Defining Correctness Conditions for Concurrent Objects in Multicore Architectures

Brijesh Dongol[1], John Derrick[2], Lindsay Groves[3], and Graeme Smith[4]

1 **Department of Computer Science, Brunel University, UK**
`Brijesh.Dongol@brunel.ac.uk`

2 **Department of Computer Science, University of Sheffield, UK**
`J.Derrick@dcs.shef.ac.uk`

3 **School of Engineering and Computer Science, Victoria University of Wellington, New Zealand**
`lindsay@ecs.vuw.ac.nz`

4 **School of Information Technology and Electrical Engineering, The University of Queensland, Australia**
`smith@itee.uq.edu.au`

Abstract

Correctness of concurrent objects is defined in terms of conditions that determine allowable relationships between histories of a concurrent object and those of the corresponding sequential object. Numerous correctness conditions have been proposed over the years, and more have been proposed recently as the algorithms implementing concurrent objects have been adapted to cope with multicore processors with relaxed memory architectures.

We present a formal framework for defining correctness conditions for multicore architectures, covering both standard conditions for totally ordered memory and newer conditions for relaxed memory, which allows them to be expressed in uniform manner, simplifying comparison. Our framework distinguishes between order and commitment properties, which in turn enables a hierarchy of correctness conditions to be established. We consider the Total Store Order (TSO) memory model in detail, formalise known conditions for TSO using our framework, and develop sequentially consistent variations of these. We present a work-stealing deque for TSO memory that is not linearizable, but is correct with respect to these new conditions. Using our framework, we identify a new non-blocking compositional condition, fence consistency, which lies between known conditions for TSO, and aims to capture the intention of a programmer-specified fence.

1998 ACM Subject Classification D.1.3 Concurrent Programming, D.2.4 Software/Program Verification, F.1.2 Concurrent Programming, F.3.1 Specifying and Verifying and Reasoning about Programs, H.2.4 Systems

Keywords and phrases Concurrent objects, correctness, relaxed memory, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.470

1 Introduction

This paper studies correctness conditions for *concurrent objects*, i.e., objects consisting of operations acting on shared data that may be executed concurrently by multiple processes. Because the operation calls of concurrent objects may overlap (as opposed to occurring one after another), their correctness is judged using a *correctness condition*, which is a relation on the behaviours of the concurrent object and its sequential specification object, i.e., a

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 470–494
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

correctness condition provides an answer to the question: In what sense does a concurrent object implement its sequential specification?

Correctness conditions for concurrent objects has been the subject of study for nearly three decades and numerous conditions have been proposed. Shavit makes the case that different correctness conditions are needed in different circumstances [31]; weaker conditions provide greater scope for optimisation, but fewer behavioural guarantees. One cannot however, continually weaken correctness conditions in search of greater performance. Programmers require strong correctness conditions that ensure an abstract specification object (whose behaviours are understandable) can be safely substituted by a concurrent object (which provides better performance) within the programs that use these objects. The existence of these two opposing goals has meant that the number of accepted correctness conditions has actually increased over time (e.g., [31, 25]), instead of being consolidated into a unified correctness notion.

Most correctness conditions, including linearizability [23], have been developed under the assumption that hardware ensures *totally ordered memory*,[1] where reads and writes within a process are guaranteed to be executed in program order. Due to their use of local buffers, modern multicore architectures are not totally ordered, and only provide relaxed memory guarantees [1, 33], meaning memory instructions may be executed in a different order to that specified by the program. Such reorderings can be avoided by introducing **`fence`** instructions in the program code, however, because **`fence`** instructions hamper performance, programmers try to limit their usage. This however, causes a direct tension between correctness and optimisation possibilities. For example, it has been shown that to ensure linearizability of many data structures under relaxed memory, there are "laws of order" that force **`fence`** instructions to be used, and hence, linearizability itself has become a bottleneck to efficiency [3]. In the face of this result, we once again look to define suitable correctness conditions weaker than linearizability [12, 32].

Although numerous correctness conditions exist (and more are proposed each year), a unified framework within which different correctness conditions can be defined and formally compared has thus far not been developed. With the advent of correctness conditions for relaxed memory architectures, it is becoming difficult to judge the comparative strengths of different conditions. This paper presents a systematic study of correctness conditions for concurrent objects executed in multicore architectures. We do not aim to characterise the memory models themselves (for such a study see [2]), but rather characterise properties of a concurrent object executing in some memory architecture.

We make the following contributions.

1. We develop a framework that enables one to *systematically develop and reason about correctness conditions for concurrent objects*. For each property we distinguish between its *order conditions*, which define allowable orderings of concrete operations, and *commit conditions*, which provide guarantees about the operations whose effect must have taken place. This distinction is the first, to our knowledge, providing a separation of concerns when defining correctness.
2. Within this framework, we formalise well-known correctness conditions for totally ordered memory, providing insight into the relationships between them.
3. To cope with relaxed memory, we define *partial* commitment conditions where the effects

[1] Architectures with totally ordered memory are also referred to as *sequentially consistent architectures* [26]. In this paper, we use sequential consistency to refer to a property on the histories of a concurrent object as done in [4]. Sequential consistency is formalised in Definition 10.

of some completed operation calls are delayed beyond their returns due to pending writes in the buffers of the calling processes.

4. We study a specific weak memory model, TSO, and formalise known correctness conditions for it (*weak flush consistency* [12] and *weak ξ-quiescent consistency* [32]) that are weaker than linearizability, as they allow more reorderings and only ensure partial commitment.
5. Using our framework, we develop a new condition, *fence consistency*, a non-blocking compositional condition that lies between the existing conditions for TSO memory.
6. We show that the Chase-Lev work-stealing deque [7] where the `put` operation returns without fencing is not linearizable under TSO memory, but does satisfy a weaker condition, *flush consistency*, which is a sequentially consistent version of the condition defined in [12]. This condition is strictly weaker than linearizability and stronger than *ξ-quiescent consistency* (which is the sequentially consistency version of the condition in [32]).
7. We prove a hierarchy for the correctness conditions in this paper based on order and commitment properties.

2 Background

This section provides the background for the rest of the paper; we introduce the Chase-Lev work-stealing dequeue (as defined by [7]), which serves as a running example for the rest of this paper. We also informally introduce notions of correctness for concurrent objects for totally ordered memory, and the Total Store Order memory model.

2.1 Work-Stealing Deque

Work-stealing *double ended queues* (abbreviated to *deques*) are often used for load balancing in multiprocessor systems. Each worker process has a deque, which it uses to record tasks to be performed. Thus, a worker executes `put` and `take` operations that, respectively, add tasks to and remove tasks from its deque. Load balancing is achieved by allowing other, so-called "thief" processes, whose own deques are empty, to execute `steal` operations that remove elements from the deque. To avoid contention between the worker and thief processes, `put` and `take` operate at different ends of the deque from `steal` operations – a worker adds and removes tasks at the tail, whereas thieves steal tasks from the head. Because the worker and thieves operate at different ends of the deqeue, contention between the worker and thieves occurs when the deque has one element. Resolving these cases is in general difficult [13]. Fig. 1 presents a simplified version of the Chase-Lev work-stealing deque. The shared state consists of an array, `items`, of tasks (represented as integers) and variables `Head` and `Tail`, which mark the part of the array containing the elements of the deque. The other variables are local to the operations in which they occur.

2.2 Correctness Conditions

Correctness of a concurrent object is judged with respect to an abstract sequential specification [22]. The abstract specification of the deque object implementation in Fig. 1 is given in Fig. 2, consisting of a deque variable dq, represented as a sequence of tasks, and atomic operations `put`, `take` and `steal`; *task* is a local variable within the `take` and `steal` operations.

An object cannot execute by itself; rather it is the *clients* of an object that execute its operations. Correctness defines a relationship between *histories* of the concrete and abstract systems, which record the interactions between the client and an object via the object's external interface. Typically, a history records invocation and return events of operation

```
    void put(int task) {
P1    tl := Tail;
P2    items[tl] := task;
P3    Tail := tl + 1; }

    int steal() {
S1    while true {
S2      hd := Head;
S3      if hd ≥ Tail
S4        return emp;
S5      task := items[hd];
S6      if cas(Head,hd,hd+1)
S7        return task; } }
```

```
     int take() {
T1     tl := Tail - 1;
T2     Tail := tl;
T3     hd := Head;
T4     if hd > tl {
T5        Tail := hd;
T6        return emp; }
T7     task := items[tl];
T8     if tl > hd
T9       return task;
T10    Tail := hd + 1;
T11    if cas(Head,hd,hd+1)
T12      return task;
T13    else return emp; } }
```

Figure 1 Chase-Lev work-stealing deque.

```
void put(int task) {
  atomic {
    dq := dq ⌒⟨ task ⟩
  } }
```

```
int steal {
  atomic {
    if dq = ⟨⟩ then
      return emp
    else
      task := head(dq) ;
      dq := tail(dq) ;
      return task } }
```

```
int take {
  atomic {
    if dq = ⟨⟩ then
      return emp
    else
      task := last(dq) ;
      dq := init(dq) ;
      return task } }
```

Figure 2 Abstract work-stealing deque.

calls. Concurrent histories may consist of both overlapping and non-overlapping operation calls, inducing a partial order on events. Correctness conditions define how, if at all, this order is maintained in the corresponding abstract history. There are several well-known existing correctness conditions for totally ordered memory [22].

- *Sequential consistency* is a simple condition requiring the order of operation calls in a concrete history for a single process to be preserved. Operation calls performed by different processes may be reordered in the abstract history even if the operation calls do not overlap in the concrete history.
- *Linearizability* strengthens sequential consistency by requiring the order of non-overlapping operations to be preserved. Operation calls that overlap in the concrete history may be reordered when mapping to an abstract history.
- *Quiescent consistency* is weaker than linearizability, but is incomparable to sequential consistency. A concurrent object is said to be quiescent at some point m in its history if none of its operations are executing at m. Quiescent consistency requires the order of operation calls separated by a quiescent point to be preserved. Operation calls that are not separated by a quiescent point may be reordered, including operations performed by the same process.

It has already been shown that the Chase-Lev deque from Fig. 1 is linearizable [7]. Since linearizability implies both sequential and quiescent consistency, the Chase-Lev deque is also both sequentially and quiescently consistent.

