The Path to Durable Linearizability

EMANUELE D'OSUALDO, MPI-SWS, Germany
AZALEA RAAD, Imperial College London, UK
VIKTOR VAFEIADIS, MPI-SWS, Germany

There is an increasing body of literature proposing new and efficient persistent versions of concurrent data structures ensuring that a consistent state can be recovered after a power failure or a crash. Their correctness is typically stated in terms of *durable linearizability* (DL), which requires that individual library operations appear to be executed atomically in a sequence consistent with the real-time order and, moreover, that recovering from a crash return a state corresponding to a prefix of that sequence. Sadly, however, there are hardly any formal DL proofs, and those that do exist cover the correctness of rather simple persistent algorithms on specific (simplified) persistency models.

In response, we propose a general, powerful, modular, and incremental proof technique that can be used to guide the development and establish DL. Our technique is (1) *general*, in that it is not tied to a specific persistency and/or consistency model, (2) *powerful*, in that it can handle the most advanced persistent algorithms in the literature, (3) *modular*, in that it allows the reuse of an existing linearizability argument, and (4) *incremental*, in that the additional requirements for establishing DL depend on the complexity of the algorithm to be verified. We illustrate this technique on various versions of a persistent set, leading to the link-free set of Zuriel et al.

CCS Concepts: • **General and reference** → Verification; • **Computer systems organization** → Multicore architectures; • **Software and its engineering** → *Formal methods*; • **Theory of computation** → *Concurrent algorithms*; **Program verification**.

Additional Key Words and Phrases: Persistency, Non-Volatile Memory, Px86, Weak Memory Models, Concurrency, Linearizability

ACM Reference Format:
Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2023. The Path to Durable Linearizability. *Proc. ACM Program. Lang.* 7, POPL, Article 26 (January 2023), 27 pages. https://doi.org/10.1145/3571219

1 INTRODUCTION

Non-volatile memory (NVM) [Kawahara et al. 2012; Lee et al. 2009; Strukov et al. 2008] is a new kind of memory, which ensures that its contents survive crashes (e.g. due to power failure) while having performance similar to that of RAM. As such, it has generated a lot of interest in the systems community, with an increasing body of work proposing persistent data structures that can be restored to a consistent state after a crash.

These persistent data structures are typically adaptations of existing *linearizable* concurrent data structures, and so their correctness is given in terms of *durable linearizability* (DL) [Izraelevitz et al. 2016], an extension of linearizability [Herlihy and Wing 1990] to account for crashes. Similar to how linearizability requires all operations to appear to execute atomically in some legal total order consistent with their real-time execution order, DL requires the same to also hold for the

Authors' addresses: Emanuele D'Osualdo, dosualdo@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Azalea Raad, azalea.raad@imperial.ac.uk, Imperial College London, UK; Viktor Vafeiadis, viktor@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/1-ART26
https://doi.org/10.1145/3571219

state recovered after a crash: it should correspond to some legal execution of a subsequence of the operations before the crash containing at least all the operations that completed before the crash.

The adaptations required to make a concurrent data structure persistent, however, are far from trivial, especially when the goal is to achieve optimal performance. The reason is that hardware implementations do not persist every write to NVM immediately, which would automatically turn any linearizable data structure to a persistent one, but rather put them into a buffer to be persisted at some later point. Writes in such buffers can moreover be persisted out of order, leading to *weak persistency* semantics, which is another layer of complexity above the *weak memory consistency* semantics of modern CPUs. To ensure that writes are persisted, programs can issue a special `flush(x)` instruction (e.g. `CLFLUSH x` on Intel-x86 machines), which blocks until all pending writes to `x` are persisted. Introducing an appropriate flush instruction after *every* memory access (i.e. both reads and writes) can restore the sane *strict persistency* [Pelley et al. 2014] model, where the order in which writes persist (the "persistency" or "non-volatile" order) agrees with the order in which they become visible to other processors in the system (the "volatile" order). Doing so, however, incurs a prohibitive cost, and so programmers of persistent libraries strive to use as few flushes as possible, which may in turn require adding auxiliary state to the algorithm or redesigning part of it.

A natural question arises. If persistent data structures are adaptations of concurrent ones, can we establish their correctness by reusing the correctness proof of their concurrent analogues? In other words, is it possible to dissect the DL requirements in a way that we can reuse the invariants established by the linearizability proof? The existing literature sadly does not provide an answer to this question. Most papers (e.g. [Friedman et al. 2018; Zuriel et al. 2019]) come with informal proof arguments in English that do not consider the intricacies of NVM semantics, while in the few papers that come with formal proofs [Raad and Vafeiadis 2018; Raad et al. 2019b] their arguments are highly entangled with a specific memory consistency/persistency model, and are not able to disentangle the concurrency aspects of the proof from the persistency ones.

In this work, we show that such modularity and reuse are possible. We present the first formal proof methodology for verifying durable linearizability of persistent algorithms. Our proof technique enjoys the following four properties:

- It is *modular*, in that it separates out the proof obligations concerning linearizability from those concerning persistency and from those concerning the recovery code, thereby allowing the reuse of an existing linearizability proof.
- It is *general* in that it is not tied to a specific model like epoch persistency [Izraelevitz et al. 2016], Px86 [Raad et al. 2020], or PArm [Raad et al. 2019b], but supports arbitrary models with different volatile and non-volatile orders.
- It is *powerful* in that it can handle the most advanced persistent algorithms in the literature. We will illustrate this point by applying it to produce the first formal DL proof of the recent link-free set of Zuriel et al. [2019], an algorithm, whose linearizability argument is already challenging, as it cannot be shown correct using fixed linearization points.
- It is *incremental* in the sense that the difficulty of the additional requirements for establishing DL is proportional to the complexity of the algorithm to be verified.

Our proof technique is captured by our Pathway Theorem in Section 4, which presents a detailed methodology for establishing DL following the "linearize-first" implementation scheme, where the effects of an operation are first committed to volatile memory and are later persisted, i.e. the operation first reaches its linearization point and later, what we call, its persistency point. The key idea is to add to a proof of linearizability a number of general conditions on commutativity of the operations restricting when operations may be linearized and persisted in different orders.

```
1  record Node:
2    key: ℕ ⊎ {+∞, −∞}
3    nxt: 𝔹 × 𝕏⊥
4
5
6  def find(h, k):
7    p = h
8    ⟨_,c⟩ = p.nxt
9    while(1):
10     if c.nxt == ⟨0,_⟩:
11       if c.key ≥ k:
12         if p.nxt == ⟨0,_⟩:
13           return ⟨p,c⟩
14         c = h
15       p = c
16     else:
17       trim(p, c)
18     ⟨_,c⟩ = c.nxt
19
20
21 def trim(p, c):
22   flush(c)
23   ⟨_,s⟩ = c.nxt
24   CAS(p.nxt, ⟨0,c⟩, ⟨0,s⟩)
25   flush(p)
26 def insert(h, k):
27   while(1):
28     ⟨p,c⟩ = find(h, k)
29     if c.key == k:
30       flush(c); flush(c.orig)
31       return false
32     n = alloc(Node)
33     n.key = k
34     n.nxt = ⟨0,c⟩
35     n.orig = p; flush(n); flush(p.orig)
36     if CAS(p.nxt, ⟨0,c⟩, ⟨0,n⟩):
37       flush(p)
38       return true
39
40 def delete(h, k):
41   while(1):
42     ⟨p,c⟩ = find(h, k)
43     if c.key≠k:
44       return false
45     ⟨b,n⟩ = c.nxt
46     if b == 0:
47       flush(c.orig)
48       if CAS(c.nxt, ⟨0,n⟩, ⟨1,n⟩):
49         trim(p, c)
50         return true
```

Fig. 1. A list-based set implementation (in black). The addition of the flushes (in red) makes it durable.

We also have a dual version of our Pathway Theorem suitable for the data structure implementations following the "persist-first" approach, e.g. SOFT [Zuriel et al. 2019], where persistency points precede linearization.

Outline. We start, in Section 2, with an informal overview of our approach. In Section 3, we develop a memory-model-agnostic definition of durable linearizability, which is used to specify persistent libraries. In Section 4, we present our proof technique culminating in our Pathway Theorem. Then, in Section 5 as an extended case study, we obtain the first formal proof of the link-free set as an application of the Pathway Theorem. We conclude with discussion of related work. All the omitted definitions and proofs, including the full verification of the link-free set can be found in the extended version of this paper [D'Osualdo et al. 2022].

2 OVERVIEW

Our goal is to produce a proof of durable linearizability, starting from a proof of standard linearizability. To make the discussion concrete, we will use a standard set implementation based on Harris [2001] as a running example.

The basic algorithm, shown in black in Fig. 1, is designed to implement a finite set of numeric keys in volatile memory. A set $S \subseteq \mathsf{Key}$ is represented in memory as a singly linked list of nodes. Each node has a key field storing an element of $\mathsf{Key} \uplus \{+\infty, -\infty\}$. Two sentinel head and tail nodes (with keys $-\infty$ and $+\infty$ resp.) are always present. The linked list is ordered in strict increasing order. Each node can be marked or unmarked; only unmarked nodes reachable from the head represent elements of S, and the marked nodes are considered deleted (they can be lazily unlinked from the list). The marking is stored as the least-significant bit of the `nxt` field, the pointer to the next node.

In our pseudo-code we represent the `nxt` field explicitly as a pair $\langle b, p\rangle$ where $b \in \{0, 1\}$ is the marking bit (0 for unmarked, 1 for marked as deleted) and p is the address of the next node.

The set operations are `insert`(h, k) and `delete`(h, k) where h is the address of the head node (fixed throughout the lifetime of the set) and $k \in \mathsf{Key}$ is the key to be inserted/deleted. A successful insert returns `true`; if the k was found to be already in the set, the operation returns `false`. Similarly, the return value of a delete indicates whether the operation was successful.

Linearizability. Linearizability [Herlihy and Wing 1990] ensures that, from the perspective of a client of the library, each call to a library function appears as a single abstract event; furthermore, these library events are ordered by a total order lin which satisfies the abstract semantics of the library (e.g. pops and pushes match), and respects the execution (or "real-time") order of calls.

A common way to prove linearizability, under sequential consistency (SC) [Lamport 1979], is through identifying the *linearization point* of each function call, namely the concrete event (i.e. memory access) in the function implementation that represents the moment when the function's effects become observable to other operations. Such a proof starts by identifying:

- A set of *abstract states*, $\mathbb{Q}$, representing the data-type abstractly presented to the client. Associated with the states is a set of allowed transitions for each operation.
- A *(volatile) representation function*, $\alpha_{\mathsf{vol}} : \mathbb{Q} \to \mathscr{P}(\mathsf{Mem})$, formalising how the specific implementation represents an abstract state in memory.
- For each execution with events E, an injective partial function, $\ell : \mathsf{Cid} \rightharpoonup E$, that identifies the linearization point of each call. The function is partial because there might be pending calls that have not reached their linearization points yet.

In our example, the abstract states are finite sets of numeric keys $S \in \mathscr{P}_{\mathsf{fin}}(\mathsf{Key})$. The transition system on abstract states asserts e.g. that a successful insert of k is only allowed on a set not containing k and leads to a state where k is added to the set. The representation function e.g. constrains the memory representing a set S to be such that all and only the keys in $S \uplus \{+\infty, -\infty\}$ are stored in unmarked nodes in the ordered linked list reachable from the head.

The linearization points for the Harris list are as follows:

- The successful `CAS` at Line 36 linearizes a successful insert.
- A failed insert linearizes at Line 10 during the call to `find` of Line 28.
- The successful `CAS` at Line 48 linearizes a successful delete.
- A failed delete linearizes at Line 10 during the call to `find` of Line 42.

Given α_{vol} and ℓ, linearizability can be then reduced to an induction over the interleaving sequence of events $e_0 \dots e_n$ of an arbitrary execution. Let M_i be the memory contents before e_i is executed. Assuming $M_i \in \alpha_{\mathsf{vol}}(q)$ for some abstract state q, one must prove that:

- if e_i is the linearization point of an operation op, then $M_{i+1} \in \alpha_{\mathsf{vol}}(q')$ for some q' such that (q, q') is a valid transition for op;
- otherwise, the concrete step e_i preserves the abstract state, i.e. $M_{i+1} \in \alpha_{\mathsf{vol}}(q)$.

We refer to the proof scheme above as "induction over execution sequences".

Linearizability in weak memory models. In declarative presentations of weak-memory models, an execution is represented as a graph of events, related through a number of relations (e.g. po, the "program order" in each thread) witnessing the execution's consistency. In such models the notion of "execution order" is in fact weaker than in SC. In particular, the program order po on memory accesses does not necessarily agree with the order in which the accesses take global effect.

To recover a global "execution order" in a proof of linearizability, we can ask the prover to provide, in addition to α_{vol} and ℓ, a strict "volatile order" vo. We can then use vo to order the

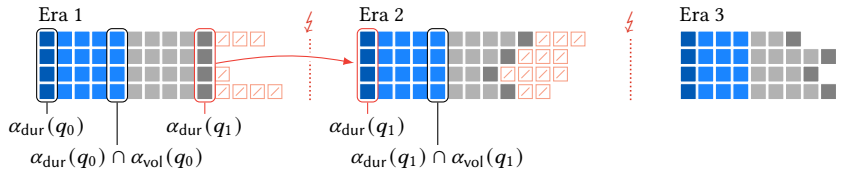


Fig. 2. An execution chain with eras separated by crashes (↯). For each era we draw the initial memory (■), the recovery events (■) the persisted writes (■, ■), and the writes which were executed but had not persisted yet when the crash happened (▨). In each era, the last persisted writes to each location (■) provide the initial memory of the next era. In the last era every write has persisted.

execution sequences in the proof of linearizability. More precisely, we would consider execution sequences $e_0 \dots e_n$ that respect vo. Any such sequence induces a memory $\mathcal{M}[\![e_0 \dots e_n]\!] \in \mathsf{Mem} \triangleq \mathbb{L} \rightharpoonup \mathbb{V}$ which assigns to each location x the last value written to it in $e_0 \dots e_n$. The linearizability proof would then be performed by induction over vo-respecting execution sequences $e_0 \dots e_n$, defining $M_i = \mathcal{M}[\![e_0 \dots e_{i-1}]\!]$.

