]).

5.2 Two-Sided Non-Commutative Operations and Mutual Exclusion

We next explore implementability of objects with non-weakly mergeable histories. We apply our framework to generalize the "laws of order" (LOO) results of [6]. The next notion of two-sided non-commutativity is a strengthening of one-sided non-commutativity defined above, and is identical to the notion of strong non-commutativity in [6]:

▶ **Definition 5.6.** Two operations $o_1, o_2 \in \mathsf{ops}(O)$ are *two-sided non-commutative in* $\mathsf{spec}(O)$ if there exist history $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$, and response values $u_1 \neq v_1$ and $u_2 \neq v_2$ in $\mathsf{rets}(O)$ such that: (i) $h_0 \cdot (p_1\colon \lfloor o_1\, u_1 \rfloor) \cdot (p_2\colon \lfloor o_2\, v_2 \rfloor) \in \mathsf{spec}(O)$; and (ii) $h_0 \cdot (p_2\colon \lfloor o_2\, u_2 \rfloor) \cdot (p_1\colon \lfloor o_1\, v_1 \rfloor) \in \mathsf{spec}(O)$.

▶ **Example 5.7.** Revisiting Example 1, in a standard set object Set the operations $\mathtt{remove}(v)$ and $\mathtt{remove}(v)$ (for any v) are strongly non-commutative. Indeed, we can take any $p_1 \neq p_2$, $h_0 = (p\colon \lfloor \mathtt{add}(v) \rfloor)$ (with any $p \in \mathsf{P}$), $u_1 = u_2 = \mathit{true}$, and $v_1 = v_2 = \mathit{false}$, and we have $(p\colon \lfloor \mathtt{add}(v) \rfloor) \cdot (p_1\colon \lfloor \mathtt{remove}(v)\ \ \mathit{true} \rfloor) \cdot (p_2\colon \lfloor \mathtt{remove}(v)\ \ \mathit{false} \rfloor) \in \mathsf{spec}(\mathsf{Set})$ and $(p\colon \lfloor \mathtt{add}(v) \rfloor) \cdot (p_2\colon \lfloor \mathtt{remove}(v)\ \ \mathit{true} \rfloor) \cdot (p_1\colon \lfloor \mathtt{remove}(v)\ \ \mathit{false} \rfloor) \in \mathsf{spec}(\mathsf{Set})$.

▶ **Example 5.8.** Consider a consensus object $\mathsf{Consensus}$ with operations $\mathtt{propose}(0)$ and $\mathtt{propose}(1)$ and return values $\{0, 1\}$. Its specification $\mathsf{spec}(\mathsf{Consensus})$ consists of all histories $h \in \mathsf{ComSeqH}(\mathsf{Consensus})$ such that every $\mathtt{propose}(v)$ invoked in h returns the same value, which is either v or the argument of one of the previously invoked $\mathtt{propose}$ operations. The operations $\mathtt{propose}(0)$ and $\mathtt{propose}(1)$ are two-sided non-commutative. Indeed, for any $p_1 \neq p_2$ and $h_0 = \varepsilon$, we have $(p_1\colon \lfloor \mathtt{propose}(0)\ \ 0 \rfloor) \cdot (p_2\colon \lfloor \mathtt{propose}(1)\ \ 0 \rfloor) \in \mathsf{spec}(\mathsf{Consensus})$ and $(p_2\colon \lfloor \mathtt{propose}(1)\ \ 1 \rfloor) \cdot (p_1\colon \lfloor \mathtt{propose}(0)\ \ 1 \rfloor) \in \mathsf{spec}(\mathsf{Consensus})$.

Examples for other objects with consensus number > 1, such as swap, compare-and-swap, fetch-and-add, queues, stacks, are constructed similarly. In [14] we show that deterministic objects with a pair of two-sided non-commutative operations must have consensus numbers > 1. (We conjecture that the converse also holds.)

