Persistence Semantics for Weak Memory

Integrating Epoch Persistency with the TSO Memory Model

AZALEA RAAD, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

Emerging non-volatile memory (NVM) technologies promise the durability of disks with the performance of volatile memory (RAM). To describe the persistency guarantees of NVM, several *memory persistency* models have been proposed in the literature. However, the formal semantics of such persistency models in the context of existing mainstream hardware has been unexplored to date. To close this gap, we integrate the *buffered epoch persistency* model with the 'total-store-order' (TSO) memory model of the x86 and SPARC architectures. We thus develop the *PTSO* (*'persistent' TSO*) model and formalise its semantics both operationally and declaratively. We demonstrate that the two characterisations of PTSO are equivalent. We then formulate the notion of *persistent linearisability* to establish the correctness of library implementations in the context of persistent memory. To showcase our formalism, we develop two persistent implementations of a queue library, and apply persistent linearisability to show their correctness.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Theory of computation → Semantics and reasoning**; **Concurrency**;

Additional Key Words and Phrases: weak memory, x86-TSO, non-volatile memory, persistency models, epoch persistency, durable linearisability

ACM Reference Format:
Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (November 2018), 27 pages. https://doi.org/10.1145/3276507

1 INTRODUCTION

Computer storage is traditionally divided into two categories: (1) fast *volatile* byte-addressable memory (e.g. SRAM and DRAM), which loses its contents in case of a power failure; and (2) slow *persistent* block-addressable storage (e.g. hard drives and flash memory), which preserves its contents in case of a power failure. Due to this split, applications typically maintain their data structures in memory and periodically write important data to disk (often in a serialised format).

There are, however, two technology trends that may well render this dichotomy obsolete, and may thus greatly affect how applications are structured. First, the size of memory has been steadily increasing; it is now common to have fairly large applications, such as in-memory databases, operate with their entire data in memory [IBM 2013; TimesTen Team 1999; Wang et al. 2014]. Second, emerging new technologies in *non-volatile memory* (NVM) such as PCM [Lee et al. 2009], STT-RAM [Kawahara et al. 2012] and memristors [Strukov et al. 2008], make it possible for processors to access data guaranteed to persist a power failure at byte-level granularity and at performance comparable to regular (volatile) RAM. It is widely believed that volatile memory will be augmented,

Authors' addresses: Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany, azalea@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2018/11-ART137
https://doi.org/10.1145/3276507

and eventually replaced with non-volatile memory, allowing for efficient access to persistent data [Intel 2014; International technology roadmap for semiconductors 2007; Pelley et al. 2014]. As such, non-volatile memory has become a popular research topic [Boehm and Chakrabarti 2016; Chakrabarti et al. 2014; Chatzistergiou et al. 2015; Coburn et al. 2011; Izraelevitz et al. 2016a; Kolli et al. 2016a,b; Nawab et al. 2017; Volos et al. 2011; Wu and Reddy 2011; Zhao et al. 2013].

Using persistent memory correctly, however, is by no means easy. A key challenge lies in ensuring correct recovery after a crash (e.g. a power failure) by maintaining the consistency of the data structures in persistent memory. This in turn requires a clear understanding of the order in which stores are propagated to the non-volatile memory. The problem is that CPUs are not directly connected to memory; instead there are multiple non-persistent caches in between. As such, memory stores are not propagated to memory at the time and in the order that the processor issues them, but rather at a later time and in the somewhat arbitrary order decided by the cache coherence protocol. This can lead to perhaps surprising outcomes. For instance, consider the simple sequential program $x := 1; y := 1$, running to completion and subsequently crashing due to a power failure. On restarting the computer, the memory may well contain $y = 1$ and yet $x = 0$; i.e. the write $x := 1$ may not have propagated to memory before the power failure.

To ensure correct recovery, one must thus control the order in which the stores are propagated to persistent memory. One simple way to do this is by requiring *strict persistency* [Pelley et al. 2014]. Under strict persistency, stores are persisted to memory in an order allowed by the underlying memory consistency model. For example, under the x86 or the ARM consistency models [Pulte et al. 2017; Sewell et al. 2010] each processor's stores would be persisted in the same order they become visible to other processors. However as Condit et al. [2009] argue, this coupling of persistence and visibility puts persistence in the critical execution path, degrading performance significantly.

To regain good performance, Pelley et al. [2014] relaxed the notion of persistency by decoupling it from the visibility of stores. They introduced the *epoch persistency* model, in which the execution of each thread is divided into several *epochs*, separated by *persist barriers*. A persist barrier ensures that stores prior to the barrier are persisted to memory before those after the barrier. In the example above, inserting a persist barrier after $x := 1$ ensures that the store to x is persisted before that of y, and thus one cannot observe the outcome $x = 0$ and $y = 1$ after a crash.

A naive way of implementing a persist barrier is to block execution until all pending stores are persisted to the memory. A more efficient way is to persist updates to memory *asynchronously* by an additional buffering layer [Condit et al. 2009; Izraelevitz et al. 2016b; Joshi et al. 2015]. This way, memory persists are buffered in a queue of write-backs to persistent memory, and execution may proceed ahead of persistence. When it is necessary to control the write-back of buffered persists explicitly (e.g. before performing I/O), a separate *syncing* instruction can be used to wait until all pending persists have been propagated and the persistent buffer is drained.

Several existing articles, including [Condit et al. 2009; Joshi et al. 2015; Kolli et al. 2016b], have investigated the feasibility of epoch persistency by studying the implementability of persist barriers and demonstrating their performance gain over other models. As such, we believe that (buffered) epoch persistency constitutes a viable and efficient persistency model. In this article, we thus formalise the buffered extension of epoch persistency by integrating it with the *'total-store-order'* (TSO) memory model. We choose the TSO memory model as the basis of our extension for two reasons: (1) it is a *mainstream* practical *weak* memory model (followed by the x86 and SPARC architectures); and (2) it has an intuitive operational semantics in terms of processor-local buffers [Sewell et al. 2010]. We call our formal model *PTSO* ('persistent TSO') and describe its semantics both *operationally* and *declaratively*, proving that the two characterisations are equivalent.

The reason for the two semantics is that while the operational semantics provides programmers with a more intuitive account of the hardware guarantees, the declarative one streamlines the construction of correctness proofs such as those for linearisability arguments [Herlihy and Wing 1990] or triangular race freedom [Owens 2010]. To construct our declarative semantics, we develop a general framework for describing declarative concurrency models in the context of persistent memory. We then present PTSO as an instance of this general framework and as an extension of the TSO model formalised by Sewell et al. [2010]. In the future, it should be possible to instantiate this framework for other consistency models, such as the ARM model [Pulte et al. 2017].

Next, we introduce a notion of correctness for persistent libraries in our general declarative framework for describing concurrency models in the presence of persistent hardware. To this end, we adapt the notion of buffered durable linearisability by Izraelevitz et al. [2016b], and describe a general proof pattern for constructing linearisability arguments in the presence of persistent memory, by identifying the linearisation points and inspecting their persistence.

To showcase the application of persistent linearisability, we develop two persistent implementations of a queue library: one as a simple lock-based implementation, other as a variant of the Michael-Scott queue implementation [Michael and Scott 1996]. In both cases, we develop a *recovery mechanism* restoring the queue data structure to a consistent state upon recovery. We then establish the correctness of both implementations by using our proof pattern for persistent linearisability.

Related Work. Although the existing literature on non-volatile memory has grown rapidly in the recent years, *formalising* persistency models, as well as *verifying* implementations under persistent memory, have largely remained unexplored to date.

On the formalisation side, the existing work on (buffered) epoch persistency has several limitations. First, there is little work on *formalising* the semantics of epoch persistency, especially in the context of weak memory models. Second, no existing work considers the *formal* integration of epoch persistency with the weak memory models of *mainstream architectures* (e.g. the x86 TSO model or the ARM memory model). Third, no existing work defines a formal *operational* semantics for epoch persistency. The latter can help provide a more intuitive description of persistence semantics and is thus especially important in making persistence programming accessible.

Pelley et al. [2014] *informally* describe the behaviour of epoch persistency under *sequentially consistent* (SC) machines; whilst Condit et al. [2009]; Joshi et al. [2015] do so under *'total-store-order'* (TSO) machines. However, neither work provides a *formal* semantics (declarative or operational) for epoch persistency. Izraelevitz et al. [2016b] give a declarative semantics for buffered epoch persistency under the release consistency model [Gharachorloo et al. 1990] using abstract executions. However, a more intuitive *operational* semantics is missing. The authors attribute their choice of the release consistency model to its relaxed nature; nevertheless, they do not explore the integration of epoch persistency with other weak memory models supported by *mainstream* hardware.

On the verification side, there is very little work exploring the correctness conditions of concurrent library implementations under persistent memory. Izraelevitz et al. [2016b] introduced the notion of *buffered durable linearisability* for linearisability under buffered epoch persistency. However, to our knowledge, neither the authors nor the existing literature have studied the application of buffered durable linearisability for showing the correctness of persistent library implementations.

Outline. The remainder of this article is organised as follows. In §2 we present an overview of our contributions and the necessary background information. In §3 we present the formal PTSO model (operational and declarative) . In §4 we formalise the notion of persistent linearisability for PTSO and present a persistent queue library implementation. In §5 we present a persistent variant of the Michael-Scott queue implementation. Finally, we discuss future work and conclude in §6.

2 OVERVIEW

We proceed with an overview of our contributions as outlined in §1.

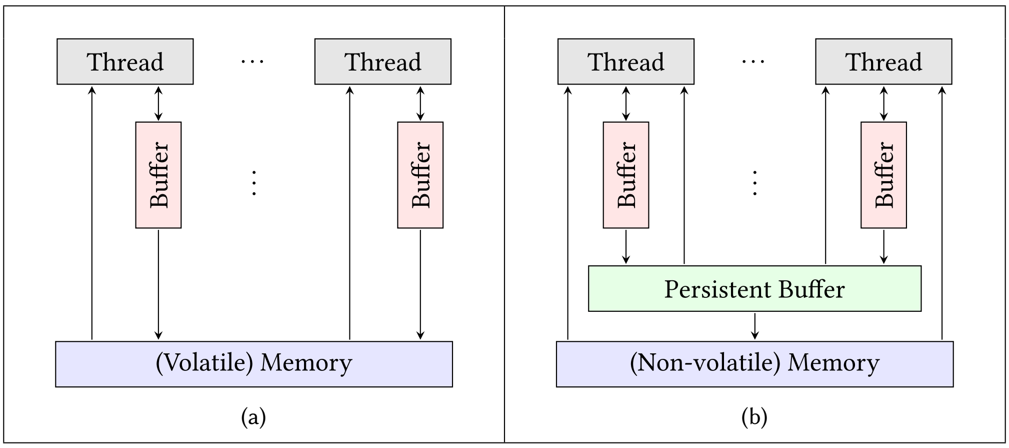

Fig. 1. The storage subsystems of the TSO (left) and PTSO (right) memory models

2.1 The PTSO Memory Model

We present an intuitive account of the PTSO memory model as an extension of the TSO memory model integrated with buffered epoch persistency.

Total Store Ordering (TSO). First introduced by the SPARC architecture [SPARC 1992], TSO is the consistency model used by the x86 architecture (both Intel and AMD) [Sewell et al. 2010]. As illustrated in Fig. 1a, in the TSO model, each thread is connected to the main (volatile) memory via a FIFO *buffer*. When a thread writes a value to a location, the write is recorded only in its buffer. When a thread reads from a location, it first consults its own buffer. If it contains buffered writes for that location, the thread reads the value of the *last* buffered write to that location; otherwise, it consults the main memory. Threads can *unbuffer* their writes by propagating them (in FIFO order) to the main memory at non-deterministic points in time. To control the unbuffering of writes, programmers can appeal to *memory barriers*. Upon encountering a memory barrier, the buffer of the executing thread is flushed to the memory, unbuffering all its pending writes.

Memory Persistency. Declarative (a.k.a. axiomatic) memory *consistency* models describe the permitted behaviours of programs by ensuring that operations on memory follow specific rules. This is done by defining a *volatile memory order*. The volatile memory order (e.g. the 'total-store-order' in case of TSO) defines the permitted behaviours of programs by constraining the visible order of loads and stores to the volatile memory (i.e. allowable visible memory states) between processors or cores. This in turn allows memory operations to be reordered, while preserving the intended program behaviour. Analogously, memory *persistency* models describe the permitted behaviours of programs upon *recovering* from a crash (e.g. due to a power failure), by defining a *persistent memory order* [Pelley et al. 2014]. The persistent memory order constrains the visible order in which stores are committed to the persistent memory. Any pair of stores ordered by persistent memory order may not be observed out of that order upon crash recovery. To distinguish the volatile and persistent memory order, memory *stores* are differentiated from memory *persists*. The former denotes the process of making a store visible to other processors, whereas the latter denotes the process of committing stores durably to non-volatile memory. As with consistency models, persistency models may be strict or relaxed.

Under strict persistency, the persist order observes all happens-before relations implied by the volatile memory order; that is, the volatile and persistent memory orders coincide. In case of the TSO model, this means that the TSO order also prescribes the order in which stores are propagated to persistent memory. As such, all persists are ordered with respect to the program order of the issuing thread. However, strict persistency introduces many persist dependencies unnecessary for correct recovery. The most common unnecessary dependency occurs due to the program-order constraint of TSO for two stores. Programs frequently persist to large, contiguous regions of memory that logically represent single objects, but which cannot persist atomically (due to their size).