An essential component of the traditional linearizability definition is that linearization agrees with po between calls. To obtain this we can require that po between linearization points be preserved by vo, i.e. $\mathrm{po}|_{LP} \subseteq \mathrm{vo}$ with where $LP \triangleq \{\ell(c) \mid c \in \mathrm{dom}(\ell)\}$.

Non-volatile memory. Non-volatile memory (NVM) introduces another level of complication. Writes to NVM persist (survive crashes) but not necessarily in the order they were executed. For example, on Px86 (describing the Intel-x86 persistency model) [Raad et al. 2020], if locations x and y are not stored in the same cache line, two sequential writes to x and y may persist in any order. To ensure the write on x persists before that on y, one must issue a `flush` on x before writing to y.

Px86 thus introduces the strict, total "non-volatile order", nvo, on durable events (i.e. writes, updates or flushes) describing in which order these events persist, and a set P of those events that have persisted before the crash. The persisted events P have to be consistent with nvo, i.e. if $(e, e') \in \mathrm{nvo}$ and $e' \in P$ then $e \in P$.

In this setting, linearizability is not an adequate correctness criterion as it does not account for crashes. A linearizable data-structure without any modifications would not be correct under Px86: a crash might leave the persisted memory in an inconsistent state. Specifically, if no flush is issued, there is no guarantee that any change at all is persisted even for operations that already returned to the client. Moreover, even if flushes are issued before returning, pending calls might have already executed their linearization points, making the update observable to other threads, but their updates might not reach the NVM before the crash. In the Harris list, for example, a key k might be inserted in the set, and observed by other concurrent inserts, but after a crash one might find that the node carrying k is not reachable from the head, or that it is, but has an uninitialized `key` field. That is, it is possible for a crash to invalidate the invariants encoded in α_{vol}.

To account for crashes, instead of single executions, formal persistency models consider *execution chains*: sequences of executions, where each execution, called an *era*, is abruptly terminated by a crash (with the exception of the last one). Figure 2 shows an example chain: the shaded area denotes the set of events that have persisted before the crash. The frontier of the persisted events represents the nvo-latest persisted writes to each location: this defines the initial memory of the next era. At the beginning of each era, a data-structure-specific recovery routine is run sequentially before resuming normal execution.

To ensure correctness in the presence of crashes, *durable linearizability* (DL) [Izraelevitz et al. 2016] requires that: (1) each execution era be linearizable; (2) the effects of every completed

(returned) call be persisted before the next crash;[1] (3) concatenating the linearizations of all eras forms a valid linearization. To achieve DL, the programmer has two main tools: flushes, and the recovery procedure run after each crash. Let us first focus on schemes that do not require recovery.

A brute-force way to ensure DL is by issuing a flush after each memory access. Specifically, flushing after a write w ensures that w is persisted before continuing; flushing after a read r ensures that the write observed by r is persisted before continuing.

When proving DL, this scheme ensures that vo includes nvo. As such, since persisted memory is simply $\mathcal{M}[\![\vec{e}]\!]$ where $\vec{e}$ is the nvo-respecting enumeration of the persisted events P (written $\mathcal{M}[\![\mathsf{nvo}|_P]\!]$), proving linearizability using ℓ and α_{vol} proves that the original volatile invariants are now maintained in persistent memory. When a crash occurs, the persisted memory belongs to $\alpha_{\mathsf{vol}}(q)$ for some legal q and the post-crash execution can continue without recovery.

While this aggressive flushing strategy allows for a straightforward adaptation of linearizability to DL, it yields poor performance. For this reason, libraries such as FliT [Wei et al. 2022] and Mirror [Friedman et al. 2021] employ alternatives that more efficiently implement a strict persistency abstraction on top of weaker models. Conceptually, these libraries improve performance by avoiding redundant flushes on the same write. Is it possible to do better? Indeed, Cohen et al. [2018] proved that any DL library can be implemented using one flush per operation, which is far lower than what using Mirror or FliT can produce. To approach this optimum, it is necessary to optimise flushes manually, and genuinely relax the order of persistency on writes.

A first optimisation. An analysis of the Harris list example reveals that flushing during `find` is not strictly necessary: when traversing keys $k_1, \ldots, k_n$ on the way to finding k, the presence (or absence) of k_i in the abstract set does not influence whether inserting/deleting k is legal. Therefore, observing a key $k_i \neq k$ in the set during traversal does not require the insertion/marking of k_i to be persisted: the result of inserting k does not reveal information about the presence or absence of k_i. This is the key insight of NVTraverse [Friedman et al. 2020], proposing a flushing scheme for tree-based data structures with traverse-and-update operations, with no flushes during traversal.

The program in Fig. 1 with the inclusion of the commands in red, instantiates the scheme for the Harris list as follows. Consider insertions: a successful insert must first persist (flush) the new node n. The second obvious flush needed is the one of `p` after the successful `CAS` which inserted n. These flushes alone, however, are insufficient: when the `CAS` on `p` swings the pointer, we cannot be sure `p` is reachable from the head *in persistent memory*. There could be a long list of pending inserts of keys $k_1 \ldots k_n$ which all executed their linearization points but have not reached the final flush; when this is the case, a concurrent insert of k_n can traverse the $k_1 \ldots k_n$ nodes without flushing them, persist k_n and k_{n-1}, and report to the client that k_n is already in the set. If a crash happens then, the node storing k_n would not be reachable from the head. The solution is to ensure `p` is reachable from the head in persistent memory by flushing the node that initially made `p` reachable, i.e. its *origin*. We thus record the origin of each node in its `orig` field. The other flushes are issued with analogous motivations.

This more sophisticated scheme cannot be proven by simply adapting the linearizability argument. In fact, the order of persistency is relaxed, and as a consequence $\mathsf{nvo} \not\subseteq \mathsf{vo}$, contrary to the brute-force approach. A synthetic example that shows this basic pattern is reproduced in Fig. 3(a), comprising two concurrent operations op_1 (on the left) and op_2 (on the right). The two `CAS` instructions represent linearization points of op_1 and op_2; `x` and `y` are distinct locations storing `0` initially. Figure 3(b)

[1]The requirement that completed calls must have persisted is only achievable if the `flush` primitive is synchronous, i.e. blocks until it takes effect. When flushes are asynchronous, the correctness criterion can be weakened to *buffered* durable linearizability that removes the constraint on completed calls. In this paper we will only consider synchronous flushes and the unbuffered version of durable linearizability.

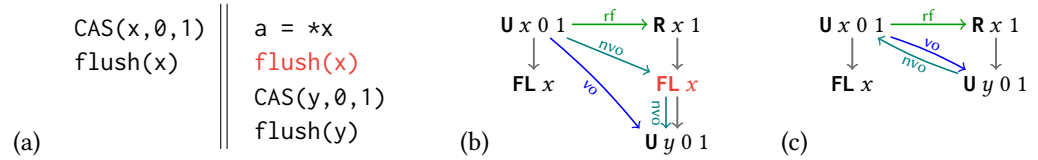


Fig. 3. Optimizing flushes might introduce disagreement between linearization and persistency orders. Figure (a) shows two parallel operations with each CAS acting as a linearization point. Removing the flush in red allows y to be persisted before x is. Figures (b) and (c) explain why in terms of an execution graphs generated by the program. Each memory access is a node labelled with **U** (successful CAS), **R** (read), or **FL** (flush). The unlabelled arrows indicate program order, rf is the "reads-from" order. Figure (b) is generated by the program with the red flush, which forces nvo and vo to order the CASes in the same way. Figure (c) is a possible execution if the red flush is removed.

shows a possible execution graph generated by the program under Px86: every memory access leads to a node, labelled with the access' effect. The linearization of op_2 is observed via a read by op_1 as signified by the "read-from" rf edge, but the linearization point on y does not depend on the value read from x. This observation of the x value, however, implies that the CAS on x comes vo-before the CAS on y. The brute-force flushing strategy would mandate the issuing of the red flush (in op_2) after the read of x, thus implying that vo on the CASes is the order in which they will be persisted (thus vo and nvo agree on them). The read of x in op_2 represents a read during a traversal, and the corresponding flush would be optimized away by the NVTraverse-style optimization. Without the red flush, the CASes can be persisted in either order, and thus it is possible for nvo and vo to order them differently. This is shown in the execution graph of Fig. 3(c) generated by the program with the red flush removed.

The overall difference induced by the optimization can be observed if a crash happens after both CASes have executed, and the CAS on y has persisted. In the unoptimized version, the CAS on x would be persisted too, yielding x = y = 1. In the optimized version, we might see x = 0 and y = 1 after the crash. The question is: when is such optimisation sound? We propose to look at the question by identifying the discrepancies between the linearizations constructed by the (volatile) linearizability argument using α_{vol}, ℓ and vo and a DL argument built from α_{vol}, ℓ and nvo. These discrepancies can be grouped in two categories: (1) two linearization points might be ordered one way by vo and the other way by nvo; and (2) operations whose linearization points are reads are not meaningfully ordered by nvo. The proof strategy we propose is to first prove (volatile) linearizability, and then prove some properties that entail that the legality of the nvo-induced order on linearization points follows from the legality of the vo-induced one. Concretely, for each execution we ask to prove:

(1) Linearizability of the execution using α_{vol}, ℓ and vo.
(2) If two linearization points $\ell(c_1)$ and $\ell(c_2)$ are such that $\ell(c_1) \xrightarrow{\text{nvo}} \ell(c_2)$ then either $\ell(c_1) \xrightarrow{\text{vo}} \ell(c_2)$, or the calls c_1 and c_2 *abstractly* commute.

By "abstractly commute" here we mean that, according to the library specification, c_1 followed by c_2 produces the same abstract state as c_2 followed by c_1. What these two conditions imply is that the sequence of linearization points in nvo is legal; therefore any prefix of it is legal. What is left to prove is that the prefix that persisted (i.e. the prefix in P) produces some abstract state q which is the one encoded in persisted memory:

(3) $\mathcal{M}[\![\text{nvo}|_P]\!] \in \alpha_{\text{vol}}(q)$.

For the Harris list, this strategy allows us to reuse the vanilla linearization argument (flushes and the `orig` field have no bearing on that proof). The proof of condition (2) is done by contraposition:

one shows that it is impossible for two non-commuting operations to be ordered in opposite ways by vo and nvo. This is because an operation on a key k always flushes the nodes relevant to k before linearizing. For example, take an insert and a vo-subsequent delete of k, where no other operation on k took place in between in vo order. The delete must have found the inserted node `c` in the linked list (we know this from the vanilla linearization proof) and therefore flushed the node that first made `c` reachable at Line 47. Since the linearization point of the insert is the update of the next pointer of `c.orig`, the flush implies that the linearization point of the insert is nvo-before that of the delete. When proving condition (3), we already know that the nvo-induced sequence is legal; we can also safely ignore linearization points which are reads because they have already been proven sound in condition (1).

Decoupling recovery. So far we have only considered schemes with a trivial recovery. This imposes a greater onus on the implementation which has to ensure that persisted memory satisfies the invariants at all times. The possibility of using a non-trivial recovery opens opportunities for much more radical optimizations. For instance, in the Harris list, if we know which nodes belong to the set, the links between the nodes are indeed redundant and can be reconstructed upon recovery.

Before examining how these more sophisticated optimisations work, we structure our proof technique in a way that decouples the correctness of the recovery and the correctness of the other library operations. To do so, we introduce a further parameter of a DL proof: the *durable* representation function $\alpha_{\mathsf{dur}} : \mathbb{Q} \to \mathscr{P}(\mathsf{Mem})$. The idea is that $\alpha_{\mathsf{vol}}(q)$ imposes stronger (or incomparable) constraints on the memory than $\alpha_{\mathsf{dur}}(q)$, and the constraints of $\alpha_{\mathsf{dur}}(q)$ are enough for the recovery to repair the memory in such a way that it belongs to $\alpha_{\mathsf{vol}}(q)$ before resuming execution.

The verification of DL then comprises two parts:

(1) Verifying that each step of the recovery takes any $M \in \alpha_{\mathsf{dur}}(q)$ to some $M' \in \alpha_{\mathsf{dur}}(q)$, and when the recovery terminates the memory is in $\alpha_{\mathsf{dur}}(q) \cap \alpha_{\mathsf{vol}}(q)$. Note that this verification can be done under SC since the recovery is sequential.
(2) Verifying that an execution of the operations, with initial memory in $\alpha_{\mathsf{dur}}(q) \cap \alpha_{\mathsf{vol}}(q)$ is DL, producing a history taking q to q' and producing a persisted memory in $\alpha_{\mathsf{dur}}(q')$.

These proof obligations allow for an inductive argument proving that any chain is durably linearizable, as illustrated in Fig. 2. The schemes so far had $\alpha_{\mathsf{dur}} = \alpha_{\mathsf{vol}}$. Let us next consider a variation of the Harris list with a non-trivial recovery called a "link-free set" [Zuriel et al. 2019].

