Verifying Correctness of Persistent Concurrent Data Structures

John Derrick[1], Simon Doherty[1], Brijesh Dongol[2(✉)], Gerhard Schellhorn[3], and Heike Wehrheim[4]

[1] University of Sheffield, Sheffield, UK
[2] University of Surrey, Guildford, UK
b.dongol@surrey.ac.uk
[3] University of Augsburg, Augsburg, Germany
[4] Paderborn University, Paderborn, Germany

Abstract. Non-volatile memory (NVM), aka persistent memory, is a new paradigm for memory preserving its contents even after power loss. The expected ubiquity of NVM has stimulated interest in the design of *persistent* concurrent data structures, together with associated notions of correctness. In this paper, we present the first formal proof technique for *durable linearizability*, which is a correctness criterion that extends linearizability to handle crashes and recovery in the context of NVM. Our proofs are based on refinement of IO-automata representations of concurrent data structures. To this end, we develop a generic procedure for transforming any standard sequential data structure into a durable specification. Since the durable specification only exhibits durably linearizable behaviours, it serves as the abstract specification in our refinement proof. We exemplify our technique on a recently proposed persistent memory queue that builds on Michael and Scott's lock-free queue.

1 Introduction

Recent technological advances indicate that future architectures will employ some form of *non-volatile memory* (NVM) that retains its contents after a system crash (e.g., power outage). NVM is intended to be used as an intermediate layer between traditional *volatile memory* (VM) and secondary storage and has the potential to vastly improve system speed and stability. Software that uses NVM has the potential to be more robust; in case of a crash, a system state before the crash may be recovered using contents from NVM, as opposed to being restarted from secondary storage. However, because the same data is stored in both a volatile and non-volatile manner, and because NVM is updated at a slower rate than VM, recovery to a consistent state may not always be possible. This is particularly true for concurrent systems, where coping with NVM requires introduction of additional synchronisation instructions into a program.

Derrick, Dongol and Doherty are supported by EPSRC grants EP/R032351/1, EP/R032556/2, EP/R019045/2; Wehrheim by DFG grant WE 2290/12-1.

M. H. ter Beek et al. (Eds.): FM 2019, LNCS 11800, pp. 179–195, 2019.
https://doi.org/10.1007/978-3-030-30942-8_12

Recently, researchers have developed *persistent* extensions to existing concurrent objects (e.g., concurrent data structures or transactional memory). This work has been accompanied by extensions to known notions of consistency, such as linearizability and opacity that cope with crashes and subsequent recovery.

In this paper, we examine correctness of the recently developed persistent queue by Friedman et al. [11], against the (also) recently developed notion of *durable linearizability* [14]. Friedman et al.'s queue extends the well-known Michael-Scott queue [20], whereas durable linearizability extends the standard notion of linearizability [12] so that completed executions are guaranteed to survive a system crash.

Our verification follows a well-established methodology: (1) we develop an operational model of durable linearizability that is parameterised by a generic sequential object (e.g., a queue data structure with enqueue and dequeue operations), (2) we prove that this operational model is sound, and (3) we establish a series of refinements between the operational model and the concrete implementation. The final (and most complex) of these steps, which establishes that the implementation refines the operational model, is fully mechanised in the KIV theorem prover [10]. It is important to note that the operational model is generic and for any particular verification one needs therefore just to establish step (3) in order to show that a particular algorithm is durable linearizable.

Ours is the first paper to address formal verification of persistent data structures. We consider the development of our sound operational characterisation of durable linearizability and the refinement proofs, including mechanisation in KIV, to be the main contributions of this paper. The mechanisation and the full version of the paper may be accessed from [17].

We present Friedman et al.'s queue in Sect. 2, durable linearizability in Sect. 3, an operational characterisation of durable linearizability in Sect. 4, and address correctness of the queue in Sect. 5.

2 A Persistent Queue

The persistent queue of Friedman et al. [11] is an extension of the Michael-Scott queue (MSQ) [20] to cope with NVM (see Algorithms 1 and 2). The MSQ uses a linked list of nodes with global head and tail pointers. The first node is a sentinel that simplifies handling of empty queues. The MSQ is initialised by allocating a dummy node with a null next pointer, then setting the global head and tail pointers to this dummy node.

The enqueue operation creates a new node that is inserted at the end of the linked list. The insertion is performed using an atomic compare-and-swap (CAS) instruction that atomically updates the *next* pointer of the last node provided this next pointer hasn't changed since it was read at the beginning of the enqueue operation. The CAS returns true if it succeeds and false otherwise. Immediately after a new node is inserted, the tail pointer is *lagging* one node behind the true tail of the queue, and hence, must be updated to point to the last node in a separate step.

Algorithm 1. Constructors

```
1: class Node
2: T val;
3: Node* next;
4: int deqID;
5: Node(T k):
6:   val(k), deqID(-1), next(null);
```

```
1: class DurableQueue
2: Node* head;
3: Node* tail;
4: T* RVals[MAX];
5: DurableQueue()
6:   T* node := new Node(T());
7:   flush(node);
8:   head := node;
9:   flush(&head);
10:  tail := node;
11:  flush(&tail);
12:  RVals[i] := null; //all i
13:  flush(&RVals[i]);
```

The dequeue operation returns `empty` if the head and tail pointer both point to the sentinel node and the tail is not lagging. If the queue is not empty, the dequeue reads from the value of the node immediately after the sentinel and atomically swings the head pointer to this next node provided it has not changed. Thereby, the next node becomes the new sentinel node of the queue.