To remedy this, Pelley et al. [2014] propose *relaxed* persistency models where the volatile and persistent memory orders are separated. As an instance of relaxed persistency, Pelley et al. [2014] introduce the *epoch persistency model.*

Epoch Persistency. The epoch persistency model was introduced by Pelley et al. [2014], inspired by the work of Condit et al. [2009]. Under epoch persistency, the execution of each thread is separated into persistence *epochs* by *persist barrier* instructions, written as **pfence**. Any two memory stores on the same thread and separated by a persist barrier are ordered by the persistent order. Persist barriers enforce that no memory stores after the barrier are persisted before those prior to the barrier. However, memory stores within the same epoch (not separated by a barrier) are concurrent and may be reordered or occur in parallel. Epoch persistency inherits some constraints from the volatile memory order: any two memory stores to the *same* location assume the order from the volatile memory order. Consequently, two persists to the same location are always ordered even if they occur in the same epoch. In order to improve performance, *persist buffering* [Condit et al. 2009; Izraelevitz et al. 2016b; Joshi et al. 2015] has been proposed to allow memory persists to occur *asynchronously.* That is, memory persists are buffered in a queue of write-backs to persistent memory. This way, persists occur after their corresponding stores and as prescribed by the persistent memory order; however, execution may proceed ahead of persistence. As such, **pfence** instructions may complete without waiting for confirmation that the stores have indeed been persisted to non-volatile memory. When it is necessary to control the write-back of buffered persists explicitly (e.g. before performing I/O), a separate *syncing* instruction, **psync**, can be used to wait until all pending persists have been committed and the persistent buffer is drained.

The Operational PTSO Model. We develop the *PTSO* ('persistent TSO') memory model by integrating (buffered) epoch persistency described above with the TSO consistency model. As illustrated in Fig. 1b, the PTSO storage system has an additional layer compared to its TSO counterpart, namely the *persistent buffer* described above. To capture persistency epochs, the persistent buffer is modelled as a FIFO queue of persistent *sub-buffers*, each representing a distinct epoch. As noted above, two stores in the same epoch may be persisted (i) in any order when they are to distinct locations; or (ii) in the volatile memory order, namely the TSO order, when they are to the same location. As such, each sub-buffer is modelled as a map associating each location with a FIFO queue.

The persistent buffer sits between the per-thread buffers and the main (non-volatile) memory and plays the same role as that of volatile memory in the TSO model with respect to per-thread buffers. More concretely, when a thread writes a value to a location, the write is recorded in its buffer as before. Threads can unbuffer their writes (in the FIFO order) by propagating them to the *persistent buffer* at non-deterministic points in time. Analogously, memory barriers (**fence** instructions) can be used to enforce the unbuffering of writes to the persistent buffer. The persistent buffer in turn may unbuffer the pending stores by propagating them (in the epoch order, while respecting the per-location order in each epoch) to the non-volatile memory at non-deterministic points in time. Persistent barriers (**pfence** instructions) can be used to enforce epoch ordering by introducing a new epoch in the persistent buffer; whilst persistent syncs (**psync** instructions) can be used to enforce the unbuffering of all pending persists to the memory and draining the persistent buffer. As before, when a thread reads from a location, it first inspects its own buffer. If it contains buffered writes for that location, then the value of the last buffered write to that location is read; otherwise, it consults the *persistent buffer*. Similarly, if the persistent buffer contains buffered stores for that location, then the value of the *last* buffered store to that location in the *latest* epoch is read; otherwise the thread consults the main memory.

The Declarative PTSO Model. We describe a general framework for declarative concurrency models in the context of persistent memory. We then present PTSO as an instance of this general definition. To model crashing programs, we define an *execution chain* $\mathcal{E}$ as a sequence $G_1; \cdots; G_n$, with each G_i describing an execution *era* between two adjacent crashes. Each execution era G_i denotes the traces of shared memory accesses generated by the program in that era. As is standard in the literature of declarative concurrency models, each G_i comprises a set of memory events (the G_i nodes), and a number of relations on events (the G_i edges). To capture the orderings imposed by persistent barrier and sync instructions, each G_i includes a *'non-volatile-order'* relation, nvo, constraining the persist order of write/update events.

2.2 Correctness of Persistent Libraries

Implementing and verifying concurrent libraries is a challenging undertaking. Traditionally, the library *implementer* is tasked with ensuring the underlying library state (e.g. a queue) remains *consistent* (e.g. the queue maintains its FIFO property), when accessed by multiple threads simultaneously. This is achieved via a suitable synchronisation mechanism, such as transactions, locks or atomic instructions, to control the concurrent accesses to shared data. On the other hand, the library *verifier* is tasked with identifying the appropriate proof techniques to formalise and establish the desired consistency guarantees. One well-known such technique is that of *linearisability* proofs as proposed by Herlihy and Wing [1990], and has been extensively used in the verification literature.

Both tasks of implementing and verifying concurrent libraries are further compounded in the presence of non-volatile hardware. Library implementers must ensure the library state remains both *consistent* and *persistent* in the presence of *crashes*. This in turn requires a clear understanding of the persistent ordering guarantees afforded by the underlying hardware. The library verifiers must accordingly adapt their arsenal of formal techniques to establish the desired *persistence* properties. In order to reason about the correctness of persistent library implementations, Izraelevitz et al. [2016b] introduced the notion of *buffered durable linearisability* as an extension of linearisability under buffered epoch persistency.

Linearisability. In traditional linearisability proofs, a library call is represented by two *call events*, *inv* and *ack*, referred to as a matching pair, corresponding to the call invocation (making the call) and acknowledgement (returning from the call). The traces of library client programs are then represented as *histories* or sequences of events. As the library is concurrently accessed by multiple threads, in a history the *inv* and *ack* events of a call by one thread may be interleaved with those of others. Intuitively, a history H is *sequential* if the events of a matching pair are not interleaved by other call events in H. A history H is *linearisable* if: 1) H can be extended (completed) to some history H_c (by appending zero or more *ack* events); 2) H_c can be truncated to a history H_t by removing every *inv* in H_t without a matching *ack*; and 3) H_t is equivalent to a *legal* sequential history H_s that preserves the 'real-time' ordering of calls in H. Intuitively, extending H to H_c captures the notion that a pending *inv* may have taken effect even though its matching *ack* has not yet been returned to the caller. Truncating H_c to H_t captures the notion that the remaining pending invocations have not yet had an effect. The notion of legal histories is library-specific; e.g. the FIFO property of queue histories. In the context of weak memory models, the 'real-time'

```
c.inc(v) ≜
  pc:=getPC(); t:=getTC();
  m:=max(v,0); lock(c); o:=c.val;
  map[t][pc]:=o+m; pfence;
  if (v > 0) {
    c.val:=o+v; pfence;
  } unlock(c); return o;

getProgress(t) ≜
  pc:=-1; v:=⊥;
  while (map[t][pc+1] != ⊥) {
    pc++; v:=map[t][pc]; }
  return (pc,v);
```

```
recover() ≜
// check that map and c are initialised;
// code redacted
  for(t in P) {
    (pc,v):=getProgress(t);
    if (pc>=0 && c.val>=v)
       P'[t]:=sub(P[t],pc+1);
    else if (pc>=0)
       P'[t]:=sub(P[t],pc);
    else P'[t]:=P[t];
  }
  run(P');
```

Fig. 2. A persistent counter implementation and its recovery mechanism with persistence code in blue

order in the linearisability definition is replaced with the weaker 'happens-before' order, defined specifically for the model considered.

Persistent linearisability. Buffered durable linearisability [Izraelevitz et al. 2016b], henceforth *persistent linearisability*, captures the notion of linearisability under (buffered) epoch persistency. When the execution of a library client program crashes $n-1$ times, its execution comprises n *eras*, where an era spans the execution period between two adjacent crashes. As such, when the execution crashes $n-1$ times, its trace is given by a *history chain* $H_1; \cdots ; H_n$, with each H_i denoting the history in the i^{th} era. Due to the asynchronous nature of persists in buffered epoch persistency, in each era (except the last) only a subset of stores may persist prior to the crash. As such, a history chain $H_1; \cdots ; H_n$ is *persistently linearisable* if there exist $P_1 \cdots P_{n-1}$ such that: (i) each P_i is a *sub-history* (prefix) of H_i; and (ii) $P_1 \cdots P_{n-1}; H_n$ is linearisable. This captures the notion that those events in the portion of H_i after P_i were buffered but not persisted before the crash.

In what follows, we present a persistent implementation of a pedagogical counter and describe intuitively how we can show its correctness by constructing a persistent linearisability proof.

2.2.1 A Persistently Linearisable Counter in PTSO. In Fig. 2 we present a persistent implementation of a simple counter library (left) and its *recovery mechanism* (right), discussed shortly. We consider a counter library with a single operation, `inc(v)`, for incrementing the value of the counter by `v`, where `v` is non-negative. We use a coarse-grained lock to control concurrent accesses to the counter. A counter at location `c` thus comprises two components, represented as two adjacent cells: (i) the counter *lock* at `c`; and (ii) the counter *value* at `c+1`, written `c.val`.

The lock on `c` is acquired by calling `lock(c)`; dually, the lock on `c` is released by calling `unlock(c)`. The lock implementation is straightforward and is elided here; we present the lock implementation later in §4.1. Ignoring the code in blue, the `inc(v)` implementation is straightforward. A call to `inc(v)` acquires the counter lock, increments its value by `v` (when `v` is non-negative), and finally releases the lock, returning the old value of the counter. That is, `m=max(v,0)` denotes the increment value as either `v` when `v` is non-negative, or 0 when `v` is a negative value.

Simplifying Assumptions. We assume that a given counter client program P is of the form $\mathtt{c}_0 || \cdots || \mathtt{c}_k$, where each $\mathtt{c}_i$ is of the form $o^i_0; \cdots ; o^i_l$, and each o^i_j is a counter operation (i.e. `inc`). We thus represent each $\mathtt{c}_i$ as an array $\mathtt{C}_i$ of length $l+1$, with each $\mathtt{C}_i[j] = o^i_j$. We then represent P as an array of length k+1 at location P, with P$[i]$ = C$_i$.[1] A client program P is executed by calling run(P). A call to run(P) spawns k+1 threads $\tau_0 \cdots \tau_k$ and sets up their contexts, with each τ_i executing C$_i$. We further assume that the context of each thread τ_i is set up such that: (1) a call to getTC() returns i; and (2) a call to getPC() returns the *'progress counter'* (or 'program counter'), namely the index of the counter operation in C$_i$ currently under execution (i.e. j when executing o^i_j).

Persistence of inc. To ensure correct crash recovery, our implementation must defensively account for the possibility of a crash at each program point. To do this, we record the relevant metadata for tracking the progress of each thread in a map of length k+1 at map, one entry per thread. When the ith thread contains l+1 instructions (i.e. c$_i$ is of the form $o^i_0; \cdots; o^i_l$), then the map entry associated with thread τ_i (i.e. map$[i]$) is an array of length l+1, with one entry per instruction. When τ_i executes its jth increment operation inc(v) with m=max(v,0), then map$[i][j]$ is updated to (o+m), where o denotes the value of the counter immediately before the increment; i.e. o+m denotes the counter value right after the increment. This is done on line 4, where the subsequent pfence instruction ensures that the thread metadata does not lag behind its progress. As we describe shortly, upon recovery we use map$[i]$ to assess the progress of τ_i after a crash. The map initialisation code is redacted here; for each thread τ_i the map$[i]$ is initialised with a $\bot$-array (an array where all entries are $\bot$) to denote that no thread has made any progress yet.

Recovery. After a crash, a client program is restored by triggering a *recovery mechanism.* The recovery mechanism of the counter client programs is given by recover() in Fig. 2. Executing recover() restores the progress of threads by generating a new program P', where each P'$[i]$ entry is a *suffix* of the original program in P$[i]$. The progress of the ith thread τ_i is assessed by calling getProgress(i) on line 17. A call to getProgress(i) traverses the array at map$[i]$, inspecting each entry in turn to locate the latest non-$\bot$ value. That is, if getProgress(i) returns (pc,v) then: (1) the effects of the first pc$-$1 inc operations of thread τ_i have persisted prior to the last crash; (2) the pcth inc operation of τ_i is of the form inc(w) for some w, o and m such that m = max(w,0) and v = o+m; and (3) the effect of this pcth inc operation may or may not have persisted prior to the last crash. As such, if the current counter value (persisted before the crash) is greater than or equal to v, then the effect of the pcth inc operation has persisted prior to the crash and thus thread progress can be advanced to pc+1. This is done on line 19 by setting P$[i]$ to sub(P$[i]$,pc+1), i.e. the suffix (subarray) of P$[i]$ starting at pc+1. On the other hand, if the counter value is less than v, then the effect of the pcth inc operation has not persisted and thus the progress is restored to pc (line 21). Lastly, if pc<0 then τ_i has made no progress prior to the crash and hence it must execute P[t] from the start (line 22).