Decoupling linearization and persistency points. The idea of the link-free set, in Fig. 4, is to adopt a Harris list data structure as the volatile representation of the set, but let the persistent representation forgo the links between nodes. More precisely, each node has a boolean `valid` field (set to 0 at allocation) indicating whether the node is a *persistent* member of the set, if not marked as deleted. Provided the validity field is updated correctly, the recovery can scan the memory for all the allocated nodes and newly arrange all the ones that are unmarked (not deleted) and valid (persisted) into a sorted linked list. This then eliminates the need for persisting the updates on links in any specific order. To ensure consistency between the volatile view and the persisted view, the link-free set adopts a "linearize first, persist second" strategy: a new node is first inserted in the volatile linked list, where it becomes visible to other threads, then its validity field is set and the node is flushed. If another operation observes the node (for example a concurrent insert of the same key returning false) then it will *help* persist the node by first setting its validity and then flushing it (possibly with a benign race, generating a redundant flush), before returning. As in the optimized version before, nodes that are traversed but do not influence the legality of some operation need not be helped to persist.

```
1  record Node:
2    key: ℕ ⊎ {+∞, −∞}
3    nxt: 𝔹 × 𝕏⊥
4    valid: 𝔹
5
6  def init():
7    t = alloc(Node)
8    t.key = +∞; t.valid = 1
9    t.nxt = ⟨0,⊥⟩
10   h = alloc(Node)
11   h.key = −∞; h.valid = 1
12   h.nxt = ⟨0,t⟩
13   return h
14
15
16 def find(h, k):
17   p = h
18   ⟨_,c⟩ = p.nxt
19   while(1):
20     if c.nxt == ⟨0,_⟩:
21       if c.key ≥ k:
22         if p.nxt == ⟨0,_⟩:
23           return ⟨p,c⟩
24         c = h
25       p = c
26     else:
27       trim(p, c)
28     ⟨_,c⟩ = c.nxt
29
30 def trim(p, c):
31   flush(c)
32   ⟨_,s⟩ = c.nxt
33   CAS(p.nxt, ⟨0,c⟩, ⟨0,s⟩)
34 def insert(h, k):
35   while(1):
36     ⟨p,c⟩ = find(h, k)
37     if c.key == k:
38       c.valid = 1
39       flush(c)
40       return false
41     n = alloc(Node)
42     n.key = k
43     n.nxt = ⟨0,c⟩
44     if CAS(p.nxt, ⟨0,c⟩, ⟨0,n⟩):
45       n.valid = 1
46       flush(n)
47       return true
48
49 def delete(h, k):
50   while(1):
51     ⟨p,c⟩ = find(h, k)
52     if c.key≠k:
53       return false
54     ⟨_,n⟩ = c.nxt
55     c.valid = 1
56     if CAS(c.nxt, ⟨0,n⟩, ⟨1,n⟩):
57       trim(p, c)
58       return true
59
60 def recover(nodes):
61   h = init()
62   for n in nodes:
63     if n.valid == 1:
64       ⟨b,_⟩ = n.nxt
65       if b == 0 and n.key ∈ Key:
66         seqInsert(h, n)
```

Fig. 4. A simple link-free set implementation.

This optimization has a number of ramifications for verification. First, we now have $\alpha_{\mathsf{vol}} \neq \alpha_{\mathsf{dur}}$: the volatile representation insists that the links must describe a sorted linked list; the durable representation only asks that there be a unique, valid and unmarked node for each key in the set. Second, the original linearization points are still valid with respect to the volatile structure, but they do not represent the point where the update they implement is made persistent. For inserts, for instance, the update to the links is the volatile linearization point, but even if that update is persisted, the inserted node would still be seen as not part of the set after a crash, until the validity field is set and persisted. The latter update makes the node persistently inserted; we call this kind of update a *persistency point*. This duplicity directly reflects the difference between α_{vol} and α_{dur}: while linearization points induce an update from a memory in $\alpha_{\mathsf{vol}}(q)$ to a memory in $\alpha_{\mathsf{vol}}(q')$, persistency points (if persisted) change the persisted memory from one in $\alpha_{\mathsf{dur}}(q)$ to one in $\alpha_{\mathsf{dur}}(q')$.

We thus ask the prover to specify persistency points using a partial function $p\colon \mathsf{Cid} \rightharpoonup P$: $p(c)$ is the persisted event that records the effect of the call c in persistent memory. The function is partial because the persistency point of a call might not have been executed/persisted yet.

The persistency points of the link-free set example are as follows:

- For successful inserts, it is the moment when the inserted node is first made valid, i.e. on Line 45 of Fig. 4 at the latest.
- For successful deletes, it coincides with the operation's linearization point, Line 56 of Fig. 4.

We have thus two overlaid linearization arguments: the volatile one on α_{vol}, ℓ, vo; and the persistent one on α_{dur}, p, nvo. On one hand, to prove DL it would suffice to provide the argument on α_{dur}, p, nvo. On the other hand, however, proving linearizability directly on nvo is challenging. First, as we noted above, we would need some special treatment of "read" operations. Second, and more important, the reasons for the legality of the sequence are typically justified by the volatile data structure, and not just the persisted one. For instance, the reason why a successful insert of k is legal is due to the traversal of the linked list providing evidence that no other unmarked node holding k is present. Since the links in persisted memory might not be consistent, this argument cannot consider persisted memory only. As such, just as we sketched above, we propose a proof scheme that allows most of the correctness argument to be done on vo. Then, we identify the potentially problematic reorderings of the linearization produced by the mismatch between ℓ and p, and vo and nvo. For those problematic reorderings we require the operations involved to commute. Together, these conditions would entail that the linearization induced by p and nvo is legal. Then, one needs to verify that the final abstract state reached through this legal linearization is in fact the one recoverably encoded in the final persisted memory.

All in all, our proof technique requires to prove, roughly:

(1) Linearizability of the execution using α_{vol}, ℓ and vo.
(2) If $p(c_1) \xrightarrow{\text{nvo}} p(c_2)$, then either $\ell(c_1) \xrightarrow{\text{vo}} \ell(c_2)$ or c_1 and c_2 abstractly commute.
(3) If $\ell(c_1) \xrightarrow{\text{vo}} \ell(c_2)$ but $p(c_1) = \bot$, then either $p(c_2) = \bot$ or c_1 and c_2 abstractly commute.
(4) Assuming the linearization induced by p and nvo on persisted events abstractly produces a state q', the persisted memory belongs to $\alpha_{\mathsf{dur}}(q')$.

Condition (2) considers pairs of calls that have linearized and persisted, but such that volatile and persistent linearizations would disagree on their ordering. In that case they are required to abstractly commute. This allows us to perform reorderings of the volatile linearization until the sequence respects nvo, while preserving its legality.

Condition (3) corrects for what we call the "voided" calls: those that are linearized in the middle of the volatile linearization, but have not persisted. They are required to commute with all the persisted calls in front of them. This ensures that we can move all of them at the end of the linearization and then remove them, while preserving legality.

Our Pathway Theorem (Section 4) formalises a generalisation of this scheme. As we show in Section 5 the scheme makes it possible to prove the volatile linearization argument first, and exploit it to deduce the invariants needed to show the commutation lemmas. The necessity of proving these lemmas can be seen as the underlying motivation for the placement of the flushes. Finally, proving that the persisted memory representation of states is correct can be done while assuming the sequence of persistency points is legal, effectively making available the relevant volatile invariants in support of the persistent argument.

Our full proof technique also supports two advanced techniques: *hindsight linearization* and *persist-first implementations*. Hindsight [O'Hearn et al. 2010] refers to linearizable operations for which it is not possible to find a fixed event representing their linearization point. Their correctness is thus proven "after the fact". Our General Pathway Theorem, presented in [D'Osualdo et al. 2022], supports hindsight by allowing a standard linearization for the other operations to be carried out first, and then adding hindsight operations, with a limited impact on the commutation conditions.

Persist-first implementations (e.g. SOFT [Raad et al. 2020] and Mirror [Friedman et al. 2021]) maintain two versions of their data, one in persistent memory and one volatile version used for enabling fast access, and write first to the persistent version and then update the volatile one. This reduces the needed flushes to the lowest theoretical bound. The commutation conditions we presented apply to the "linearize-first" implementations. In Section 4.4 we present a Persist-First Pathway Theorem which, using dual commutation conditions, applies to persist-first schemes.

3 OPERATIONAL MODEL AND DURABLE LINEARIZABILITY

3.1 Preliminaries

Relations. We write $[X]$ for the identity relation on X, rel^+ for transitive closure, rel^* for reflexive transitive closure, $\mathsf{rel}^?$ for reflexive closure, $\mathsf{rel}^! \triangleq \{(x,y) \in \mathsf{rel} \mid \nexists z.(x,z) \in \mathsf{rel} \wedge (z,y) \in \mathsf{rel}\}$, and rel^{-1} for inverse. We say rel is *acyclic* if rel^+ is irreflexive.

Sequences. We use the notation $\vec{e}$ to range over finite sequences, $|\vec{e}|$ for the length of the sequence, $\vec{e}(i)$ for the item at position $0 \le i < |\vec{e}|$ in the sequence, $(\vec{e})_{i]}$ for the sequence $\vec{e}(0) \dots \vec{e}(i)$, and $(\vec{e})_{|i}$ for the sequence $\vec{e}(i) \dots \vec{e}(|\vec{e}|-1)$. The empty sequence is denoted by ϵ. We sometimes implicitly coerce $\vec{e}$ to the set of its items. Given a set A we write A^* for the set of finite sequences of elements of A. We write $\vec{e} \cdot \vec{e}'$ for the concatenation of the two sequences. Given sequences $\vec{e}, \vec{e}' \in A^*$, we say $\vec{e}$ is a *scattered subsequence* of $\vec{e}'$, if all the items of $\vec{e}$ appear in $\vec{e}'$ in the same order. The expression $\vec{e}|_B$ denotes the longest scattered subsequence of $\vec{e}$ consisting only of elements of B, e.g. $cabcbacb|_{\{a,b\}} = abbab$. For $\vec{e} \in A^*$, we also write $\vec{e} \setminus B$ for $\vec{e}|_{A \setminus B}$.

Definition 3.1 (Enumeration). Given a relation $\mathsf{rel} \subseteq A \times A$, and a finite set $X \subseteq A$ with n elements, we write $\mathsf{rel}^{\uparrow}_X$ for the set of enumerations $x_0 \dots x_n$ of X such that $\forall i, j \le n.(x_i, x_j) \in \mathsf{rel} \Rightarrow i < j$. Notice that if rel is an acyclic relation, then $\mathsf{rel}^{\uparrow}_X \neq \emptyset$. If rel is a strict total order on X, then $\mathsf{rel}^{\uparrow}_X = \{\vec{e}\}$ and we write $\mathsf{rel}^{\uparrow}_X$ for $\vec{e}$ directly. We omit X when clear from the context.

Partial functions. We write $f \colon A \rightharpoonup B$ if f is a partial function from A to B, i.e. a function of type $f \colon A \to B \uplus \{\bot\}$. The *domain* of f is $\mathrm{dom}(f) \triangleq \{a \in A \mid f(a) \neq \bot\}$. The *range* of f is $\mathrm{rng}(f) \triangleq \{f(a) \in A \mid a \in A, f(a) \neq \bot\}$. We say f is finite if its domain is finite.

3.2 Actions and Events

We make a number of simplifying modelling choices. First, we only model the scenario where the whole of working memory is NVM. Second, we abstract memory management issues and we will assume memory is managed and garbage collected. We thus include an atomic allocation primitive but no de-allocation. Third, the algorithms we are interested in do not use pointer arithmetic, so we will only model pointers as opaque references and prove no null-dereference is possible. Finally, we model the heap as a uniform collection of structured records, with some fixed finite set of field names $\mathbb{F}$. None of these choices are fundamental.

A *location* $l \in \mathbb{L} \triangleq \mathbb{X} \times \mathbb{F}$ is a pair of an address $x \in \mathbb{X}$ and a field name $\mathtt{f} \in \mathbb{F}$, and we will write them as $x.\mathtt{f}$. The set $\mathbb{V}$ collects all possible values associated to fields. The set of locations is partitioned into *cache lines* $\mathbb{CL}$. The fields of an address are assumed to fit in a single cache line, so we postulate that: $\forall \vec{l} \in \mathbb{CL} \colon x.\mathtt{f} \in \vec{l} \Rightarrow \forall \mathtt{f}' \in \mathbb{F} \colon x.\mathtt{f}' \in \vec{l}$.

The set Act is the set of *actions* α which are of the form:

$$\alpha ::= \mathbf{R}\, x.\mathtt{f}\, v \mid \mathbf{W}\, x.\mathtt{f}\, v \mid \mathbf{U}\, x.\mathtt{f}\, v\, v' \mid \mathbf{MF} \mid \mathbf{FL}\, x \mid \mathbf{A}\, x \mid \mathbf{Ret}\, v \mid \mathbf{Err}$$

where $x \in \mathbb{X}$, $\mathtt{f} \in \mathbb{F}$, $v, v' \in \mathbb{V}$. We include the standard read (**R**), write (**W**) and update (**U**) actions, memory fences (**MF**), flushes (**FL**), and three non-standard actions. Allocation actions **A** x initialise all the fields at a fresh x with zero. Return actions **Ret** v represent the return instruction of a library

call; we will use them as the atomic event representing the whole invocation in the abstract traces of linearizable libraries. An error action **Err** is emitted when reading from or writing to a location with invalid address ($\bot$ or not allocated).

Each action (bar **Ret** v) mentions a single address x which we can access with $\mathrm{addr}(\alpha)$. Similarly, $\mathrm{loc}(\alpha)$ is the location mentioned in an action, if any. As an exception, $\mathrm{loc}(\mathbf{A}\ x)$ is the set $\{x.\mathtt{f} \mid \mathtt{f} \in \mathbb{F}\}$ since an allocation initializes all fields to zero. The value of a return action is $\mathrm{val}(\mathbf{Ret}\ v) \triangleq v$. We also speak of the *read value* ($\mathrm{val_r}$) and *written value* ($\mathrm{val_w}$) of an action: $\mathrm{val_r}(\mathbf{R}\ x.\mathtt{f}\ v) \triangleq \mathrm{val_r}(\mathbf{U}\ x.\mathtt{f}\ v\ v') \triangleq v$, $\mathrm{val_w}(\mathbf{W}\ x.\mathtt{f}\ v') \triangleq \mathrm{val_w}(\mathbf{U}\ x.\mathtt{f}\ v\ v') \triangleq v'$, and $\mathrm{val_w}(\mathbf{A}\ x) \triangleq 0$. We assume a fixed set of operation names Op. For the set library $\mathrm{Op} = \{\mathtt{insert}, \mathtt{delete}\}$.

We also assume an enumerable universe of *events* $\mathbb{E}$ equipped with three functions:

- $\mathrm{act}\colon \mathbb{E} \to \mathrm{Act}$, associating an action to every event. We lift functions on actions to events in the obvious way, e.g. $\mathrm{loc}(e) = \mathrm{loc}(\mathrm{act}(e))$ and write $(e : \alpha)$ to indicate that $\mathrm{act}(e) = \alpha$.
- $\mathrm{cid}\colon \mathbb{E} \to \mathrm{Cid}_\bot \uplus \{\mathrm{rid}\}$, associating a *call identifier* to every event and $\bot$ to client events. Here Cid is a fixed enumerable set of call identifiers, and rid is a special identifier reserved for the call to the recovery procedure; We require $\mathrm{cid}(e) \neq \bot$ if $(e : \mathbf{Ret}\ _)$.
- $\mathrm{call}\colon \mathrm{Cid} \to \mathrm{Call}$, where $\mathrm{Call} \triangleq (\mathrm{Op} \times \mathbb{V}^*)$, returns the operation called and its parameters.