A key feature of MSQ is a *helping mechanism* where a different thread from the original enqueue may advance the tail pointer if it is lagging. In the case of a dequeue, this only occurs if head and tail pointers are equal, but the queue is not empty.

Friedman et al. [11] adapt MSQ to a system comprising both VM and NVM. In such systems, computations take place in VM as normal, but data is periodically flushed to NVM by the system. In addition to system controlled flushes, a programmer may introduce explicit **flush** events that transfer data from VM to NVM. Only data in NVM persists after a crash (e.g., power loss). A persistent data structure must enable recovery from such an event, as opposed to a full system restart. In doing this, it must ensure some notion of consistency in the presence of crashes and a subsequent recovery operation. Following Friedman et al. [11], the notion of consistency we use is durable linearizability (see Sect. 3).

The persistent queue uses the same underlying data structure as MSQ (see Algorithm 1), but nodes contain an additional field, `deqID` (initialised to -1), which holds the ID of the thread that removed the node from the queue. In addition to the head and tail pointers, it uses an array of pointers, `RVals`, with one index for each thread, containing either `null` (which is the initial value), of a pointer to a cell which itself either contains `empty` (which signifies that the thread last saw an empty queue), or a value (which is the value that was last dequeued). Unlike MSQ, the persistent dequeue operation does not return a value; instead the returned value for `tid` is stored in the cell pointed to by `RVals[tid]`.

Persistent Enqueue. The basic structure (see Algorithm 2) is the same as the enqueue of MSQ. In addition, to ensure that the linked list data structure

Algorithm 2. Enqueue and dequeue methods of Friedman et al. [11]

```
1: procedure ENQ(T val)
2: Node* node := new Node(val);
3: flush(node);
4: while true do
5:   Node* last := tail;
6:   Node* nxt := last→next;
7:   if (last = tail)
8:     if (nxt = null)
9:       if CAS(&last→next,nxt,node)
10:        flush(&last→next);
11:        CAS(&tail, last, node);
12:        return;
13:    else
14:      flush(&last→next);
15:      CAS(&tail, last, nxt);
```

```
1: procedure DEQ(int tid)
2: T* newRVal := new T();
3: flush(newRVal);
4: RVals[tid] := newRVal;
5: flush(&RVals[tid]);
6: while true do
7:    Node* first := head;
8:    Node* last := tail;
9:    Node* nxt := first→next;
10:   if (first = head)
11:     if (first = last)
12:       if (nxt = null)
13:         *RVals[tid] := empty;
14:         flush(RVals[tid]);
15:         return;
16:       flush(&last→next);
17:       CAS(&tail, last, nxt);
18:     else
19:       T val := nxt→val;
20:       if CAS(&nxt→deqID,-1,tid)
21:         flush(&nxt→deqID);
22:         *RVals[tid] := val;
23:         flush(RVals[tid]);
24:         CAS(&head, first, nxt);
25:         return;
26:       else
27:         T* addr:=RVals[nxt→deqID];
28:         if (head = first)
29:           flush(&nxt→deqID);
30:           *addr := val;
31:           flush(addr);
32:           CAS(&head,first,nxt);
```

is recoverable after a crash, nodes and next pointers have to be persisted after being modified in VM.

This is achieved by using three **flush** operations in lines 3, 10 and 14. The first ensures that the node is persisted before it is inserted into the queue; the second and third ensure that the next pointer of a lagging tail pointer is persisted before the tail is advanced. Note that updates to tail do not need to be explicitly flushed because it can be recomputed during recovery by traversing the persistent list.

Persistent Dequeue. The basic structure of the dequeue operation also resembles the dequeue of MSQ. In addition it uses variables `RVals` and `deqID` to guarantee durable linearizability. `RVals` is an array of pointers to cells that are used to store the value returned by each dequeue. A dequeue creates a new cell at

Line 2, then flushes it at Line 3. The pointer to this cell is stored in `RVals` at Line 4, and this pointer is made persistent at Line 5.

The `deqID` field is used to logically mark nodes that are dequeued, which occurs at the successful CAS at Line 20. This logical dequeue is made persistent by flushing the `deqID` at Line 21. After a node has been logically dequeued, the dequeued value is stored in the cell pointed to by `RVals[tid]` (see Line 22) where `tid` is the thread ID of the dequeuing thread. This dequeued value is made persistent at Line 23. A dequeue by thread `tid` stores `empty` in `RVals[tid]` if the queue is empty in Line 13, and this value is made persistent at Line 14.

The persistent dequeue operation employs an additional helping mechanism to ensure that these new fields are made persistent in the correct order. In particular, a node that has been logically dequeued in VM must be made persistent before another dequeue is allowed to succeed. Therefore, if a thread recognises that `deqID` is not -1 at Line 20, it helps the other thread by flushing the `deqID` field, writing the dequeued value into the cell pointed to by `RVals[nxt→tid]`, flushing this cell, and finally advancing the head pointer. Note that the helping thread may be delayed between the read at Line 27 and the write at Line 30, and the original thread `tid` may begin a new dequeue operation in this interval. In this case, since `tid` allocates a fresh cell at Line 2, the helping thread's write at Line 30 will harmlessly modify a previous cell.

After a crash, and prior to resuming normal operation, persistent data structures must perform a *recovery* operation that restores the state of the data structure in VM from NVM. The recovery procedure proposed by Friedman et al. is multithreaded (and complex), so we elide its details here. Instead, we provide a simpler single-threaded recovery operation (see Sect. 5.1).