We prove in [14] that two-sided non-commutative operations imply non-weakly mergeability:

▶ **Lemma 5.9.** *Let O be a deterministic object and $o_1, o_2 \in \mathsf{ops}(O)$ be two-sided non-commutative operations in $\mathsf{spec}(O)$. Then, there exist $h_0 \in \mathsf{ComSeqH}(O)$, processes $p_1 \neq p_2$ and response values $u_1, u_2 \in \mathsf{rets}(O)$ such that $(p_1\colon \lfloor o_1\, u_1 \rfloor)$ and $(p_2\colon \lfloor o_2\, u_2 \rfloor)$ are not weakly mergeable in $\mathsf{spec}(O)$ after h_0.*

We now apply the merge theorem and the properties $\mathrm{SCM}^{\mathsf{w}}, \mathrm{TSO}^{\mathsf{w}}, \mathrm{RA}^{\mathsf{w}}$ from Table 1 to obtain the lower bounds of LOO under SCM along with impossibilities for TSO and RA (see [14] for the proof):

▶ **Theorem 5.10.** *Let O be a deterministic object with a pair of strongly non-commutative operations $o_1, o_2 \in \mathsf{ops}(O)$ in $\mathsf{spec}(O)$. Let $\mathcal{I}$ be a spec-available implementation of O that is consistent under a memory model M. Then, there exist $p_1 \in \mathsf{P}$ and $\pi_1 \in \mathsf{traces}(\mathcal{I}(o_1, p_1))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$:*

(a) if $M = \mathrm{SCM}$, then σ_1 either has an RMW or is not RBW; and
(b) if $M \in \{\mathrm{TSO}, \mathrm{RA}\}$, then σ_1 either has an RMW or is not LTF (i.e., has a fence in the middle).

Since deterministic objects with a pair of two-sided non-commutative operations have consensus numbers > 1, their wait-free implementations must rely on RMWs [20]. We therefore consider their obstruction-free implementations to obtain upper bounds in the absence of RMWs.[2] In [14], we show that every object in this class has an obstruction-free implementation under TSO with a fence pattern optimal w.r.t. our lower bounds in

[2] It is known that every deterministic object has read/write obstruction-free linearizable implementations in SCM [21].

contention-free executions, i.e., when a process runs solo for long enough to complete its operation. For that, we use a variant of a universal construction from [33] instantiated on top of a TSO-based obstruction-free consensus algorithm. The latter is obtained from shared memory Paxos [18] using our fence-insertion strategy.

Finally, in §B, we derive lower bounds for mutual exclusion. We define an object Lock that can be implemented by means of an entry section of a mutual exclusion algorithm. We show that Lock has a pair of non-weakly mergeable histories, and apply the merge theorem to obtain the lower bounds of LOO for SCM and their counterparts for TSO and RA. A matching upper bound for TSO is obtained by adding a single fence to the entry section of the Bakery algorithm [28].

5.3 Snapshot and Counter

We next explore implementability of snapshot [1] (Snapshot) and (non-monotone) counter (Counter). The former is known to be universal w.r.t. a large class of objects implementable in read/write SCM [5], and the latter has been studied extensively as a building block for randomized consensus (e.g., [4, 2]).

In §C, we revisit and formalize Example 2, and obtain lower bounds on memory events and fence structure that must be exhibited by any consistent and spec-available implementation of snapshot and counter under the memory models we consider. Specifically, we obtain the following for snapshot:

▶ **Theorem 5.11.** *Let $\mathcal{I}$ be a spec-available implementation of* Snapshot *that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(\mathtt{update}(w), p))$ for some $i \in \{1..m\}$ and $w \in W$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\mathtt{scan}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) if $M = \text{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not RBW;
(b) if $M = \text{TSO}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not LTF (i.e., has a fence in the middle); and
(c) if $M = \text{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is not LTF.

We show that a similar lower bounds holds for Counter for $\pi_1 \in \mathsf{traces}(\mathcal{I}(o, p))$ with $o \in \{\mathtt{inc}, \mathtt{dec}\}$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\mathtt{read}, p'))$.