The following sets group events by their action type:

$$\mathbb{U} \triangleq \{e \in \mathbb{E} \mid e : \mathbf{U}\ x.\mathtt{f}\ v\ v'\} \qquad \mathbb{MF} \triangleq \{e \in \mathbb{E} \mid e : \mathbf{MF}\} \qquad \mathbb{FL} \triangleq \{e \in \mathbb{E} \mid e : \mathbf{FL}\ x\}$$
$$\mathbb{W} \triangleq \{e \in \mathbb{E} \mid (e : \mathbf{W}\ x.\mathtt{f}\ v) \vee (e : \mathbf{A}\ x)\} \qquad \mathbb{WU} \triangleq \mathbb{W} \cup \mathbb{U} \qquad \mathbb{D} \triangleq \mathbb{W} \cup \mathbb{U} \cup \mathbb{FL}$$
$$\mathbb{R} \triangleq \{e \in \mathbb{E} \mid e : \mathbf{R}\ x.\mathtt{f}\ v\} \qquad \mathbb{RU} \triangleq \mathbb{R} \cup \mathbb{U}$$

We also group events based on their call identifier:

$$\mathbb{RET} \triangleq \{e \in \mathbb{E} \mid e : \mathbf{Ret}\ v, \mathrm{cid}(e) \in \mathrm{Cid}\} \qquad \mathbb{C}_i \triangleq \{e \in \mathbb{E} \mid \mathrm{cid}(e) = i\}$$
$$\mathbb{E}^{\mathrm{lib}} \triangleq \{e \in \mathbb{E} \setminus \mathbb{RET} \mid \mathrm{cid}(e) \neq \bot\} \qquad \mathrm{cid}_= \triangleq \{(e_1, e_2) \mid \mathrm{cid}(e_1) = \mathrm{cid}(e_1) \neq \bot\}$$

The set $\mathbb{RET}$ collects all return events associated with calls (excluding the one of the recovery), the set $\mathbb{C}_i$ collects all events of the call identified by i, the set $\mathbb{E}^{\mathrm{lib}}$ includes all internal library events (returns are considered to be visible by the client). The relation $\mathrm{cid}_=$ relates all events belonging to the same call. Subscripting a set of events with a location selects the events for which that location is relevant: for each of the sets of events $\mathbb{S}$ defined above, their location-subscripted variant is $\mathbb{S}_l = \mathbb{S} \cap \mathbb{E}_l$ and $\mathbb{S}_L = \mathbb{S} \cap \mathbb{E}_L$, where $\mathbb{E}_l \triangleq \{e \in \mathbb{E} \mid l \in \mathrm{loc}(e)\}$, and $\mathbb{E}_L \triangleq \{e \in \mathbb{E} \mid L \cap \mathrm{loc}(e) \neq \emptyset\}$.

3.3 Executions

We adopt the declarative approach of weak memory model specifications, where executions are represented using graphs of events and dependency relations. The events of an execution should be understood as the concrete low-level instructions issued by a closed multi-threaded program. In the context of the verification of a library, this closed program would be an arbitrary client issuing both instructions produced by calls to the library, and arbitrary instructions on its own locations (which are assumed to be disjoint from the ones manipulated by the library).

Definition 3.2 (Execution). An *execution* is a structure $G = \langle E, I, P, \mathrm{po}, \mathrm{rf}, \mathrm{mo}, \mathrm{nvo}\rangle$ where

- $E \subseteq \mathbb{E}$ is a finite set of events. In the context of the execution G, the sets $\mathbb{W}$, $\mathbb{R}$, etc should be understood as subsets of E. Moreover, $G.\mathrm{Cid} = \mathrm{cid}(E)$.
- $I \subseteq \mathbb{W}$ is the set of *initialisation events*, with $\forall e_1, e_2 \in I\colon e_1 \neq e_2 \Rightarrow \mathrm{loc}(e_1) \cap \mathrm{loc}(e_2) = \emptyset$ and $\forall e \in I\colon \mathrm{cid}(e) = \bot$. Moreover, there are no double allocations: for all $e \in E$ with $(e : \mathbf{A}\ x)$, $\forall e' \in I\colon \mathrm{addr}(e') \neq x$ and $\forall e' \in E\colon ((e' : \mathbf{A}\ y) \wedge e' \neq e) \Rightarrow x \neq y$.
- $P \subseteq \mathbb{D}$ is the set of *persisted events*, with $I \cup \mathbb{FL} \subseteq P$.

- $\mathsf{po} \subseteq E \times E$ is the *'program-order' relation*, a strict partial order with $I \times (E \setminus I) \subseteq \mathsf{po}$.
- $\mathsf{rf} \subseteq \mathbb{WU} \times \mathbb{RU}$ is the *'reads-from' relation* between events of the same location with matching values; i.e. $\forall (a, b) \in \mathsf{rf}$: $\mathrm{loc}(a) = \mathrm{loc}(b) \wedge \mathrm{val_w}(a) = \mathrm{val_r}(b)$. Moreover, rf is total and functional on its range, i.e. every read or update is related to exactly one write or update.
- $\mathsf{mo} \subseteq E \times E$ is the *'modification-order'*, required to be a disjoint union of relations $\{\mathsf{mo}_l\}_{l \in \mathbb{L}}$, such that each mo_l is a strict total order on $\mathbb{WU}_l$, and $I_l \times (\mathbb{WU}_l \setminus I_l) \subseteq \mathsf{mo}_l$.
- $\mathsf{nvo} \subseteq \mathbb{D} \times \mathbb{D}$ is the *'non-volatile-order'*, required to be a strict total order on $\mathbb{D}$, such that $I \times (\mathbb{D} \setminus I) \subseteq \mathsf{nvo}$ and $(e_1, e_2) \in \mathsf{nvo} \wedge e_2 \in P \Rightarrow e_1 \in P$.

The derived *happens-before* relation is defined as $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$.

A memory model is characterised by the subset of executions that are feasible, called *consistent* executions. Different models can be used by adopting a different consistency criterion.

Although our proof technique applies independently of the choice of consistency criterion, we will articulate it on the Px86 memory model, defined in full in [D'Osualdo et al. 2022]. An execution G is *Px86-consistent* if there exists a strict *total store order* $\mathsf{tso} \subseteq G.E \times G.E$ representing the global order in which durable instructions are observed to affect the memory, which satisfies the usual x86 axioms [Sewell et al. 2010]. In addition, tso must satisfy the following three conditions:

$$\forall \vec{l} \in \mathbb{CL}\colon [\mathbb{D}_{\vec{l}}]; \mathsf{tso}; [\mathbb{D}_{\vec{l}}] \subseteq \mathsf{nvo} \qquad [\mathbb{FL}]; \mathsf{tso}; [\mathbb{D}] \subseteq \mathsf{nvo} \qquad E \cap \mathbb{FL} \subseteq P$$

The first condition requires that all the durable events on the same cache line be persisted in the same order in which they affected the volatile memory. The second condition says that durable events tso-following a flush will be persisted after the flush (and thus after all the durable events on the flushed cache line that happened tso-before the flush). The third condition characterises the *synchronous* flush semantics: flushes are persisted as soon as they are included in an execution.

This model deviates slightly from Px86_{sim} of Raad et al. [2020], where nvo only preserves tso on durable events on the same *location*, not the same *cache line*. Our stronger semantics is consistent with the actual hardware implementations [SNIA 2017, §10.1.1]. In fact, the algorithms of [Zuriel et al. 2019] are only correct and optimal under the stronger model we adopt in this paper. For instance, for the `insert` in Fig. 4 it is crucial that the `valid` and `key` fields get persisted together, or a crash might leave a valid node in memory with an uninitialized key.

Definition 3.3 (Memory). The *memory* relative to some sequence of events $\vec{e}$, is the finite partial function $\mathcal{M}[\![\vec{e}]\!] \colon \mathbb{L} \rightharpoonup \mathbb{V}$ defined as: $\mathcal{M}[\![\vec{e}]\!](l) \triangleq \mathrm{val_w}(\vec{e}(i))$ where $i = \max\{j \mid \vec{e}(j) \in \mathbb{WU}_l\}$. The function is undefined on l if $\vec{e} \cap \mathbb{WU}_l = \emptyset$. If $\mathrm{rel} \subseteq \mathbb{E} \times \mathbb{E}$ is a strict total order on $\mathbb{WU}$, then $\mathrm{rel}^{\uparrow}_{\mathbb{WU}} = \{\vec{e}\}$ for some $\vec{e}$, and we write $\mathcal{M}[\![\mathrm{rel}]\!]$ for $\mathcal{M}[\![\vec{e}\,]\!]$. We write Mem for the set of finite partial functions from locations to values.

A set of initial events, by definition, contains at most one write event per location. Memories and sets of initial events are therefore in a 1-to-1 correspondence modulo identity of events. By virtue of this, we shall implicitly coerce memories into sets of initial events and vice versa.

Definition 3.4 (Chain). A *chain* is a sequence $G_0 \ldots G_n$ of consistent executions such that $G_0.I = \emptyset$, $G_{i+1}.I = \mathcal{M}[\![G_i.\mathsf{nvo}|_{G_i.P}]\!]$ for every $0 \leq i < n$, and $G_n.P = G_n.\mathbb{D}$.

A library implementation is a pair $\mathcal{I} = \langle \mathcal{I}_{\text{op}}, \mathcal{I}_{\text{rec}} \rangle$ where $\mathcal{I}_{\text{op}}$ describes the implementation of each operation, and $\mathcal{I}_{\text{rec}}$ describes the implementation of the recovery. An *execution of* $\mathcal{I}$ is a consistent execution where the recovery is run sequentially at the beginning, and then arbitrary client events and calls of operations run concurrently. We only consider executions where libraries and clients work on disjoint locations. A chain of $\mathcal{I}$ is a chain of executions of $\mathcal{I}$. The formal definitions are unsurprising and relegated to [D'Osualdo et al. 2022].

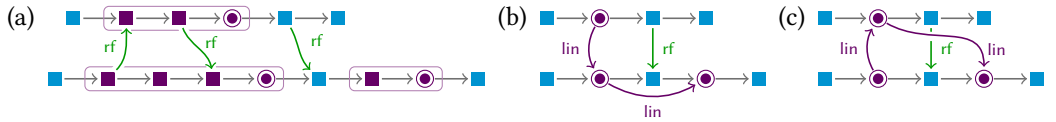


Fig. 5. The concrete execution graph of Figure (a) shows client events (■), and library events (■) and return events (◉) generated by calls (encircled). Figures (a) and (b) represent the only two possible abstract executions of Figure (a): the rf edge between client events constrains the possible linearization orders.

An execution of a library implementation contains both client events, and internal events that are conceptually opaque to the client. From the perspective of the client, all operation calls should be viewable as instantaneous return events giving back control to the client, ordered by some (legal) total order. Abstract executions encode such a client-side view of an execution. Since some calls might not have returned yet, the abstract execution can insert the return events for the calls that have not returned yet but have already conceptually carried out their work. An important constraint is that the abstract execution sees all return events (including the inserted ones) as being persisted. This means that the operations need to ensure their updates have been persisted before returning, to be consistent with their abstract execution.

Definition 3.5 (Abstract execution). Fix an execution G. A set of *completion events* for G is a set $C \subseteq \mathbb{RET} \setminus G.\mathbb{RET}$ such that $\forall e \in C. \exists e' \in G.E.\ \mathsf{cid}(e) = \mathsf{cid}(e') \wedge G.\mathsf{ret}(e') = \bot$.

Given a set C of completion events for G, we define the execution $G^\sharp_C$ as follows:

- $G^\sharp_C.E = E^\sharp \triangleq (G.E \setminus \mathbb{E}^{\mathsf{lib}}) \uplus C$.
- $G^\sharp_C.P = (G.P \setminus \mathbb{E}^{\mathsf{lib}})$.
- $G^\sharp_C.\mathsf{po} = (G.\mathsf{po} \cup C_{\mathsf{po}})^+|_{E^\sharp}$ where $C_{\mathsf{po}} = (G.E \times C) \cap \mathsf{cid}_=$
- $G^\sharp_C.\mathsf{rf} = G.\mathsf{rf}|_{E^\sharp}$, $G^\sharp_C.\mathsf{mo} = G.\mathsf{mo}|_{E^\sharp}$, and $G^\sharp_C.\mathsf{nvo} = G.\mathsf{nvo}|_{E^\sharp}$.

An *abstract execution* of G is a tuple $\langle G^\sharp_C, \mathsf{lin}\rangle$ consisting of the execution $G^\sharp_C$, and a strict total order lin on $P_{\mathbb{RET}} \triangleq G.\mathbb{RET} \uplus C$ such that $G^\sharp_C.\mathsf{hb}|_{P_{\mathbb{RET}}} \subseteq \mathsf{lin}$. Henceforth we use $G^\sharp$ as a meta-variable ranging over the possible abstract executions of G.

Figure 5 shows an example. As shown in Fig. 5(a), an execution G will in general include client events (i.e. $G.E \setminus \mathbb{E}^{\mathsf{lib}} \neq \emptyset$). In particular this means that $G^\sharp_C.\mathsf{hb}$ would include edges between client events, and between client events and return events. The requirement that lin should preserve those edges encodes the idea that the linearization order should never contradict the client-observable ordering of calls. In the example, Fig. 5(b) and Figure 5(c) are the only abstract executions of G that respect $G^\sharp_C.\mathsf{hb}$: the rf edge between client events is preserved in the abstract execution, requiring that the call of the thread at the top is linearized before the second call of the thread at the bottom.

Definition 3.6 (Histories). A *history* is a sequence $h \in \mathsf{Hist} \triangleq (\mathsf{Call} \times \mathbb{V})^*$. Given a sequence of events $e_0 \ldots e_n \in \mathbb{RET}^*$, their history is defined as

$$\mathsf{hist}(e_0 \ldots e_n) \triangleq \langle \mathsf{call}(\mathsf{cid}(e_0)), v_0\rangle \ldots \langle \mathsf{call}(\mathsf{cid}(e_n)), v_n\rangle$$

where $e_i : \mathbf{Ret}\ v_i$. The legal histories of a library are specified as a set $\mathcal{L} \subseteq \mathsf{Hist}$.