2.3 Total Store Order (TSO) Memory

Modern multi-core architectures use local buffers to allow more efficient use of shared memory (see Fig. 3). For optimisation purposes, many architectures only provide relaxed

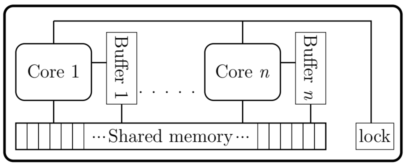

Figure 3 TSO architecture.

```
word x=0, y=0;
Process p {          Process q {
p1: x := 1 ;         q1: y := 1 ;
p2:  r1 := y }       q2: r2 := x }
```

Figure 4 TSO example.

memory guarantees. We consider Total Store Order (TSO) memory as implemented by x86 processors. A general definition of TSO is given in [1], an operational semantics in [30], and an interval-based semantics in [14].

Here, a write by a processor core is not immediately committed to shared memory. Instead it is enqueued as a pending write in the local buffer and only becomes visible to other processes after it is *flushed*, which commits the pending write in the buffer to shared memory. Hence, there is a discrepancy between the time at which a write is executed and the time at which the effect of the write becomes visible to other processes. In TSO, pending writes are flushed in a FIFO order. In addition, using a method known as *Intra-Process Forwarding* [1], when reading a memory location, a processor core fetches the value of the last pending write from its local buffer if available and from shared memory otherwise. Due to pending writes and intra-process forwarding, from an external perspective, read and write instructions within a process appear to be reordered [1], i.e., total memory order is not maintained.

▶ **Example 1.** Consider the program in Fig. 4, where processes `p` and `q` modify shared variables `x` and `y`, both of which are initialised to 0. Under totally ordered memory, when the program terminates, at least one of `r1` or `r2` would have the value 1. However, under TSO memory, it is possible for the program to terminate so that both `r1` and `r2` read the original values of `x` and `y`, i.e., both `r1` and `r2` are 0 at termination. One such execution sequence is ⟨ `p1`, `p2`, `q1`, `q2`, `flush(p)`, `flush(p)`, `flush(q)`, `flush(q)` ⟩, where `p1` denotes execution of the statement at line `p1` (similarly `p2`, etc.), and `flush(p)` denotes execution of a hardware-controlled flush event for process p (similarly `flush(q)`). The write to `x` at `p1` is not seen by process `q` until `p`'s buffer is flushed, and symmetrically for the write to `y` at `q1`. Hence, it is possible for `q` to read a value 0 for `x` at `q2` even though `q2` is executed after `p1`.

To avoid instruction reordering, a core may acquire a global *lock* (depicted in Fig. 3), which prevents all other cores from accessing shared memory. This *lock* is used to implement (coarse-grained) atomic operations such as **`cas`** [30]. In particular, a **`cas`** operation locks the buffer, performs the compare and swap, fully flushes the buffer, then releases the lock.

▶ **Example 2.** Suppose we wish to establish the postcondition that either `r1` or `r2` has value 1 for the program in Fig. 4. The only possibility is to introduce **`fence`** instructions between `p1` and `p2`, and between `q1` and `q2` in Fig. 4. Note that both **`fence`** instructions are necessary, otherwise, `r1` = `r2` = 0 remains a possible outcome of the program.

3 Defining Correctness for Concurrent Objects

The correctness conditions described in Section 2.2 are all defined in terms of an abstract sequential history which is related in a certain way to a given execution of the concurrent object in question. This relationship can be defined more precisely in terms of a mapping function that maps elements in the concrete history to those of the corresponding abstract history. Mapping functions are inspired by the encoding of linearizability in [10], which has led to a *complete* simulation-based method for proving linearizability [29]. Our conditions for relaxed memory are also amenable to integration with such proof methods.

3.1 A Framework for Specifying Correctness

As already discussed, correctness conditions are defined in terms of histories of the abstract (sequential) and concrete (concurrent) systems. In order to make these comparable, while capturing the relevant information about concurrent executions, histories record just the invocation and return of each operation call. In a concurrent history, operation calls may be interleaved with those of other processes, so the invocation and return of a given call may be separated by any number of invocations and returns of other processes, while in a sequential history, the invocation of an operation call is immediately followed by its corresponding return.

For correctness conditions under totally ordered memory it turns out that invocation and response events are all that must be recorded. However, relaxed memory architectures often require additional events such as buffer flushes to be recorded [5, 17, 6]. Thus, assuming that process identifiers have type P, operations names have type I, and the inputs and outputs of operations have type V, we define events and histories as follows: [2]

$$Event \;\widehat{=}\; inv\langle\!\langle P \times I \times V\rangle\!\rangle \mid ret\langle\!\langle P \times I \times V\rangle\!\rangle \qquad\qquad History \;\widehat{=}\; \mathrm{seq}\, Event_C$$

where $Event_C \supseteq Event$ is the set of concrete events; the set $Event_C$ will be specialised in later sections. We say that two events e_1 and e_2 are *matching* if they form an invocation/return pair for the same operation performed by the same process:

$$matching(e_1, e_2) \;\widehat{=}\; inv?(e_1) \wedge ret?(e_2) \wedge e_1.pr = e_2.pr \wedge e_1.i = e_2.i$$

where $inv?$ and $ret?$ are true for invocation and return events, respectively, and $e.pr$ and $e.i$ denote the process and operation corresponding to an event e, respectively; similarly, $e.v$ denotes e's input/outputs. Indices m and n form a *matching pair* in a history h if they identify a pair of matching events and there is no invocation or return performed by the same process between them:

$$\begin{aligned} mp(m, n, h) \;\widehat{=}\;& matching(h(m), h(n)) \wedge \\ & \forall\, k : \mathrm{dom}\, h \bullet m < k < n \wedge h(k).pr = h(m).pr \Rightarrow h(k) \notin Event \end{aligned}$$

Note that in the case of totally ordered memory $Event_C = Event$, i.e., all elements of a history h are in $Event$. Hence, the second conjunct of $mp(m, n, h)$ simplifies to $\forall\, k : \mathrm{dom}\, h \bullet m < k < n \Rightarrow h(k).pr \neq h(m).pr$. However, this is not the case for the histories in Section 4, and there, the consequent of the second conjunct does not trivially reduce to false.

An index m is a *pending invocation* in history h if $h(m)$ is an invocation that is not followed by a matching return in h:

$$pi(m, h) \;\widehat{=}\; inv?(h(m)) \wedge \forall\, k : \mathrm{dom}\, h \bullet m < k \Rightarrow \neg matching(h(m), h(k))$$

A history is *sequential* if it is either empty or an alternating sequence of matching invocations and returns starting with an invocation:

$$\begin{aligned} sequential(h) \;\widehat{=}\;& h = \langle\,\rangle \vee (inv?(h(0)) \wedge \forall\, k : \mathrm{dom}\, h \bullet inv?(h(k)) \wedge k+1 \in \mathrm{dom}\, h \Rightarrow \\ & \qquad matching(h(k), h(k+1)) \end{aligned}$$

[2] We generally use Z mathematical notation [34]. This definition says that any element of *Event* is of the form $inv(p, i, v)$ or $ret(p, i, v)$, where $p \in P$, $i \in I$ and $v \in V$. We also write $inv(p, op)$ for invocations with no inputs, and $ret(p, op)$ for returns with no outputs. In this paper, we assume that sequences are indexed from 0 onwards.

As in [23], we assume each process calls at most one operation at a time. We say a history h is *well formed* if each $h_{|p}$ is sequential, where $h_{|p}$ denotes a history h restricted to all events by process p; for the rest of this paper we assume that all histories are well-formed. A history h is *legal* if each return is preceded by some matching invocation:

$$legal(h) \;\widehat{=}\; \forall\, n : \mathrm{dom}\, h \bullet ret?(h(n)) \Rightarrow \exists\, m : \mathrm{dom}\, h \bullet m < n \wedge mp(m, n, h)$$

A correctness condition between a concurrent history h and a sequential history hs is defined in terms of a *mapping function*, $f : \mathbb{N} \nrightarrow \mathbb{N}$, which is an injective partial function from indices of h to indices of hs. Injectivity ensures that each element of h occurs at most once in hs, while partiality provides the flexibility needed to represent delayed operation calls under relaxed memory architectures (where some completed operation calls may not appear in the abstract history).

▶ **Example 3.** Consider concurrent history h and sequential history hs below:

$$\begin{aligned} h &\;\widehat{=}\; \langle inv(q_1, \mathtt{steal}), inv(w, \mathtt{put}, 1), ret(q_1, \mathtt{steal}, \mathtt{emp}), ret(w, \mathtt{put})\rangle \\ hs &\;\widehat{=}\; \langle inv(q_1, \mathtt{steal}), ret(q_1, \mathtt{steal}, \mathtt{emp})\rangle \end{aligned}$$

The mapping function from h to hs is $\{0 \mapsto 0, 2 \mapsto 1\}$. In this example, we assume that due to TSO the `put` operation has not yet taken effect.

In our framework, one only needs to define predicates on h and f; the corresponding sequential history is $hs = \{f(k) \mapsto h(k) \mid k \in \mathrm{dom}\, f\}$.

▶ **Example 4.** Sequence $h = \langle a, b, c, d\rangle$ is the set of mappings $\{0 \mapsto a, 1 \mapsto b, 2 \mapsto c, 3 \mapsto d\}$. Hence, if $f = \{2 \mapsto 0, 0 \mapsto 1, 1 \mapsto 2, 3 \mapsto 3\}$ then $hs = \{0 \mapsto c, 1 \mapsto a, 2 \mapsto b, 3 \mapsto d\} = \langle c, a, b, d\rangle$.

We distinguish between two types of predicates on h and f: *order conditions*, which describe the allowable orders of events when mapping h to hs (via f), and *commitment conditions*, which describe the events of h that must occur in hs (due to occurrence of their corresponding index in f). We write $P(\overline{v})$ if P is a predicate with free variables $\overline{v}$.

▶ **Definition 5.** Suppose $Q(h, f)$ is a predicate on history h and mapping function f, $\overline{m}$ is a vector over type $\mathbb{N}$, $P(h, \overline{m})$ is a predicate on h and $\overline{m}$, and $QR(f, \overline{m})$ and $QD(f, \overline{m})$ are predicates on f and $\overline{m}$. We say that $Q(h, f)$ is:

- an *order condition* iff $Q(h, f)$ is of the form $\forall\, \overline{m} : \mathrm{dom}\, f \bullet P(h, \overline{m}) \Rightarrow QR(f, \overline{m})$, where $QR(f, \overline{m})$ is a predicate on the range of f and $\overline{m}$ only, and
- a *commitment condition* iff $Q(h, f)$ is of the form $\forall\, \overline{m} : \mathrm{dom}\, h \bullet P(h, \overline{m}) \Rightarrow QD(f, \overline{m})$, where $QD(f, \overline{m})$ is a predicate on the domain of f and $\overline{m}$ only.