3 Durable Linearizability

We now define *durable linearizability* [14], a central correctness condition for persistent concurrent data structures. Like linearizability, durable linearizability is defined over *histories* recording the *invocation* and *response* events of operations executed on the concurrent data structure. Unlike linearizability, durably linearizable histories include crash events.

Formally, we let Σ be the set of operations. For a queue, $\Sigma = \{\texttt{Enq}, \texttt{Deq}\}$. A history is a sequence of events, each of which is either (a) an invocation of an operation op by a thread $t \in T$ with values $\boldsymbol{v}$, written $inv_t(op, \boldsymbol{v})$, (b) responses of op in thread t with value v, written $res_t(op, v)$, or (c) a system-wide crash c.

Given a history h, we let $ops(h)$ denote h restricted to non-crash events, and $h_{|t}$ denote h restricted to (non-crash) events of thread $t \in T$. The crash events partition a history into $h = h_0 c_1 h_1 c_2 ... h_{n-1} c_n h_n$, such that n is the number of crash events in h, c_i is the ith crash event and $ops(h_i) = h_i$ (i.e., h_i contains no crash events). We call the subhistory h_i the *i-th era* of h. For a history h and events e_1, e_2, we write $e_1 <_h e_2$ whenever $h = h_0 e_1 h_1 e_2 h_2$.

A history h is said to be *sequential* iff every invocation event (except if it is the last event in h) is immediately followed by its corresponding response event;

it is *well formed* if and only if (a) $h_{|t}$ is sequential for every thread t and (b) each thread id appears in at most one era. Any invocation that is not followed by its response event is called a *pending* invocation. We consider well-formed histories only. A history h defines a *happens-before* ordering on the events occuring in h by letting $e_1 \prec_h e_2$ iff $e_1 <_h e_2$ and e_1 is a response and e_2 an invocation event. Linearizability (and durable linearizability) requires a notion of a *legal history*, which we define using a sequential object. Every history of a sequential object is both sequential and legal.

Definition 1 (Sequential Object). *A* sequential object *over a base type Val is a 5-tuple* $(\Sigma, S, s_0, in, \rho)$ *where*

- Σ *is an* alphabet *of operations,* S *is a set of states and* s_0 *the initial state,*
- $in : \Sigma \to \mathbb{N}$ *is an* input function *telling us the number of inputs an operation* $op \in \Sigma$ *takes, and*
- $\rho : S \times \Sigma \times Val^* \to S \times (Val \cup \{\texttt{empty}, \bot\})$ *is a partial* transition function.

We assume outputs of operations to consist of a single value which possibly is the symbol `empty` or no value denoted by $\bot$. In the following we let $\boldsymbol{v} = v_1 v_2 \dots v_n$ denote a string of n elements and write $\#\boldsymbol{v}$ to denote its length n. We write $inv_t(op, \boldsymbol{v})$ for an invocation of the operation op with $n = \#\boldsymbol{v}$ inputs by thread t and let Inv be the set of all such invocations. Similarly, we let Res be the set of all responses.

The *legal* histories of a sequential object $\mathbb{S} = (\Sigma, S, s_0, in, \rho)$ are defined as follows. We write $s \xrightarrow{inv_t(op,\boldsymbol{v})res_t(op,v)} s'$ for $\rho(s, op, \boldsymbol{v}) = (s', v)$ and $t \in T$. For a sequence w of invocations and responses, we write $s \xrightarrow{w} s'$ iff either $w = \langle\rangle$ and $s = s'$, or $w = u \circ w'$ and there exists an s'' such that $s \xrightarrow{u} s''$ and $s'' \xrightarrow{w'} s'$. The set of *legal histories* of $\mathbb{S}$ is given by $legal_{\mathbb{S}} = \{w \in (Inv \cup Res)^* \mid \exists s \in S.\ s_0 \xrightarrow{w} s\}$.

Example 2. A sequential queue, $\mathbb{Q}$, storing elements of type V is defined by $\Sigma = \{\texttt{Enq}, \texttt{Deq}\}$, $in(\texttt{Enq}) = 1, in(\texttt{Deq}) = 0$, $q_0 = \langle\rangle$, and

$$\begin{aligned}\rho = &\{\big((q, \texttt{Enq}, v), (q \cdot v, \bot)\big) \mid v \in V \wedge q \in V^*\} \cup \\ &\{\big((v \cdot q, \texttt{Deq}, \varepsilon), (q, v)\big) \mid v \in V \wedge q \in V^*\} \cup \{\big((\langle\rangle, \texttt{Deq}, \varepsilon), (\langle\rangle, \texttt{empty})\big)\}\end{aligned}$$

where ε is the empty string, $\langle\rangle$ is the empty sequence and $\cdot$ is used for sequence concatenation. For $\mathbb{Q}$, the history h below is sequential and legal

$$h = \langle inv_1(\texttt{Enq}, a), res_1(\texttt{Enq}, \bot), inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$$

whereas the history $h \cdot \langle inv_3(\texttt{Deq}, \varepsilon), res_3(\texttt{Deq}, b)\rangle$ is sequential but not legal.

For the definition of durable linearizability some more notation is needed. We write $h \equiv h'$ if $h_{|t} = h'_{|t}$ for all threads t. We let $compl(h)$ (the completion) be the set of histories that can be obtained from h by appending (some) missing responses at the end, and use $trunc(h)$ to remove pending invocations from a history h (or a set of histories). Following Herlihy and Wing [12], h is *linearizable*

if there is some $h' \in trunc(compl(h))$ and some legal sequential history h_S such that (i) $h' \equiv h_S$ and (ii) $\forall e_1, e_2 \in h' : e_1 \prec_{h'} e_2 \Rightarrow e_1 \prec_{h_S} e_2$.