Definition 3.7 (Durable linearizability). A library with legal histories $\mathcal{L}$ and implementation $\mathcal{I}$ is *linearizable* if for all executions G of $\mathcal{I}$, there exist an abstract execution $\langle G^\sharp, \mathsf{lin}\rangle$ of G such that $\mathsf{hist}(\mathsf{lin}^\uparrow) \in \mathcal{L}$. The library is *durably linearizable* if for every chain $G_0 \ldots G_n$ of $\mathcal{I}$ there are abstract executions $\langle G^\sharp_0, \mathsf{lin}_0\rangle, \ldots, \langle G^\sharp_n, \mathsf{lin}_n\rangle$ such that each $\langle G^\sharp_i, \mathsf{lin}_i\rangle$ is an abstract execution of G_i and $\mathsf{hist}(\mathsf{lin}_0^\uparrow \cdot \ldots \cdot \mathsf{lin}_n^\uparrow) \in \mathcal{L}$.

3.4 Library Specifications

Legal histories are all that is needed to specify the desired abstract behaviour of a linearizable library. We introduce an abstract-machine-based way of specifying legal histories, that will allow us to give a more structured proof technique for proving persistent linearizability.

Definition 3.8 (Library Specification). A *library specification* is an abstract machine that accepts legal histories of library calls. Formally, a specification is a tuple $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ where $\mathbb{Q}$ is a set of *abstract states*, $\Delta\colon \mathsf{Call} \times \mathbb{V} \to \mathscr{P}(\mathbb{Q} \times \mathbb{Q})$ is the transition relation indexed by a call and return value, and $\bar{q} \in \mathbb{Q}$ is the initial abstract state. The *legal histories of* $\langle \mathbb{Q}, \Delta \rangle$ *from* q *to* q' form the set $\mathcal{L}(q, q') \subseteq \mathsf{Hist}$, defined as the smallest such that $\epsilon \in \mathcal{L}(q, q)$, and if $h \in \mathcal{L}(q, q')$ and $(q', q'') \in \Delta(\mathit{call}, v)$ then $h \cdot \langle \mathit{call}, v \rangle \in \mathcal{L}(q, q'')$. The *legal histories of* $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ are defined as $\mathcal{L}(\bar{q}) \triangleq \bigcup_{q \in \mathbb{Q}} \mathcal{L}(\bar{q}, q)$. If $c \in \mathsf{Cid}$, we may write $\Delta(c, v)$ for $\Delta(\mathsf{call}(c), v)$.

A specification is *deterministic* if $\forall \langle \mathit{call}, v \rangle\colon \forall q, q_1, q_2\colon (q, q_1), (q, q_2) \in \Delta(\mathit{call}, v) \Rightarrow q_1 = q_2$.

The legal histories for a set data structure can be formalised as follows. Assume a numeric totally ordered type of keys Key. The legal histories of a library implementing a finite set of keys are the legal histories of the following (deterministic) library specification. The abstract states form the set $\mathsf{KSet} \triangleq \mathscr{P}_{\mathsf{fin}}(\mathsf{Key})$. The transition relation is defined as:

$$\Delta(\mathtt{insert}, k, \mathtt{true}) = \{(S, S \uplus \{k\}) \mid k \notin S\} \qquad \Delta(\mathtt{delete}, k, \mathtt{true}) = \{(S, S \setminus \{k\}) \mid k \in S\}$$
$$\Delta(\mathtt{insert}, k, \mathtt{false}) = \{(S, S) \mid k \in S\} \qquad \Delta(\mathtt{delete}, k, \mathtt{false}) = \{(S, S) \mid k \notin S\}$$

We define a natural notion of equivalence on histories which we will use to justify the history manipulations in our Pathway Theorem.

Definition 3.9 (Equivalent histories). Fix some specification $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ and histories $h_1, h_2 \in \mathsf{Hist}$; we define $h_1 \equiv h_2$ to hold when $\forall q, q' \in \mathbb{Q}\colon h_1 \in \mathcal{L}(q, q') \Leftrightarrow h_2 \in \mathcal{L}(q, q')$.

Our proof strategy for durable linearizability exploits some notions of independence between operations: commutativity, and the weaker voidability.

Definition 3.10 (Commutativity). Fix some specification $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ and let $\langle c, v \rangle, \langle c', v' \rangle \in \mathsf{Call} \times \mathbb{V}$. We say $\langle c, v \rangle$ *commutes with* $\langle c', v' \rangle$, written $\langle c, v \rangle \propto \langle c', v' \rangle$, if $\langle c, v \rangle \langle c', v' \rangle \equiv \langle c', v' \rangle \langle c, v \rangle$.

For example $\langle \mathtt{insert}, k, b \rangle$ commutes with $\langle \mathtt{insert}, k', b' \rangle$ if $k \neq k'$.

Definition 3.11 (Voidable call). Let $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ be a library specification, $\langle c, v \rangle \in \mathsf{Call} \times \mathbb{V}$, and $h \in \mathsf{Hist}$. We say $\langle c, v \rangle$ is *h-voidable* if $\forall q \in \mathbb{Q}\colon (\langle c, v \rangle \cdot h) \in \mathcal{L}(q) \Rightarrow h \in \mathcal{L}(q)$.

Lemma 3.12 (Voidability in KSet). *For the set specification, the following hold*

- $\langle \mathtt{insert}, k, \mathtt{false} \rangle$ *and* $\langle \mathtt{delete}, k, \mathtt{false} \rangle$ *are* h-voidable *for every* h.
- $\langle \mathtt{insert}, k, \mathtt{true} \rangle$ *is* h-voidable *if and only if* h *contains no calls to operations on the key* k*, or* $\langle \mathtt{insert}, k, \mathtt{true} \rangle \cdot h$ *is not legal.*
- $\langle \mathtt{delete}, k, \mathtt{true} \rangle$ *is* h-voidable *if and only if* h *contains no calls to operations on the key* k*, or* $\langle \mathtt{delete}, k, \mathtt{true} \rangle \cdot h$ *is not legal.*

Note that operations which affect the abstract state may still be voidable. Moreover, if the operation *op* commutes with all the operations in h then *op* is h-voidable, but the converse is not necessarily true. For example, a failed insert of k is h-voidable when h consists of a successful delete of k, but the two calls do not commute.

4 A PROOF TECHNIQUE FOR PERSISTENT LINEARIZABILITY

In this section we formalize, through series of lemmas, our methodology for proving durable linearizability. The first step is to decouple the verification of recovery and of the operations, by identifying an interface between the two in the form of the durable and recovered state representation functions. Then linearizability is reduced to an induction over appropriate sequences of events. Finally, Section 4.3 presents our Pathway Theorem. Throughout the section, we fix some arbitrary library specification $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$.

4.1 Decoupling Recovery

As a first step, we decouple the verification of recovery and of the operations, so that they can be combined in persistently linearizable chains. To do so, we specify two invariants, the durable and recovered memories. Assume the library is specified using the style of Definition 3.8. The proof technique we propose requires the definition of two functions $\alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \colon \mathbb{Q} \to \mathscr{P}(\mathsf{Mem})$:

- $\alpha_{\mathsf{dur}}(q)$ is the set of all durable memory representations of q,
- $\alpha_{\mathsf{rec}}(q)$ is the set of all recovered memory representations of q.

We say a memory M encodes a durable state q if $M \in \alpha_{\mathsf{dur}}(q)$, or that it encodes a recovered state q if $M \in \alpha_{\mathsf{rec}}(q)$. For the recovery, $\alpha_{\mathsf{dur}}(q)$ acts both as a precondition, and as an invariant that must be preserved by each of its steps; $\alpha_{\mathsf{rec}}(q)$ is the postcondition of the recovery. When verifying the operations, one assumes that the initial memory has been recovered. At any point in time, the code of operations maintains the invariant $\exists q \colon \alpha_{\mathsf{dur}}(q)$. Technically, we start by defining when we consider a recovery sound with respect to α_{dur} and α_{rec}.

Definition 4.1 (Sound recovery). Given $\alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \colon \mathbb{Q} \to \mathscr{P}(\mathsf{Mem})$, a recovery implementation $\mathcal{I}_{\mathsf{rec}}$ is said $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-*sound* if, for any execution G of $\langle \mathcal{I}_{\mathsf{op}}, \mathcal{I}_{\mathsf{rec}} \rangle$, with $G.I \in \alpha_{\mathsf{dur}}(q)$ for some $q \in \mathbb{Q}$, and with $\mathcal{I}_{\mathsf{op}}$ arbitrary, the following hold:

(1) if $G.\mathbb{RET} \cap \mathbb{C}_{\mathsf{rid}} \neq \emptyset$ then $\mathcal{M}[\![G.\mathsf{po}|_{G.I \cup \mathbb{C}_{\mathsf{rid}}}]\!] \in \alpha_{\mathsf{rec}}(q)$;
(2) $\forall w \in G.\mathbb{C}_{\mathsf{rid}} \colon \forall q \in \mathbb{Q}, \vec{e}, \vec{e}' \colon \mathcal{M}[\![G.I \cdot \vec{e} \cdot w \cdot \vec{e}']\!] \in \alpha_{\mathsf{dur}}(q) \Leftrightarrow \mathcal{M}[\![G.I \cdot \vec{e} \cdot \vec{e}']\!] \in \alpha_{\mathsf{dur}}(q)$.

Condition 4.1(1) considers the recovery run from a memory encoding a *durable* state q, and ensures that, when the recovery returns, the volatile memory encodes the *recovered* state q. Condition 4.1(2) requires that any write issued by the recovery, preserve the durable state encoded by the memory. More precisely, the writes of the recovery should be irrelevant for α_{dur}. This requirement implies that any crash occurring during recovery will leave the memory in a recoverable state, without altering the encoded abstract state. It is also used in the verification of operations, to argue that the abstract state of the persisted memory at the time of a crash is not affected if some recovery events have not been persisted.

Definition 4.2 ($\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-Linearizability). Consider an implementation of operations $\mathcal{I}_{\mathsf{op}}$. We say $\mathcal{I}_{\mathsf{op}}$ is $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-*linearizable* if, for every $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-sound $\mathcal{I}_{\mathsf{rec}}$, all $q \in \mathbb{Q}$, all G execution of $\langle \mathcal{I}_{\mathsf{op}}, \mathcal{I}_{\mathsf{rec}} \rangle$ with $G.I \in \alpha_{\mathsf{dur}}(q)$, there is a $q' \in \mathbb{Q}$ such that:

(1) there is an abstract execution $\langle G^\sharp, \mathsf{lin} \rangle$ of G with $\mathsf{hist}(\mathsf{lin}^\uparrow) \in \mathcal{L}(q, q')$;
(2) $\mathcal{M}[\![G.\mathsf{nvo}|_{G.P}]\!] \in \alpha_{\mathsf{dur}}(q')$.

Condition 4.2(1) asks to find a linearization that is a legal history from abstract state q to q'; Condition 4.2(2) requires that the persisted memory encode the same apparent final state q'.

Theorem 4.3. *If, for some $\alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \colon \mathbb{Q} \to \mathsf{Mem}$ with $\emptyset \in \alpha_{\mathsf{dur}}(\bar{q})$, $\mathcal{I}_{\mathsf{rec}}$ is $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-sound and $\mathcal{I}_{\mathsf{op}}$ is $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-linearizable, then $\langle \mathcal{I}_{\mathsf{rec}}, \mathcal{I}_{\mathsf{op}} \rangle$ is durably linearizable.*

4.2 Linearizability Proofs

The first component of a linearizability proof is the volatile order.

Definition 4.4. A *volatile order of G* is a transitive relation $\mathsf{vo} \subseteq G.E \times G.E$ such that:

(1) $G.I \times (G.E \setminus G.I) \subseteq \mathsf{vo}$
(2) $G.\mathbb{C}_{\mathsf{rid}} \times (G.E \setminus (G.I \cup G.\mathbb{C}_{\mathsf{rid}})) \subseteq \mathsf{vo}$
(3) $[\mathbb{WU} \cap G.\mathbb{C}_{\mathsf{rid}}]; \mathsf{po}; [\mathbb{WU} \cap G.\mathbb{C}_{\mathsf{rid}}] \subseteq \mathsf{vo}$

If an order rel satisfies condition 4.4(1), then for every $\vec{e} \in G.\mathsf{rel}^{\uparrow}$ we have that for some $i_0 < |\vec{e}|$, $(\vec{e})_{i_0\rfloor} = G.I$. Henceforth, we write $\iota_0(\vec{e})$ for such i_0. If rel also satisfies condition 4.4(2), there is some j such that $i \leq j \Leftrightarrow \vec{e}(i) \in G.I \cup G.\mathbb{C}_{\mathsf{rid}}$. We write $\iota(\vec{e})$ for such index j. Finally, if rel also satisfies 4.4(3) then $\mathcal{M}[\![(\vec{e})_{\iota(\vec{e})\rfloor}]\!] = \mathcal{M}[\![G.\mathsf{po}|_{G.I \cup \mathbb{C}_{\mathsf{rid}}}]\!]$.

Next, we introduce linearization strategies, which package in a tuple the components needed to define a linearization through the identification of linearization points.

Definition 4.5 (Linearization strategy). Given an execution G, a *linearization strategy* for G is a tuple $\langle \pi, r, \mathsf{rel}, \alpha \rangle$ where:

- $\pi \colon G.\mathsf{Cid} \rightharpoonup G.E$ is an *injective* finite partial function, which identifies so-called *linearization event* (if any), for the calls of G;
- $r \colon G.\mathsf{Cid} \rightharpoonup \mathsf{Val}$ associates to each call a return value;
- $\mathsf{rel} \subseteq G.E \times G.E$ is an acyclic relation on events;
- $\alpha \colon \mathbb{Q} \to \mathscr{P}(\mathsf{Mem})$ is the state representation function.

We require:

(1) $\forall c \in \mathrm{dom}(\pi)\colon \mathrm{cid}(\pi(c)) \neq \bot \wedge \mathrm{cid}(\pi(c)) \neq \mathsf{rid}$,
(2) $\mathrm{dom}(\pi) \subseteq \mathrm{dom}(r)$, and
(3) $\forall e \in G.\mathbb{RET}\colon r(\mathrm{cid}(e)) = \mathrm{val}(e)$.