The wait-free linearizable implementations of both Snapshot and Counter under SCM are well-known [1, 5]. The implementation of `update` operation uses collect followed by a write, and the implementation of `scan` uses a sequence of three collects. Counter can be implemented on top of a snapshot using a single call to `update` to implement increment and decrement, and a single call to `scan` to implement read. Both implementations exhibit a single read-after-write across a consecutive pair of update and read, and are therefore optimal w.r.t. to the above lower bounds.

To obtain optimal upper bounds for TSO, the above algorithms are modified using the fence-insertion strategy discussed above that inserts a fence at the end of `update` for snapshot, and at the end of the increment and decrement for counter. Optimal implementations under RA are left for future work.

Max-register vs. snapshot and counter. Our analysis yields the first sharp separation between max-register on the one hand and snapshot and counter on the other in terms of their implementability under RA using only reads, writes, and fences. Specifically, as we show above, max-register can be implemented under RA using fences only at the beginning and the end of `read` and `write`. On the other hand, our lower bounds for snapshot and counter show that this fence placement is insufficient to correctly implement these objects under RA. We are unaware of prior results separating these objects. In particular, all of them are equivalent w.r.t. their power to solve consensus under SCM [20].

6 Related Work

Our mergeability approach is inspired by the work of Kawash [25], who showed that, without fences and RMWs, the critical section problem, as well as certain producer/consumer coordination problems, cannot be solved in a variety of weak memory models that were studied at that time (including TSO). However, while Kawash considers specific tasks, we derive a general result by relating mergeability of traces in the underlying memory model to mergeability at the level of the implemented object histories.[3] Moreover, we also use different mergeability properties to differentiate between weak memory models.

We have already discussed how the results of [6], which were based on a covering technique [13], are obtained by a simpler merge-based argument. The main advantage of our approach is its applicability beyond "strongly non-commutative operations" (see Lem. 5.9), as well as the fact that we directly handle weak memory models, which is only implicit in [6]. In addition, [6] is restricted to deterministic objects and implementations, while our merge theorem avoids these assumptions by stating more precise availability requirements.

Through consensus numbers, Herlihy [20] already showed that for some of the objects we consider here, such as sets, queues and stacks, RMW operations are required in any lock-free linearizable implementation. This result does not have any implication for obstruction-freedom. In fact, for every object, there is a read/write obstruction-free linearizable implementation under SCM (as consensus is universal and read/write obstruction-free solvable [20, 21]). Due to our results, if the object has non-weakly mergeable operations, any such implementation cannot be RBW.

For several objects such as snapshot, counter, max register, work stealing and even relaxations of queues, stacks, and data sketches, there have been proposed lock-free or wait-free read/write linearizable (or variants of it) RBW implementations under SCM [5, 1, 3, 15, 16, 34]. None of these works relate the possibility of such implementations with mergeability properties of the objects implemented. Morrison and Afek [30] show how memory fences can be eliminated on TSO in the implementation of work stealing by assuming that the store buffers are bounded in size, and using this bound in the thief implementation to guarantee that a write is propagated to main memory after a number of subsequent writes. In contrast, the store buffers in the TSO model we study are unbounded, and hence their implementation is not considered linearizable.

In the weak memory literature, some works studied *robustness* of concurrent implementations under TSO and RA, where a robust implementation cannot have any non-SCM behaviors [11, 9, 27, 10, 29]. We note, however, that robustness does not entail that linearizability under SCM is transferred to linearizability under TSO or RA (a register implementation that uses one shared variable is robust, but fences are needed to ensure linearizability under TSO and RA). This is different from the fence-insertion strategy in §5.1 that transfers linearizability under SCM to linearizability under TSO. Other works studied alternatives to linearizability for TSO and RA [12, 35, 32], whereas we take standard linearizability as a correctness criterion.

[3] We also note that Kawash's merge strategy for TSO traces is unnecessarily complex, while our proofs directly exploit the local store buffers for avoiding inter-thread communication.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 James Aspnes. Time-and space-efficient randomized consensus. In *PODC*, page 325–331, New York, NY, USA, 1990. ACM. doi:10.1145/93385.93433.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. doi:10.1145/2108242.2108244.