Constructing Histories for Persistent Linearisability. The typical way of proving a library implementation linearisable is to locate the *linearisation points* of each operation implementation. The linearisation points are points in the implementation source code which, when executed, are deemed to perform the entire observable effect of the operation instantaneously, and thus define the order in which the concurrent library operations are to be linearised. For instance, in our implementation inc has two linearisation points depending on the value of v: (i) if v is non-negative, the linearisation point is on line 6, i.e. when the counter is incremented; (ii) if v is negative, the linearisation point is on line 3, when the counter value is read.

To identify a linearising history H_s of H, recall that we must first complete H to H_c (by extending it with zero or more acknowledgement events), completing those pending invocations that have taken effect (executed their linearisation points) even though their matching *ack*s have not yet been

[1] Note that we do not make assumptions about the thread *IDs*; nor do we assume that recovery restores the same threads (with same IDs). Rather, as the number of threads in P is known in advance, each thread is distinguished by its index in P.

reached. We then truncate H_c to H_t by removing the pending invocations that have not yet had an effect (not reached their linearisation points). Linearisation points can be used to decide H_c and H_t for *persistent* linearisability as follows. For each library invocation event *inv* without a matching *ack* event: (i) if its linearisation point is *persisted* before the crash, the matching *ack* is added to H_c; and (ii) if its linearisation point is *not persisted*, the *inv* is removed from H_t. Finally, we must construct a sequential history H_s that enumerates the library events in H_t. As each `inc` operation acquires the counter lock at the beginning, this global lock acquisition imposes a total 'happens-before' order on the execution of `inc` operations in H_t. Using this global order, we can thus construct a sequential history H_s that enumerates the library events in H_t in the 'happens-before' order.

Persistent Programming Pattern. While developing the persistent counter in Fig. 2, as well as the persistent queues presented later in §4 and §5, we noted that our persistent implementations all follow a certain *pattern*. More concretely, the implementation of each library operation (e.g. `inc`) adheres to the same structure as follows: (1) update the metadata for tracking the progress of the executing thread; (2) execute a `pfence` instruction; (3) carry out the effect of the library operation; (4) execute a `pfence` instruction. For instance, given the implementation of `inc` in Fig. 2, steps (1) and (2) correspond to lines 2-4; whilst steps (3) and (4) correspond to lines 5-6. We believe that this pattern can be used to develop implementations of other concurrent libraries in the presence of persistent hardware. In particular, the first two steps ensure that the recovery metadata of each thread does not lag behind its progress; conversely, the last two steps ensure that the progress of each thread does not lag behind its recovery metadata. Therefore, when executing the recovery mechanism (`recover()`) upon crash recovery, the persisted progress of each thread τ may at most be one step behind its persisted metadata. As such, it suffices to ascertain whether the effect of the *last* recorded instruction for τ has persisted and to restore the progress of τ accordingly. For instance, in the recovery mechanism of the counter implementation in Fig. 2, for each thread τ it suffices to check if the effect of the last recorded `pc` value (obtained on line 17) has persisted and to restore the progress of τ to either `pc+1` or `pc`, as required.

Lifting the Simplifying Assumptions. As discussed on page 7, we assume that our client programs are of a certain shape. This in turn allows us to use a simple progress counter `pc` to record the necessary metadata for tracking how far a thread has advanced along its execution. However, in a more realistic system, one needs to lift these simplifying assumptions and adapt metadata maintenance accordingly. For instance, one way to achieve this is to *log* the call stack of each thread to ensure correct recovery. However, as we demonstrated, programming under persistent memory is subtle even when ignoring the details of how recovery metadata is maintained. As our aim here is to highlight these subtleties, we opt for our simplifying assumptions to streamline metadata maintenance and avoid distracting the reader with the specifics of a particular maintenance strategy.

3 THE PTSO MEMORY MODEL: EPOCH PERSISTENCY FOR TSO

We describe a simple programming language for TSO; we then extend it with persistence primitives to obtain a language for PTSO. We present the PTSO *operational* semantics in §3.1; and present the PTSO *declarative* semantics in §3.2. We establish the equivalence of the two semantics in §3.3.

A Programming Language for TSO. To keep our presentation concise, we employ a simplified concurrent programming language as given in Fig. 3, ignoring the highlighted fragments. We assume a finite set Loc of memory locations; a finite set Reg of registers (local variables); a finite set Val of values; a finite set TId of thread identifiers; and any standard interpreted language for expressions Exp containing at least registers and values. We use `x`, `y`, `z` as metavariables for locations; v for values; τ for thread identifiers; and e for expressions. The sequential fragment of

Basic domains

$\mathtt{x} \in \textsc{Loc}$	Memory locations
$a \in \textsc{Reg}$	Registers
$v \in \textsc{Val}$	Values
$\tau \in \textsc{TId}$	Thread identifiers

Programs

$\mathsf{P} \in \textsc{Prog} \triangleq \textsc{TId} \xrightarrow{\text{fin}} \textsc{Com}$

Expressions and sequential commands

$\textsc{Exp} \ni e ::= v \mid a \mid e{+}e \mid \cdots$

$\textsc{Com} \ni \mathsf{c} ::= \mathbf{skip} \mid \mathbf{if}\ (e)\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ (e)\ \mathbf{do}\ c$
$\quad \mid c; c \mid a := e \mid a := \mathtt{x} \mid \mathtt{x} := e \mid \mathbf{fence}$
$\quad \mid a := \mathbf{CAS}(\mathtt{x}, e, e) \mid a := \mathbf{FAA}(\mathtt{x}, e)$
$\quad \mid \mathbf{pfence} \mid \mathbf{psync} \mid \mathbf{recover}$

Fig. 3. A simple concurrent programming language extended with persistence primitives as highlighted

the language is given by the Com grammar and includes the standard constructs of **skip** (no-op), conditionals, loops, sequential composition, and local variable assignment. The $a := \mathtt{x}$ denotes an atomic memory *read* from location x; similarly, the $\mathtt{x} := e$ denotes an atomic memory *write* to location x. The **fence** denotes a *memory barrier* instruction. As discussed above, when thread τ encounters a **fence** instruction, the store buffer of τ is flushed to memory, unbuffering all pending writes. The $a := \mathbf{CAS}(\mathtt{x}, e, e')$ denotes the atomic 'compare-and-swap' operation, where the value of location x is compared against e: if the values match then the value of x is set to e' and 1 is returned in a; otherwise x is left unchanged and 0 is returned in a. Analogously, the $a := \mathbf{FAA}(\mathtt{x}, e)$ denotes the atomic 'fetch-and-add' operation, where the value of location x is incremented by e and its old value is returned in a. The **CAS** and **FAA** are collectively known as *atomic update* or RMW ('read-modify-write') instructions. Upon their execution, the memory content of a location is read and subsequently modified depending on its old value. To ensure their atomicity, RMW instructions act as memory barriers and may only proceed when the store buffer of the executing thread is flushed of all pending writes. Moreover, the resulting update is committed directly to the memory, bypassing the thread buffer. This is to ensure that the resulting update is immediately visible to other threads. Note that the behaviour of update instructions with respect to thread buffers differs from that of write instructions: writes are added to the thread buffer; updates bypass the thread buffer and flush it of pending writes. Lastly, we model a multi-threaded program P as a function mapping each thread to its (sequential) program. We write $\mathsf{P} = \mathsf{c}_1 || \cdots || \mathsf{c}_n$ when $dom(\mathsf{P}) = \{\tau_1 \cdots \tau_n\}$ and $\mathsf{P}(\tau_i) = \mathsf{c}_i$.

A Programming Language for PTSO. As presented in Fig. 3, the programming language of PTSO is that of TSO, extended with persistence primitives, namely persistent barrier (**pfence**) and persistent sync (**psync**) instructions. As discussed in §2.2, when a program crashes it restarts a recovery mechanism. The recovery mechanism is *program-specific* and must be provided alongside the program itself. For instance, the recovery program for a database may choose to roll-back or roll-forward those queries rendered incomplete due to a crash. To model this, we extend our programming language with an abstract recovery instruction, **recover**, which is to be executed after a program crash. Later in §4, we present a recovery mechanism for two different queue libraries.

Formalising the PTSO Memory Model. Sewell et al. [2010] described the TSO semantics both *operationally* via a small-step transition system, and *declaratively* via execution graphs. They then established the equivalence of the two semantics. Inspired by their formalism, in §3.1 we revisit their operational semantics and describe how we extend it to support epoch persistency. Later in §3.2, we similarly extend their declarative semantics for epoch persistency. In §3.3 we establish the equivalence of our two semantics by means of an intermediate transition system.

3.1 The PTSO Operational Semantics

We describe the PTSO operational semantics by separating the transitions of its *program* and *storage* subsystems. The former describe the steps in program execution, e.g. how and when a conditional branch is triggered. The latter describe how the storage subsystem (comprising the non-volatile memory, the persistent buffer and per-thread buffers as depicted in Fig. 1b) determine the execution steps, whilst simultaneously being updated by the transitions. The PTSO operational semantics is then defined by combining the transitions of the program and storage subsystems.

3.1.1 Program Transitions. The transitions of the PTSO program subsystem are given in Fig. 4a. Program transitions are defined in terms of the transitions of their constituent threads. Thread transitions are of the form: $\mathsf{c}, \mathsf{s} \xrightarrow{\tau:\mathsf{l}} \mathsf{c}', \mathsf{s}'$, where $\mathsf{c}, \mathsf{c}' \in \textsc{Com}$ denote sequential programs as described by the grammar in Fig. 3. The $\mathsf{s}, \mathsf{s}' \in \textsc{Store}$ denote *variable stores*, mapping local variables (registers) to their associated values. The τ:l marks the transition by recording the identifier of the executing thread τ, as well as the transition *label* l. A label may be: ϵ, to denote silent transitions of no-ups; $\mathsf{R}(\mathtt{x}, v)$ to denote reading v from location $\mathtt{x}$; $\mathsf{W}(\mathtt{x}, v)$ to denote writing v to location $\mathtt{x}$; $\mathsf{U}(\mathtt{x}, v, v')$ to denote a successful update (RMW) instruction modifying $\mathtt{x}$ to v'm when its value matches v; F to denote the execution of a memory barrier; PF to denote the execution of a persistent barrier; and PS to denote the execution of a persistent sync. The thread transition can thus be read as, given variable store s, thread τ can take an l step to reduce c to c', whilst updating the variable store to s'.

Most thread transitions are standard; all but the highlighted transitions are identical to their TSO counterparts. The (T-CAS0) transition describes the reduction of a 'compare-and-swap' instruction $a := \mathbf{CAS}(\mathtt{x}, e, e')$ when unsuccessful; i.e. when the value read (v) is different from $\mathsf{s}(e)$. Accordingly the value of a in the store is updated to 0, reflecting the failed **CAS**. The (T-CAS1) transition dually describes the reduction of a **CAS** instruction when successful. Note that in the failure case, as no update takes place, the transition is marked with a read label $\mathsf{R}(\mathtt{x}, v)$ and not an update label as in the success case. The (T-FAA) transition can be described analogously. The (T-Fence) transition describes the execution of a memory barrier (**fence**) instruction, reducing to **skip** with label F, leaving the variable store unchanged. Similarly, the highlighted (T-PFence) and (T-PSync) transitions describe the execution of persistence primitives **pfence** and **psync**.

Program transitions are of the form: $\mathsf{P}, \mathsf{S} \xrightarrow{\tau:\mathsf{l}} \mathsf{P}', \mathsf{S}'$, where $\mathsf{P}, \mathsf{P}' \in \textsc{Prog}$ denote multi-threaded programs as defined by the grammar in Fig. 3. The $\mathsf{S}, \mathsf{S}' \in \textsc{SMap}$ denote *variable store maps*, associating threads with their variable stores. Program transitions are described by simply lifting the transitions of their constituent threads.

3.1.2 Storage Transitions. The transitions of the PTSO storage subsystem are given in Fig. 4b and are of the form: $\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{l}} \mathsf{M}', \mathsf{PB}', \mathsf{B}'$. The $\mathsf{M}, \mathsf{M}' \in \textsc{Mem}$ denote the *non-volatile memory*, modelled as a (finite) map from locations to values. The $\mathsf{PB}, \mathsf{PB}' \in \textsc{PBuff}$ denote the *persistent buffer*, represented as a sequence (FIFO queue) of *persistent sub-buffers*. Each persistent sub-buffer, $\mathsf{pb} \in \textsc{PSBuff}$, captures an epoch and is modelled as a (finite) map associating each location with a sequence of values. The $\mathsf{B}, \mathsf{B}' \in \textsc{BMap}$ denote the *buffer map*, associating each thread with its *buffer*. Lastly, a buffer, $\mathsf{b} \in \textsc{Buff}$, is modelled as a sequence of location-value pairs.