To reduce clutter, for predicates R, R_1 and R_2 on a history h, mapping function f, and boolean operator $\oplus$, we define:

$$\begin{aligned} R_1 \equiv R_2 &\;\widehat{=}\; \forall\, h, f \bullet legal(h) \Rightarrow R_1(h, f) = R_2(h, f) \\ R_1 \Rrightarrow R_2 &\;\widehat{=}\; \forall\, h, f \bullet legal(h) \wedge R_1(h, f) \Rightarrow R_2(h, f) \\ (R_1 \oplus R_1)(h, f) &\;\widehat{=}\; R_1(h, f) \oplus R_2(h, f) \end{aligned}$$

Using these concepts, we now define what it means for a mapping function to be valid. We formalise this definition using an order property *vmf_ord*, which ensures that for any matching pair m, n in h mapped by f, index $f(n)$ immediately follows $f(m)$:

$$vmf\_ord(h, f) \;\widehat{=}\; \forall\, m, n : \mathrm{dom}\, f \bullet mp(m, n, h) \Rightarrow f(n) = f(m) + 1$$

and a commitment property vmf_com, which ensures that for any matching pair m, n in h, the invocation $h(m)$ is mapped by f iff the return $h(n)$ is also mapped by f:

$$vmf\_com(h, f) \quad \widehat{=} \quad \forall\, m, n : \mathrm{dom}\, h \bullet mp(m, n, h) \Rightarrow (m \in \mathrm{dom}\, f \Leftrightarrow n \in \mathrm{dom}\, f)$$

Note that vmf_ord conforms to the structure of an order condition as defined in Definition 5, since predicate $P(h, \overline{m})$ is instantiated to $mp(m, n, h)$ and $QR(f, \overline{m})$ is instantiated to $f(n) = f(m) + 1$. Similarly, vmf_com conforms to the structure of a commitment condition as defined in Definition 5; $P(h, \overline{m})$ is instantiated to $mp(m, n, h)$ and $QD(h, \overline{m})$ instantiated to $m \in \mathrm{dom}\, f \Leftrightarrow n \in \mathrm{dom}\, f$.

We say a function f is a *valid mapping function* if, for any history h, the domain of f is contained in the domain of h, the range of f is a consecutive sequence starting from 0, only invocation/return events are mapped by f, matching pairs in h are mapped to consecutive events in the target abstract history, and f only maps matching pairs. Assuming $[m..n]$ is the set of naturals from m to n inclusive, we formalise validity for mapping functions as follows:

$$\begin{aligned} VMF(h, f) \quad \widehat{=} \quad & \mathrm{dom}\, f \subseteq \mathrm{dom}\, h \wedge (\exists\, n : \mathbb{N} \bullet \mathrm{ran}\, f = [0..n-1]) \wedge \\ & (\forall\, n : \mathrm{dom}\, f \bullet h(n) \in Event) \wedge vmf\_ord(h, f) \wedge vmf\_com(h, f) \end{aligned}$$

We can now define a correctness condition to be a conjunction of ordering and commitment conditions, along with a requirement that we have a valid mapping function.

▶ **Definition 6.** A *correctness condition* is a predicate $R(h, f)$ over a history h and mapping function f, whose definition has the form:

$$R(h, f) \quad \widehat{=} \quad VMF(h, f) \wedge (\textstyle\bigwedge_i OC_i(h, f)) \wedge (\bigwedge_j CC_j(h, f))$$

where each OC_i is an order condition and each CC_j is a commitment condition.

Note that the conjunct vmf_ord in VMF means that pending invocations in h are never mapped by f. However, when formalising correctness conditions, one must also consider *incomplete histories*, which contain pending invocations whose effects have already taken place and are observable to other processes [23].

▶ **Example 7.** Consider a history $HE_1 \mathrel{\widehat{=}} \langle inv(w, \mathtt{put}, 7), inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, 7)\rangle$ of the Chase-Lev deque (Fig. 1). This history is incomplete because the invocation of the `put` operation has not returned. However, its effect has clearly taken place because the `steal` operation returns 7.

To reason about such histories, Herlihy and Wing [23] consider *history extensions*, which are constructed from a history h by concatenating a sequence of returns corresponding to some of the pending invocations of h. For example, HE_1 may be extended to $HE_1 \frown \langle ret(w, \mathtt{put})\rangle$ to enable the extended history to be mapped abstractly. Note that a history may have several possible extensions. For example, for the history:

$$HE_2 \quad \widehat{=} \quad \langle inv(w, \mathtt{put}, 7), inv(q_1, \mathtt{steal}), ret(w, \mathtt{put}), inv(q_2, \mathtt{steal})\rangle$$

the following are some of many possible extensions:

$$\begin{aligned} HE_3 \quad & \widehat{=} \quad HE_2 \frown \langle ret(q_1, \mathtt{steal}, emp)\rangle \\ HE_4 \quad & \widehat{=} \quad HE_2 \frown \langle ret(q_2, \mathtt{steal}, 7), ret(q_1, \mathtt{steal}, emp)\rangle \end{aligned}$$

Pending invocations in an incomplete history may remain pending in the extended history. For example, in H_3, the second steal operation is still pending. Herlihy and Wing define a function *complete* to remove all pending histories from a history, and define linearizability of a history h in terms of $complete(he)$, where he is some extension of h. However, reasoning about $complete(he)$ is often cumbersome because removal of pending invocations causes the indices of he to shift. This is exacerbated by the non-determinism of history extensions.

In our framework, because correctness is defined using an explicit mapping function, we can avoid using the *complete* function. In particular, after extending an incomplete history with return events, we can simply leave out pending invocations when mapping this extended history, simplifying the definitions and the proofs. We now lift correctness to the level of concurrent objects. This definition is tied to the fact that every concurrent object is inherently an implementation of some sequential abstract counterpart.

▶ **Definition 8.** A concurrent object C implementing an abstract object A is *correct* with respect to a correctness condition R, denoted $C \models_A R$, iff for any legal history h of C, there exists an extension he of h, a mapping function f such that $R(he, f)$ holds, and a valid sequential history hs of A such that $hs = \{f(k) \mapsto he(k) \mid k \in \operatorname{dom} f\}$.

The next theorem states that if a concurrent object implements an abstract object for some notion of correctness, then it also implements the abstract object with respect to a weaker correctness condition.

▶ **Theorem 9.** *Suppose C is a concurrent object, A an abstract object and R_1, R_2 are correctness conditions such that $R_1 \Rrightarrow R_2$. If $C \models_A R_1$ then $C \models_A R_2$.*

The proof is straightforward by expanding the definitions and using the fact that *legal* is *extension closed*, i.e., if $legal(h)$ holds and he is an extension of h, then $legal(he)$ holds.

3.2 Specifying Correctness Conditions for Totally Ordered Memory

We now use our framework to formalise the conditions for totally ordered memory from Section 2.2: sequential consistency, linearizability and quiescent consistency. There are already existing formalisations of each of these in the literature, e.g., using partial orders. However, using our framework, we are able to distinguish between the different types of properties that form each condition.

Each correctness condition in Section 2.2 implies a *total* commitment condition, which means that all completed operation calls in a given history h must be mapped by f to some operation call in a sequential history.

$$total(h, f) \quad \widehat{=} \quad \forall\, m : \operatorname{dom} h \bullet h(m) \in Event \wedge \neg pi(m, h) \Rightarrow m \in \operatorname{dom} f$$

Sequential consistency is defined in terms of an order condition sc, which states operation calls in h by the same process are not reordered by f when mapped to a sequential history.

$$\begin{aligned} sc(h, f) \quad \widehat{=} \quad \forall\, m, n : \operatorname{dom} f \bullet\ & m < n \wedge h(m).pr = h(n).pr \wedge \\ & ret?(h(m)) \wedge inv?(h(n)) \Rightarrow f(m) < f(n) \end{aligned}$$

▶ **Definition 10.** A concurrent object C implementing an abstract object A is *sequentially consistent* iff $C \models_A SC$, where [3] $SC \mathrel{\widehat{=}} VMF \wedge sc \wedge total$.

[3] Note that by definition of $\equiv$ and pointwise lifting, $SC(h, f) \equiv VMF(h, f) \wedge sc(h, f) \wedge total(h, f)$ for any history h and mapping function f.

Linearizability [23] is a straightforward extension to sequential consistency, strengthening the order condition so that an operation call is not reordered with another operation call that is invoked after the first operation returns.

$$lin(h, f) \quad \widehat{=} \quad \forall\, m, n : \operatorname{dom} f \bullet m < n \wedge ret?(h(m)) \wedge inv?(h(n)) \Rightarrow f(m) < f(n)$$

▶ **Definition 11.** A concurrent object C implementing an abstract object A is *linearizable* iff $C \models_A LIN$, where $LIN \mathrel{\widehat{=}} VMF \wedge lin \wedge total$.

It is straightforward to link this definition to the formalisation by Derrick *et al* [10], which has in turn been linked with Herlihy and Wing's original definition.

Quiescent consistency, as informally described by Shavit [31], has been formalised in [9] and is defined in terms of bijections between a concurrent history and its corresponding abstract history. We first define a *quiescent point* as an index m in a history h at which there are no pending invoked operation calls. We use $h[m..n]$ to denote the *projection* of the elements of h from index m to n, inclusive, i.e., $h[m..n] = \langle h(m), h(m+1), \ldots, h(n-1), h(n)\rangle$.

$$qp(m, h) \quad \widehat{=} \quad \forall\, n : \operatorname{dom} h \bullet (n \le m => \neg pi(n, h[0..m]))$$

The ordering condition for quiescent consistency states that f does not reorder two indices in h separated by a quiescent point.

$$qc\_ord(h, f) \quad \widehat{=} \quad \forall\, m, k, n : \operatorname{dom} f \bullet m < k < n \wedge qp(k, h) \Rightarrow f(m) < f(n)$$

▶ **Definition 12.** A concurrent object C implementing an abstract object A is *quiescent consistent* iff $C \models_A QC$, where $QC \mathrel{\widehat{=}} VMF \wedge qc\_ord \wedge total$.

A benefit of our formalisation is that it is now straightforward to formally prove that linearizability implies both sequential consistency and quiescent consistency, the former is because $lin \Rrightarrow sc$ holds, while the latter is because $lin \Rrightarrow qc$. It is well known that $SC \Rrightarrow LIN$ and $QC \Rrightarrow LIN$ are both false; constructing counter-examples is straightforward [22].

4 Correctness Conditions for Total Store Order Memory

We now explore notions of correctness for concurrent objects in relaxed memory architectures. In particular, we focus on the potential for optimisation for TSO architectures. To simplify development of correctness conditions, we present each correctness condition as an instantiation of a number of high-level steps. We formalise two recently defined notions of correctness [12, 32], develop sequentially consistent variations of these, then develop a new correctness condition, *fence consistency*.