The function π identifies the event at which each operation is seen to have taken effect; since not all calls might have reached that event, the function is partial. The function r records the return values; it is required to agree with any return event present in the execution, but it might assign return values to calls that have not returned yet, but have already linearized. The relation rel is the one used to order the linearization events to induce the linearized sequence of calls. The representation function α formalises how an abstract state can be represented in memory.

We overload the 'hist' symbol so we can extract histories from sequences of events, through linearization strategies.

Definition 4.6. Let $\langle \pi, r, \mathsf{rel}, \alpha \rangle$ be a linearization strategy. We define:

$$\mathrm{hist}^r_\pi(\vec{e}) \triangleq \begin{cases} \epsilon & \text{if } \vec{e} = \epsilon \\ \langle \mathrm{call}(c), r(c) \rangle \cdot \mathrm{hist}^r_\pi(\vec{e}') & \text{if } \vec{e} = e \cdot \vec{e}' \wedge \pi(c) = e \\ \mathrm{hist}^r_\pi(\vec{e}') & \text{if } \vec{e} = e \cdot \vec{e}' \wedge \forall c\colon \pi(c) \neq e \end{cases}$$

We defined the notion of strategy generically because we will instantiate it in two ways in proofs: once with the "volatile" parameters $\pi = \ell$, $\mathsf{rel} = \mathsf{vo}$, $\alpha = \alpha_{\mathsf{vol}}$, and once with the "persistent" parameters $\pi = p$, $\mathsf{rel} = \mathsf{nvo}$, $\alpha = \alpha_{\mathsf{dur}}$. In both cases, we want to use the strategy to validate an execution by induction on rel-preserving sequences of events.

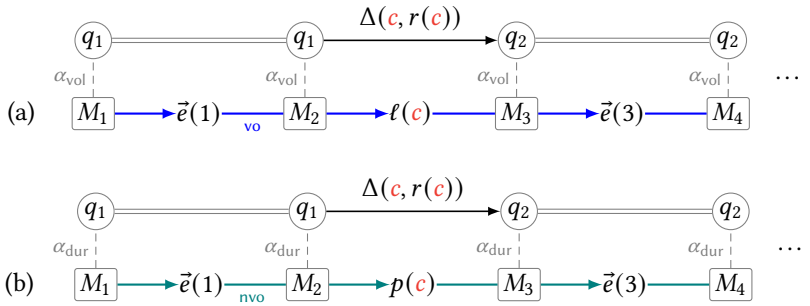


Fig. 6. Examples of (a) validating volatile strategy, and (b) validating persistent strategy. Events from an enumeration $\vec{e}$ of vo (resp. nvo) are examined in sequence; each event may modify the volatile (resp. persisted) memory M_i which is related to some abstract state q_i via α_{vol} (resp. α_{dur}). Only linearization events are allowed to update the abstract state, and the update needs to be legal according to Δ.

Definition 4.7 (Validating strategy). The strategy $\langle \pi, r, \text{rel}, \alpha \rangle$ ι_0*-validates* G if for all $\vec{e} \in \text{rel}^{\uparrow}$, all $q_0 \in \mathbb{Q}$ such that $\mathcal{M}[\![(\vec{e})_{\iota_0(\vec{e})\rfloor}]\!] \in \alpha(q_0)$, all $\iota_0(\vec{e}) < i < |\vec{e}|$, and all $q \in \mathbb{Q}$, if $\text{hist}^r_\pi((\vec{e})_{i-1\rfloor}) \in \mathcal{L}(q_0, q)$ and $\mathcal{M}[\![(\vec{e})_{i-1\rfloor}]\!] \in \alpha(q)$, then:

(1) if $\vec{e}(i) \neq \pi(c)$ for all c, then $\mathcal{M}[\![(\vec{e})_{i\rfloor}]\!] \in \alpha(q)$;
(2) if $\vec{e}(i) = \pi(c)$ for some c, then $\mathcal{M}[\![(\vec{e})_{i\rfloor}]\!] \in \alpha(q')$ for some q' such that $(q, q') \in \Delta(c, r(c))$.

The strategy ι*-validates* G if it satisfies the above conditions using ι instead of ι_0.

The idea of Definition 4.7 is illustrated in Fig. 6. When applied to volatile strategies (Fig. 6(a)), showing that an execution is validated by the strategy is done by induction on sequences $\vec{e}$ respecting vo. For each position i in the sequence, assuming the vo-induced memory $M_i = \mathcal{M}[\![(\vec{e})_{i-1\rfloor}]\!]$ at that point encodes some state q through α_{vol}, one checks if the event $\vec{e}(i)$ is a linearization point or not according to ℓ. If it is, one checks that the event transforms the memory to one that encodes some q' that is legal for the linearizing operation. If it is not, one checks that the encoded state is preserved. The end result is to have proven the linearization induced by the strategy is legal.

The same proof scheme can be applied to persistent strategies (Fig. 6(b)), with the aim of proving that the persistency points produce a persisted memory that encodes the expected state.

Lemma 4.8. *Let G be an execution of $\langle \mathcal{I}_{\text{op}}, \mathcal{I}_{\text{rec}} \rangle$ for some $\langle \alpha_{\text{dur}}, \alpha_{\text{rec}} \rangle$-sound $\mathcal{I}_{\text{rec}}$, with $G.I \in \alpha_{\text{dur}}(q)$:*

(1) $\langle \ell, r, \text{vo}, \alpha_{\text{vol}} \rangle$ ι-validates $G \Rightarrow \forall \vec{e} \in G.\text{vo}^{\uparrow} : \exists q' : \text{hist}^r_\ell(\vec{e}) \in \mathcal{L}(q, q')$.
(2) $\langle p, r, \text{nvo}|_{G.P}, \alpha_{\text{dur}} \rangle$ ι_0-validates $G \Rightarrow \forall \vec{e} \in \text{nvo}^{\uparrow}_{G.P} : \exists q' : \text{hist}^r_p(\vec{e}) \in \mathcal{L}(q, q') \wedge \mathcal{M}[\![\vec{e}]\!] \in \alpha_{\text{dur}}(q')$.

Remark 4.9 (Completeness). As shown by [Schellhorn et al. 2014], identifying linearization points is a complete technique for proving linearizability, provided that (1) linearization points can be dependent on future events, and (2) a single event can linearize multiple calls. Since we define a linearization strategy given a full execution G, the first item above is fully supported. Even when extended with support for hindsight, however, linearization strategies cannot fully accommodate the second item, since the maps from calls to linearization events are required to be injective. This is an obstacle when multiple writer operations linearize together (e.g. in the elimination stack, where a push and pop cancel out). In principle, it is possible to allow a linearization map ℓ to return, for each call, a pair $\langle e, i \rangle$ of an event and a $i \in \mathbb{N}$. This way, ℓ can be injective and still associate multiple calls to the same event (their relative order being determined by i). For the sake of simplicity, we use here the simpler, incomplete definition.

4.3 The Pathway Theorem

We are now ready for our Pathway Theorem, which we first state and then explain.

Theorem 4.10 (Pathway Theorem). *Consider a library with deterministic specification* $\langle \mathbb{Q}, \Delta, \bar{q} \rangle$ *and operations implementation* $\mathcal{I}_{\mathsf{op}}$. *Let* $\alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}}, \alpha_{\mathsf{vol}} \colon \mathbb{Q} \to \mathcal{P}(\mathsf{Mem})$ *with* $\forall q \colon \alpha_{\mathsf{rec}}(q) \subseteq \alpha_{\mathsf{vol}}(q)$. *To prove* $\mathcal{I}_{\mathsf{op}}$ *is* $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$*-linearizable, it is sufficient to prove the following. Fixing arbitrary* $q \in \mathbb{Q}$, $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$*-sound* $\mathcal{I}_{\mathsf{rec}}$, *and* G *execution of* $\langle \mathcal{I}_{\mathsf{op}}, \mathcal{I}_{\mathsf{rec}} \rangle$ *with* $G.I \in \alpha_{\mathsf{dur}}(q)$, *find:*

- *a volatile order* vo
- $\ell \colon G.\mathsf{Cid} \rightharpoonup G.E$
- $r \colon G.\mathsf{Cid} \rightharpoonup \mathbb{V}$
- $p \colon G.\mathsf{Cid} \rightharpoonup G.P$
- *a persisted readers set* $PR \subseteq G.\mathsf{Cid} \setminus \mathrm{dom}(p)$

such that:

(1) $\mathrm{dom}(p) \cup PR \subseteq \mathrm{dom}(\ell)$.

(2) $\forall c \in PR\colon\ \Delta(c, r(c)) \subseteq [\mathbb{Q}]$.

(3) $\forall c, c' \colon \ell(c) \xrightarrow{\mathsf{hb}} \ell(c') \Rightarrow \ell(c) \xrightarrow{\mathsf{vo}} \ell(c')$.

(4) $\forall c \in \mathrm{dom}(\ell) \colon \exists e_1, e_2 \in G.\mathbb{C}_c \colon e_1 \xrightarrow{\mathsf{hb}^?} \ell(c) \xrightarrow{\mathsf{hb}^?} e_2$.

(5) $\langle \ell, r, \mathsf{vo}, \alpha_{\mathsf{vol}} \rangle$ *ι-validates* G.

(6) $\mathrm{cid}(G.\mathbb{RET}) \subseteq \mathrm{dom}(p) \cup PR$.

(7) For any $c, c' \in \mathrm{dom}(p)$, *either:*
$\langle \mathrm{call}(c), r(c) \rangle \propto \langle \mathrm{call}(c'), r(c') \rangle$ *or* $p(c) \xrightarrow{\mathsf{nvo}} p(c') \Rightarrow \ell(c) \xrightarrow{\mathsf{vo}} \ell(c')$.

(8) For any $c \in \mathrm{dom}(\ell) \setminus (\mathrm{dom}(p) \cup PR)$, *and all* $\vec{e} \in \mathsf{vo}^{\uparrow}_{G.E}$:
if $\vec{e} = \vec{e}' \cdot \ell(c) \cdot \vec{e}''$ *then* $\langle \mathrm{call}(c), r(c) \rangle$ *is h-voidable, where* $h = \mathrm{hist}^r_\ell(\vec{e}'')|_{(\mathrm{dom}(p) \cup PR)}$.

(9) $\mathrm{hist}^r_p(\mathsf{nvo}^{\uparrow}_{G.P}) \in \mathcal{L}(q) \Rightarrow \langle p, r, \mathsf{nvo}|_{G.P}, \alpha_{\mathsf{dur}} \rangle$ *ι₀-validates* G.

The theorem follows the high-level description of Section 2. To prove $\langle \alpha_{\mathsf{dur}}, \alpha_{\mathsf{rec}} \rangle$-linearizability of G, we have to provide two linearization strategies: the volatile one $\langle \ell, r, \mathsf{vo}, \alpha_{\mathsf{vol}} \rangle$ and the persistent one $\langle p, r, \mathsf{nvo}|_{G.P}, \alpha_{\mathsf{dur}} \rangle$. Since the persistency points must be durable events, $\mathrm{dom}(p)$ only represents the persisted "writer" calls, i.e. calls that induce an abstract state change. The persisted readers set PR indicates which of the "reader" calls should be considered (logically) persisted.

Then the theorem asks us to check a number of conditions. Condition 4.10(1) checks that every persisted operation has a linearization point. Condition 4.10(2) ensures that the persistent readers are indeed readers.

The goal of conditions 4.10(3) and 4.10(4) is to make sure the final linearization respects the hb order between the calls, as required by Definition 3.5. In particular, we do not want the linearization to contradict the po ordering between calls. In fact vo might not preserve hb in general: typically vo reconstructs a global notion of time which might contradict some po edges, for examples the ones between write and read events. Condition 4.10(3) states that vo has to preserve hb on linearization points, which are typically either reads or updates. Condition 4.10(4) requires a linearization point to be hb-between two events of the call it linearizes. This ensures that an hb edge between calls implies an hb edge between the respective linearization points. If linearization points are local to the call they linearize, the condition is trivially satisfied.

The conditions so far represent well-formedness constraints that usually hold by construction. The conditions that follow are the ones where the actual algorithmic insight is involved.

Condition 4.10(5) requires to prove regular volatile linearizability using the volatile linearization strategy. This is typically a straightforward adaptation of the proof of a non-durable version of the

same data structure (e.g. Harris list for the link-free set). The adaptation simply needs to ensure that the events added to make the data structure durable preserve the encoded state when executed.

When making a linearizable data structure durable, after having decided how the persisted memory represents an abstract state (through α_{dur}), the main design decision is which flushes to issue and when. Conditions 4.10(6), 4.10(7), and 4.10(8) are the ones that check that all the flushes needed for correctness are issued by the operations. This can help guiding optimizations, as redundant flushes would be the ones not contributing to the proofs of these conditions.

Condition 4.10(6) reflects the *unbuffered* nature of durable linearizability: it requires every returned call to be considered as persisted. The only way to ensure a writer operation is persisted before returning is to issue a (synchronous) flush on the address of the persistency point of the call, which we dub the "flush before return" policy.

Condition 4.10(7) deals with the discrepancies between the volatile and the persistent linearization orders. Specifically, it consider two calls that have both persisted. If they abstractly commute, their relative order in the linearization does not matter. Otherwise, it must be the case that they are ordered by the volatile linearization strategy in the same order as they are by the persistent strategy. In implementations, this is typically achieved by making every call execute their linearization point after having made sure that the persistency point of earlier non-commuting calls are flushed.

Condition 4.10(8) deals with the second source of discrepancies between volatile and persistent linearizations: voided calls, i.e. calls that the volatile argument sees as having linearized, but have not persisted. A simpler (but less general) version of the condition would mirror the previous condition using commutativity:

(8′) For any $c \in \mathrm{dom}(\ell) \setminus (\mathrm{dom}(p) \cup PR)$ *and* $c' \in \mathrm{dom}(p) \cup PR$*, either:*
$\langle \mathrm{call}(c), r(c)\rangle \propto \langle \mathrm{call}(c'), r(c')\rangle$ *or* $\ell(c') \xrightarrow{\mathsf{vo}} \ell(c)$.

Condition (8′) considers the situation where the volatile linearization places a voided call c before a persisted call c'. If such calls commuted, it would be possible to rearrange the volatile linearization until all the voided calls appear at the end, while preserving legality of the sequence. Then the persistent linearization would be a prefix of the volatile one, which is also a legal sequence. Condition (8′) therefore asks that the calls either commute, or that the voided call is vo-after the persisted one.