Recall that when a thread reads from a location $\mathtt{x}$, it first consults its own buffer, followed by the persistent buffer (if no write to $\mathtt{x}$ was found in the thread buffer), and finally the non-volatile memory (if no store to $\mathtt{x}$ was found in the thread buffer or the persistent buffer). This lookup chain is captured by the $\mathtt{read}(\mathsf{M}, \mathsf{PB}, \mathsf{B}(\tau), \mathtt{x})$ function in the premise of the ($\textsc{M-Read}^*$) transition. The definition of the $\mathtt{read}(\mathsf{M}, \mathsf{PB}, \mathsf{B}(\tau), \mathtt{x})$ function is given at the bottom of Fig. 4b. The PTSO

Per-thread transitions: $\textsc{Com} \times \textsc{Store} \xrightarrow{\textsc{TId}:\textsc{Lab}} \textsc{Com} \times \textsc{Store}$ $\qquad \mathsf{s} \in \textsc{Store} \triangleq \textsc{Reg} \overset{\text{fin}}{\rightharpoonup} \textsc{Val}$

Program transitions: $\textsc{Prog} \times \textsc{SMap} \xrightarrow{\textsc{TId}:\textsc{Lab}} \textsc{Prog} \times \textsc{SMap}$ $\qquad \mathsf{S} \in \textsc{SMap} \triangleq \textsc{TId} \overset{\text{fin}}{\rightarrow} \textsc{Store}$

$$\mathsf{l} \in \textsc{Lab} \triangleq \left\{ \epsilon, \mathsf{R}(\mathtt{x}, v), \mathsf{W}(\mathtt{x}, v), \mathsf{U}(\mathtt{x}, v, v'), \mathsf{F}, \mathsf{PF}, \mathsf{PS} \;\middle|\; \mathtt{x} \in \textsc{Loc} \land v, v' \in \textsc{Val} \right\}$$

$$\frac{\mathsf{s}(e){\neq}0 \Rightarrow \mathsf{c}{=}\mathsf{c}_1 \quad \mathsf{s}(e){=}0 \Rightarrow \mathsf{c}{=}\mathsf{c}_2}{\textbf{if } (e) \textbf{ then } \mathsf{c}_1 \textbf{ else } \mathsf{c}_2, \mathsf{s} \xrightarrow{\tau:\epsilon} \mathsf{c}, \mathsf{s}} \;(\text{T-If}) \qquad \frac{\mathsf{c}_1, \mathsf{s} \xrightarrow{\tau:\mathsf{l}} \mathsf{c}_1', \mathsf{s}'}{\mathsf{c}_1; \mathsf{c}_2, \mathsf{s} \xrightarrow{\tau:\mathsf{l}} \mathsf{c}_1'; \mathsf{c}_2, \mathsf{s}'} \;(\text{T-Seq1}) \qquad \frac{}{\textbf{skip}; \mathsf{c}, \mathsf{s} \xrightarrow{\tau:\epsilon} \mathsf{c}, \mathsf{s}} \;(\text{T-Seq2})$$

$$\frac{}{\textbf{while } (e) \textbf{ do } \mathsf{c}, \mathsf{s} \xrightarrow{\tau:\epsilon} \textbf{if } (e) \textbf{ then } (\mathsf{c}; \textbf{while } (e) \textbf{ do } \mathsf{c}) \textbf{ else skip}, \mathsf{s}} \;(\text{T-While})$$

$$\frac{\mathsf{s}' = \mathsf{s}[a \mapsto \mathsf{s}(e)]}{a := e, \mathsf{s} \xrightarrow{\tau:\epsilon} \textbf{skip}, \mathsf{s}'} \;(\text{T-ReadL}) \qquad \frac{\mathsf{s}' {=} \mathsf{s}[a \mapsto v]}{a := \mathtt{x}, \mathsf{s} \xrightarrow{\tau:\mathsf{R}(\mathtt{x}, v)} \textbf{skip}, \mathsf{s}'} \;(\text{T-Read}) \qquad \frac{}{\mathtt{x} := e, \mathsf{s} \xrightarrow{\tau:\mathsf{W}(\mathtt{x}, \mathsf{s}(e))} \textbf{skip}, \mathsf{s}} \;(\text{T-Write})$$

$$\frac{v \neq \mathsf{s}(e) \quad \mathsf{s}' = \mathsf{s}[a \mapsto 0]}{a := \textbf{CAS}(\mathtt{x}, e, e'), \mathsf{s} \xrightarrow{\tau:\mathsf{R}(\mathtt{x}, v)} \textbf{skip}, \mathsf{s}'} \;(\text{T-CAS0}) \qquad \frac{\mathsf{s}' = \mathsf{s}[a \mapsto 1]}{a := \textbf{CAS}(\mathtt{x}, e, e'), \mathsf{s} \xrightarrow{\tau:\mathsf{U}(\mathtt{x}, \mathsf{s}(e), \mathsf{s}(e'))} \textbf{skip}, \mathsf{s}'} \;(\text{T-CAS1})$$

$$\frac{\mathsf{s}' = \mathsf{s}[a \mapsto v]}{a := \textbf{FAA}(\mathtt{x}, e), \mathsf{s} \xrightarrow{\tau:\mathsf{U}(\mathtt{x}, v, v+\mathsf{s}(e))} \textbf{skip}, \mathsf{s}'} \;(\text{T-FAA}) \qquad \frac{}{\textbf{fence}, \mathsf{s} \xrightarrow{\tau:\mathsf{F}} \textbf{skip}, \mathsf{s}} \;(\text{T-Fence})$$

$$\frac{}{\textbf{pfence}, \mathsf{s} \xrightarrow{\tau:\mathsf{PF}} \textbf{skip}, \mathsf{s}} \;(\text{T-PFence}) \qquad \frac{}{\textbf{psync}, \mathsf{s} \xrightarrow{\tau:\mathsf{PS}} \textbf{skip}, \mathsf{s}} \;(\text{T-PSync}) \qquad \frac{\mathsf{P}(\tau), \mathsf{S}(\tau) \xrightarrow{\tau:\mathsf{l}} \mathsf{c}, \mathsf{s}}{\mathsf{P}, \mathsf{S} \xrightarrow{\tau:\mathsf{l}} \mathsf{P}[\tau \mapsto \mathsf{c}], \mathsf{S}[\tau \mapsto \mathsf{s}]} \;(\text{P-Step})$$

(a) Program transitions in PTSO

Storage transitions: $\textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap} \xrightarrow{\textsc{TId}:\textsc{Lab}} \textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap}$

$$\frac{\mathtt{read}(\mathsf{M}, \mathsf{PB}, \mathsf{B}(\tau), \mathtt{x}) = v}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{R}(\mathtt{x}, v)} \mathsf{M}, \mathsf{PB}, \mathsf{B}} \;(\text{M-Read}^*) \qquad \frac{}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{W}(\mathtt{x}, v)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto (\mathtt{x}, v).\mathsf{B}(\tau)]} \;(\text{M-Write})$$

$$\frac{\mathsf{B}(\tau){=}\epsilon \quad \mathsf{PB}{=}\mathsf{pb}.\mathsf{PB}' \quad \mathtt{read}(\mathsf{M}, \mathsf{PB}, \epsilon, \mathtt{x}){=}v_r}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{U}(\mathtt{x}, v_r, v_w)} \mathsf{M}, \mathsf{pb}[\mathtt{x} \mapsto v_w.\mathsf{pb}(\mathtt{x})].\mathsf{PB}', \mathsf{B}} \;(\text{M-RMW}^*) \qquad \frac{\mathsf{B}(\tau) = \epsilon}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{F}} \mathsf{M}, \mathsf{PB}, \mathsf{B}} \;(\text{M-Fence})$$

$$\frac{\mathsf{B}(\tau) = \mathsf{b}.(\mathtt{x}, v) \quad \mathsf{PB} = \mathsf{pb}.\mathsf{PB}'' \quad \mathsf{PB}' = \mathsf{pb}[\mathtt{x} \mapsto v.\mathsf{pb}(\mathtt{x})].\mathsf{PB}''}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}', \mathsf{B}[\tau \mapsto \mathsf{b}]} \;(\text{M-BProp}^*)$$

$$\frac{\mathsf{PB} = \mathsf{PB}'.\mathsf{pb} \quad \mathsf{pb}(\mathtt{x}) = S.v}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}[\mathtt{x} \mapsto v], \mathsf{PB}'.(\mathsf{pb}[\mathtt{x} \mapsto S]), \mathsf{B}} \;(\text{M-PBProp}) \qquad \frac{\mathsf{PB} \neq \epsilon}{\mathsf{M}, \mathsf{PB}.\mathsf{pb}_0, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}, \mathsf{B}} \;(\text{M-PBPropE})$$

$$\frac{\mathsf{B}(\tau) = \epsilon}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{PF}} \mathsf{M}, \epsilon.\mathsf{PB}, \mathsf{B}} \;(\text{M-PFence}) \qquad \frac{\mathsf{B}(\tau) = \epsilon \quad \mathsf{PB} = \mathsf{pb}_0}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{PS}} \mathsf{M}, \mathsf{PB}, \mathsf{B}} \;(\text{M-PSync})$$

with

$$\mathsf{M} \in \textsc{Mem} \triangleq \textsc{Loc} \overset{\text{fin}}{\rightarrow} \textsc{Val} \qquad \mathtt{read}(\mathsf{M}, \mathsf{PB}, \mathsf{b}, \mathtt{x}) \triangleq \begin{cases} v & \text{if } \mathtt{rd}_{\mathtt{b}}(\mathsf{b}, \mathtt{x}) = v \\ v & \text{if } \mathtt{rd}_{\mathtt{pb}}(\mathsf{PB}, \mathtt{x}) = v \\ \mathsf{M}(\mathtt{x}) & \text{otherwise} \end{cases}$$

$$\mathsf{PB} \in \textsc{PBuff} \triangleq \textsc{Seq}\,\langle \textsc{PSBuff} \rangle$$

$$\mathsf{pb} \in \textsc{PSBuff} \triangleq \textsc{Loc} \overset{\text{fin}}{\rightarrow} \textsc{Seq}\,\langle \textsc{Val} \rangle \qquad \mathtt{rd}_{\mathtt{b}}((\mathtt{y}, v).\mathsf{b}, \mathtt{x}) \triangleq \begin{cases} v & \text{if } \mathtt{x}{=}\mathtt{y} \\ \mathtt{rd}_{\mathtt{b}}(\mathsf{b}, \mathtt{x}) & \text{otherwise} \end{cases} \qquad \mathtt{rd}_{\mathtt{b}}(\epsilon, \mathtt{x}) \text{ undef}$$

$$\mathsf{B} \in \textsc{BMap} \triangleq \textsc{TId} \overset{\text{fin}}{\rightarrow} \textsc{Buff}$$

$$\mathsf{b} \in \textsc{Buff} \triangleq \textsc{Seq}\,\langle \textsc{Loc} \times \textsc{Val} \rangle \qquad \mathtt{rd}_{\mathtt{pb}}(\mathsf{pb}.\mathsf{PB}, \mathtt{x}) \triangleq \begin{cases} v & \text{if } \exists s.\ \mathsf{pb}(\mathtt{x}) = v.s \\ \mathtt{rd}_{\mathtt{pb}}(\mathsf{PB}, \mathtt{x}) & \text{otherwise} \end{cases} \qquad \mathtt{rd}_{\mathtt{pb}}(\epsilon, \mathtt{x}) \text{ undef}$$

and $\quad \mathsf{pb}_0 \triangleq \lambda \mathtt{x}.\epsilon$

(b) Storage transitions in PTSO where starred* rules denote those changed from their TSO counterparts

Fig. 4. The program and storage transitions in PTSO with persistence extensions highlighted

Operational semantics: $\textsc{Prog} \times \textsc{SMap} \times \textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap} \Rightarrow \textsc{Prog} \times \textsc{SMap} \times \textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap}$

$$\frac{\mathsf{P}, \mathsf{S} \xrightarrow{\tau:\epsilon} \mathsf{P}', \mathsf{S}'}{\mathsf{P}, \mathsf{S}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}', \mathsf{S}', \mathsf{M}, \mathsf{PB}, \mathsf{B}} \ (\textsc{SilentP}) \qquad \frac{\mathsf{P}, \mathsf{S} \xrightarrow{\tau:\mathsf{l}} \mathsf{P}', \mathsf{S}' \quad \mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathsf{l}} \mathsf{M}', \mathsf{PB}', \mathsf{B}'}{\mathsf{P}, \mathsf{S}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}', \mathsf{S}', \mathsf{M}', \mathsf{PB}', \mathsf{B}'} \ (\textsc{Step})$$

$$\frac{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}', \mathsf{PB}', \mathsf{B}'}{\mathsf{P}, \mathsf{S}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}, \mathsf{S}, \mathsf{M}', \mathsf{PB}', \mathsf{B}'} \ (\textsc{SilentM}) \qquad \frac{}{\mathsf{P}, \mathsf{S}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \textbf{recover}, \mathsf{S}_0, \mathsf{M}, \mathsf{PB}_0, \mathsf{B}_0} \ (\textsc{Crash})$$

with $\mathsf{S}_0 \triangleq \lambda\tau.\emptyset$, $\mathsf{B}_0 \triangleq \lambda\tau.\epsilon$ and $\mathsf{PB}_0 \triangleq \mathsf{pb}_0$.

Fig. 5. The PTSO operational semantics with thread and storage transitions in Fig. 4

(M-Read*) differs from its TSO counterpart transition in that the TSO lookup chain does not contain the persistent buffer, but rather the relevant thread buffer and the volatile memory.