4 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi:10.1016/0196-6774(90)90021-6.

5 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *SPAA*, pages 340–349. ACM, 1990. doi:10.1145/97444.97701.

6 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498. ACM, 2011. doi:10.1145/1926385.1926442.

7 Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers. Long-lived counters with polylogarithmic amortized step complexity. *Distributed Comput.*, 36(1):29–43, 2023. URL: https://doi.org/10.1007/s00446-022-00439-5, doi:10.1007/S00446-022-00439-5.

8 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1926385.1926394, doi:10.1145/1926385.1926394.

9 Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

10 Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. Reasoning about TSO programs using reduction and abstraction. In *CAV*, pages 336–353, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96142-2_21.

11 Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *ICALP (2)*, pages 428–440, 2011.

12 Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, pages 87–107, Berlin, Heidelberg, 2012. Springer.

13 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993. URL: https://doi.org/10.1006/inco.1993.1065, doi:10.1006/INCO.1993.1065.

14 Armando Castañeda, Gregory Chockler, Brijesh Dongol, and Ori Lahav. What cannot be implemented on weak memory? *CoRR*, abs/2405.16611, 2024. URL: https://doi.org/10.48550/arXiv.2405.16611, arXiv:2405.16611, doi:10.48550/ARXIV.2405.16611.

15 Armando Castañeda and Miguel Piña. Read/write fence-free work-stealing with multiplicity. *J. Parallel Distributed Comput.*, 186:104816, 2024. URL: https://doi.org/10.1016/j.jpdc.2023.104816, doi:10.1016/J.JPDC.2023.104816.

16 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Set-linearizable implementations from read/write operations: Sets, fetch &increment, stacks and queues with multiplicity. *Distributed Comput.*, 36(2):89–106, 2023. doi:10.1007/s00446-022-00440-y.

17 Edsger W. Dijkstra. EWD123: Cooperating Sequential Processes. Technical report, 1965. URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html.

18 Eli Gafni and Leslie Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, feb 2003.

19 Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

20 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

21 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

23 Intel. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: system programming guide, Part*, 2(11):1–64, 2011.

24 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, pages 175–189, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009850.

25 J. Y. Kawash. *Limitation and capabilities of weak memory consistency systems.* PhD thesis, University of Calgary, 2000. doi:10.11575/PRISM/19939.

26 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL*, pages 649–662, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837643, doi:10.1145/2837614.2837643.

27 Ori Lahav and Roy Margalit. Robustness against release/acquire semantics. In *PLDI*, pages 126–141, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3314221.3314604, doi:10.1145/3314221.3314604.

28 Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

29 Roy Margalit and Ori Lahav. Verifying observational robustness against a C11-style memory model. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi:10.1145/3434285.

30 Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *ASPLOS*, pages 413–426. ACM, 2014. doi:10.1145/2541940.2541987.

31 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, Berlin, Heidelberg, 2009. Springer. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27, doi:10.1007/978-3-642-03359-9_27.

32 Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290381.

33 Michel Raynal. Distributed universal construcitons: a guided tour. *Bulleting of EATCS: Distributed Computing Column*, (121), 2011.

34 Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. In *DISC*, volume 179 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.2.

35 Abhishek Kr Singh and Ori Lahav. An operational approach to library abstraction under relaxed memory concurrency. *Proc. ACM Program. Lang.*, 7(POPL):1542–1572, 2023. doi:10.1145/3571246.

36 SPARC International Inc. *The SPARC architecture manual (version 9).* Prentice-Hall, 1994.

A Fence-optimal Max Register Under RA

The pseudocode of a linearizable wait-free implementation of MaxReg under RA is given in Algorithm 1. The function *collect*(M) reads one by one, in an arbitrary order, the entries of M, and returns an array with the read values. The algorithm is fence-optimal. It uses one fence at the beginning and one fence at the end of every operation, thus matching the lower bounds of Thm. 5.5. The correctness proof appears in [14].