The case study of Section 5, however, does not satisfy (8′), but satisfies the more permissive 4.10(8), which uses voidability instead of commutativity: the condition asks to prove that voided calls are voidable with respect to the rest of the linearization ahead of them. More precisely, one considers the linearization induced by vo and ℓ, and finds the linearization point of a voided call c. Then one needs to check that the call is voidable with respect to the volatile history of non-voided calls ahead (h). This means it can be removed from the linearization without affecting its legality.

The condition is typically enforced by making sure that possibly conflicting calls in h issue a flush on the address of the persistency point of c before persisting themselves: if that is the case, then either h does not contain conflicting calls, or c has been persisted, which means it is not voided.

When the conditions presented so far hold, the legality of the volatile linearization implies the legality of the persistent linearization. What remains to prove is that the persistency points modify the persisted memory so that it encodes the output state expected given the legal linearization. This is checked by condition 4.10(9), which allows us to *assume* the persisted linearization is legal (a fact that follows from the other conditions) and asks us to prove by induction on nvo that each persistency point modifies the memory so that the encoded state reflects the expected change. Proving this does not involve flushes, but merely checks that the persistency points enact changes that are compatible with the durable state interpretation α_{dur}.

4.4 The Persist-First Pathway Theorem

The Pathway Theorem applies to schemes that first linearize operations and then persist them, as codified by condition 4.10(1). These schemes are natural as in hardware writes are first propagated in working memory and then persisted asynchronously. Persist-first schemes store the data structure in two redundant versions, one used for durability and a purely volatile one. They reverse the usual scheme by first committing the effects of an operation in the persistent representation and then linearizing it in the volatile representation.

To the best of our knowledge, every durable data structure in the literature adopts either a linearize-first or a persist-first scheme, never mixing the two. We therefore opted for providing a separate theorem for persist-first, although in principle it is possible to provide a joined version.

The Persist-First Pathway Theorem, shown in [D'Osualdo et al. 2022] coincides with the Pathway Theorem, except for conditions 4.10(1), 4.10(7), and 4.10(8). The obvious substitute of 4.10(1) is $\mathrm{dom}(\ell) \subseteq \mathrm{dom}(p)$. This excludes the presence of voided calls, but allows for what we call "prematurely persisted" calls: calls that get persisted at some point in the persistent linearization, but have not linearized yet. To account for this, condition 4.10(7) is modified to additionally require any persisted and linearized call c to be persisted nvo-before any prematurely persisted c', unless c and c' commute.

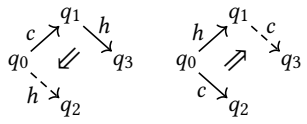


Fig. 7. Duality between voidability and appendability.

Finally, just as condition 4.10(8) of the Pathway Theorem requires voided calls to be "voidable", the Persist-First Pathway Theorem requires prematurely persisted calls to be "appendable".

Definition 4.11 (Appendable call). Let $\langle c, v\rangle \in \mathsf{Call} \times \mathbb{V}$, and $h \in \mathsf{Hist}$. We say $\langle c, v\rangle$ is *h-appendable* if, for all $q \in \mathbb{Q}$ we have $\big(c \in \mathcal{L}(q) \wedge h \in \mathcal{L}(q)\big) \Rightarrow (h \cdot \langle c, v\rangle) \in \mathcal{L}(q)$.

Figure 7 shows in which sense voidability and appendability can be considered duals. For persist-first, the voidability condition 4.10(8) is replaced with one that requires each prematurely persisted call c to be h-appendable where h is the history induced by vo and ℓ, that comes vo-after the persistency point of c.

5 CASE STUDY: THE LINK-FREE SET

In this section we sketch how our Pathway Theorem can be used to provide a formal proof of the link-free set of Zuriel et al. [2019] with respect to the Px86 memory model. The full argument is presented in [D'Osualdo et al. 2022].

5.1 State Representation

We begin by formalizing the intended memory representation of a set. We use a distinguished address *head* as the entry point of the data structure for the current era. We have two overlaid representations: the recoverable and the volatile ones.

Definition 5.1 (Durable state representation). We define $\alpha_{\mathsf{dur}} : \mathsf{KSet} \to \mathscr{P}(\mathsf{Mem})$ as the function such that $M \in \alpha_{\mathsf{dur}}(S)$ if and only if $M = M_{\mathsf{s}} \uplus M_{\mathsf{d}} \uplus M_{\mathsf{g}}$ where

$$M_{\mathsf{s}} = \biguplus_{k \in S} [x_k.\mathtt{key} \mapsto k, x_k.\mathtt{nxt} \mapsto \langle 0, _\rangle, x_k.\mathtt{valid} \mapsto 1]$$
$$M_{\mathsf{d}} = \biguplus_{y \in X_{\mathsf{d}}} [y.\mathtt{key} \mapsto _, y.\mathtt{nxt} \mapsto \langle 1, _\rangle, y.\mathtt{valid} \mapsto 1]$$
$$M_{\mathsf{g}} = \biguplus_{y \in X_{\mathsf{g}}} [y.\mathtt{key} \mapsto _, y.\mathtt{nxt} \mapsto \langle 0, _\rangle, y.\mathtt{valid} \mapsto 0]$$

for some sets of addresses $X_s = \{x_k | k \in S\}$, X_d, and X_g. Intuitively, M_s collects the nodes representing members of S, M_d collects deleted nodes, M_g collects garbage nodes.[2]

Definition 5.2 (Volatile state representation). We define $\alpha_{vol} : \mathsf{KSet} \to \mathscr{P}(\mathsf{Mem})$ as the function such that $M \in \alpha_{vol}(S)$ iff $M = M_s \uplus M_d \uplus M_u$ where $x_{-\infty} = \mathit{head}$, $S \cup \{+\infty, -\infty\} = \{k_1, \ldots, k_n\}$, and

$$M_s = \biguplus_{1 \le i \le n} \left[x_{k_i}.\mathtt{key} \mapsto k_i, x_{k_i}.\mathtt{nxt} \mapsto \langle 0, _\rangle, x_{k_i}.\mathtt{valid} \mapsto _\right]$$
$$M_d = \biguplus_{y \in X_d} [y.\mathtt{key} \mapsto _, y.\mathtt{nxt} \mapsto \langle 1, _\rangle, y.\mathtt{valid} \mapsto 1]$$
$$M_u = \biguplus_{y \in X_u} [y.\mathtt{key} \mapsto _, y.\mathtt{nxt} \mapsto \langle 0, _\rangle, y.\mathtt{valid} \mapsto 0]$$

for some sets of addresses X_d, X_u, and $X_s = \{x_{k_1}, \ldots, x_{k_n}\}$ and such that

$$\forall x, y \in \mathbb{X}. \left(M(x.\mathtt{key}) < +\infty \wedge M(x.\mathtt{nxt}) = \langle _, y \rangle\right) \Rightarrow M(x.\mathtt{key}) < M(y.\mathtt{key}) \tag{1}$$
$$\forall i < n. \exists m \ge 0. \exists y_1, \ldots, y_m \in X_d. \exists y_{m+1} = x_{k_{i+1}}: M(x_{k_i}.\mathtt{nxt}) = \langle 0, y_1 \rangle \wedge \left(\bigwedge_{1 \le j \le m} M(y_j.\mathtt{nxt}) = \langle 1, y_{j+1} \rangle\right) \tag{2}$$
$$\forall y \in X_d. \exists y' \in X_s \cup X_d. M(y.\mathtt{nxt}) = \langle _, y' \rangle \tag{3}$$

That is, M_s is a sorted linked list of valid unmarked nodes representing members of the store, possibly interleaved with deleted notes; M_d represents deleted nodes, and M_u represents uninitialised nodes. Note that even links in M_d and M_u are sorted, although they are not required to form a list. Moreover, the sortedness constraint (1) implies $k_i < k_{i+1}$ for all $i < n$.

The recovery function's goal is to repair some memory in $\alpha_{dur}(S)$ so that it belongs to $\alpha_{vol}(S)$. The recovered state representation is therefore $\alpha_{rec}(S) \triangleq \alpha_{dur}(S) \cap \alpha_{vol}(S)$.

Showing that the recovery function is sound is easy. The first soundness condition requires to prove that from some memory in $\alpha_{dur}(q)$, upon termination, the recovery produced a memory in $\alpha_{rec}(q)$. This is easy to establish with standard sequential reasoning. The second condition requires the recovery to never introduce intermediate memories which are not in $\alpha_{dur}(S)$. Since the recovery only modifies the `nxt` fields, which are not constrained by α_{dur}, the condition follows immediately.

5.2 Volatile Linearizability

We sketch how to prove linearizability by providing a suitable pre-linearization strategy. To start, we need to pick an appropriate volatile order. For Px86 a natural choice is the *global happens before* order [Alglave 2012], which reconstructs the possible order of events from a global point of view:

$$\mathsf{ghb} \triangleq \left((\mathsf{po} \setminus ((\mathbb{W} \cup \mathbb{FL}) \times \mathbb{R}))|_{\mathbb{E} \setminus \mathbb{RET}} \cup \mathsf{mo} \cup (\mathsf{rf}^{-1}; \mathsf{mo}) \cup (\mathsf{rf} \setminus \mathsf{po})\right)^+$$

The Pathway Theorem asks to provide a set of events X which includes all linearization points and on which vo preserves hb. For the link-free set, we let $X = \mathbb{RU}$, since $[\mathbb{RU}]; \mathsf{hb}; [\mathbb{RU}] \subseteq \mathsf{ghb}$.

Lemma 5.3 (ghb includes hb on $\mathbb{RU}$). $[\mathbb{RU}]; \mathsf{hb}; [\mathbb{RU}] \subseteq \mathsf{ghb}$.

Next, we have to pick linearization points by defining ℓ and return values r.

Definition 5.4 (Linearization points). Given an execution G of the link-free set of Fig. 4, we define the finite partial functions $\ell : G.\mathsf{Cid} \rightharpoonup G.\mathbb{RU}$ and $r : G.\mathsf{Cid} \rightharpoonup \mathbb{V}$ as the smallest such that:

- if $\mathsf{call}(c) = \langle \mathtt{insert}, _ \rangle$ then
  - If there is an event $(e : \mathbf{U}\ p.\mathtt{nxt}\ \langle 0, _\rangle\ \langle 0, _\rangle) \in G.\mathbb{C}_c$ generated at Line 44 then $\ell(c) = e$ and $r(c) = \mathtt{true}$.

[2] For simplicity we assume head and tail nodes (i.e. nodes with key $\pm\infty$) do not survive a crash; they would otherwise also accumulate as garbage nodes since recovery re-allocates them.

  - If there is an event $(r : \mathbf{Ret}\ \texttt{false}) \in G.\mathbb{C}_c$ then there is at least one read event of $G.\mathbb{C}_c$ generated by Line 20 in the call to `find` at Line 36; let $(e : \mathbf{R}\ c.\texttt{nxt}\ \langle 0, _\rangle)$ be the po-last such event. Then we set $\ell(c) = e$ and $r(c) = \texttt{false}$.
- if $\text{call}(c) = \langle \texttt{delete}, _\rangle$ then
  - If there is an event $(e : \mathbf{U}\ c.\texttt{nxt}\ \langle 0, _\rangle\ \langle 1, _\rangle) \in G.\mathbb{C}_c$ generated at Line 56, then $\ell(c) = e$ and $r(c) = \texttt{true}$.
  - If there is an event $(r : \mathbf{Ret}\ \texttt{false}) \in G.\mathbb{C}_c$, then there is at least one event in $G.\mathbb{C}_c$ generated by Line 20 in the call to `find` at Line 51; let $(e : \mathbf{R}\ c.\texttt{nxt}\ \langle 0, _\rangle)$ be the po-last such event. Then we set $\ell(c) = e$ and $r(c) = \texttt{false}$.

By construction we have $\forall c \in \text{dom}(\ell)$: $\text{cid}(\ell(c)) = c$, ℓ is injective, $\text{cid}(G.\mathbb{RET}) \subseteq \text{dom}(\ell)$, $\text{dom}(r) = \text{dom}(\ell)$, and $(e : \mathbf{Ret}\ v) \in G.\mathbb{RET} \Rightarrow r(\text{cid}(e)) = v$.

It is easy to check that $\langle \ell, r, \text{ghb}, \alpha_{\text{vol}} \rangle$ is a linearization strategy. With these parameters we could already prove standard linearizability by showing that $\langle \ell, r, \text{ghb}, \alpha_{\text{vol}} \rangle$ ι-validates every execution.

Theorem 5.5. *Assume an arbitrary $\langle \alpha_{\text{dur}}, \alpha_{\text{rec}} \rangle$-sound $\mathcal{I}_{\text{rec}}$. If G is an execution of $\langle \mathcal{I}_{\text{op}}^{LFS}, \mathcal{I}_{\text{rec}} \rangle$, then $\langle \ell, r, \text{ghb}, \alpha_{\text{vol}} \rangle$ ι-validates G.*

We omit the proof as it is subsumed by the proof on the optimized algorithm provided in the extended version of this paper [D'Osualdo et al. 2022].

5.3 Persistency Points

We now define the final parameters needed to instantiate the Pathway Theorem: the persistency points through p and the persisted readers PR.

Definition 5.6 (Persistency points). Given an execution G of the link-free set in Fig. 4, we define the finite partial function $p \colon G.\text{Cid} \rightharpoonup G.P$ as the smallest such that:

- if $\text{call}(c) = \langle \texttt{insert}, _\rangle$ and $\ell(c) = (e : \mathbf{U}\ p.\texttt{nxt}\ _\ \langle 0, n\rangle)$ then $p(c) = \min_{\text{nvo}}\{e' \mid (e' : \mathbf{W}\ n.\texttt{valid}\ 1) \in G.P\}$ (undefined if $G.P$ contains no write to n.`valid`);
- if $\text{call}(c) = \langle \texttt{delete}, _\rangle$ and $\ell(c) = (e : \mathbf{U}\ _.\texttt{nxt}\ \langle 0, _\rangle\ \langle 1, _\rangle)$ then $p(c) = e$.