For durable linearizability, this definition is now simply lifted to histories with crashes.

Definition 3 (Durable Linearizability [14]). *A history h is* durably linearizable *if it is well formed and $ops(h)$ is linearizable.*

Informally, durable linearizability guarantees that even after a crash the state of the concurrent object remains consistent with the abstract specification. This means that the effect of any operations that completed before a crash are preserved after the crash. The effect of operations that did not complete before a crash may or may not be preserved. For example, the concurrent history

$$hc = \langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Deq}, \varepsilon), res_1(\texttt{Enq}, \bot), c, inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$$

is durably linearizable since $ops(hc) = \langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Deq}, \varepsilon), res_1(\texttt{Enq}, \bot), inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$ is linearizable with respect to the history h in Example 2. On the other hand the history

$$\langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Enq}, b), res_1(\texttt{Enq}, \bot), c, inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, \texttt{empty})\rangle$$

is not durably linearizable since the effect of the completed operation $\texttt{Enq}(a)$ is not preserved after the crash.

Our methodology for proving durable linearizability does not use Definition 3 directly; instead it uses the following characterisation, which defines the set of all durably linearizable histories for a sequential object.

We let $\textsc{Lin}(\mathbb{S})$ be the set of histories linearizable wrt. the legal histories of sequential object $\mathbb{S}$ and define

$$\textsc{DurLin}(\mathbb{S}) = \{h \in (Inv \cup Res \cup \{c\})^* \mid ops(h) \in \textsc{Lin}(\mathbb{S})\}$$

For a given concurrent durable data structure implementing a sequential object $\mathbb{S}$, proving its correctness thus amounts to showing that all histories of the implementation are in $\textsc{DurLin}(\mathbb{S})$. To this end, for a given $\mathbb{S}$, we develop an operational model $\textsc{DurAut}(\mathbb{S})$ whose behaviours generate $\textsc{DurLin}(\mathbb{S})$. We then use a standard refinement approach to show that the implementation model is a refinement of $\textsc{DurAut}(\mathbb{S})$. This is enough to guarantee that the original implementation is durably linearizable.

4 An Operational Model for Durable Linearizability

The operational model for durable linearizability is formalised in terms of an *Input/Output automaton (IOA)* [18]. This framework is often used for proving linearizability via refinement [9].

$inv_t(op, \boldsymbol{v})$
Pre: $pc(t) = ready$
$\#\boldsymbol{v} = in(op)$
Eff: $pc(t) := inv(op)$
$\boldsymbol{val}(t) := \boldsymbol{v}$

$do_t(op)$
Pre: $pc(t) = inv(op)$
Eff: $pc(t) := res(op)$
$(s, out(t)) := \rho(s, op, \boldsymbol{val}(t))$

$res_t(op, v)$
Pre: $pc(t) = res(op)$
$v = out(t)$
Eff: $pc(t) := ready$

run_t
Pre: $pc(t) = idle$
Eff: $pc(t) := ready$

$crash$
Pre: $true$
Eff: $pc := \lambda t : T.$**if** $pc(t) \neq idle$ **then** $crashed$ **else** $pc(t)$

$$states(A) : pc : T \to \{idle, ready, crashed\} \cup \{inv(op), res(op) \mid op \in \Sigma\}$$
$$s : S \text{ (the state of the sequential object)}$$
$$\boldsymbol{val} : T \to Val^* \text{ (values of inputs)}$$
$$out : T \to Val \text{ (the value of the output)}$$
$$start(A) : \forall t \in T : pc(t) = idle \wedge \boldsymbol{val}(t) = \epsilon \wedge out(t) = \epsilon \wedge s = s_0$$

Fig. 1. Durable automaton $A = \textsc{DurAut}(\mathbb{S})$ for $\mathbb{S} = (\Sigma, S, s_0, in, \rho)$

Definition 4. *An* IOA *is a labeled transition system A with*

- *a set of* states $states(A)$*,*
- *a set of* start *states* $start(A) \subseteq states(A)$*,*
- *a set of* actions $acts(A)$*, and*
- *a transition relation* $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ *(so that the actions label the transitions).*

The set $acts(A)$ is partitioned into *internal* actions, $internal(A)$ and *external* actions, $external(A)$.[1] The internal actions represent events of the system that are not visible to the environment, whereas the external actions represent the automaton's interactions with its environment.

An *execution* of an IOA A is a sequence $\sigma = s_0 a_1 s_1 a_2 s_2 a_3 \ldots$ of alternating states and actions such that $s_0 \in start(A)$ and for each i, $(s_i, a_{i+1}, s_{i+1}) \in trans(A)$. A *reachable* state of A is a state appearing in an execution of A. An *invariant* of A is any superset of the reachable states of A (equivalently, any predicate satisfied by all reachable states of A). A *trace* of A is any sequence of (external) actions obtained by projecting onto the external actions of any execution of A. The set of traces of A, $traces(A)$, represents A's externally visible behaviour. If every trace of an automaton C is also a trace of an automaton A, then we say that C *implements* or *refines* A.

For an arbitrary sequential object $\mathbb{S}$, we next construct a durable automaton $\textsc{DurAut}(\mathbb{S})$ (see Fig. 1) whose traces are histories in $\textsc{DurLin}(\mathbb{S})$ only. This automaton can serve as a specification automaton in a refinement proof. The state of this automaton incorporates the state s of the sequential object $\mathbb{S}$, plus for every thread $t \in T$:

[1] In the standard IOA setting, external actions are further subdivided into input and output actions; this distinction is not needed for this current work.