4.1 Minimising `fence` instructions in TSO

Correctness conditions that hold for a concurrent object under totally ordered memory may no longer hold in the presence of relaxed memory. For our running example, under TSO memory, consider the following scenario. After initialisation, suppose two complete `put` operations as well as their `flush`es have been executed. Thus, the deque is of size two with tasks a_0 and a_1 at array indices 0 and 1, and `Head` $= 0$ and `Tail` $= 2$. Suppose w invokes a `take`, which executes up to line `T4` without executing any `flush`es, setting its local variables `hd` and `tl` to 0 and 1, respectively. Now suppose two thief processes q_1 and q_2 invoke and execute `steal` operations up to completion, stealing both a_0 and a_1. The worker may now continue executing `take`, and return some unspecified value for `task` because the test at `T8`

```
    int take() {
T1    tl := Tail - 1;
T2    Tail := tl;
T3    fence ;
T4    hd := Head;
T5    if hd > tl {
T6       Tail := hd;
T7       return emp; }
T8    task := items[tl];
T9    if tl > hd
T10     return task;
T11   Tail := hd + 1;
T12   if cas(Head,hd,hd+1)
T13     return task;
T14   else return emp; } }
```

Figure 5 Chase-Lev `take` operation modified for TSO.

succeeds. Such an execution cannot be proved to implement Fig. 2 for any sensible definition of correctness.

Liu et al. [27] have shown that linearizability can be restored provided (i) a **`fence`** is introduced immediately after `P3` in the `put` operation, and (ii) the `take` operation in Fig. 1 is replaced by the `take` in Fig. 5, where a **`fence`** has been introduced after `T2`. As memory barriers in the form of **`fence`** instructions are expensive, our question is: *Are there conditions weaker than linearizability that would allow only one* **`fence`** *to be used such that the behaviours one obtains are still sensible?* Although removing a single **`fence`** instruction may not seem like a big change, because a client may execute several `put` operations consecutively, there is a potential for a high level of efficiency gains. Furthermore, since data structures such as deques are used to implement underlying system mechanisms such as schedulers [16] and operating system kernels [28], avoiding **`fence`** instructions can provide system-wide benefits.

It turns out that a **`fence`** after `T2` is needed to avoid the scenario described above. In the other case, it turns out that the object is not linearizable, because the following is possible:

$$\langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), inv(q, \mathtt{steal}), ret(q, \mathtt{steal}, emp)\rangle \qquad (1)$$

This occurs because the effect of a `put` operation only occurs after the write at `P3` is flushed. Therefore, the `steal` operation may read an older value causing it to return `emp`. We argue that such histories should be allowed – it is perfectly sensible for the `steal` and `put` operations to be reordered because the effect of the `put` has merely been delayed by buffer effects, whereby the `put` operation continues to execute beyond its return event. We therefore, set out to formally define correctness conditions that would accept histories such as (1), e.g., for the Chase-Lev deque under TSO memory where no **`fence`** instructions are introduced after `P3`.

Behaviours in TSO memory in which the effect of an operation is delayed beyond its return are already accepted as being correct for many implementations, e.g., spinlock [12, 28], Burns' mutex [35] and the sequence lock [32]. However, a precise notion of correctness in these scenarios has thus far not been developed. Even less is known about the implications of accepting more histories than allowed by linearizability.

4.2 Defining Correctness Conditions

It turns out that there are several possibilities for interpreting correctness for delayed operations. We describe a sequence of steps for defining correctness conditions for TSO memory, where each step identifies an aspect of the condition that must be considered. Picking a particular instantiation at each step, leads to a particular correctness condition.

▶ Step 1 (Determine the events to be recorded in histories). For the conditions on totally ordered memory in Section 2.2, histories only needed to record invocation/response events.

For TSO memory, it is often necessary to record additional events, with rules on how these events are recorded. Formally, these additional events are recorded by instantiating $Event_C$.

For example, for weak ξ-quiescent consistency (as defined in Section 4.3), we record an additional event $\xi(p)$, thus $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$. Event $\xi(p)$ is triggered (i.e., recorded) if either (i) a transition causes the buffer of process p to become empty, or (ii) if process p returns from an operation call when p's buffer is empty. For case (i), $\langle \xi(p)\rangle$ is concatenated, while for case (ii) the two-event sequence $\langle ret(p, op, v), \xi(p)\rangle$ is concatenated to the end of the history[4]. Note that a transition causing p's buffer to become empty may be caused by a CPU-controlled buffer flush, which may occur after p has already returned. We assume $\xi(p)$ is not recorded if the buffer of p is already empty in the prestate of a non-return transition, and if p returns when its buffer is non-empty, then only $\langle ret(p, op, v)\rangle$ is concatenated to the end of the history. $\square$

We explain the next two steps assuming $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$ has been fixed as defined in Step 1. For our examples below, we assume that the deque is initially empty, w denotes the worker process, and q, q_1, q_2 and q_3 denote thief processes.

▶ Step 2 (Determine what operations can be reordered). A common feature of the correctness conditions discussed in Section 2.2 is that operation calls whose active intervals overlap may be reordered. For totally ordered memory, an operation call may be considered to be active from its invocation to its return.

In the context of TSO memory, because some operation calls may return with non-empty buffers, there is additional flexibility in defining what counts as an active operation [12, 32, 35]. One possibility is to think of an operation call by a process p as being active until buffer of process p becomes empty. Consider the following history, which is possible for the deque in Fig. 1 but with the `take` operation from Fig. 5.

$$\begin{aligned} HC_1 \;\widehat{=}\;& \langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q_1, \texttt{steal}), ret(q_1, \texttt{steal}, emp), \xi(q_1),\\ & inv(q_2, \texttt{steal}), ret(q_2, \texttt{steal}, emp), \xi(q_2), \xi(w), inv(q_3, \texttt{steal}),\\ & \xi(q_3), ret(q_3, \texttt{steal}, x), \xi(q_3)\rangle \end{aligned}$$

HC_1 cannot be linearized with respect to the abstract deque (Fig. 2) – HC_1 restricted to invocations and responses only is sequential and the `steal` occurs after the `put` has completed, yet the `steal` returns empty. However, in the context of TSO memory with the interpretation that returned operations calls by process p are active until p's buffer is empty, HC_1 can be explained by the following sequential history:

$$\begin{aligned} &\langle inv(q_1, \texttt{steal}), ret(q_1, \texttt{steal}, emp), inv(q_2, \texttt{steal}), ret(q_2, \texttt{steal}, emp),\\ & inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q_3, \texttt{steal}), ret(q_3, \texttt{steal}, x)\rangle \end{aligned}$$ $\square$

It turns out that there are varying ways of defining active operations. In this paper we explore two possibilities: the first (inspired by [32]) allows an operation call to be active as long as the buffer of its calling process is non-empty, and the second (inspired by [12]) is more restricted, allowing an operation call to be active only as long as the final write corresponding to the operation call has not been flushed.

▶ Step 3 (Determine the commitment conditions). The conditions in Section 2.2 for totally ordered memory are all total, i.e., any operation call that has returned must be mapped to some abstract operation call. Total conditions are appropriate for such architectures because

[4] Note that there are other alternatives to recording case (ii) in the history; e.g., one could use a special "return empty" event that is distinct from ret events to obtain $Event_C ::= Event \mid ret_\xi\langle\langle P \times O \times V\rangle\rangle \mid \xi\langle\langle P\rangle\rangle$.

hardware guarantees that each write is immediately committed to shared memory when the write instruction is executed, making its effect visible to other concurrent threads.

On the other hand, in relaxed memory models, write instructions may be cached in local buffers, and thus not seen by other processes until the buffers are flushed. Hence, when an operation call returns, the effect of the operation may not have have occurred in shared memory. We refer to a returned operation call that has taken effect as a *committed* operation call and as *uncommitted*, otherwise. To take delayed operation calls (due to buffer effects) into account, we allow correctness conditions to be defined using *partial commitment conditions*, allowing some completed operation calls to not be mapped to any abstract operations. When specifying partial commitment conditions, it turns out that one must additionally define conditions that dictate when an operation must become committed.

For TSO memory, one possible instantiation of this step is to require that *all operation calls of process p that have returned prior to $\xi(p)$ occurring must have committed.* For example, consider the following history:

$$\begin{aligned} HC_2 \ \widehat{=}\ & \langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), \xi(w), inv(w, \texttt{put}, y), ret(w, \texttt{put}), \\ & inv(q, \texttt{steal}), ret(q, \texttt{steal}, x), \xi(q)\rangle \end{aligned}$$

History HC_2 cannot be judged consistent against sequential histories $\langle\,\rangle$ or $\langle inv(w, \texttt{put}, x), ret(w, \texttt{put})\rangle$ because due to $\xi(w)$, the first `put` operation must be committed, and due to $\xi(q)$, the `steal` must have also been committed. Note that $\xi(p)$ represents that latest point at which commitments of completed operations of process p must occur; the commitment condition does not prevent operations from committing earlier. Thus, for example, both sequential histories below satisfy the requirement:

$\langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q, \texttt{steal}), ret(q, \texttt{steal}, x)\rangle$

$\langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(q, \texttt{steal}), ret(q, \texttt{steal}, x), inv(w, \texttt{put}, y), ret(w, \texttt{put})\rangle$ ◻

Note that the conditions in Section 2.2 can also be defined using these three steps. For all three conditions, $Event_C ::= Event$ (Step 1), and the commitment condition is *total*, which states completed operation calls must appear in any corresponding sequential history (Step 3). The three conditions only differ in terms of their order properties, *sc*, *lin* and *qc*, which are different instantiations of Step 2.

4.3 Weak ξ-Quiescent Consistency

Smith et al. [32] prove correctness of a sequence lock algorithm in TSO memory with respect to quiescent consistency [31]. An object is considered to be quiescent in a history if none of its operations calls are pending in the history and the buffer of each process that has called an operation of the object is empty. Note that the buffer of a process calling an operation may become empty after the operation has returned. Reordering of operations across a quiescent point is disallowed.

This condition may be formalised by instantiating the steps in Section 4.2. For Step 1, we use $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$ because we must reason about empty buffers. We say such a history h is *legal* iff $h_{|Event}$ (i.e., h restricted to invocations and return events) is legal. Histories of this type are extension closed. For Step 2, as in [32], we say an operation call is active until the object becomes quiescent. Finally, for Step 3, we require that all operation calls be committed when the object becomes quiescent – until then operation calls remain uncommitted.