Algorithm 1 MaxReg implementation in RA. Algorithm for proces p_i.

```
Shared variables:
  $int[n]\ \ M = [0, \dots, 0]$

procedure READ( )
    fence()
    $m[] = collect(M)$
    fence()
    return $max(m[])$

procedure WRITE($v$)
    fence()
    $m[] = collect(M)$
    if $max(m[]) < v$ then
        $M[i] = v$
    fence()
    return ack
```

B Mutual Exclusion

We use the merge theorem (Thm. 4.8) for the case of non-weakly mergeable histories and the mergeability results for the memory models to establish minimum synchronization requirements for mutual exclusion. Our result for SCM reproves the corresponding lower bound of [6].

Consider a (non-standard) lock object Lock with ops(Lock) $\triangleq$ {`acquire`} and rets(Lock) $\triangleq$ {`ack`}. Its specification is given by

$$\mathsf{spec}(\mathsf{Lock}) \triangleq \{\varepsilon\} \cup \{(p\colon \overset{\texttt{acquire}}{\vdash\!\!\!\longrightarrow\!\!\dashv}) \mid p \in \mathsf{P}\}.$$

The histories $(p\colon \overset{\texttt{acquire}}{\vdash\!\!\!\longrightarrow\!\!\dashv})$ and $(p'\colon \overset{\texttt{acquire}}{\vdash\!\!\!\longrightarrow\!\!\dashv})$ where $p \neq p'$ are not weakly mergeable. Thus, by the merge theorem and properties SCM$^{\mathsf{w}}$, TSO$^{\mathsf{w}}$, and RA$^{\mathsf{w}}$ in Table 1, we have:

▶ **Theorem B.1.** *Let $\mathcal{I}$ be a spec-available implementation of* Lock *that is consistent under a memory model M. Then, there exist $p \in \mathsf{P}$ and $\pi \in \mathsf{traces}(\mathcal{I}(\texttt{acquire}, p))$ such that the following hold for $\sigma = \pi|_{\mathsf{M}}$:*

(a) if $M =$ SCM, then σ either has an RMW event or is not RBW; and
(b) if $M \in \{TSO, \mathrm{RA}\}$, then σ either has an RMW event or a fence.

The proof of this theorem is identical to that of Thm. 5.10, which appears in [14]. Since the implementation of the entry section of a mutual exclusion algorithm can be used to implement `acquire`, we obtain that entry section of a solo-terminating mutual exclusion algorithm on SCM has to use a RAW pattern or an RMW; and on TSO or RA, it must use an RMW or a fence.

There exist many algorithms implementing starvation-free mutual exclusion under SCM, which use the RAW pattern to implement the entry section. As before, their counterparts under TSO can be obtained by adding a fence between every pair of consecutive write and read (§5.1). For example, the transformation of Bakery algorithm [28] only requires a single fence to separate a write-only block at the beginning of the entry section from the read-only block right afterwards. The resulting implementation is therefore tight. Mutual exclusion under RA with an RMW or a fence has several verified implementations [27].

C Lower and Upper Bounds for Snapshot and Counter

Lower bounds for snapshot. Consider a (single-writer) snapshot object Snapshot storing a vector of a length $|\mathsf{P}|$ over a set of values W (also represented as function in $\mathsf{P} \to W$) with the initial vector of $\langle \bot, \dots, \bot \rangle$. The operations are $\{\texttt{update}(w) \mid w \in V\} \cup \{\texttt{scan}\}$, and its return values are $\{\texttt{ack}\} \cup (\mathsf{P} \to W)$. The specification $\mathsf{spec}(\mathsf{Snapshot})$ consists of all complete sequential histories where each `scan` event returns v such that $v(p)$ is the value written by the last preceding `update` by process p, or $\bot$ if no such `update` exists.