We also define $PR = \{c \mid (r : \mathbf{Ret}\ \texttt{false}) \in G.\mathbb{C}_c\}$. By construction we have that p is injective, $\text{dom}(p) \cap PR = \emptyset$ and $\text{dom}(p) \cup PR \subseteq \text{dom}(\ell)$.

5.4 Applying the Pathway Theorem

We have now picked all the parameters we need to apply the Pathway Theorem: we set vo = ghb, $X = G.\mathbb{RU}$, ℓ and r as defined in Definition 5.4, p and PR as defined in Definition 5.6. We can now check each condition of the Pathway Theorem.

Conditions 4.10(1) and 4.10(2) are satisfied by construction. Condition 4.10(3) follows from Lemma 5.3. Condition 4.10(4) is straightforward from $\text{cid}(\ell(c)) = c$ (which allows us to pick $e_1 = e_2 = \ell(c)$). Condition 4.10(5) is a direct consequence of Theorem 5.5. The rest of the conditions represent the crucial steps in the correctness proof.

Flush before returning. Condition 4.10(6) reflects the *unbuffered* nature of durable linearizability: it requires every returned call to be considered as persisted. The only way to enforce something is persisted before returning is to issue a (synchronous) flush on the address of the persistency point of the call. In our case, the "read-only" calls that have returned are included in PR by construction. For successful inserts/deletes we can prove that if they returned: (1) they have issued a flush on the address of their persistency point, and (2) the persistency point happens nvo-before the flush. Since $G.\mathbb{FL} \subseteq G.P$, we conclude that if those calls returned, their persistency point was persisted.

Commuting calls. To prove 4.10(7) we focus on persisted calls $c, c' \in \mathrm{dom}(p)$ that do not commute, that is persisted successful inserts or deletes of the same key k. As a preliminary step, we can show that any such c, c' are related by ghb in some order. Thus, since nvo is total on $G.P$, condition 4.10(7) can be reduced to proving that if $\ell(c) \xrightarrow{\text{ghb}} \ell(c')$ then $p(c) \xrightarrow{\text{nvo}} p(c')$. By transitivity, we can focus on calls whose linearization points are adjacent, i.e. there are no linearization points of calls on key k between c and c' in ghb order. Since we already established legality of the ghb-induced sequence of linearization points, we can focus on legal pairs of calls. This leaves us with two cases:

- c is a successful insert of k, and c' a adjacent successful delete of k. From the volatile invariants on the ghb-induced memory, we know that c' will be deleting the node n inserted by c. This means that $n.\texttt{valid}$ will be written before $\ell(c')$ (either by c or c'). The earliest such write is $p(c)$ which would then be nvo-before $\ell(c') = p(c')$ as desired.
- c is a successful delete of k, and c' a adjacent successful insert of k. Wme can prove that before $\ell(c)$ is executed, at least one call to `trim` on the deleted node has been run. The flush at line 31 would then ensure the desired nvo order.

Voided calls. We now check condition 4.10(8), which asks that every call c which reached its linearization point but has not persisted must be voidable. Lemma 3.12 tells us that the only possible problematic calls are successful inserts or deletes, ghb-followed by operations on the same key.

Lemma 5.7. *Take any* $c \in \mathrm{dom}(\ell) \setminus (\mathrm{dom}(p) \cup PR)$*, and all* $\vec{e} \in \mathrm{vo}^{\uparrow}_{G.E}$ *such that* $\vec{e} = \vec{e}' \cdot \ell(c) \cdot \vec{e}''$*. Then* $\langle \mathrm{call}(c), r(c) \rangle$ *is* h*-voidable, where* $h = \mathrm{hist}^r_\ell(\vec{e}'')|_{(\mathrm{dom}(p) \cup PR)}$.

The proof proceeds by contradiction: assume $\langle \mathrm{call}(c), r(c) \rangle$ is not h-voidable. We have two cases: either c is a successful insert or a successful delete, of some key k. Let us focus on the insert case, as the delete case is analogous. $\vec{e}''$ must contain the linearization point of a *persisted* call involving the same key. Let c' be the id of such call. We can show that if c' persisted, the persistency point of c must have been executed and persisted. We therefore reach a contradiction with the assumption that $c \in \mathrm{dom}(\ell) \setminus (\mathrm{dom}(p) \cup PR)$. Since we already established legality of histories induced by ℓ and ghb, we are left with the following cases:

- call $c' = \langle \texttt{insert}, k \rangle$, $r(c) = \texttt{false}$: then lines 38 and 39 have been both executed so at least one event setting `valid` to 1 has been persisted, implying $c \in \mathrm{dom}(p)$.
- call $c' = \langle \texttt{delete}, k \rangle$, $r(c) = \texttt{true}$ then line 55 has been executed; since c' has persisted, we know that the successful CAS at line 56 also persisted; since `c.nxt` and `c.valid` fit in the same cache line, both writes have been then persisted, implying $c \in \mathrm{dom}(p)$.

Persisted memory correctness. We have finally arrived at condition 4.10(9). Until now, we mostly considered ghb-induced sequences of events and invariants on memory. When nvo was involved, we only needed to prove that some writes/flushes were inserted in some crucial points. This is intentional: the fundamental correctness argument typically rests on the volatile invariants. The Pathway Theorem allows us to focus on those for as long as possible. What the proof so far entails is the legality of the (persisted) volatile history, and, the fact that the nvo-induced history is legal and equivalent to the volatile one. What is missing is to prove that the contents of the *persisted* memory encode the output state of the (legal) history, with respect to α_{dur}.

Concretely, consider proving $\langle p, r, \mathrm{nvo}|_{G.P}, \alpha_{\mathsf{dur}} \rangle$ ι_0-validates G, for the link-free set. The first condition that Definition 4.7 asks us to prove is that events that are not persistency points preserve the encoded abstract state. This is easy to prove: the only non-trivial writes affect the links, which are ignored by α_{dur}; the other fields are only updated on uninitialized nodes. The second condition we need to prove is that persistency points induce the desired legal transitions. Proving this directly is challenging. When considering the persistency point of a successful insert of k, for example, we

need to prove that when the `valid` field of the new node n is first set to 1, the persisted memory is in $\alpha_{\mathsf{dur}}(S)$ for some S with $k \notin S$. The reason why this is true is exclusively due to the volatile invariants which apply to the ghb-induced memory, memory that may be encoding (via α_{vol}) some different $S'' \neq S$. The proof strategy embedded in our Pathway Theorem resolves the mismatch by letting us *assume* (by virtue of the other conditions) that the nvo-induced history is legal. In particular, in the insert case, we are allowed to assume $k \notin S$, which by $\alpha_{\mathsf{dur}}(S)$ would imply that after the persistency point, exactly one valid unmarked node would be holding the key k. Finally, another notable simplification introduced by this proof strategy, is that we can safely ignore the "read-only" operations like failed inserts and deletes, since their place in the linearization has already been provided through the volatile argument.

5.5 Optimizations and Extensions

The full link-free algorithm includes a number of further optimizations and a wait-free `contains` operation. The extended version of this paper [D'Osualdo et al. 2022] presents a formal proof of the full algorithm by using a generalization of the Pathway Theorem.

In particular, the full algorithm optimizes the `find` function by removing lines 22 and 24 from Fig. 4: it is not necessary to check that the predecessor is unmarked before returning. This optimization introduces the need for hindsight linearization of failed deletes. The issue is that, we can no longer identify a point in the program where the p and c returned by `find` are both unmarked, adjacent and reachable. This makes it impossible to find any particular event that can serve as the linearization of a failed delete of some k. In fact, when we check p is unmarked it might be too early: a node n holding k might be ahead in the list and unmarked. Then n might have been marked before we reached it. By the time we reach c, a new n' holding k might have been added behind in the list, and so at this point we are too late to linearize the delete. The operation is still correct: between reading p unmarked and reading c unmarked, there must have been a point when no unmarked node in between them held k. This kind of "after the fact" argument is called a "hindsight lemma" in [O'Hearn et al. 2010].

The general version of our Pathway Theorem supports hindsight by means of a partial map $\ell_{\mathsf{h}}(\vec{e})\colon \mathsf{Cid} \rightharpoonup \mathbb{N}$, which associates to each call c that needs hindsight, an index $\ell_{\mathsf{h}}(\vec{e})(c)$ indicating where in the (vo-ordered) sequence of events $\vec{e}$ the call is logically linearized. Crucially, ℓ_{h} can be specified after ℓ has been defined and has been used to prove $\langle \ell, r, \mathsf{ghb}, \alpha_{\mathsf{vol}} \rangle$ ι-validates the execution. The definition of ℓ_{h} can therefore assume the history induced by ℓ and $\vec{e}$ is legal, and find a position where the hindsight calls linearize, in the same way we informally argued above.

6 RELATED WORK

The literature includes attempts at strengthening and simplifying the original notion of linearizability [Herlihy and Wing 1990] such as strict linearizability [Aguilera and Frolund 2003], as well as sophisticated proof strategies for establishing linearizability (e.g. [O'Hearn et al. 2010]). As the original definition of linearizability was based on the strong sequential consistency (SC) [Lamport 1979] memory model, Burckhardt et al. [2012] later adapted linearizability to the weaker TSO model [Sewell et al. 2010], while Batty et al. [2013] adapted it to a fragment of the even weaker C11 model [Batty et al. 2011]. Raad et al. [2019a] developed a general framework for specifying various correctness conditions for concurrent libraries, including linearizability.

In order to account for the durability guarantees of implementations in the context of persistent memory, Izraelevitz et al. [2016] extended linearizability to durable linearizability (DL). As with the original notion of linearizability, this original DL definition was tied to the strong SC model. Raad and Vafeiadis [2018] later developed a weak persistency model known as PTSO and adapted the notion of DL to PTSO. Raad et al. [2020, 2019b] subsequently developed the PARMv8 and Px86

models, respectively formalising the (weak) persistency semantics of the ARMv8 and Intel-x86 architectures, and accordingly adapted DL to account for PARMv8 and Px86. Unlike our memory-model-agnostic approach here, these DL definitions are tied to specific persistency models.

Some memory models admit the so-called data-race freedom property (DRF), which guarantees that, in absence of data-races, only SC behaviour is observable. When DRF applies, proving linearizability would be simpler, but the order in which writes are persisted may still be different from the SC order; our technique would then still be useful and provide a viable proof strategy for DL.

The existing literature includes several examples of durable libraries and data structures. The most notable example is PMDK [Intel 2015], a collection of libraries for persistent programming. However, as of yet the PMDK libraries lack formal specifications and have not been formally verified. Friedman et al. [2018] developed several durable queue libraries over the Px86 model; however, they provide an informal argument (in English) that their implementations are correct (satisfy DL) and do not provide a formal correctness proof. Similarly, Zuriel et al. [2019] developed two durable set implementations (over Px86), including the link-free set we verify here. Once again, they do not provide a formal correctness proof of their implementations, and instead present an informal argument without accounting for the intricacies of the underlying Px86 model.

Raad et al. [2020, 2019b] develop durable variants of the Michael-Scott queue [Michael and Scott 1996] over the PARMv8 and Px86 models, and formally prove that their implementations are correct. These implementations are much simpler than those we verify here (e.g. they do not involve hindsight reasoning). Moreover, unlike our approach here, their proofs are non-modular in that they do not separate the linearizability, persistency and recovery proof obligations. As such, they do not provide any insights that can be adapted to reason about other durable implementations.

Derrick et al. [2021] proposed a sound and complete refinement-based proof technique for DL in the context of SC, which they use to prove a queue from [Friedman et al. 2018]. Their thread-local simulation technique could in principle be combined with our Pathway Theorem, yielding a powerful technique for DL under SC. We leave this exploration to future work.

ACKNOWLEDGMENTS

This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

Marcos K. Aguilera and Svend Frolund. 2003. Strict Linearizability and the Power of Aborting. https://www.hpl.hp.com/techreports/2003/HPL-2003-241.html

Jade Alglave. 2012. A formal hierarchy of weak memory models. *Formal Methods Syst. Des.* 41, 2 (2012), 178–210. https://doi.org/10.1007/S10703-012-0161-5

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *POPL 2013*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/2429069.2429099

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *ESOP 2012 (LNCS, Vol. 7211)*. Springer, Heidelberg, Germany, 87–107. https://doi.org/10.1007/978-3-642-28869-2_5

Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *SPAA*. ACM, 259–269. https://doi.org/10.1145/3210377.3210400

John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2021. Verifying correctness of persistent concurrent data structures: a sound and complete method. *Formal Aspects Comput.* 33, 4-5 (2021), 547–573. https://doi.org/10.1007/s00165-021-00541-8

Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2022. The Path to Durable Linearizability (Extended Version). *CoRR* arxiv:2211.07631 (2022). https://doi.org/10.48550/arxiv.2211.07631

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *PLDI*. ACM, 377–392. https://doi.org/10.1145/3385412.3386031

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In *PLDI*. ACM, 1218–1232. https://doi.org/10.1145/3453483.3454105

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC (Lecture Notes in Computer Science, Vol. 2180)*. Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

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

Intel. 2015. Persistent Memory Programming. http://pmem.io/

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *DISC (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

Takayuki Kawahara, Kenchi Ito, Riichiro Takemura, and Hideo Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectron. Reliab.* 52, 4 (2012), 613–627. https://doi.org/10.1016/J.MICROREL.2011.09.028

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*. ACM, 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* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *PODC*. 85–94. https://doi.org/10.1145/1835698.1835722

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *ISCA*. IEEE Computer Society, 265–276. https://doi.org/10.1109/ISCA.2014.6853222

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019a. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. 3, POPL (2019). https://doi.org/10.1145/3290381

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 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019b. Weak Persistency Semantics from the Ground Up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135, 27 pages. https://doi.org/10.1145/3360561

Gerhard Schellhorn, John Derrick, and Heike Wehrheim. 2014. A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. *ACM Trans. Comput. Log.* 15, 4 (2014), 31:1–31:37. https://doi.org/10.1145/2629496

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 (jul 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Storage Networking Industry Association (SNIA). 2017. NVM Programming Model (NPM). SNIA Technical Position. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf Version 1.2.

Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83. https://doi.org/10.1109/IOLTS.2019.8854427

Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: a library for simple and efficient persistent algorithms. In *PPoPP*. ACM, 309–321. https://doi.org/10.1145/3503221.3508436

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 128:1–128:26. https://doi.org/10.1145/3360554

Received 2022-07-07; accepted 2022-11-07