- a program counter fixing whether the thread is still *idle*, is *ready* to be started, is *crashed* (i.e., has been active during a crash), or is currently executing an operation,
- possible input values of the thread's operations and a possible output value.

The transition relation of the automaton is – as usual – given in the form of pre- and postconditions of actions. For every operation op in the sequential object, the automaton has actions $inv(op)$, $do(op)$ and $res(op)$, where $do(op)$ corresponds to execution of the abstract operation op, potentially changing the state of the sequential object. We use $inv_t(op, \boldsymbol{v})$ and $res_t(op, v)$ for $inv(op)_t(\boldsymbol{v})$ and $res(op)_t(v)$, respectively. Any step of the implementation that refines $do(op)$ is a step that persists the corresponding operation op (i.e., a *persistence point*, see Sect. 5). Persistence points in durable linearizability are analogous to linearization points in linearizability [9]. Note that a thread may only invoke an operation if it is *ready*. We furthermore have a dedicated *crash* action that may be executed at any time that sets all active threads to *crashed*. To ensure that crashed threads are confined to a single era, we use a separate action *run* that enables idle threads to become *ready*. While $inv(op)$, $res(op)$ and *crash* are external actions, *run* and $do(op)$ are internal.

The theorem below ensures that traces of the durable automaton are the durably linearizable histories of $\mathbb{S}$.

Theorem 5. *If $\mathbb{S}$ is a sequential object, then traces*(DurAut($\mathbb{S}$)) $\subseteq$ DurLin($\mathbb{S}$).

Proof. Let $\sigma = cs_0 a_1 cs_1 \dots a_n cs_n$ be an execution of DurAut($\mathbb{S}$) and let $cs_i.s$, $cs_i.out$ etc. be the components of state cs_i. Let tr be the trace of σ. We construct the history h by making the following changes to tr (in this order).

Completion For every a_i being a do action $do_t(op)$ in σ without matching $res_t(op)$, we add $res_t(op, v)$ such that $v = cs_i.out(t)$ to the end of tr.
Truncation We remove all $inv_t(op, \boldsymbol{v})$ without matching response.

Next, we need to construct a legal sequential history h_S such that $ops(h) \equiv h_S$. Let $i_1, \dots, i_k$ be the indices of σ such that a_{i_j} is a do action $do_t(op)$. Then $\rho(cs_{i_{j-1}}.s, op, \boldsymbol{v}) = (cs_{i_j}.s, cs_{i_j}.out(t))$ by definition of the durable automaton. We set

$$w_{i_j} = inv_t(op, \boldsymbol{v})\, res_t(op, cs_{i_j}.out(t)) \ .$$

We let $h_S = w_{i_1} \dots w_{i_k}$ and $h_S \in legal(\mathbb{S})$.

Now assume $e_1 \prec_h e_2$. By definition, $e_1 = res_t(op, v)$ and $e_2 = inv_{t'}(op', \boldsymbol{v})$ for some $t, t' \in T$. Then e_1 has not been added to the trace tr by completion since responses are added to the end. By construction of the durable automaton threads execute *inv*, *do* and *res* operations in this ordering only. Hence the execution σ contains an action $do_t(op)$ prior to e_1 and an action $do_{t'}(op')$ following e_2. Hence $e_1 \prec_{h_S} e_2$. □

In fact, we believe that the two sets in Theorem 5 are equal. However, we do not need this property for our proof methodology.

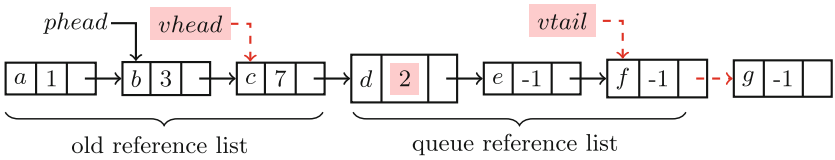


Fig. 2. Possible state of persistent queue; volatile data represented using shading and volatile pointers represented using dashed arrows

5 Correctness of the Persistent Queue

In this section we present a formal verification of the persistent queue. In Section 5.1, we describe a model of the queue. In Sect. 5.2 we describe the application of the refinement-based verification to this example, where we establish the relationship between an intermediate automaton and the durable automaton. Section 5.3 describes the persistence points in the concrete implementation that are used in the proof, and Sect. 5.4 describes the main invariants and abstraction relations.

5.1 Modelling the Persistent Queue

To verify durable linearizability we need to model the persistent queue. The persistent queue contains two versions of each variable: one in VM and one in NVM. We model this in the automaton by two mappings $ps, vs : Loc \to X$, where Loc is a set of locations and X is a generic set that contains enqueued values, references, thread ids, etc. Mappings ps and vs represent the persistent and volatile states, respectively. A flush of location k updates the value of $ps(k)$ to $vs(k)$, while recovery moves data from $ps(k)$ to $vs(k)$. All other operations take place in $vs(k)$.