An index m is quiescent iff the last completed operation call for each process p has been followed by $\xi(p)$. We say that $p \in P$ is *quiescent between* indices m and n of history h iff m is a return for p, the buffer of p becomes empty at some point between m and n, and p does not invoke any new operation between m and n. Thus, we define the following, where $empty?(e)$ holds iff $e = \xi(p)$ for some $p \in P$, and $inv_p?(e) \mathrel{\widehat{=}} inv?(e) \wedge e.pr = p$, and predicates $ret_p?$ and $empty_p?$ are similarly defined.

$$\begin{aligned} qb(m, n, p, h) \quad \widehat{=} \quad & m < n \wedge ret_p?(h(m)) \wedge (\exists\, k : m+1..n \bullet h(k) = \xi(p)) \wedge \\ & (\forall\, k : m+1..n \bullet \neg inv_p?(h(k))) \end{aligned}$$

Using qb, we define a quiescent point m in history h as follows.

$$qp_\xi(m, h) \quad \widehat{=} \quad \forall\, p : P, n : \mathrm{dom}\, h \bullet n < m \wedge inv_p?(h(n)) \Rightarrow \exists\, k : \mathrm{dom}\, h \bullet qb(k, m, p, h)$$

Then we define the order and commitment conditions for weak ξ-consistency as follows. Order condition wqc_ord_ξ states that two events separated by a quiescent point are not reordered.

$$wqc\_ord_\xi(h, f) \quad \widehat{=} \quad \forall\, m, n : \mathrm{dom}\, f \bullet (\exists\, k : \mathbb{N} \bullet m < k < n \wedge qp_\xi(k, h)) \Rightarrow f(m) < f(n)$$

The commitment condition wqc_com_ξ requires all operation calls occurring before a quiescent point to be committed.

$$wqc\_com_\xi(h, f) \quad \widehat{=} \quad \forall\, m, n : \mathrm{dom}\, h \bullet m \le n \wedge qp_\xi(n, h) \wedge h(m) \in Event \Rightarrow m \in \mathrm{dom}\, f$$

▶ **Definition 13.** A concurrent object C implementing an abstract object A is *weakly ξ-quiescent consistent* iff $C \models_A WQC_\xi$, where $WQC_\xi \mathrel{\widehat{=}} VMF \wedge wqc\_ord_\xi \wedge wqc\_com_\xi$.

Weak ξ-quiescent consistency is a straightforward generalisation of quiescent consistency, and hence, we omit example histories here. This definition is weak because it does not guarantee sequential consistency, i.e., operations calls within a process may be reordered. However, weak ξ-quiescent consistency follows the condition defined in [32], which in turn is based on quiescent consistency [9, 22, 31]. In Section 4.5, we motivate a new correctness condition, then strengthen WQC_ξ accordingly so that it guarantees sequential consistency. Defining such variations in our framework is straightforward.

4.4 Weak Flush Consistency

We now formalise the correctness condition defined by Derrick et al. [12], which we refer to in this paper as *weak flush consistency*. Informally, weak flush consistency captures the idea that an operation call may be considered to be active only until the last pending write corresponding to the operation call is flushed. This differs from weak ξ-quiescent consistency, since an operation call may become inactive even if the buffer of the calling process is non-empty.

Derrick et al. [12] define weak flush consistency in terms of linearizability of *transformed* histories. Their (deterministic) transformation algorithm proceeds as follows: (i) the final flush corresponding to each operation call is located, (ii) the actual return is moved to this flush, and (iii) all remaining flushes are removed from the history. The standard definition of linearizability is then applied to the transformed histories. For example, consider the following history, where $\phi^k(p)$ denotes k consecutive flush events of process p:

$$\begin{aligned} &\langle inv(w, \mathtt{put}, x), ret(w, \mathtt{put}), inv(w, \mathtt{put}, y), inv(q, \mathtt{steal}), \phi(q), \phi^3(w), \\ &\ ret(q, \mathtt{steal}, emp), ret(w, \mathtt{put})\rangle \end{aligned}$$

This history is transformed to

$$\langle inv(w, \mathtt{put}, x), inv(w, \mathtt{put}, y), inv(q, \mathtt{steal}), ret(w, \mathtt{put}), ret(q, \mathtt{steal}, emp), ret(w, \mathtt{put})\rangle$$

then judged consistent because it is linearizable with respect to the sequential history:

$$\langle \mathit{inv}(q, \texttt{steal}), \mathit{ret}(q, \texttt{steal}, \mathit{emp}), \mathit{inv}(w, \texttt{put}, x), \mathit{ret}(w, \texttt{put}), \mathit{inv}(w, \texttt{put}, y), \mathit{ret}(w, \texttt{put})\rangle$$

Using our framework, we can define weak flush consistency directly, i.e., without performing such a history transformation. This involves two small extensions to the *Event* type. First, we record each flush event. Second, because the number of writes each operation call performs is potentially non-deterministic, we additionally record each write event to identify the last flush corresponding to each operation call. Thus, for Step 1, we define $\mathit{Event}_C ::= \mathit{Event} \mid \omega\langle\!\langle P\rangle\!\rangle \mid \phi\langle\!\langle P\rangle\!\rangle$. Here, $\omega(p)$ denotes a write by process p, and $\phi(p)$ records a flush for process p. We assume $\mathit{write}?(e)$ and $\mathit{flush}?(e)$ hold iff event e is a write and flush, respectively. We assume that only writes and flushes executed by the concurrent object in question are recorded in the histories, and that writes are executed between matching invocations and responses. Hence, we say such a history h is *legal* iff $h_{|\mathit{Event}}$ is legal and $\omega(p)$ only occurs in h when p is executing some operation. Legality of histories of this type are also extension closed.

For Step 2, we say an operation call that completes after executing l writes remains active until each of these l writes have been flushed, or until the return occurs, whichever is later. To formalise this, we define a function *num*, which counts the number of events in h that satisfy event predicate *ep* (mapping an event to a boolean) up to and including index m:

$$\mathit{num}(\mathit{ep}, m, h) \quad \widehat{=} \quad \mathit{size}(\{h(k) \mid 0 \le k \le m \land \mathit{ep}(h(k))\})$$

The order condition for weak flush consistency counts the number of writes, say l, that have occurred when an operation call, say L, by process p returns. Any operation invoked after these l flushes have occurred may not be reordered with L. Thus, we obtain:

$$\begin{array}{lcl} \mathit{wflc\_ord}(h, f) & \widehat{=} & \forall\, m, n : \operatorname{dom} f \bullet \\ & & \left(\begin{array}{l}\exists\, p : P \bullet m < n \land \mathit{ret}_p?(h(m)) \land \mathit{inv}?(h(n)) \land \\ \qquad \mathit{num}(\mathit{write}_p?, m, h) \le \mathit{num}(\mathit{flush}_p?, n, h)\end{array}\right) \Rightarrow f(m) < f(n) \end{array}$$

Note that $\mathit{num}(\mathit{write}_p?, m, h)$ counts all writes by process p since initialisation.

For Step 3, we say that completed operation calls whose last write has been flushed must commit. Weak flush consistency has two commitment conditions. The first condition, $\mathit{wflc\_com}_1$, states that an operation call is committed whenever all writes executed by that operation call have been flushed and the call returns. The second, $\mathit{wflc\_com}_2$, requires than an operation call L that executes l writes that are not flushed before L returns is committed whenever l flushes of the calling process have occurred.

$$\begin{array}{lcl} \mathit{wflc\_com}_1(h, f) & \widehat{=} & \forall\, n : \operatorname{dom} h \bullet \\ & & \left(\begin{array}{l}\exists\, p : P \bullet \mathit{ret}_p?(h(n)) \land \\ \qquad \mathit{num}(\mathit{flush}_p?, n, h) = \mathit{num}(\mathit{write}_p?, n, h)\end{array}\right) \Rightarrow n \in \operatorname{dom} f \\ \mathit{wflc\_com}_2(h, f) & \widehat{=} & \forall\, k, n : \operatorname{dom} h \bullet \\ & & \left(\begin{array}{l}\exists\, p : P \bullet n < k \land \mathit{ret}_p?(h(n)) \land \mathit{flush}_p?(k) \\ \qquad \mathit{num}(\mathit{write}_p?, n, h) = \mathit{num}(\mathit{flush}_p?, k, h)\end{array}\right) \Rightarrow n \in \operatorname{dom} f \end{array}$$

▶ **Definition 14.** A concurrent object C implementing an abstract object A is *weakly flush consistent* iff $C \models_A \mathit{WFLC}$, where $\mathit{WFLC} \mathrel{\widehat{=}} \mathit{VMF} \land \mathit{wflc\_ord} \land \mathit{wflc\_com}_1 \land \mathit{wflc\_com}_2$.

▶ **Example 15.** Consider the histories below, neither of which is linearizable, where $\omega^k(p)$ denotes k consecutive $\omega(p)$ events.

$$\langle inv(w, \texttt{put}, x), \omega^3(w), ret(w, \texttt{put}), \phi^2(w), inv(q, \texttt{steal}), \phi(w), \omega^3(q), \phi^3(q), ret(q, \texttt{steal}, emp)\rangle \quad (2)$$

$$\langle inv(w, \texttt{put}, x), \omega^3(w), ret(w, \texttt{put}), \phi^3(w), inv(q, \texttt{steal}), \omega^3(q), \phi^3(q), ret(q, \texttt{steal}, emp)\rangle \quad (3)$$

History (2) is weakly flush consistent; the `steal` operation is invoked before the final flush of the `put` occurs, and hence the active intervals of the `put` and `steal` overlap, allowing the `steal` to be ordered before the `put`, i.e., (2) is flush consistent with respect to sequential history $\langle inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp), inv(w, \texttt{put}, x), ret(w, \texttt{put})\rangle$. On the other hand, (3) is not weakly flush consistent because the active intervals of `put` and `steal` do not overlap, namely, the `steal` is invoked after the final flush of `put` has occurred, and hence, cannot be ordered before the `put`.

4.5 Sequential Consistency for TSO Memory

Both weak ξ-quiescent consistency and weak flush consistency allow operation calls for the same process to be reordered, i.e., sequential consistency may be violated. If sequential consistency is required, the following history should be judged incorrect even though both `put` operation calls are "active" over the interval in which the `steal` occurs.

$$HC_3 \;\widehat{=}\; \langle inv(w, \texttt{put}, x), ret(w, \texttt{put}), inv(w, \texttt{put}, y), ret(w, \texttt{put}), inv(q, \texttt{steal}), \xi(q), ret(q, \texttt{steal}, y), \xi(q), \xi(w)\rangle$$

Because `put` operations add elements to the end of the deque and `steal` operations remove elements from the beginning, the only way to explain HC_3 is by reordering the first two `put` calls, violating sequential consistency. It turns out that sequential consistency is an important property. In fact, it is equivalent to *observational refinement* between a concrete object and its abstract specification for *data independent* clients [15]. The notion of observational refinement is based on observing the initial and final values of variables of client programs.