As described earlier, when a thread writes value v to a location x, this is recorded in its buffer as the $(\mathtt{x}, v)$ pair. This is captured by the (M-Write) transition. In the (M-RMW*) transition, when executing an RMW instruction on location x (i.e. a **CAS** or **FAA**) a similar lookup chain is followed to determine the value of x, as with a read. Recall that RMW instructions act as memory barriers. As such, the execution of an RMW may proceed only when the thread buffer is drained, as stipulated by the premise $\mathsf{B}(\tau)=\epsilon$. Moreover, the resulting update is committed directly to the persistent buffer, bypassing the thread buffer. This is to ensure that the resulting update is immediately visible to other threads. The difference between (M-RMW*) and its TSO analogue lies in the lookup chain (as described above), and in committing the update to the persistent buffer, rather than the memory.

The (M-Fence) transition describes the execution of a memory barrier by ensuring that the buffer of the executing thread is fully drained ($\mathsf{B}(\tau)=\epsilon$). Analogously, the (M-PFence) transition describes the execution of a persistent barrier by introducing a new empty epoch ϵ and appending it to the existing ones. Recall that when thread τ executes a persistent barrier, it ensures that all stores prior to the barrier (in program order) are persisted before all those following the barrier. How are we then to ensure this for those writes in the thread buffer, not yet flushed to the persistent buffer? To this end, the execution of a persistent barrier requires that the thread buffer be drained and thus acts as a memory barrier. The (M-PSync) transition describes the execution of a persistent sync by ensuring that the persistent buffer is fully drained ($\mathsf{PB}=\mathsf{pb}_0$). That is, the persistent buffer comprises a single empty sub-buffer (epoch) captured by pb_0. The empty sub-buffer pb_0 is defined at the bottom of Fig. 4b and associates each location with an empty sequence. Note that ensuring that the persistent buffer is empty has the same effect as issuing a new epoch. As such, a persistent sync additionally mimics a persistent barrier. Recall that when a thread executes a persistent sync, it ensures that all stores prior to the sync (in program order) are persisted before proceeding with the execution. As such, in order to ensure that the pending writes in the thread buffer are persisted, the execution of a persistent sync requires that the thread buffer be drained and thus acts as a memory barrier. That is, a persistent sync subsumes both a memory barrier and a persistent barrier.

The (M-BProp*) describes the non-deterministic propagation of pending writes in a thread buffer to the latest epoch in the persistent buffer. The PTSO (M-BProp*) differs from its TSO counterpart in that the pending writes in a thread buffer are written back to the persistent buffer and not to the memory. Analogously, the (M-PBProp) describes the propagation of stores in the persistent buffer to the main memory. Note that as described earlier, epochs are propagated in FIFO order. Moreover, in each epoch, the stores to the same location are propagated in FIFO order. However, the stores to different locations in an epoch are not ordered and may be propagated in any order. Lastly, the (M-PBPropE) describes a silent transition where a drained (empty) epoch is evicted from the persistent buffer.

3.1.3 Combined Transitions. The PTSO operational semantics is defined by combining the transitions of the program and storage subsystems, as presented in Fig. 5. The (SilentP) rule describes the case when the program subsystem takes a silent step and thus the storage subsystem is left unchanged. Analogously, the (SilentM) rule describes the case when the storage subsystem takes a silent step and hence the program remains unchanged. The (Step) rule describes the case when the program and storage subsystems both take the same transition (with the same label) and thus the transition effect is that of their combined effects. Lastly, the (Crash) rule describes the case when the program crashes: the execution is restarted with the recovery program **recover**; the memory is left unchanged as it is non-volatile; the variable stores and buffers of all threads are lost and are thus reset to empty; and the persistent buffer is similarly reset to a single empty epoch.

3.2 The PTSO Declarative Semantics

We describe a framework for declarative concurrency models in the context of persistent memory. We then present PTSO as an instance of this general definition.

Notation. Given a relation r on a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$; irreflexive(r) for $\nexists a.\ (a,a) \in r$; and acyclic(r) for irreflexive(r^+). Given two relations r_1 and r_2, we write $\mathsf{r}_1;\mathsf{r}_2$ for their (left) relational composition, i.e. $\{(a,b) \mid \exists c.\ (a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$. Lastly, when r is a strict partial order, we write $\mathsf{r}|_{\mathrm{imm}}$ for the *immediate* edges in r: $\{(a,b) \in \mathsf{r} \mid \nexists c.\ (a,c) \in \mathsf{r} \wedge (c,b) \in \mathsf{r}\}$.

Definition 1 (Events)**.** An *event* is a tuple $\langle n, \tau, \mathsf{l} \rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \mathrm{TId}$ is a thread identifier, and $\mathsf{l} \in \mathrm{Lab}$ is an event label as defined in Fig. 4a.

We typically use a, b and e to range over events. The functions $\mathtt{tid}$, $\mathtt{lab}$, $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathsf{r}}$ and $\mathtt{val}_{\mathsf{w}}$ respectively project the thread identifier, label, type (in $\{\mathtt{R}, \mathtt{W}, \mathtt{U}, \mathtt{F}, \mathtt{PF}, \mathtt{PS}\}$), location, and read/written values of an event, where applicable. Given a relation r on events, we write $\mathsf{r}_{\mathtt{x}}$ for $\{(a,b) \in r \mid \mathtt{loc}(a) = \mathtt{loc}(b) = \mathtt{x}\}$. Similarly, given a set A of events, we write A_x for $\{a \in A \mid \mathtt{loc}(a) = x\}$.

Execution Chains and Execution Graphs. In the literature of declarative models, the traces of shared memory accesses generated by a program are commonly represented as a set of *execution graphs*, where each graph G comprises: (i) a set of events denoting the nodes of the graph; and (ii) a number of relations on events, denoting the sundry edges of the graph. It is common practice to consider *complete* executions only, i.e. those that do not *fail* (crash) and terminate successfully. However, this latter assumption renders this model unsuitable for capturing the crashing behaviour of executions in presence of persistent memory. Instead, we model an *execution chain* $\mathcal{E}$ as a sequence $G_1; \cdots; G_n$, with each G_i describing an execution *era* between two adjacent crashes. More concretely, when an execution of program P crashes $n{-}1$ times, we model this as the chain $\mathcal{E} = G_1; \cdots; G_n$, where (1) G_1 describes the initial era between the start of execution up to the very first crash; (2) for all $i \in \{2 \cdots n{-}1\}$, G_i denotes the i^{th} execution era, recovering from the $(i{-}1)^{\text{st}}$ crash; and (3) G_n describes the final execution era terminating successfully.

Definition 2 (Execution graphs)**.** An *execution graph*, G, is a tuple $(E^0, E^P, E, \mathrm{po}, \mathsf{rf}, \mathsf{vo}, \mathsf{nvo})$, where:

- E^0 is a set of *initialisation events*, comprising a single event with label $\mathtt{W}(\mathtt{x}, v)$, for each location $\mathtt{x} \in \mathrm{Loc}$, where $v \in \mathrm{Val}$.
- E^P is a set of *persistent events* with $E^0 \subseteq E^P$.
- E is a set of *events* with $E^P \subseteq E$. The set of *read* events in E is denoted by $R \triangleq \{e \in E \mid \mathtt{typ}(e) = \mathtt{R}\}$; the sets of *write* events W, *update* events U, *memory barrier* events F, *persistent barrier* events PF and *persistent sync* events PS are defined analogously.

- $\mathsf{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, with $E_0 \times (E \setminus E_0) \subseteq \mathsf{po}$.
- $\mathsf{rf} \subseteq (W \cup U) \times (R \cup U)$ denotes the *'reads-from' relation* between store and lookup events of the same location with matching values; i.e. $(a, b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a)=\mathtt{loc}(b) \wedge \mathtt{val}_{\mathtt{w}}(a)=\mathtt{val}_{\mathtt{r}}(b)$. Moreover, rf is total and functional on its co-domain, i.e. every lookup event is related to exactly one store event.
- $\mathsf{vo} \subseteq E \times E$ is the *'volatile-order'*, defined as a strict partial order with $E_0 \times (E \setminus E_0) \subseteq \mathsf{vo}$.
- $\mathsf{nvo} \subseteq E \times E$ is the *'non-volatile-order'*, defined as a strict partial order with $E_0 \times (E \setminus E_0) \subseteq \mathsf{nvo}$.

In the context of an execution graph G – we often use the "G." prefix to make this explicit – the persistent events E^P include those events whose effects (if any) have reached the non-volatile memory; e.g. those stores that have persisted. As discussed earlier, the 'volatile-order' vo constrains the visible order of loads and stores to the volatile memory (i.e. allowable visible memory states) between processors or cores. For instance, the 'total-store-order' in case of the TSO memory model constitutes the vo order in TSO. Analogously, the 'non-volatile-order' nvo constrains the visible order in which stores are committed to the persistent memory.

Definition 3 (Execution chains). An *execution chain* $\mathcal{E}$ is a sequence $G_1; \cdots ; G_n$ of execution graphs such that for each $i \in \{1, \cdots, n{-}1\}$ and $G_i = (E_i^0, E_i^P, E_i, \mathsf{po}_i, \mathsf{rf}_i, \mathsf{vo}_i, \mathsf{nvo}_i)$:

- $\forall \mathtt{x} \in \mathrm{Loc}.\ \exists w.\ w \in E_1^0 \wedge \mathtt{lab}(w)=\mathtt{W}(\mathtt{x}, 0)$
- $\forall \mathtt{x} \in \mathrm{Loc}.\ \exists w, v.\ w \in E_{i+1}^0 \wedge \mathtt{lab}(w)=\mathtt{W}(\mathtt{x}, v) \wedge \exists e \in \max\left(\mathsf{nvo}_i|_{E_i^P \cap (U_{\mathtt{x}} \cup W_{\mathtt{x}})}\right).\ \mathtt{val}_{\mathtt{w}}(e)=v;$
- $E_n^P = E_n$.

The first constraint ensures that in the first era all locations are initialised with value 0; similarly, the second constraint ensures that in each subsequent $i{+}1^{\text{st}}$ era all locations are initialised with a value persisted by a store in the previous (i^{th}) era maximally (in nvo_i order). The last constraint ensures that the execution of the final era is complete (does not crash) by stipulating that all its events be durable (persistent). That is, we assume that in the absence of a crash, all memory events are eventually persisted.

In the literature of declarative concurrency models, the set of execution graphs associated with a program can be defined straightforwardly by induction on the structure of programs (see e.g. [Vafeiadis and Narayan 2013]). Analogously, the set of execution chains associated with a given program in persistent settings can be defined by induction on the number of execution eras. For each era, the set of execution graphs can be defined by induction over the structure of programs and their associated recovery mechanisms. Each execution of a program P has a particular program *outcome*, prescribing the final values of local variables in each thread. In the definition above, the execution outcomes are almost unrestricted as there are very few constraints on the constituent execution eras and their respective po, rf, vo and nvo relations. Such restrictions and thus the permitted outcomes of a program are determined by defining the set of *valid* execution chains, defined specifically for the model under consideration. In what follows, we define the set of *PTSO-valid* executions.

Definition 4 (PTSO-validity). An execution chain $\mathcal{E} = G_1; \cdots ; G_n$ is *PTSO-valid*, iff each execution graph $G_i \in \mathcal{E}$ is PTSO-valid. An execution graph $G{=}(E^0, E^P, E, \mathsf{po}, \mathsf{rf}, \mathsf{tso}, \mathsf{nvo})$ is *PTSO-valid* iff:

- tso is total on $E \setminus R$; (TSO-TOTAL)
- $\mathsf{po} \setminus (W \times R) \subseteq \mathsf{tso}$; (TSO-PO)
- $\mathsf{rf} \subseteq \mathsf{tso} \cup \mathsf{po}$; (TSO-RF1)
- $\forall (w, r) \in \mathsf{rf}.\ \forall w' {\in} W \cup U.\ (w', r) \in \mathsf{tso} \cup \mathsf{po} \wedge \mathtt{loc}(w')=\mathtt{loc}(r) \Rightarrow (w, w') \notin \mathsf{tso}$. (TSO-RF2)
- nvo is a strict total order on $PE \times PE$, with $PE{=}W \cup U \cup PF \cup PS$; (NVO-TOTAL)
- for each location $\mathtt{x} \in \mathrm{Loc}$, $\mathsf{nvo}_{\mathtt{x}} \subseteq \mathsf{tso}$; (NVO-TSO)

- $[PS \cup PF]; \mathsf{tso}; [PE] \;\cup\; [PE]; \mathsf{tso}; [PS \cup PF] \subseteq \mathsf{nvo}$. (NVO-PSF)
- $dom(\mathsf{nvo}; [E^P]) \subseteq E^P$; (NVO-PRE)
- $R \cup F \cup PS \subseteq E^P$. (NVO-PERS)

Note that in PTSO-valid execution graphs, the 'volatile-order' vo is given by the 'total-store-order' tso of the TSO memory model. For clarity, we have thus replaced vo with tso in our definition. The (TSO-TOTAL), (TSO-PO), (TSO-RF1) and (TSO-RF2) constraints are borrowed directly from those of TSO-consistency as proposed by Sewell et al. [2010].

The (NVO-TOTAL) states that nvo totally orders those events that affect the non-volatile memory, i.e. those events in *PE*. Recall that under epoch persistency, for each location x the volatile and non-volatile memory orders (tso and nvo in PTSO) agree. This is captured by (NVO-TSO).