In order to help illustrate the structure of the queue, Fig. 2 depicts a possible state of the persistent queue. Each node contains three values: a data value, a thread id (possibly -1, which is the initial value), and a next pointer. Variables *phead* and *vhead* are the persistent and volatile head pointers, respectively, and *vtail* is the volatile tail pointer. In the KIV model $phead = ps(head)$, $vhead = vs(head)$, etc. The values depicted by shading and the dashed arrows in the figure are volatile; in Fig. 2, these are the `deqID` of node d and the next pointer of node f, as well as the volatile head and tail pointers. Here enqueues for nodes labelled a to f have all taken place and persisted, whereas node labelled g has been partly enqueued but not yet persisted. Nodes labelled a to c have been dequeued and persisted, but the node labelled d has been marked for dequeue, but not persisted. Here, the *phead* is lagging behind *vhead*; in an execution, *phead* may be lagging by an arbitrary amount as the flush of *vhead* is controlled by the system as opposed to an explicit flush statement in the program code.

The persistent contents of the queue (which we refer to as the *queue reference list*) corresponds to the abstract queue $\langle d, e, f \rangle$. In addition, our proof makes use

$inv_t(op, \boldsymbol{v})$
Pre: $pc(t) = ready$
$\#\boldsymbol{v} = in(op)$
Eff: $pc(t) := inv(op)$
$\boldsymbol{val}(t) := \boldsymbol{v}$
$obsEmp(t) := false$

$do_t(op)$
Pre: $pc(t) = inv(op)$
Eff: $pc(t) := res(op)$
$(q, out(t)) := \rho(q, op, \boldsymbol{val}(t))$

$res_t(Enq)$
Pre: $pc(t) = res(Enq)$
Eff: $pc(t) := ready$

$res_t(Deq, v)$
Pre: $pc(t) = res(Deq)$
$v = out(t)$
Eff: $pc(t) := ready$

$do_t(Deq)$
Pre: $pc(t) = inv(Deq)$
$obsEmp(t)$
Eff: $pc(t) := res(Deq)$
$out(t) := \texttt{empty}$

$checkEmp_t$
Pre: $pc(t) = inv(Deq)$
$q = \langle\rangle$
Eff: $obsEmp(t) := true$

run_t
Pre: $pc(t) = idle$
Eff: $pc(t) := ready$

$crash$
Pre: $true$
Eff: $pc := \lambda t : T.\textbf{if } pc(t) \neq idle \textbf{ then } crashed \textbf{ else } pc(t)$

$$
\begin{aligned}
states(IDQ) : pc &: T \to \{idle, ready, crashed\} \cup \{inv(op), res(op) \mid op \in \Sigma\} \\
q &: V^* \text{ (the state of the sequential queue)} \\
val &: T \to V^* \text{ (values of inputs)} \\
out &: T \to V \cup \{\bot\} \text{ (the value of the output)} \\
obsEmp &: T \to \mathbb{B}
\end{aligned}
$$

Fig. 3. The intermediate automaton *IDQ*

of the so called old reference list, which are elements that had been persistently enqueued, but have also been persistently dequeued.

5.2 Refinement-Based Verification

As outlined in Sect. 3, we verify durable linearizability by proving refinement between the implementation model and DurAut($\mathbb{Q}$) using the IO automata formalism introduced in Sect. 4. Refinement can be proven via *forward* or *backward simulations* [19]; such simulations allow a step-by-step comparison between the operations using an abstraction relation. In our proof, we establish a backward simulation between the intermediate automaton and DurAut($\mathbb{Q}$) as well as a forward simulation between the implementation of the persistent queue and the intermediate automaton. The proof uses an intermediate automaton that resolves non-determinism at the abstract level as used in existing proofs of MSQ [8,9]. Since refinement guarantees trace inclusion, this is sufficient to show that the persistent queue is durably linearizable.

The intermediate automaton *IDQ*, presented in Fig. 3, is similar to the durable automaton for the queue datatype, DurAut($\mathbb{Q}$) (see Fig. 1 instantiated for the queue from Example 2). As with DurAut($\mathbb{Q}$), it has variables *pc*, *val* and *out*, which play the same role, and variable q instantiates the state s.

Furthermore, all its actions except for *checkEmp* are also actions of DurAut($\mathbb{Q}$), and have essentially the same effect. For *IDQ* we get the following property[2].

Theorem 6. *traces*(*IDQ*) $\subseteq$ *traces*(DurAut($\mathbb{Q}$)) .

The additional features of *IDQ* exist to model a behaviour where a dequeue thread first *observes* that the queue is empty, and later decides to return `empty`, at a point when the queue may no longer *be* empty. The observation is modelled by a $checkEmp_t$ action, which records in the $obsEmp_t$ variable the fact that the queue was empty during the execution of t's dequeue operation. In this automaton, it is possible for a thread t to execute a $do_t(Deq)$ transition and set the output value to `empty` whenever $obsEmp(t)$ has been set to *true*. We note that the queue may not actually be empty when this transition takes place, but this does not affect soundness of the proof method since $obsEmp(t)$ being set to *true* indicates that the queue has been empty at *some point* during the operation's execution. Further details of this technique, in the context of linearizability, may be found in [4,8,9].

5.3 Identification of Persistence Points

To match executions of the concrete implementation with the abstract level, we must identify the *persistence points* of the implementation, which are atomic events whose execution causes the effect of the corresponding operations to take effect at the abstract level. These are analogous to standard linearizability, where proofs proceed via identification of *linearization points* [9]. In durable linearizability, persistence points are typically statements (flush events) that cause the operation under consideration to become durable. Thus these statements must be simulated by the abstract *do* operation. Note that persistent points must occur after an operation has taken effect in NVM, but before the operation returns.

In MSQ, the enqueue operation linearizes upon successful execution of the CAS at line 9. However, in the persistent queue, this line is not the persistence point of the operation, rather it is the first operation that *flushes the effect of this CAS*, i.e., the first flush of the next pointer to the enqueued element. This may occur in line 10 of the same thread, line 14 executed by another thread, or due to a system-controlled flush. Despite there being several possible choices for the persistence point, it is possible to prove forward simulation with respect to the $do(Enq)$ operation of the intermediate automaton *IDQ*.