> An implementation C of a data structure is an *observational refinement* of an implementation A of the same data structure, if every observable behaviour of any client program using C can also be observed when the program uses A instead. [15, pg 412]

Data independence states that each process accesses only local variables or resources in its client operations. We therefore define sequentially consistent versions of weak ξ-quiescent consistency and weak flush consistency. This is straightforward in our framework and involves adding the already defined order condition *sc* as a conjunct, i.e., the sequentially consistent versions are obtained via a different instantiation of Step 2.

▶ **Definition 16.** Suppose C is a concurrent object implementing an abstract object A. We say C is

- *ξ-quiescent consistent* iff $C \models_A QC_\xi$, where $QC_\xi \;\widehat{=}\; WQC_\xi \wedge sc$
- *flush consistent* iff $C \models_A FLC$, where $FLC \;\widehat{=}\; WFLC \wedge sc$.

It is possible to prove that the deque in Fig. 1 is flush consistent under TSO memory. The argument is complex and combines the most sophisticated types of reasoning from linearizability proofs (i.e., those that require reasoning about future behaviour [29, 21] and about linearization points in other operations [11, 8]) with the additional complexities of reasoning about delayed operations [12, 32]. We use the term *commit point* to refer to the atomic program statement that causes the effect of an operation to be felt abstractly; this is analogous to a *linearization point* in a linearizability proof [10, 8]. We provide a proof sketch for this argument below.

▶ **Proposition 17.** *The work-stealing deque in Fig. 1 is flush consistent under TSO with respect to the abstract deque in Fig. 2.*

Proof. A standard representation relation is used to relate the concrete array used by Fig. 1 with the abstract sequence in Fig. 2. We assume the existence of a program counter variable pc_p for each process p, with value *idle* if p is currently not invoking any operation and value $P1, \ldots, P3, S1, \ldots S7, T1, \ldots T14$ corresponding to the labels in Fig. 1, otherwise.

To cope with delayed operations, we record operation calls that have returned without committing in an auxiliary variable $g \in P \to \operatorname{seq} Event$. Each invocation by process p appends a corresponding invoke event in $g(p)$, and each return that has not committed appends the corresponding return event to $g(p)$. Therefore, $g(p)$ records the sequence of uncommitted calls by p in real-time order. To ensure sequential consistency, pending operations are committed by process p by removing the first two elements from $g(p)$ (which must be an invocation/return), then executing the corresponding operation abstractly. The commit points of the algorithm are as follows.

- A `flush` by a worker process of a pending write to *Tail* when $pc_w \in \{idle, T1, T2\}$. The only possible pending operation in this case is `put`, which is committed.
- A `flush` by a worker process of a pending write to *Tail* when $pc_w = T3$, which commits the first pending operation $g(w)$. There are two cases depending on the value of $head(g(w))$. If $head(g(w)).i =$ `put`, then the `flush` corresponds to committing a completed `put` (that has already returned). Otherwise, there are 3 further cases. The two simpler cases are: (i) if $Tail > Head$ in the post state, then the `flush` must commit the currently executing `take` operation, and (ii) if $Tail < Head$, then the `flush` must commit a `take` that returns *emp*. The difficult case is if $Tail = Head$ in the post state, which signifies the case where there is only one task in the deque, causing the current `take` operation to race with a `steal` operation executed by another process. If there is an active `steal` operation at `S4` or `S5`, there are two possible outcomes, depending on the *future* execution of the program:
  - the active `steal` operation succeeds and the `take` returns *emp* (via `T14`), or
  - the `take` succeeds and the `steal` tries again.

  One of the two outcomes must be determined at this `flush` because any other operations `steal` invoked (by a third process) after the `flush` has been executed will return *emp*. Note that if there is no active `steal` operation at line `S4` or `S5`, then the only possible future outcome is that the `take` succeeds.
- The `fence` at `T3`. This commits all pending `put` operations. Then, commits the executing `take`, or a `take` and a `steal` depending on whether $Tail > Head$, $Tail < Head$ or $Tail = Head$ holds in the post state, as in the `flush` case described above.
- A successful `cas` at `S6`. Assuming the `steal` is executed by a process $p \neq w$, there are two cases, depending on the value of $g(p)$. If $g(p) = \langle inv(p, steal, \perp)\rangle$, then the `steal` has not yet been committed earlier, and the successful `cas` commits the `steal`. Otherwise, $g(p) = \langle ret(p, steal, tout)\rangle$ holds, i.e., the `steal` has been committed earlier (by a `flush` or `fence` as above), and hence, the `cas` is not a commit point.

◀

Interestingly, the `cas` at `T12` is not a commit point because the `take` operation must have already been committed either at the `fence` at `T3` or an earlier `flush`.

Using a weaker condition than linearizability has allowed us to prove correctness with respect to a standard sequential abstract specification. This differs from [6, 17], where linearizability is established, but the abstract specification differs from what one would expect. In particular, [6] uses an abstraction that executes using TSO semantics, whereas [17] includes additional non-determinism to cope with buffer effects at the concrete level.

4.6 Fence Consistency

ξ-quiescent consistency is simple, but provides relatively weak guarantees about a program's behaviour; flush consistency on the other hand is a conservative weakening of linearizability (providing strong behavioural guarantees), but is relatively complex as both writes and flushes need to be recorded in a history. Following the formalisations in the preceding sections, we identify a new correctness condition, *fence consistency*, that is weaker than flush consistency, but stronger than ξ-quiescent consistency. The condition aims to capture the fact that in many cases, if the buffer of a process p becomes empty at say index m of a history, then completed operation calls for p before m should not be reordered with operations invoked after m. This means that the event corresponding to a "buffer becoming empty" is a barrier to reordering, which is precisely the intention of a programmer specified **fence** instruction. This condition is potentially useful for developing algorithms that offer more optimisation possibilities than flush consistency.

Fence consistency has the characteristics of both flush consistency and ξ-quiescent consistency. Like ξ-quiescent consistency, for Step 1 we instantiate $Event_C ::= Event \mid \xi\langle\langle P\rangle\rangle$ and record $\xi(p)$ for $p \in P$ events in the same way. Step 2 is similar to flush consistency: we say an operation call is active until the buffer of the calling process is empty; operation calls may only be reordered if the intervals in which they are active overlap. For Step 3, we require operation calls executed by process p to be committed when p's buffer becomes empty. Finally, we require sequential consistency. The ordering condition for fence consistency states that if $\xi(p)$ occurs at an index k of history h, then any operation calls of process p completed (i.e., returned) before k are not reordered with an operation call by any process invoked after k, i.e.,

$$\begin{aligned} fc\_ord(h,f) \;\widehat{=}\; & \forall\, m,n : \mathrm{dom}\, f \bullet ret?(h(m)) \wedge inv?(h(n)) \wedge \\ & (\exists\, k : \mathrm{dom}\, h \bullet m < k < n \wedge empty?(h(k)) \wedge h(m).pr = h(k).pr) \\ & \Rightarrow f(m) < f(n) \end{aligned}$$

Furthermore, if $\xi(p)$ occurs at an index k of history h, then all completed operation calls of p occurring before k must have been committed, i.e.,

$$\begin{aligned} fc\_com(h,f) \;\widehat{=}\; & \forall\, n,k : \mathrm{dom}\, h \bullet n < k \wedge ret?(h(n)) \wedge empty?(h(k)) \wedge \\ & h(n).pr = h(k).pr \Rightarrow n \in \mathrm{dom}\, f \end{aligned}$$

▶ **Definition 18.** A concurrent object C implementing an abstract object A is *fence consistent* iff $C \models_A FC$, where $FC \mathrel{\widehat{=}} VMF \wedge sc \wedge fc\_ord \wedge fc\_com$.

▶ **Example 19.** In a fence consistent history, $\xi(p)$ does not prevent an operation call for p that is still executing from being reordered. This is for good reason. For example, consider the following history of the Chase-Lev deque:

$$\langle inv(w,\texttt{put},x), \xi(w), ret(w,\texttt{put}), inv(q,\texttt{steal}), ret(q,\texttt{steal},emp), \xi(q), \xi(w)\rangle \qquad (4)$$

Here, we know that even though an $\xi(w)$ occurs between $inv(w,\texttt{put},x)$ and $ret(w,\texttt{put})$, process w's buffer is non-empty when the `put` returns because $ret(w,\texttt{put})$ is not immediately followed by $\xi(w)$. Therefore, a `steal` operation may read an old value of *Tail*. Fence consistency states that the `put` operation is active until the second $\xi(w)$ occurs, allowing (4) to be judged consistent with respect to to the sequential history:

$$\langle inv(q,\texttt{steal}), ret(q,\texttt{steal},emp), inv(w,\texttt{put},x), ret(w,\texttt{put})\rangle \qquad \square$$

▶ **Example 20.** Fence consistency does not require an operation call to commit unless $\xi(p)$ occurs after the operation call has returned. For example, the history

$$\langle inv(w, \texttt{put}, x), \xi(w), ret(w, \texttt{put}), inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp), \xi(q)\rangle \tag{5}$$

is fence consistent with respect to history $\langle inv(q, \texttt{steal}), ret(q, \texttt{steal}, emp)\rangle$, where the effect of the put has not yet been reflected abstractly.

The next two theorems establish that fence consistency is both *non-blocking* and *compositional*. The non-blocking property pertains to *total operations*, which are operations for which a return is well-defined in any system state. A correctness condition is non-blocking iff total operations can always complete, i.e., are never prevented from completing by the correctness condition itself. A correctness condition R is compositional if for any multi-object system, the system as a whole satisfies R iff each object of the system satisfies R, which ensures that R can be proved in a modular manner. The lack of compositionality of sequential consistency was a key motivation for Herlihy and Wing to introduce linearizability, which is compositional [23], and hence, we see compositionality as being an important property.

▶ **Lemma 21.** *Suppose $m \in dom\, h$ such that $pi(m, h)$ holds, $h(m).i$ is a total operation, and e is an event such that $matching(h(m), e)$ holds. Then, $FC(h, f) \Rightarrow \exists f' \bullet FC(h \frown \langle e\rangle, f')$.*

▶ **Lemma 22.** *Suppose h is a history, f is a mapping function, hr is a sequence of returns and hr' a permutation of hr. Then $FC(h \frown hr, f) \Rightarrow \exists f' \bullet FC(h \frown hr', f')$.*

▶ **Theorem 23** (Fence consistency is non-blocking)**.** *Suppose $FC(he, f)$, where he extends history h and f is a matching function. If $m \in dom\, h$ is an index such that $pi(m, h)$ and $h(m).i$ is a total operation, then there exists an event e such that $matching(h(m), e)$, an extension he' of $h \frown \langle e\rangle$, and a mapping function f' such that $FC(he', f')$.*

Proof. Suppose $he = h \frown hr$. Because $h(m).i$ is total, the return event e is well defined. We must now show that he is fence consistent.