The (NVO-PSF) describes the epoch orderings imposed by persistent barriers and syncs: if the execution of $e \in PE$ is ordered with respect to that of a persistent barrier or sync, then their persists are also accordingly ordered. The (NVO-PRE) states that nvo is prefix-closed with respect to durable (persistent) events: if an event e is durable, then all those preceding it in nvo order are also durable.

The (NVO-PERS) states that the events in $R \cup F \cup PS$ are always *durable* and are thus included in the set of persistent events E^P. In the case of *PS*, this is because their execution is carried out synchronously. Once a persistent sync is executed, its effect (flushing all pending persists) is immediately committed to the non-volatile memory. In the case of $R \cup F$, their execution has no bearing on the persistent layer (the persistent buffer and the non-volatile memory) of the system – they leave the persistent layer unchanged – and thus their effect is vacuously durable. By contrast, write/update events (in $W \cup U$) and persistent barrier events (in *PF*) may or may not be durable. If the program crashes before a write/update is persisted to the non-volatile memory, its effect is lost, rendering it volatile. Similarly, if the program crashes before the epoch introduced by a persistent barrier $pf \in PF$ is flushed to non-volatile memory, the orderings imposed by pf are lost.

3.3 Equivalence of the PTSO Operational and Declarative Semantics

The PTSO operational semantics presented in §3.1 is *equivalent* to the PTSO declarative semantics in §3.2. We formalise this in Thm. 1 below. We refer the reader to the technical appendix [Raad and Vafeiadis 2018] for the full proof. To establish the equivalence of the two semantics we must show that for all programs P, if $\mathsf{P}, \mathsf{S}_0, \mathsf{M}_0, \mathsf{PB}_0, \mathsf{B}_0 \Rightarrow^* \mathbf{skip}||\cdots||\mathbf{skip}, \mathsf{S}, \mathsf{M}, \mathsf{PB}, \mathsf{B}$, then we can construct a PTSO-valid execution chain $\mathcal{E}$ with the same program outcome (S); and vice versa.

To this end, we develop an *intermediate* semantics as an *event-annotated* transition system. More concretely, we describe the intermediate semantics by separating the transitions of its program and storage subsystems, as before. The transitions of the *annotated program subsystem* are of the form $\mathsf{P}, \mathsf{S} \xrightarrow{\lambda} \mathsf{P}', \mathsf{S}'$, where λ is an *annotated label*, recording the memory event $e \in E$ (Def. 1) making the transition. For instance, when executing a barrier instruction **fence**, the annotated label λ is a *fence* label, $\mathsf{F}\langle f \rangle$, where $f \in F$ is a memory barrier event. Similarly, when executing a read instruction $a := \mathrm{x}$, the annotate label λ is a *read* label, $\mathsf{R}\langle r, e \rangle$, where $r \in R$ is a *read* event, and $e \in W \cup U$ is a write/update event, denoting the event responsible for writing the value read by r. That is, e denotes the write/update event that r *reads from*. Tracking the write/update events this way allows us to construct the rf relation when constructing the corresponding PTSO-valid execution chains.

Similarly, the transitions of the storage subsystem are of the form $M, PB, B \xrightarrow{\lambda} M' PB', B'$, where M, M' are the *event-annotated memory*; PB, PB' are the *event-annotated persistent buffer*; and B, B' are the *event-annotated buffer maps*. An annotated memory M is a map from locations to *write/update* events. That is, for each location x, rather than recording its value, we record the write/update event responsible for setting x to its current value. An annotated persistent buffer PB is analogously augmented to record the write/update *events* to be propagated. *Mutatis mutandis* for B.

The intermediate semantics is then obtained by combining the transitions of the program and storage subsystems. The combined transitions are of the form: $\mathsf{P}, \mathsf{S}, M, PB, B, \mathcal{H}, \pi \Rightarrow \mathsf{P}', \mathsf{S}', M', PB', B', \mathcal{H}', \pi'$. The π denotes the *execution path* in the current execution era, and is of the form $\lambda_n. \cdots .\lambda_1$, modelled as a sequence of annotated labels. That is, each time the combined system takes a λ step, the current execution path is extended by appending λ at the end. Recording the execution path in π allows us to construct the po, tso and nvo relations of the current execution era. The $\mathcal{H}$ denotes an *execution history*, tracking the execution paths of the previous eras. That is, at any point during the execution, if the execution has encountered n crashes, then the history $\mathcal{H}$ contains n entries, $\pi_1 \cdots \pi_n$, with each π_i tracking the execution path in the i^{th} era. Recording the history $\mathcal{H}$ allows us to construct the execution graphs of the previous eras.

To establish the equivalence of our two semantics we must show that: (i) the PTSO operational semantics is equivalent to the PTSO intermediate semantics; and (ii) the PTSO intermediate semantics is equivalent to the PTSO declarative semantics. Proof of part (i) is by straightforward induction on the structure of $\Rightarrow$ transitions. Proof of part (ii) is more involved. As discussed above, to construct PTSO-valid execution chains from the intermediate semantics, we appeal to the events recorded in the storage subsystem, as well as the order of events in execution paths and histories. Dually, to construct the intermediate transitions with the same outcome given a PTSO-valid execution chain $\mathcal{E}$, we must construct the relevant execution path and histories given the execution eras in $\mathcal{E}$ and their sundry relations. We refer the reader to the technical appendix [Raad and Vafeiadis 2018] for the full details of the proof.

Theorem 1 (Semantics equivalence). *The PTSO operational semantics in §3.1 is* equivalent *to the PTSO declarative semantics in §3.2.*

Proof. The full proof is given in the technical appendix [Raad and Vafeiadis 2018].

4 LINEARISABILITY FOR EPOCH PERSISTENCY

In §4.1 we present a formal definition of linearisability in the presence of non-volatile hardware. In §4.2 we develop a persistent queue library together with its recovery mechanism in the PTSO language, and demonstrate that our implementation is persistently linearisable.

4.1 Persistent Linearisability

Izraelevitz et al. [2016b] formulated the notion of buffered durable linearisability using abstract executions and histories as discussed in §2.2. Here, we adapt their definition to that of execution graphs (Def. 2). To this end, we define *library events* as the set of events (Def. 1) extended with library call events, namely *inv* and *ack* events. To identify each *inv* and *ack* pair uniquely, we assume a finite set of *call identifiers*, CId, ranged over by ι. The labels of matching pairs are thus of the form $\mathtt{I}(\iota, m, v_a)$ and $\mathtt{A}(\iota, m, v_r)$, where m denotes the name of the library operation called, v_a denotes the invocation argument, and v_r denotes the return value.

Definition 5 (Library events). Given a library $\mathcal{L}$ and its associated set of *library operations* $\textsc{Op}_{\mathcal{L}} \subseteq \textsc{String}$, A *library event* of $\mathcal{L}$ is a tuple $\langle n, \tau, \mathsf{l} \rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \textsc{TId}$ is a thread identifier, and $\mathsf{l} \in \textsc{Lab} \cup \left\{ \mathtt{I}(\iota, m, v_a), \mathtt{A}(\iota, m, v_r) \,\middle|\, \iota \in \textsc{CId} \land m \in \textsc{Op}_{\mathcal{L}} \land v_a, v_r \in \textsc{Val} \right\}$ is an event label with Lab as defined in Fig. 4a. The set of *invocations* I is defined as $\left\{ e \,\middle|\, \mathtt{typ}(e) = \mathtt{I} \right\}$. The set of *acknowledgements* A is defined analogously. The set of *matching call pairs* is defined as $\mathsf{Duals} \triangleq \left\{ (e_i, e_a) \,\middle|\, \exists \iota, m.\ \mathtt{lab}(e_i) = \mathtt{I}(\iota, m, -) \land \mathtt{lab}(e_a) = \mathtt{A}(\iota, m, -) \right\}$.

The function `cid` returns the call identifier of an event. The definitions of execution graphs (Def. 2), execution chains (Def. 3) and PTSO-validity (Def. 4) are then simply lifted to admit library events. As such, we write e.g. *library execution graph* for an execution graph whose events are library events. Given a library execution graph G, we assume that call identifiers are unique across matching pairs in $G.E$, i.e. no two invocation (acknowledgement) events have the same call identifier.

As discussed in §2.2, the 'happens-before' relation is commonly used in the definition of linearisability in the context of weak memory models. As such, in our following definition of persistent linearisability we use the widely-used 'happens-before' relation defined as $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$, denoting a strict partial order on the execution events.

Definition 6 (Persistent linearisability). A history (sequence of events) H *linearises* a library execution graph $G = (E^0, E^P, E, \mathsf{po}, \mathsf{rf}, \mathsf{vo}, \mathsf{nvo})$ iff there exist E_c and E_t such that:

- $E_c \in \mathtt{comp}(E^P)$ with $\mathtt{comp}(S) \triangleq \left\{ S' \supseteq S \;\middle|\; \begin{array}{l} S' \setminus S \subseteq A \land \forall e_a \in S' \setminus S. \\ \quad \exists e_i \in S.\ (e_i, e_a) \in \mathsf{Duals} \\ \quad \land \nexists e'_a \in S.\ (e_i, e'_a) \in \mathsf{Duals} \\ \quad \land \forall e'_a \in S' \cap A.\ \mathtt{cid}(e_a){=}\mathtt{cid}(e'_a) \Rightarrow e_a{=}e'_a \end{array} \right\}$;
- $E_t = \mathtt{trunc}(E_c)$ with $\mathtt{trunc}(S) \triangleq (I \cup A) \cap \left(S \setminus \left\{ i \in I \;\middle|\; \nexists a \in S.\ (i, a) \in \mathsf{Duals} \right\}\right)$;
- H is an enumeration of E_t such that: $\forall a, b \in E_t.\ (a, b) \in \mathsf{hb} \Rightarrow a <_H b$, where $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$ denotes the *'happens-before' relation*, and $a <_H b$ denotes that a appears before b in H;
- H is *sequential*, i.e. is of the form $i_1; a_1; \cdots; i_m; a_m$, with each $(i_k, a_k) \in \mathsf{Duals}$.

An execution chain $\mathcal{E} = G_1; \cdots; G_n$ of library $\mathcal{L}$ is *persistently linearisable* iff there exist $H_1 \cdots H_n$ such that: i) each H_i linearises G_i; and ii) $H_1; \cdots; H_n$ is a legal history of $\mathcal{L}$.

Definition 7 (Library linearisability). A PTSO implementation of library $\mathcal{L}$ is *persistently linearisable* iff all its PTSO-valid library execution chains are persistently linearisable.

4.2 A Persistently Linearisable Queue Library in PTSO

In Fig. 6 we present a persistent implementation of a queue library (left) and its recovery mechanism (right) in the PTSO language. We consider a queue library with two standard operations, `enq(v)` and `deq()`, for adding and removing elements from the queue, respectively. We use a coarse-grained lock to control concurrent accesses to the queue. The contents of the queue are stored as an array, with the head index recorded at a designated location. A queue at location `q` thus comprises three components, represented as three adjacent cells: (i) the queue *lock* at `q`, written `q.lock`, recording an integer which may be 0 (when unlocked) or 1 (when locked); (ii) the queue *contents* at `q+1`, written `q.data`, recording the location at which the contents array resides; and (iii) the queue *head* at `q+2`, written `q.head`, recording the index of the first entry in `q.data`.

The lock on `q` is acquired by calling `lock(q)`, where the calling thread spins until `q.lock` holds 0, at which point its value is atomically set to 1 (via a **CAS** operation). Dually, the lock on `q` is released by calling `unlock(q)`, where `q.lock` is set to 0. To keep our presentation simple, we assume that the array at `q.data` can grow dynamically, and elide the details of array management. Ignoring the code in blue, the `enq(v)` and `deq()` implementations are straightforward. A call to `enq(v)` creates a new node `n` with value `v`, acquires the lock, traverses the queue starting at the head `q.head` until it finds an empty (`null`) entry, inserts the new node `n` at this location, and finally releases the lock. Analogously, a call to `deq()` acquires the lock and retrieves the head entry at `q.head` (which may hold `null`when the queue is empty) in `n`. If `n` is not `null` (the queue is not empty), the head index is duly incremented by one. The lock is then released and `n` is returned.