The verification of the empty dequeue follows a similar pattern to the verification of the empty dequeue of MSQ. The persistence point is conditional on the future execution of the operation, thus we refer to the persistence point as a *potential persistence point* (this is similar to the concept of potential linearization points [4,8,9]). The empty dequeue potentially takes effect at line 9 if the value

[2] For the proof, see the full paper at [17].

loaded for `nxt` is `null`, but this decision is not resolved until later in the operation (line 12). Using the intermediate automaton (Fig. 3), it allows the proof to proceed via forward simulation, like earlier proofs of linearizability [8,9].

A non-empty dequeue linearizes in VM when the node that is dequeued is marked for deletion by updating its `deqID` field at line 20. Like the enqueue, the persistence point of the dequeue is the first flush of this `deqID` field. This may occur either at line 21 of the same thread, line 29 of another thread, or a system flush. Again, we show that each of these steps simulates the $do(Deq)$ operation of the intermediate automaton.

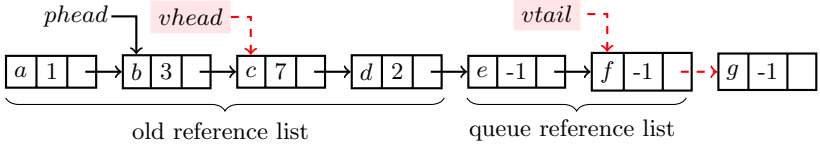
5.4 Key Invariants and Abstraction Relation

There are several key properties that the persistent queue must maintain in order to ensure correctness. These are formalised as invariants in our proof. Here we describe them in plain English:

1. We keep track of two sublists: *old reference list*, which are elements that have been dequeued, and *queue reference list*, which are elements that form the current queue. Formalising the structure of these lists and ensuring global correctness of the invariant is one of the most difficult parts of the proof. This is particularly true for steps that correspond to persistence points (see below) since the volatile pointers can be "lagging" immediately after persistence.
2. All nodes of the queue must be reachable in NVM (i.e., *ps*) from *phead*. This means that the nodes including the next pointers must be made persistent prior to inserting a new node.
3. All nodes in the old reference list must have a `deqID` field different from -1 in *ps*, indicating that they have been dequeued.
4. All nodes in the queue reference list must have a `deqID` field value -1 in *ps*.
5. Only the first node in queue reference list may have a `deqID` field value different from -1 in *vs*.
6. Pointers *phead* and *ptail* may be lagging behind *vhead* and *vtail*, respectively. However, *vhead* may not overtake *vtail*.

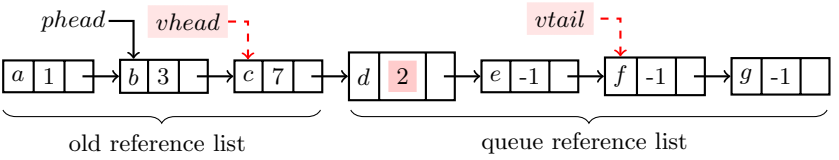
We now describe the state of the queue after execution of some key steps of the algorithm.

Dequeue Persistence Point. A node is considered to be dequeued if its logical deletion is flushed, i.e., the `deqID` marked by a thread id is flushed. For the queue depicted in Fig. 2, the queue immediately after the volatile `deqID` of node d is flushed is as follows.



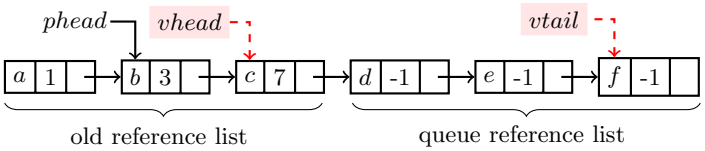
The abstract queue corresponding to this queue is $\langle e, f \rangle$. Note that in the queue above, *vhead* pointer is now lagging and must be updated to point to the new sentinel node d.

Enqueue Persistence Point. A node is considered to be enqueued if it can be reached from *phead* in NVM, and its `deqID` field in persistent memory is -1. Consider again the queue in Fig. 2. The queue immediately after the next pointer of f becomes persistent is as follows.



Note that this transformation must be performed before moving *vtail*, otherwise the nodes after g could be lost upon system crash. In the queue above *vtail* is lagging, and hence, must be updated before a new node can be enqueued. As soon as the next pointer of f becomes persistent, the node g is considered to be part of the queue, i.e., the abstract queue corresponding to the queue above is $\langle d, e, f, g \rangle$.

Crash and Recovery. Finally, consider the queue in Fig. 2 after a crash and recovery:



The volatile `deqID` value for node d is restored from persistent memory, but the node g is lost.

Abstraction Relation and Mechanisation in KIV. These invariants enable us to prove a refinement between the implementation and *IDQ* in Fig. 3. The main part of the abstraction relation states that the abstract queue corresponds to values in the queue reference list. For an enqueue, the first flush that persists the next pointer (i.e., the effect of line 9) must match $do_t(op)$ with $op = Enq$. For a non-empty dequeue, the first flush that persists `deqID` must match $do_t(op)$ with $op = Deq$ in Fig. 3. An empty dequeue must match $checkEmp_t$ when it loads `nxt` at line 9 and $do_t(Deq)$ if the test at line 12 succeeds.