- If $m \in \mathrm{dom}\, f$, then because $pi(m, h) \wedge vmf\_com(he, f)$ holds, $e \in \mathrm{ran}\, hr$. Furthermore, by Lemma 22, there must exist an hr' such that $\mathrm{ran}(hr') = \mathrm{ran}(hr) \backslash \{e\}$ (i.e., $\langle e\rangle \frown hr'$ is a permutation of hr) and $FC(h \frown \langle e\rangle \frown hr', f')$ holds.
- Otherwise, i.e., $m \notin \mathrm{dom}\, f$, we have

$$\begin{array}{ll} FC(he, f) & \\ \Rightarrow \exists f' \bullet FC(he \frown \langle e\rangle, f') & \text{, by Lemma 21} \\ \Leftrightarrow \exists f' \bullet FC(h \frown hr \frown \langle e\rangle, f') & \text{, definition of } he \\ \Leftrightarrow \exists f'' \bullet FC(h \frown \langle e\rangle \frown hr, f') & \text{, Lemma 22} \end{array}$$ ◀

Compositionality refers to histories of multiple concurrent objects [22]. To formalise this, one must consider histories in which the object corresponding to each event may be distinguished, and hence the event types above must be extended with object names. We assume $e.obj$ returns the object corresponding to event e. For an object z and history h, we let $h_{|z}$ denote the subhistory of h with all events of object z. Prior to our new result that fence consistency is compositional (Theorem 24), we give a new proof of compositionality for linearizability.

▶ **Theorem 24** (Linearizability is compositional)**.** *For any history h, there exists an extension he of h and a mapping function f such that $LIN(he, f)$ if, and only if, for each object z, there exists an extension he_z of $h_{|z}$ and a mapping function f_z such that $LIN(he_z, f_z)$.*

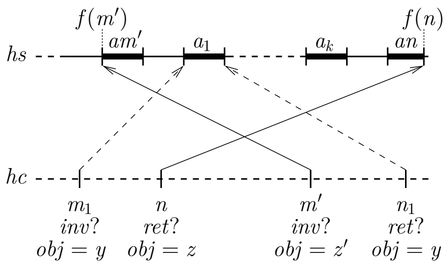

Figure 6 Assumed minimal reordering.

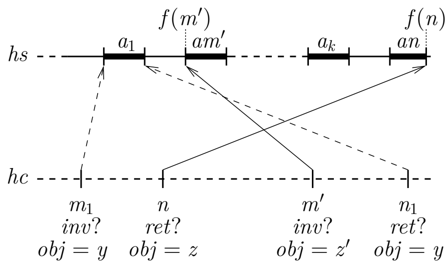

Figure 7 Swapped ordering.

Proof. The only if direction is trivial.

For the other direction, for each object z, suppose $LIN(he_z, f_z)$ holds for some extension he_z of $h_{|z}$ and mapping function f_z. The proof is by contradiction. Suppose that for every extension he of h, and every mapping f to a sequential history, we have:

$$\begin{aligned} & \neg LIN(he, f) \\ = \ & \neg VMF(he, f) \vee \neg lin(he, f) \vee \neg total(he, f) && \text{, by definition} \\ = \ & VMF(he, f) \wedge total(he, f) \Rightarrow \neg lin(he, f) && \text{, by logic} \end{aligned}$$

Thus, we assume $VMF(he, f) \wedge total(he, f)$ and prove $\neg lin(he, f)$.

For $\neg lin(he, f)$ to hold, by definition, there must exist indices n, m' in he such that $n < m' \wedge ret?(he(n)) \wedge inv?(he(m')) \wedge f(n) > f(m')$, i.e., n and m' are indices of operation calls where $he(n)$ returns before $he(m')$, and f reorders these calls when mapping to hs. Suppose $he(n).obj = z$ and $he(m').obj = z'$, and let an and am' be the operation calls corresponding to $he(n)$ and $he(m')$ in hs, respectively. If $z = z'$, we get an immediate contradiction to the assumption that there exists an he_z and f_z such that $LIN(he_z, f_z)$ holds. Therefore, we assume $z \neq z'$.

Now, pick an f such that the number of reordered operation calls that invalidate $lin(he, f)$ are minimal, then pick $n, m' \in \mathrm{dom}\, f$ such that $f(n) - f(m')$ is minimal and n, m' violate $lin(he, f)$. Because $ret?(he(n)) \wedge inv?(he(m'))$, the smallest possible value of $f(n) - f(m')$ is 3, which occurs if, in hs, the operation call an occurs immediately after am'. However, in this case, because $z \neq z'$, operation calls an and am' commute, i.e., there must exist another valid sequential history in which the order of an and am' are swapped, and we obtain a contradiction to $f(n) > f(m')$. Therefore, assume $f(n) - f(m') > 3$, i.e., some finite number of operation calls $a_1, a_2, \ldots a_k$, occur between $f(m')$ and $f(n)$ in hs.

Consider a_1, and suppose the invocation/return events corresponding to a_1 occur at m_1 and n_1 in he, respectively. We must have $m_1 < n \wedge m' < n_1$, otherwise, we obtain a contradiction to minimality of $f(n) - f(m')$ (see Fig. 6). Let $y = a_1.obj$ be the object corresponding to a_1. We now have two cases.

- Case $y \neq z'$. Because operations of different objects commute, calls a_1 and am' may be swapped in hs, to produce another valid sequential history hs' and mapping f' such that $f'(n) - f'(m') < f(n) - f(m')$, contradicting minimality of $f(n) - f(m')$ (see Fig. 7).
- Case $y = z'$. Here, a_1 and am' may not be swapped, however, we must have $a_2.obj = a_3.obj = \cdots = a_k.obj = z'$, otherwise, we may swap the first a_i such that $a_i.obj \neq z'$ with each of $am', a_1 \ldots a_{i-1}$ to again contradict minimality of $f(n) - f(m')$. However, we now have $a_k.obj = z'$ and $an.obj = z$, i.e., $a_k.obj \neq an.obj$, so a_k and an may be swapped to produce a valid sequential history, giving us our final contradiction. ◀

▶ **Theorem 25** (Fence consistency is compositional). *For any history h, there exists an extension he of h and a mapping function f such that $FC(he, f)$ if, and only if, for each object z, there exists an extension he_z of $h_{|z}$ and a mapping function f_z such that $FC(he_z, f_z)$.*

Proof. The only if direction is trivial.

For the other direction, for each object z, suppose $FC(he_z, f_z)$ holds for some extension he_z of $h_{|z}$ and mapping function f_z. The proof is by contradiction. Suppose that for every extension he of h, and every mapping f to a sequential history, we have:

$$\begin{aligned} & \neg FC(he, f) \\ = {} & \neg\, VMF(he, f) \vee \neg sc(he, f) \vee \neg fc\_ord(he, f) \vee \neg fc\_com(he, f) && \text{, by definition} \\ = {} & VMF(he, f) \wedge sc(he, f) \wedge fc\_com(he, f) \Rightarrow \neg fc\_ord(he, f) && \text{, by logic} \end{aligned}$$

Assuming $VMF(he, f) \wedge sc(he, f) \wedge fc\_com(he, f)$ holds, we attempt to prove $\neg fc\_ord(he, f)$. For $\neg fc\_ord(he, f)$, by definition, there must exist indices n, m' in he such that

$$n < l < m' \wedge f(n) > f(m') \tag{6}$$

$$ret?(he(n)) \wedge empty?(he(l)) \wedge inv?(he(m')) \wedge he(n).pr = he(l).pr \tag{7}$$

By (6), f reorders operation calls $he(n)$ and $he(m')$, and by (7), n, l and m' are indices corresponding to a return, empty and invocation, respectively and both $he(n)$ and $he(l)$ correspond to the same process. We assume $he(n).obj = z$ and $he(m').obj = z'$, and let an and am' be the operation calls corresponding to $he(n)$ and $he(m')$ in hs, respectively. If $z = z'$, we get an immediate contradiction to the existence of an extension he_z of $h_{|z}$ and mapping function f_z such that $FC(he_z, f_z)$. Therefore, we assume $z \neq z'$.

Pick an f as well as indices n and m' as in Theorem 24. For the minimal value of $f(n) - f(m')$ (i.e., if $f(n) - f(m') = 3$) we obtain a contradiction as in Theorem 24. Therefore, assume $f(n) - f(m') > 3$, i.e., some finite number of operation calls $a_1, a_2, \ldots a_k$, occur between $f(m')$ and $f(n)$ in hs. Consider a_k, and suppose the invocation/return events corresponding to a_k occur at m_k and n_k in he, respectively. We have that $a_k.obj \neq z$ (otherwise we get a contradiction to minimality by swapping a_k and am') and cases:

- If $l < m_k$ or $n_k < l$, because $a_k.obj \neq z$ we obtain an immediate contradiction to the assumption that an and am' are different objects such that $f(n) - f(m')$ is minimal.
- Else if $m_k < n \wedge m' < n_k$, the proof proceeds as in Theorem 24.
- Else if $n < m_k < l$, then $a_k.obj = z$, otherwise we can swap a_k and an to contradict minimality of $f(n) - f(m')$. In fact $a_i.obj = z$ must hold for all $1 \leq i \leq k$. But now $am'.obj \neq a_1.obj$, and hence can be swapped, once again contradicting minimality.
- Finally, if $l < n_k < m'$, we obtain our final contradiction to minimality of $f(n) - f(m')$ using a similar argument to $n < m_k < l$. ◀

5 A Correctness Condition Hierarchy

As we've seen, there are several different correctness conditions that are appropriate for different types of algorithms over different memory architectures. The literature includes many others (e.g., k-linearizability [20], eventual consistency [36], quantitative quiescent consistency [25]), which we have not covered in this paper.