As described in §2.2, we assume that client programs are of the form $\mathtt{c}_0 || \cdots || \mathtt{c}_k$; that each $\mathtt{c}_i$ is of the form $o^i_0; \cdots; o^i_l$, where each o^i_j is a library operation (`enq` or `deq`); and that the client program is represented as an array at `P`. A client program `P` is executed by calling `run(P)`, with each thread context set up as discussed in §2.2. As before, to ensure correct recovery, the metadata for tracking the progress of each thread is recorded in a map at `map`.

```
q.enq(v) ≜
 pc:=getPC(); t:=getTC();
 n:=newNode(v,t,pc);
 map[t][pc]:=n; pfence;
 lock(q); h:=q.head;
 while (q.data[h] != null)
  h:=h+1;
 q.data[h]:=n;
 pfence; unlock(q);

q.deq() ≜
 pc:=getPC(); t:=getTC();
 lock(q);h:=q.head;n:=q.data[h];
 map[t][pc]:=n;
 if (n != null) {
  t':=n.t; pc':=n.pc;
  map[t'][pc']:=⊤ }
 pfence;
 if (n != null) {
  q.head:=h+1; pfence; }
 unlock(q); return n;

lock(q) ≜
 while (!CAS(q.lock,0,1)) skip;

unlock(q) ≜ q.lock:=0;

isIn(q,n) ≜
 h:=q.head; c:=q.data[h];
 while (c != null) {
  if (n==c) return true;
  else { h:=h+1; c:=q.data[h]; }
 } return false;

getProgress(t) ≜
 pc:=-1; n:=⊥;
 while (map[t][pc+1] != ⊥) {
  pc++; n:=map[t][pc]; }
 return (pc,n);

start() ≜
 lq:=newQueue();
 s:=P.size; lmap:=newMap(s);
 for (t in P)
  lmap[t]:=newArray(P[t].size,⊥);
 pfence;
 q:=lq; map:=lmap; run(P);

recover() ≜
 if (q==null || map==null)
  goto start();
 for(t in P) enq[t]:=-1;
 unlock(q);
 for(t in P) { // deq recovery
  (pc,n):=getProgress(t);
  if (pc>=0 && isDeq(P[t][pc])) {
   if (n==null)
    P'[t]:=sub(P[t],pc+1);
   else {
    if (inIn(q,n))
     P'[t]:=sub(P[t],pc);
    else
     P'[t]:=sub(P[t],pc+1);
    t':=n.t; pc':=n.pc;
    enq[t']:=max(enq[t'],pc'+1);}
  }
  else if (pc<0) P'[t]:=P[t];
 }
 for(t in P) { // enq recovery
  (pc,n):=getProgress(t);
  if (pc>=0 && isEnq(P[t][pc])) {
   if (pc < enq[t])
    P'[t]:=sub(P[t],enq[t]);
   else if (n==⊤ || isIn(q,n))
    P'[t]:=sub(P[t],pc+1);
   else
    P'[t]:=sub(P[t],pc); }
 } run(P');
```

Fig. 6. A persistent queue implementation and its recovery mechanism with persistence code in blue

Initialisation. The `start()` commences the execution of the client program stored at location P by initialising the metadata necessary for crash recovery. It thus creates a new (empty) queue at q, together with a recovery map of the relevant size (the number of threads in P) at map, and launches the execution by calling `run(P)`. As before, when the i^{th} thread contains l+1 instructions (`P[`i`].size` = l+1), then its associated map entry (i.e. map[i]) is an array of length l+1, with one entry per instruction. For each i^{th} thread τ_i the `map[`i`]` entry is initialised with a $\bot$-instantiated array of the appropriate size (i.e. `P[`i`].size`) to denote that τ_i has made no progress as of yet. The `pfence` on line 40 ensures that if the execution of `start()` crashes, then recovery does not observe a partially initialised `map`.

Persistence of Queue Operations. Recall that we track the progress of each thread in `map` to ensure correct crash recovery. In particular, when τ_i executes its j^{th} operation, *prior* to carrying out the relevant queue update, it updates `map[`i`][`j`]` to `n`, where `n` denotes the node being added or removed. This is done on lines 4 and 13 of `enq` and `deq`, where the subsequent `pfence` instructions (lines 4 and 17) ensure that the thread metadata does not lag behind its progress.

Upon recovery, the progress of the i^{th} thread τ_i is assessed by calling `getProgress(`i`)` on line 48. A call to `getProgress(`i`)` traverses the array at `map[`i`]` in order to locate the latest non-$\bot$ value. That is, if `getProgress(`i`)` returns `(`j`,n)` then: (1) the effects of the first `pc`-1 operations of τ_i have persisted prior to the last crash; (2) the `pc`$^{\text{th}}$ operation of τ_i was attempting to enqueue/dequeue node `n`; and (3) the effect of this `pc`$^{\text{th}}$ operation may or may not have persisted prior to the last crash. As such, if `getProgress(`i`)` returns $(j,$`n`$)$ and o^i_j (the j^{th} operation of τ_i) is a `deq`, node `n` may or may not have been removed by τ_i when the crash occurred. One can then inspect the queue to ascertain whether the execution of o^i_j was completed and persisted. If `n` is in the queue, then the crash occurred before the removal of `n` was persisted and thus recovery must resume executing τ_i from o^i_j. On the other hand, if n is not in the queue, then recovery must resume τ_i from o^i_{j+1}. Similarly, if o^i_j is an `enq`, one can *in most cases* determine the progress of τ_i by inspecting the queue. If `n` is in the queue, then the crash occurred after the insertion of `n` was persisted and thus recovery must resume τ_i from o^i_{j+1}. However, if `n` is not in the queue, it may be the case that τ_i added `n` to the queue, while another thread later removed `n` from the queue, prior to the crash.

To understand this better, consider `P=q.enq(v)`$\,||\,$`(q.deq();`$o^1_1;o^1_2$`)`. Let us suppose thread τ_0 executing `enq(v)` acquires the lock before τ_1, adds `v` to the queue and thus sets `map[0][0]` to `n` for some `n` with value `v`. Thread τ_1 later acquires the lock, executes `deq()` and removes `n` from the queue, and subsequently crashes while executing o^1_2. Let us assume that all writes persisted before the crash, i.e. `map[0][0]=n`. In this scenario, even though the execution of τ_0 was finalised and fully persisted, we cannot ascertain this by simply inspecting the queue, as `n` is removed by τ_1.

To remedy this, the `deq` operations must *help* advance the progress of `enq` operations. That is, when removing a node `n`, we can confirm that `n` was indeed added to the queue, and thus the progress of the thread responsible for inserting it must be advanced accordingly. To this end, for each node `n` added to the queue, the representation of `n` additionally records the metadata of the thread responsible for adding it to the queue. More concretely, when the j^{th} operation of τ_i adds node `n` to the queue, as part of its representation `n` records: 1) the thread index i at location `n+1`, written `n.t`; and 2) the operation index j at location `n+2`, written `n.pc`. When removing `n` via `deq`, the implementation updates the current progress of the thread responsible for inserting `n` (i.e. `n.t`) in `map` if necessary (lines 14-16). That is, when `n.t` $= \tau_i$ and `n.pc` $= j$, as τ_i has successfully enqueued `n` via its j^{th} operation, its current recorded progress in `map[`i`][`j`]` is updated to the designated value $\top$, to indicate that the insertion of `n` is indeed successful. As we describe shortly, upon recovery, when `map[`i`][`j`]` $= \top$ and o^i_j (the j^{th} operation of τ_i) is an enqueue operation, we can infer that the effect of o^i_j has persisted successfully and can thus advance the progress of τ_i accordingly. In the example above, this ensures that τ_1 sets `map[0][0]` to $\top$ when removing `n`, thus ensuring that recovery realises the completion of τ_0 operations.

Lastly, the `pfence` instructions on lines 9 and 19 ensure that the thread progress does not lag behind its recovery metadata in `map`. Note that the queue implementation in Fig. 6 follows the persistent programming pattern discussed on page 9.

Recovery. The recovery mechanism of a queue client program at location `P` is triggered by calling `recover()`. The first two lines ensure that `q` and `map` have been initialised; otherwise `start()` is called. As discussed above, the `deq` calls help advance the progress of their counterpart `enq` calls. Analogously, the recovery program can also use the progress of `deq` calls prior to crash to restore the progress of `enq` calls correctly. To this end, the `enq` array (initialised on line 45) tracks the progress of `enq` calls as observed by `deq` calls. After releasing the queue lock, recovery restores the progress of threads by generating a new program `P'`, where each `P'`$[\tau_i]$ entry is a *suffix* of the original program in `P`$[\tau_i]$. This restoration is done in two passes: first for threads executing a `deq` operation prior to crash (lines 47-61), and then for those executing an `enq` (62-71).

Recall that the progress of thread τ prior to crash can be ascertained by calling `getProgress`(τ). For each dequeuing thread τ, when `getProgress`(τ) returns `(pc,n)`, if `n=null` (the queue was empty when τ attempted a `deq`) then its effect has (trivially) persisted and thus its progress can be advanced to `pc+1`. This is done on line 51 by setting `P`$[\tau]$ to `sub(P`$[\tau]$`,pc+1)`, i.e. the subarray of `P`$[\tau]$ starting at `pc+1`. On the other hand if `n≠null`, then the effect of τ (removing `n`) may or may not have persisted. Recall that to determine the progress of τ one can inspect the queue to ascertain whether it contains `n`. This is done by calling `isIn(q,n)`. As discussed above, the τ progress can be restored accordingly to either `pc` when `n` is still in the queue (line 54), or `pc+1` when `n` is not in the queue (line 56). In both cases, we can confirm that the thread responsible for enqueuing `n` has persisted past the operation inserting `n`. When `n.t=`τ' and `n.pc=pc'`, the `enq`$[\tau']$ entry is thus set to the maximum value observed for τ' so far, i.e. `max(enq`$[\tau']$`,pc'+1)` – see line 58.

For each enqueuing thread τ, when `getProgress`(τ) returns `(pc,n)`, if the progress recorded for τ lags behind that observed by dequeuing operations (`pc<enq`$[\tau]$), then progress is duly set to `enq`$[\tau]$ on line 66. On the other hand, if the progress is not lagging, then the effect of τ (adding `n`) may or may not have persisted. Inspecting the queue, one can then restore the τ progress accordingly to either `pc+1` when `n` is in the queue (line 68), or `pc` when `n` is not in the queue (line 71). Moreover, recall that dequeuing threads help advance the progress of enqueuing threads by updating the relevant entry to the designated value $\top$. As such, when `n` $=$ $\top$ (line 67), we can deduce that the node inserted by the `pc`th operation has been removed by a dequeuing thread prior to the crash, and thus the progress of τ can be advanced to `pc+1` accordingly.

Lastly, for each thread τ, when `getProgress`(τ) returns `(pc,n)`, observe that when `pc<0` then τ has made no progress prior to the crash and hence it must execute `P`$[\tau]$ from the start (line 60).

4.2.1 Persistent Linearisability of the Implementation. We demonstrate that the implementation in Fig. 6 is persistently linearisable. To do this, we describe a proof pattern for constructing persistent linearisation histories using *linearisation points*. We then present an informal argument of the persistent linearisability of our queue implementation and state it formally as a theorem. We refer the reader to the technical appendix [Raad and Vafeiadis 2018] for the full proof.

Constructing Histories for Persistent Linearisability. Recall that to show a library implementation is *persistently* linearisable, one must show that each constituent era is persistently linearisable. Proving that an execution era G is persistently linearisable is similar to showing it is linearisable. Intuitively, to show an era G is linearisable, one must identify a history that linearises its *library events* (i.e. *inv* and *ack* events in $G.E$); whereas to show G is *persistently linearisable*, one must identify a history that linearises its *persistent library events* (i.e. those in $G.E^P$).

To identify a linearising history H, recall that we must first complete E_P to E_c by extending it with zero or more acknowledgement events, completing those pending *inv* events that have taken effect (executed their linearisation points) even though their matching *ack*s have not yet been reached. We then truncate E_c to E_t by removing pending invocation methods that have not yet had an effect (not reached their linearisation points). As discussed in §2.2, linearisation points can be used to decide E_c and E_t for *persistent* linearisability as follows. For each library invocation event *inv* without a matching *ack* event: (i) if its linearisation point is *persisted* (i.e. in E^P), the matching *ack* is added to E_c; and (ii) if its linearisation point is *not persisted* (i.e. not in E^P), the *inv* is removed from E_t. Finally, we must construct a sequential history H that enumerates the library events in E_t. This construction is dictated by the 'happens-before' relation induced by the implementation and is thus implementation-specific.

Persistent Linearisability of the Implementation in Fig. 6. As discussed above, the construction of linearisable histories is guided by the linearisation points. In our implementation, the `enq` operation has a single linearisation point on line 8, when `n` is appended to the end of `q.data`. The `deq` operation has two linearisation points depending on `q`: (i) if `q.data` is empty, the linearisation point is on line 12, when `null` is read and `q` is left unchanged; (ii) if `q.data` is not empty, the linearisation point is on line 19, when `q.head` is advanced.

Note that the placement of `pfence` instructions in our implementation, together with the PTSO-validity of executions ensure that in each era G, each thread has *at most one* pending *inv* event without a matching persisted *ack*. This is due to (TSO-PO), (NVO-PSF) and (NVO-PRE) in Def. 4.

To show that an execution era G of our implementation is persistently linearisable, we construct the E_c and E_t sets as described above. Finally, we must construct a sequential history H that enumerates the library events in E_t. As each queue operation acquires the global queue lock at the beginning, this global lock acquisition imposes a total 'happens-before' (hb) order on the execution of queue operations in G. Using this global order, we can thus construct a sequential history H that enumerates the library events in E_t in the hb order.

Recall that as part of the linearisability proof one must additionally show that the combined histories of execution eras form a *legal* history. As discussed earlier, the notion of a legal history is library-specific. For the queue library, a history H is a legal history if it respects the FIFO property. We formalise this in Def. 8 below. We then state the persistent linearisability of our implementation in Thm. 2, with its full proof given in the technical appendix [Raad and Vafeiadis 2018].