This refinement has been interactively, mechanically proven in the KIV theorem prover [10] (see [17] for the KIV proof and the encodings), which has been used extensively in the verification of concurrent data structures (e.g., [4,24]). The proof of the invariant in KIV is simplified via the use of a *rely condition* [15] that captures interference from a thread's environment in an abstract manner. Roughly speaking, a rely condition is a relation over the states of an automaton that must preserve the invariant of each transaction, and that must abstract the

transitions of each transaction. Similar techniques have been used in previous proofs of concurrent algorithms [6].

6 Conclusion

There are numerous approaches to proving (standard) linearizability of concurrent data structures (e.g., [1,24,27]; see [9] for an overview), including specialisations to cope with weak memory models (e.g., [2,5,7,22,25,26]). The recent development of NVM has been accompanied by persistent versions of well-known concurrent constructs, including concurrent objects [3,11], synchronisation primitives [13,21] and transactional memory [16]. This paper has focussed on a persistent queue [11], against the recently developed notion of durable linearizability [14].

Development of objects implemented for NVM presents a similar challenge to weak memory, in the sense that there are multiples levels of memory to consider. Moreover, caches and registers are volatile, while cache flush instructions allow reordering with store instructions in accordance with the memory model of the system (e.g., [23]). Correctness in the presence of crashes and recovery can be affected by the order in which elements are persisted, which necessitates the use of programmer-controlled flush operations, increasing complexity. Unfortunately, proofs of correctness (e.g., of durable linearizability) are either given informally or are entirely lacking. This gives little confidence in the correctness of the underlying persistent objects.

Verification of persistent memory algorithms is inherently more complex than in the standard setting. Since an operation only takes effect after a flush event, helping is inevitably required to bring the data structure into a consistent state and for an operation to take effect. For proofs by refinement, these additional helping steps have to be considered in the simulation proof. This ultimately complicates the invariant since the helping is performed by another thread (including a system thread). Moreover, since the state of the data structure can be "lagging" immediately after helping is performed, precisely formalising the underlying helping mechanism further complicates the invariant. Future work will consider how best to manage this additional proof complexity.

Acknowledgements. We thank Lindsay Groves for comments that have helped improve this paper.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_23
2. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: Giacobazzi, R., Cousot, R. (eds.) Symposium on Principles of Programming Languages, POPL, pp. 235–248. ACM (2013). https://doi.org/10.1145/2429069.2429099

3. Cohen, N., Aksun, D.T., Larus, J.R.: Object-oriented recovery for non-volatile memory. PACMPL **2**(OOPSLA), 153:1–153:22 (2018)
4. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_25
5. Derrick, J., Smith, G., Groves, L., Dongol, B.: A proof method for linearizability on TSO architectures. In: Hinchey, M.G., Bowen, J.P., Olderog, E.-R. (eds.) Provably Correct Systems. NMSSE, pp. 61–91. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-48628-4_4
6. Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Proving opacity of a pessimistic STM. In: OPODIS, LIPIcs, vol. 70, pp. 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
7. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Making linearizability compositional for partially ordered executions. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 110–129. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_7
8. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_7
9. Dongol, B., Derrick, J.: Verifying linearisability: a comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015). https://doi.org/10.1145/2796550
10. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV–overview and verifythis competition. Softw. Tools Technol. Transf. (STTT) **17**(6), 677–694 (2015)
11. Friedman, M., Herlihy, M., Marathe, V.J., Petrank, E.: A persistent lock-free queue for non-volatile memory. In: Krall, A., Gross, T.R. (eds.) ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 28–40. ACM (2018). https://doi.org/10.1145/3178487.3178490
12. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
13. Huang, Y., Pavlovic, M., Marathe, V.J., Seltzer, M., Harris, T., Byan, S.: Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In: USENIX Annual Technical Conference, pp. 967–979. USENIX Association (2018)
14. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille, C., Ilcinkas, D. (eds.) DISC 2016. LNCS, vol. 9888, pp. 313–327. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53426-7_23
15. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
16. Joshi, A., Nagarajan, V., Cintra, M., Viglas, S.: DHTM: durable hardware transactional memory. In: ISCA, pp. 452–465. IEEE Computer Society (2018)
17. KIV proofs for the durable linearizable queue (2019). http://www.informatik.uni-augsburg.de/swt/projects/Durable-Queue.html
18. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC, pp. 137–151. ACM, New York (1987). https://doi.org/10.1145/41840.41852

19. Lynch, N., Vaandrager, F.W.: Forward and backward simulations part I: untimed systems. Inf. Comput. Inf. Control - IANDC **121**, 214–233 (1995). https://doi.org/10.1006/inco.1995.1134
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of 15th ACM Symposium on Principles of Distributed Computing, pp. 267–275 (1996)
21. Pavlovic, M., Kogan, A., Marathe, V.J., Harris, T.: Brief announcement: persistent multi-word compare-and-swap, In: PODC, pp. 37–39. ACM (2018)
22. Raad, A., Doko, M., Rozic, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. PACMPL **3**(POPL), 68:1–68:31 (2019). https://dl.acm.org/citation.cfm?id=3290381
23. Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model. PACMPL **2**(OOPSLA), 137:1–137:27 (2018)
24. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Logic **15**(4), 31:1–31:37 (2014). https://doi.org/10.1145/2629496
25. Travkin, O., Mütze, A., Wehrheim, H.: SPIN as a linearizability checker under weak memory models. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 311–326. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_21
26. Travkin, O., Wehrheim, H.: Verification of concurrent programs on weak memory models. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 3–24. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_1
27. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40