▶ **Proposition C.1.** *Let $w, w' \in W$, $p_1, p_2, p_3 \in \mathsf{P}$, and $h_1, h_2 \in \mathsf{ComH}(\mathsf{Snapshot})$, such that $w \neq w'$, $i \neq j$, $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$, and the following hold:*

- $h_1 \sqsubseteq (p_1\colon \left|\underline{\texttt{update}(w)\qquad}\right|) \cdot (p_3\colon \left|\underline{\texttt{scan}\;\; v}\right|)$, *where $v = \lambda p.$ **if** $p = p_1$ **then** w **else** $\bot$; and*
- $h_2 \sqsubseteq (p_2\colon \left|\underline{\texttt{update}(w')\qquad}\right|) \cdot (p_2\colon \left|\underline{\texttt{scan}\;\; v'}\right|)$, *where $v' = \lambda p.$ **if** $p = p_2$ **then** w' **else** $\bot$.*

Then, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after ε.

Next, we use the merge theorem (instantiated for the case of non-weakly mergeable histories) together with Prop. C.1 and the mergeability results SCM$^{\mathsf{w}}$, TSO$^{\mathsf{w}}$, and RA$^{\mathsf{w}}$ from Table 1 to establish lower bounds on implementability of snapshot.

▶ **Theorem 5.11.** *Let $\mathcal{I}$ be a spec-available implementation of Snapshot that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(\texttt{update}(w), p))$ for some $i \in \{1..m\}$ and $w \in W$, and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{scan}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$:*

(a) if $M = \mathrm{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not RBW;
(b) if $M = \mathrm{TSO}$, then $\sigma_1 \cdot \sigma_2$ either has an RMW or is not LTF (i.e., has a fence in the middle); and
(c) if $M = \mathrm{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is not LTF.

Proof. First, consider the case of $M \in \{\mathrm{SCM}, \mathrm{TSO}\}$. Let p_1, p_2 be distinct processes and consider the histories

$$h_1 = (p_1\colon \left|\underline{\texttt{update}(1)\qquad}\right|) \cdot (p_1\colon \left|\underline{\texttt{scan}\;\; v}\right|) \quad \text{and} \quad h_2 = (p_2\colon \left|\underline{\texttt{update}(1)\qquad}\right|) \cdot (p_2\colon \left|\underline{\texttt{scan}\;\; v'}\right|),$$

where $v = \lambda p.\,\textbf{if } p = p_1 \textbf{ then } w \textbf{ else } \bot$ and $v' = \lambda p.\,\textbf{if } p = p_2 \textbf{ then } w' \textbf{ else } \bot$. Then, by Prop. C.1, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after $h_0 = \varepsilon$. Clearly, we also have $h_1, h_2 \in \mathsf{spec}(\mathsf{Snapshot})$, and since $\mathcal{I}$ is spec-available, it is available w.r.t. both h_1 and h_2.

Thus, by Thm. 4.8, there exist $\pi'_1, \pi'_2 \in \mathsf{traces}(\mathcal{I})$ such that $h_1 = \pi'_1|_{\mathsf{Snapshot}}$ and $h_2 = \pi'_2|_{\mathsf{Snapshot}}$, and $\sigma'_1 = \pi'_1|_{\mathsf{M}}$ and $\sigma'_2 = \pi'_2|_{\mathsf{M}}$ are not weakly mergeable in M. Observe that $\pi'_1 = \pi_1 \cdot \pi_2$ where $\pi_1 \in \mathsf{traces}(\mathcal{I}(\texttt{update}(1), p_1))$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{scan}, p_1))$. Let $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}$. Then, $\sigma'_1 = \sigma_1 \cdot \sigma_2$. Thus, the required follows properties SCM$^{\mathsf{w}}$ and TSO$^{\mathsf{w}}$ in Table 1.

Next, we consider the case of $M = \mathrm{RA}$. Let $p_1, p_2, p_3 \in \mathsf{P}$ be distinct processes, and consider the histories:

$h_1 = \langle p_1{:}\texttt{inv}(\texttt{update}(1)), p_3{:}\texttt{inv}(\texttt{scan}), p_1{:}\texttt{res}(\texttt{ack}), p_3{:}\texttt{res}(v)\rangle$ and
$h_2 = (p_2\colon \left|\underline{\texttt{update}(2)\qquad}\right|) \cdot (p_2\colon \left|\underline{\texttt{scan}\;\; v'}\right|)$,

where $v = \lambda p.\, \textbf{if}\ p = p_1\ \textbf{then}\ w\ \textbf{else}\ \bot$ and $v' = \lambda p.\, \textbf{if}\ p = p_2\ \textbf{then}\ w'\ \textbf{else}\ \bot$. Then, by Prop. C.1, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Snapshot})$ after $h_0 = \varepsilon$. Note that $h_2 \in \mathsf{spec}(\mathsf{Snapshot})$. Consider the following sequential history of $\mathsf{spec}(\mathsf{Snapshot})$:

$$h^1_{\mathsf{seq}} = (p_1\colon \vdash\!\underline{\texttt{update}(1)\qquad}\!\dashv) \cdot (p_3\colon \vdash\!\underline{\texttt{scan}\quad v}\!\dashv)$$

By assumption, $\mathcal{I}$ is available w.r.t. h_2 and h^1_{seq}.

Then, by Thm. 4.8, there exist π_1 and π_2 such that:

- $\pi_i|_{\mathsf{Snapshot}} = h_i$ for $i \in \{1, 2\}$.
- $\pi_i \in \mathsf{traces}(\mathcal{I})$ for $i \in \{1, 2\}$.
- $\pi_i|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ for $i \in \{1, 2\}$.
- $\mathsf{proc\text{-}set}(\pi_i) = \mathsf{proc\text{-}set}(h_i)$ for $i \in \{1, 2\}$.
- For every $\pi'_1 \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_1)$ such that $\pi'_1|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ and $\pi_1|_{\mathsf{Snapshot}} = h_1$ and $\pi'_2 \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_2)$ such that $\pi'_2|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ and $\pi_2|_{\mathsf{Snapshot}} = h_2$, $\pi'_1|_{\mathsf{M}}$ and $\pi'_2|_{\mathsf{M}}$ are not weakly mergeable in RA.

Let π'_1 be the sequence obtained from π_1 by:

- moving $\langle p_1, \texttt{inv}(\texttt{update}(1))\rangle$, $\langle p_3, \texttt{inv}(\texttt{scan})\rangle$ and all leading fences to the beginning of the sequence; and
- moving $\langle p_1, \texttt{res}(\texttt{ack})\rangle$, $\langle p_3, \texttt{res}(v)\rangle$ and all trailing fences to the end of the sequence.

In this rearrangement we keep the internal order among moved events as it is in π_1. Then, $\pi'_1 \in \mathsf{reorder}_{\mathsf{sproc}}(\pi_1)$ and $\pi_1|_{\mathsf{Snapshot}} = h_1$. Moreover, among memory events, we only moved fences which are no-ops under SCM. Thus, $\pi_1|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$ implies $\pi'_1|_{\mathsf{M}} \in \mathsf{traces}(\mathrm{SCM})$. By taking $\pi'_2 = \pi_2$, we obtain that $\pi'_1|_{\mathsf{M}}$ and $\pi'_2|_{\mathsf{M}}$ are not weakly mergeable in RA. Finally, by property RA^{W} in Table 1, we obtain that $\pi'_1|_{\mathsf{M}}$ is not LTF, or it contains some RMW event. This implies that either $\pi'_1|_{\mathsf{M}}|_{p_1}$ or $\pi'_1|_{\mathsf{M}}|_{p_3}$ are not LTF or contain some RMW event. ◀

Lower bounds for counter. Consider a counter object $\mathsf{Counter}$ with the initial value of 0, and the increment (`inc`), decrement (`dec`), and read (`read`) operations. Then, we have:

▶ **Proposition C.2.** *Let $p_1, p_2, p_3 \in \mathsf{P}$ and $h_1, h_2 \in \mathsf{ComH}(\mathsf{Counter})$ such that $\mathsf{proc\text{-}set}(h_1) \cap \mathsf{proc\text{-}set}(h_2) = \emptyset$ and the following hold:*

- $h_1 \sqsubseteq (p_1\colon \vdash\!\underline{\texttt{inc}\qquad\quad}\!\dashv) \cdot (p_3\colon \vdash\!\underline{\texttt{read}\qquad 1}\!\dashv)$*; and*
- $h_2 \sqsubseteq (p_2\colon \vdash\!\underline{\texttt{dec}\qquad\quad}\!\dashv) \cdot (p_2\colon \vdash\!\underline{\texttt{read}\quad -1}\!\dashv)$

Then, h_1 and h_2 are not weakly mergeable in $\mathsf{spec}(\mathsf{Counter})$ after ε.