Definition 8 (Legal queue history). A history H is a legal history of a queue library iff:

- $\forall e \in H.\ \mathtt{lab}(e) \in \{\mathtt{I}(-, \mathtt{enq}, -), \mathtt{A}(-, \mathtt{enq}, -), \mathtt{I}(-, \mathtt{deq}, -), \mathtt{A}(-, \mathtt{deq}, -)\}$; and
- $\mathsf{fifo}(\epsilon, H)$ holds, where

$$\begin{aligned}\mathsf{fifo}(s, H) \overset{\text{def}}{\iff}\ & H{=}\epsilon \lor (\exists n, H', \iota.\ n{\neq}\mathtt{null} \land H{=}\mathtt{I}(\iota, \mathtt{enq}, n); \mathtt{A}(\iota, \mathtt{enq}, ()); H' \land \mathsf{fifo}(s; n, H')) \\ & \lor(\exists n, H', \iota, s'.\ n{\neq}\mathtt{null} \land s{=}n; s' \land H{=}\mathtt{I}(\iota, \mathtt{deq}, ()); \mathtt{A}(\iota, \mathtt{deq}, n); H' \land \mathsf{fifo}(s', H')) \\ & \lor(\exists H', \iota.\ s{=}\epsilon \land H{=}\mathtt{I}(\iota, \mathtt{deq}, ()); \mathtt{A}(\iota, \mathtt{deq}, \mathtt{null}); H' \land \mathsf{fifo}(s, H'))\end{aligned}$$

Theorem 2. *The queue implementation in Fig. 6 is persistently linearisable.*

Proof. The full proof is provided in the technical appendix [Raad and Vafeiadis 2018].

5 PERSISTENT MICHAEL-SCOTT QUEUE LIBRARY IN PTSO

In Fig. 7 we present a *persistent* variant of the lock free Michael-Scott (MS) queue [Michael and Scott 1996] implementation (left) and its recovery mechanism (right) in the PTSO language. For simplicity, in our variant of the Michael-Scott queue we do not track the *tail* pointer.

As before, the queue contents are stored as an array that may grow dynamically. A queue at `q` comprises two components, represented as two adjacent cells: (i) the queue contents at `q`, written `q.data`, recording the location of the contents array; and (ii) the queue head at `q+1`, written `q.head`. As before, we assume that a client program is represented as an array at `P`; and that `P` is executed by calling `run(P)`, with each thread context set up as discussed in §2.2. Similarly, we record the relevant metadata for tracking the progress of each thread in a map at `map`.

```
q.enq(v) ≜
  pc:=getPC(); t:=getTC();
  n:=newNode(v,t,pc,null);
  map[t][pc]:=n; pfence;
  h:=q.head;
  find: while (q.data[h] != null)
    h:=h+1;
  if (!CAS(q.data[h],null,n))
    goto find;
  pfence;

q.deq() ≜
  pc:=getPC(); t:=getTC();
  try: h:=q.head; n:=q.data[h];
  if (n!=null && !CAS(n.deq,null,t))
    goto try;
  map[t][pc]:=n;
  if (n != null) {
    t':=n.t; pc':=n.pc;
    map[t'][pc']:=⊤;
  } pfence;
  if (n!=null) {
    q.head:=h+1; pfence;
  }
  return n;
```

```
recover() ≜
  if (q==null || map==null)
    goto start();
  for(t in P) enq[t]:=-1;
  for(t in P) { // deq recovery
    (pc,n):=getProgress(t);
    if (pc>=0 && isDeq(P[t][pc])){
      if (n==null)
        P'[t]:=sub(P[t],pc+1);
      else {
        if (inIn(q,n)) {
          P'[t]:=sub(P[t],pc);
          n.deq:=null; }
        else
          P'[t]:=sub(P[t],pc+1);
        t':=n.t; pc':=n.pc;
        enq[t']:=max(enq[t'],pc'+1);}
    } else if (pc<0) P'[t]:=P[t]; }
  for(t in P) { // enq recovery
    (pc,n):=getProgress(t);
    if (pc>=0 && isEnq(P[t][pc])){
      if (pc < enq[t])
        P'[t]:=sub(P[t],enq[t]);
      else if (n==⊤ || isIn(q,n))
        P'[t]:=sub(P[t],pc+1);
      else
        P'[t]:=sub(P[t],pc); }
  } pfence;
  run(P');
```

Fig. 7. A persistent Michael-Scott queue implementation and its recovery with persistence code in blue

Queue Operations. The implementation of enq(v) in Fig. 7 is rather similar to its counterpart implementation in §4.2. The difference of the two lies in the synchronisation mechanism used to control concurrent accesses. More concretely, rather than a global queue lock, the enq implementation in Fig. 7 uses a **CAS** instruction to *atomically* append the new node n to the end of the queue.

The implementation of deq() in Fig. 7 is also similar to its counterpart implementation in §4.2. As before, when thread τ executes its jth operation which is a deq, the implementation proceeds by retrieving the first queue entry (at the head index) in n. To correctly track its progress, recall that the map entry at map[τ][j] must be accordingly updated to n. Let us assume $n\neq$null. As the MS queue is lock free, another thread τ' may concurrently attempt to execute its kth operation which is also a deq, thus updating its map[τ'][k] entry to n. Now consider the scenario when τ' wins the race to remove n, and subsequently the program crashes. When recovering, the recovery mechanism observes that n is removed from the queue and rightfully advances the progress of τ' to k+1. Following the same observation, it also advances the progress of τ to j+1, *even though* τ did not complete its dequeue before the crash.

To correctly track the progress of dequeue operations, the representation of each node `n` additionally includes a 'deq' field, written `n.deq`, recording the index of the thread that successfully removed it. When creating a new node `n`, the `n.deq` is initialised with `null` (see line 3). When seeking to remove `n`, the executing thread τ *signals* to other threads its intention to remove `n` by attempting to atomically set `n.deq` to τ via a **CAS** instruction. This is done on line 14 of the implementation. If the **CAS** fails, this is because another thread is in the process of removing `n` and τ must thus retry dequeuing. On the other hand, if the **CAS** is successful, this will prevent other threads from racing with τ to remove `n`. The rest of the `deq` implementation then proceeds as before. The `map[`τ`]` entry is updated, the progress of the thread responsible for enqueuing `n` is advanced if necessary, and the head pointer is updated (if `n`$\neq$`null`). Note that the queue implementation in Fig. 7 follows the persistent programming pattern discussed on page 9.

Initialisation and Recovery. The execution of a client program is commenced by running `start()` in Fig. 6 as discussed in §4.2. The recovery mechanism of the Michael-Scott queue library is given by `recover()` in Fig. 7. The code of `recover()` is almost identical to its counterpart in Fig. 6. The only difference between the two lies in lines 37 and 52. Line 37 captures the case when a crash occurs during a `deq` execution, *before* the dequeuing thread successfully removes a node `n` and persists its effect. As such, upon recovery `n.deq` is set to `null` to facilitate the re-execution of `deq` to remove `n`. Line 52 ensures that the changes made to the queue by the recovery (when resetting `n.deq` on line 37) are persisted before those of the restarted execution.

Persistent Linearisability of the Implementation in Fig. 7. The linearisation points of our implementation are analogous to those of their counterparts in §4.2. The linearisation point of `enq` is on line 8; the `deq` has two linearisation points depending on `q.data`: (i) if `q.data` is empty, the linearisation point is on line 13; (ii) if `q.data` is not empty, the linearisation point is on line 22. To show that an execution era G of our implementation is persistently linearisable, we construct the E_c and E_t sets using the linearisation points as described in §4.2.

Note that the linearisation points of `enq` operations, as well as those of `deq` in case (ii) above, are *write* and *update* instructions and are thus ordered by the total-store-order $G.\mathsf{tso}$. To construct a sequential history H, we extend $G.\mathsf{tso}$ to a total order tso_t, where *all* linearisation points are ordered with respect to one another. We then construct H as an enumeration of the library events such that the order between their linearisation points is respected. That is, H is of the form $inv_1; ack_1; \cdots ; inv_m; ack_m$, where for all $i, j \in \{1 \cdots m\}$ we have: $i < j$ iff the linearisation point associated with (inv_i, ack_i) is tso_t-ordered before that of (inv_j, ack_j).

Lastly, we demonstrate that the combined histories of execution eras form a legal queue history as given in Def. 8. We state the persistent linearisability of our implementation in Thm. 3 below; we refer the reader to the technical appendix [Raad and Vafeiadis 2018] for the full proof.

Theorem 3. *The Michael-Scott queue implementation in Fig. 7 is persistently linearisable.*

Proof. The full proof is provided in the technical appendix [Raad and Vafeiadis 2018].

Towards a Non-Blocking Implementation. Observe that in contrast to the original Michael-Scott queue implementation in [Michael and Scott 1996], the `deq` implementation of the variant presented in Fig. 7 is blocking in that a thread attempting to dequeue may spin until the current dequeuing thread completes its dequeue operation (lines 14-15). This adaptation was merely to keep the presentation simple. In the technical appendix, we present a fully non-blocking persistent variant of the Michael-Scott queue together with its recovery mechanism, and demonstrate its persistent linearisability. We refer the reader to [Raad and Vafeiadis 2018] for the full details.

6 CONCLUSIONS AND FUTURE WORK

We developed the PTSO memory model by combining the buffered epoch persistency model with the TSO memory model, which underpins the x86 and SPARC architectures. We demonstrated the use of those semantics by verifying two queue implementations against our formal semantics. To our knowledge, the work presented here is the first to study persistence semantics *formally* in the context of a *mainstream architecture*.

We believe that the approach presented here can be used to combine epoch-persistency with any *multi-copy-atomic* architecture, e.g. ARMv8. More concretely, for operational semantics, the underlying machine transitions must be adapted to incorporate the additional persistent buffer. For declarative semantics, the existing axioms must be extended with those of epoch-persistency in Def. 4 (i.e. the (NVO-TOTAL), (NVO-TSO), (NVO-PSF), (NVO-PRE) and (NVO-PERS) axioms), with tso replaced by vo (volatile order), as determined by the underlying architecture (see Def. 2).

As directions of future work, we thus plan to build on top of the work presented here in two ways. First, we plan to explore the semantics of epoch persistency when integrated with other consistency models. Of particular interest is the recent ARM consistency model by Pulte et al. [2017]. The authors formalised the semantics of the ARM memory model both operationally and declaratively. As with the work presented here, we plan to extend both characterisations with persistence primitives under the epoch persistency model and establish their equivalence. The declarative semantics of the ARM memory model fits our general framework for describing declarative concurrency models in the presence of persistent memory. This will allow us to apply correctness conditions such as that of persistent linearisability to verify the correctness of library implementations under the ARM memory model.

Second, we plan to develop reasoning techniques for verifying the correctness of persistent programs and libraries. We plan to pursue this in two different directions. First, utilising our formal declarative semantics (those presented in this article as well as those in future work), we can formulate the notion of persistent linearisability in the context of other persistency models and architectures. Moreover, we can adapt other correctness conditions, such as that of triangular race freedom by Owens [2010], for persistent memory under different persistency models. Second, taking advantage of our formal operational semantics, we plan to develop program logics that would allow us to verify properties of persistent programs. This can be achieved by either extending existing program logics for weak memory with persistence primitives, or developing new program logics for currently unsupported models.

ACKNOWLEDGMENTS

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, under the European Union Horizon 2020 Framework Programme (grant agreement number 683289).

REFERENCES

Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016).* ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. https://doi.org/10.1145/1961296.1950380

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 15–26. https://doi.org/10.1145/325096.325102

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

IBM. 2013. IBM solidDB. (2013). https://www.ibm.com/support/knowledgecenter/en/SSPK3V_7.0.0/master/welcome.kc.html

Intel. 2014. Intel architecture instruction set extensions programming reference. (2014). https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

International technology roadmap for semiconductors. 2007. Process Integration, devices, and structures. (2007). http://www.maltiel-consulting.com/ITRS_2011-Process-Integration-Devices-Structures.pdf

Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016a. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016b. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

T. Kawahara, K. Ito, R. Takemura, and H. Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability* 52, 4 (2012), 613 – 627. https://doi.org/10.1016/j.microrel.2011.09.028 Advances in non-volatile memory technology.

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016a. High-Performance Transactions for Persistent Memories. *SIGPLAN Not.* 51, 4 (March 2016), 399–411. https://doi.org/10.1145/2954679.2872381

Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016b. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael James Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *DISC*.

Scott Owens. 2010. Reasoning About the Implementation of Concurrency Abstractions on x86-TSO. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 478–503. http://dl.acm.org/citation.cfm?id=1883978.1884011

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

Azalea Raad and Viktor Vafeiadis. 2018. Technical Appendix. (2018). http://plv.mpi-sws.org/ptso/

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

SPARC. 1992. *The SPARC Architecture Manual: Version 8.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. 2008. The missing memristor found. *Nature* 453 (2008), 80 – 83.

TimesTen Team. 1999. In-memory Data Management for Consumer Transactions the Timesten Approach. *SIGMOD Rec.* 28, 2 (June 1999), 528–529. https://doi.org/10.1145/304181.304244

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &*

Applications. 867–884.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. https://doi.org/10.1145/2248487.1950379

Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using Restricted Transactional Memory to Build a Scalable In-memory Database. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14).* ACM, New York, NY, USA, Article 26, 15 pages. https://doi.org/10.1145/2592798.2592815

Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11).* ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2063384.2063436

Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46).* ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744