Using our framework, for the conditions we have considered, it is possible to formally establish a hierarchy based on order and commitment properties. We first link ξ-quiescent, fence and flush consistency. Fence and flush consistency are defined on histories that consider slightly different aspects of a system's behaviour. To relate the two conditions, we consider histories in which writes and flushes as well as buffer empty events are recorded. To this end, we define $Event_\xi ::= Event \mid \xi\langle\langle P\rangle\rangle$, $Event_\phi ::= Event \mid \omega\langle\langle P\rangle\rangle \mid \phi\langle\langle P\rangle\rangle$ and $Event_C ::= Event_\xi \mid Event_\phi$, with the understanding that $\langle\phi(p), \xi(p)\rangle$ is concatenated to the history if the flush $\phi(p)$ causes the buffer of process p to become empty, while $\langle ret(p, op, out), \xi(p)\rangle$ is concatenated whenever $ret(p, op, out)$ occurs and the buffer of p is empty. A history h is *legal* if both $h_{|Event_\phi}$ and $h_{|Event_\xi}$ are legal. This definition of *legal* is also extension closed.

▶ **Proposition 26.**
1. *$FLC \Rrightarrow FC$, but not vice versa, and*
2. *$FC \Rrightarrow QC_\xi$, but not vice versa.*

Proof.
1. The forward direction follows by expanding the definitions of FLC and FC, then using the fact that both $wflc\_ord \Rrightarrow fc\_ord$ and $wflc\_com_1 \wedge wflc\_com_2 \Rrightarrow fc\_com$. To show the other direction does not hold, consider the following history, where $p, q \in P$ and `enq`, `deq` are enqueue and dequeue operations on a concurrent queue, respectively. The history is clearly a legal fence consistent history, but it is not flush consistent.
$$\langle inv(p, \texttt{enq}, 1), \omega(p), ret(p, \texttt{enq}), inv(p, \texttt{enq}, 2), \omega(p), \phi(p), inv(q, \texttt{deq}), \\ ret(p, \texttt{enq}), \phi(p), \xi(p), ret(q, \texttt{deq}, emp), \xi(q)\rangle$$
2. The forward direction follows by expanding the definitions of FC and WQC_ξ, then using the fact that both $fc\_ord \Rrightarrow wqc\_ord_\xi$ and $fc\_com \Rrightarrow wqc\_com_\xi$ hold.
To show the other direction does not hold, consider the following history, where $q_1, q_2, q_3 \in P$ and `enq`, `deq` are enqueue and dequeue operations on a concurrent queue, respectively. The history is clearly ξ-quiescent consistent, but not fence consistent.
$$\langle inv(q_1, \texttt{enq}, 1), ret(q_1, \texttt{enq}), inv(q_2, \texttt{enq}, 2), \xi(q_1), inv(q_3, \texttt{enq}, 3), ret(q_2, \texttt{enq}), \\ \xi(q_2), ret(q_3, \texttt{enq}), \xi(q_3), inv(q_1, \texttt{deq}), ret(q_1, \texttt{deq}, 3)\rangle$$

◀

It is also possible to link correctness conditions on totally ordered memory with those on TSO memory. It is straightforward to prove the following propositions.

▶ **Proposition 27.** *If h is legal, then for any mapping function f, $LIN(h, f) \Rightarrow FLC(h, f)$.*

One may think of totally ordered memory as being a special case of TSO memory where the buffer is always empty. Here, using the strategy for recording histories of type $History_\xi$ described in Section 4.2, under totally ordered memory, the return for each process p is immediately followed by $\xi(p)$, i.e.,

$$ret\_emp(h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h, p : P \bullet ret_p?(h(n)) \Rightarrow n+1 \in \mathrm{dom}\, h \wedge empty_p?(h(n+1))$$

▶ **Proposition 28.** *If h is legal and $ret\_emp(h)$ holds, then for any mapping function f, $LIN(h, f) \Leftrightarrow FC(h, f)$ and $QC(h, f) \Leftrightarrow WQC_\xi(h, f)$.*

One may also think of totally ordered memory as being a type of TSO memory where flushes occur immediately after each write. Here, using the strategy for recording histories of type $History_\phi$ defined in Section 4.4, we obtain the following property, which states that for any process p, a flush for p immediately follows each write by p.

$$imm\_fl(h) \quad \widehat{=} \quad \forall\, n : \mathrm{dom}\, h, p : P \bullet \omega_p?(h(n)) \Rightarrow n+1 \in \mathrm{dom}\, h \wedge \phi_p(h(n+1))$$

▶ **Proposition 29.** *Suppose $h \in History_\phi$ is legal and $imm\_fl(h)$ holds. Then for any mapping function f, $LIN(h, f) \Leftrightarrow FLC(h, f)$.*

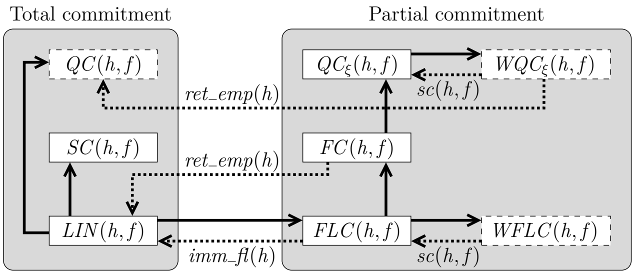

Figure 8 Reltionships between correctness conditions for a history h and mapping function f. The arrows represent implication with dashed versions representing conditional implication; the label on each arrow represents the required condition. The conditions within solid boxes ensure sequential consistency.

An overview of the hierarchy is presented in Fig. 8, where we assume h is a legal history and f a mapping function. Each solid arrow in Fig. 8 denotes implication, and each dashed arrow denotes conditional implication with the condition corresponding to the label. Note that Fig. 8 allows one to easily deduce transitivity properties, e.g., if $FC(h, f)$ and $ret\_emp(h)$ hold, then $SC(h, f)$ holds.

6 Conclusions

Correctness of a concurrent object is defined with respect to a correctness condition, which is a relation on its behaviours against those of a sequential specification object. Algorithms implementing such objects must cope with the additional challenges of distributed memory; the low-level effects of write buffers in most modern processors (e.g., x86, ARM) only provide *relaxed memory* guarantees. The end goals of programmers that use concurrent objects and designers that develop algorithm for concurrent objects differ, a large number of correctness conditions have been defined in the literature. We have provided a framework within which these can be formalised, which in turn allows the relative strengths of different conditions to be compared.

Within our framework, we have defined well-known conditions for totally ordered memory (sequential consistency, linearizability and quiescent consistency), as well as newly developed conditions for TSO memory (weak ξ-quiescent consistency and weak flush consistency). To characterise implementations that are also sequentially consistent, we define stricter variations of both conditions that guarantee sequential consistency. We identify and develop a new compositional condition for TSO architectures, fence consistency, which is weaker than flush consistency, but stricter than ξ-quiescent consistency. Notable in our framework is the capability of specifying partial commitment properties, which provide the flexibility needed to cope with delayed operation effects due to relaxed memory.

The study of correctness conditions for concurrent objects under relaxed memory is new [12, 32, 35, 17, 6]. Of these, [12, 17, 6, 35] consider linearizability, but [17, 6] facilitate optimisation (via `fence` removal) by weakening the abstract specification, while [12, 35] weaken definition of linearizability so that the interval of execution for an operation is expanded. We believe flush consistency to be equivalent to the weaker definition of linearizability given in [35], however because Travkin et al. [35] do not provide a formal definition, this is difficult to verify. Jagadeesan et al [24] provide a framework that enables one to develop correctness conditions for many different relaxed memory models by decoupling buffer effects from the correctness conditions at hand, however, they only formalise sequential consistency and linearizability. More recently, Batty et al. [5] have developed methods for proving observational refinement directly for C11 specifications, which are weaker than TSO. They develop a compositional correctness condition that is stricter than linearizability when C11 is restricted to totally ordered behaviours. Using our framework to formalise their correctness condition to compare them to the conditions in this paper is a subject of further study. Future work will also consider correctness conditions for software transactional memory [19].

This paper has mainly considered correctness conditions from an algorithm designer's perspective. Satisfying the requirements of programmers introduces another dimension to this problem (e.g., [18]). Our work does not differ from, say [17, 6], in that observational refinement is only assured for data independent clients. Extending our results to cope with other real-world issues such as ownership transfer (like [17, 6]) is a subject of future work.

References

1 S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

2 J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.

3 H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In T. Ball and M. Sagiv, editors, *POPL*, pages 487–498. ACM, 2011.

4 H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

5 M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 235–248. ACM, 2013.

6 S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In H. Seidl, editor, *ESOP 2012*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

7 D. Chase and Y. Lev. Dynamic circular work-stealing deque. In P. B. Gibbons and P. G. Spirakis, editors, *SPAA*, pages 21–28. ACM, 2005.

8 R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set. In *CAV*, volume 4144 of *LNCS 4144*, pages 475–488. Springer, 2006.

9 J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM*, pages 200–214. Springer, 2014.

10 J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.

11 J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisabilty with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.

12 J. Derrick, G. Smith, and B. Dongol. Verifying linearizability on TSO architectures. In E. Albert and E. Sekerinski, editors, *iFM*, pages 341–356. Springer, 2014.

13 S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In P. B. Gibbons and M. Adler, editors, *SPAA*, pages 216–224. ACM, 2004.

14 B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under Total Store Order memory. In Z. Liu, J. Woodcock, and H. Zhu, editors, *ICTAC*, volume 8049 of *LNCS*, pages 177–194. Springer, 2013.

15 I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

16 M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In J. W. Davidson, K. D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 212–223. ACM, 1998.

17 A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In M. Aguilera, editor, *DISC 2012*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.

18 A. Gotsman and H. Yang. Linearizability with ownership transfer. *Logical Methods in Computer Science*, 9(3), 2013.

19 R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

20 T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and Sokolova A. Quantitative relaxation of concurrent data structures. In R. Giacobazzi and R.dhia Cousot, editors, *POPL*, pages 317–328. ACM, 2013.

21 T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR*, pages 242–256. Springer, 2013.

22 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

23 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

24 R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness – compositional reasoning under relaxed memory models (extended abstract). In Felleisen M and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 492–511. Springer, 2013.

25 R. Jagadeesan and J. Riely. Between linearizability and quiescent consistency – quantitative quiescent consistency. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *ICALP II*, volume 8573 of *LNCS*, pages 220–231. Springer, 2014.

26 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

27 F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In J. Vitek, H. Lin, and F. Tip, editors, *PLDI*, pages 429–440. ACM, 2012.

28 S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In T. D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

29 G. Schellhorn, H. Wehrheim, and J. Derrick. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. on Computational Logic*, 2014.

30 P. Sewell, S. Sarkar, S. Owens, F.Z. Nardelli, and M.O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

31 N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

32 G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *FACS*. Springer, 2014. To appear (accepted 21 July, 2014).

33 D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.

34 J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

35 O. Travkin, A. Mütze, and H. Wehrheim. SPIN as a linearizability checker under weak memory models. In V. Bertacco and A. Legay, editors, *HVC*, volume 8244 of *LNCS*, pages 311–326. Springer, 2013.

36 W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.