Then, the following can be obtained by instantiating the proof of Thm. 5.11 to use Prop. C.2.

▶ **Theorem C.3.** *Let $\mathcal{I}$ be a spec-available implementation of $\mathsf{Counter}$ that is consistent under a memory model M. Then, there exist $p, p' \in \mathsf{P}$, $\pi_1 \in \mathsf{traces}(\mathcal{I}(o, p))$ where $o \in \{\texttt{inc}, \texttt{dec}\}$ and $\pi_2 \in \mathsf{traces}(\mathcal{I}(\texttt{read}, p'))$ such that the following hold for $\sigma_1 = \pi_1|_{\mathsf{M}}$ and $\sigma_2 = \pi_2|_{\mathsf{M}}\}$:*

(a) if $M = \mathrm{SCM}$, then $\sigma_1 \cdot \sigma_2$ either has a RMW event or is not RBW; and
(b) if $M = \mathrm{TSO}$, then $\sigma_1 \cdot \sigma_2$ has either a RMW event or is non-LTF (i.e., has a fence in the middle).
(c) if $M = \mathrm{RA}$, then (i) either σ_1 or σ_2 has an RMW, or (ii) either σ_1 or σ_2 is non-LTF.

Upper bounds. There is a wait-free snapshot implementation [1] that is linearizable under SCM, in which `scan` performs a sequence of reads, and `update` performs a sequence of reads followed by a write, Using the fence insertion strategy in §5.1, a linearizable wait-free implementation of snapshot under TSO is obtained from such implementations by adding a single fence at the end of `update`.

▶ **Theorem C.4.** *For $M \in \{\mathrm{SCM}, \mathrm{TSO}\}$, there exists a linearizable wait-free implementation of snapshot* $\mathsf{Snapshot}_M$ *under M such that:*

(a) $\mathsf{Snapshot}_{\mathrm{SCM}}$ *uses only a sequence of reads followed by a write to implement* `update` *and only reads to implement* `scan`*, and*
(b) $\mathsf{Snapshot}_{\mathrm{TSO}}$ *uses only a sequence of reads followed by a write and a fence at the end to implement* `update`*, and only reads to implement* `scan`*.*

Observe that Thm. C.4 (a) implies that any pair of consecutive `update` and `scan` is RBW, which is tight in the lower bound of Thm. 5.11 (a). Likewise, Thm. C.4 (b) is tight in the lower bound of Thm. 5.11 (b), which stipulates that a fence is needed somewhere within consecutively executed `update` and `scan`.

A linearizable wait-free counter can be implemented on top of a snapshot instance as follows: each process p_i stores its contribution to the current counter value in a local variable c_i initialized to 0. To increment (resp., decrement) the counter, p_i increments (resp., decrements) c_i, and then invokes `update`(c_i) to share its contribution with other processes. To read the counter, a process calls `scan` and returns the sum of the values stored in the returned vector.

▶ **Theorem C.5.** *For $M \in \{\mathrm{SCM}, \mathrm{TSO}\}$, there exists a linearizable wait-free implementation of counter* $\mathsf{Counter}_M$ *under M such that:*

(a) $\mathsf{Counter}_{\mathrm{SCM}}$ *uses only writes to implement* `inc` *and* `dec` *and only reads to implement* `read`*, and*
(b) $\mathsf{Counter}_{\mathrm{TSO}}$ *uses only writes and a fence at the end to implement* `inc` *and* `dec`*, and only reads to implement* `read`*.*

As in the case of snapshot, the synchronization strategy stipulated by this result is optimal w.r.t. the lower bound of Thm. C.3. The optimal implementations of snapshot and counter under RA are left for future work.