Detectable Sequential Specifications for Recoverable Shared Objects

Nan Li
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Abstract

The recent commercial release of persistent main memory by Intel has sparked intense interest in recoverable concurrent objects. Such objects maintain state in persistent memory, and can be recovered directly following a system-wide crash failure, as opposed to being painstakingly rebuilt using recovery state saved in slower secondary storage. Specifying and implementing recoverable objects is technically challenging on current generation hardware precisely because the top layers of the memory hierarchy (CPU registers and cache) remain volatile, which causes application threads to lose critical execution state during a failure. For example, a thread that completes an operation on a shared object and then crashes may have difficulty determining whether this operation took effect, and if so, what response it returned. Friedman, Herlihy, Marathe, and Petrank (DISC'17) recently proposed that this difficulty can be alleviated by making the recoverable objects *detectable*, meaning that during recovery, they can resolve the status of an operation that was interrupted by a failure. In this paper, we formalize this important concept using a *detectable sequential specification (DSS)*, which augments an object's interface with auxiliary methods that threads use to first declare their need for detectability, and then perform detection if needed after a failure. Our contribution is closely related to the nesting-safe recoverable linearizability (NRL) framework of Attiya, Ben-Baruch, and Hendler (PODC'18), which follows an orthogonal approach based on ordinary sequential specifications combined with a novel correctness condition. Compared to NRL, our DSS-based approach is more portable across different models of distributed computation, compatible with several existing linearizability-like correctness conditions, less reliant on assumptions regarding the system, and more flexible in the sense that it allows applications to request detectability on demand. On the other hand, application code assumes full responsibility for nesting DSS-based objects. As a proof of concept, we demonstrate the DSS in action by presenting a detectable recoverable lock-free queue algorithm and evaluating its performance on a multiprocessor equipped with Intel Optane persistent memory.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Computer systems organization → Reliability

Keywords and phrases persistent memory, concurrency, fault tolerance, correctness, detectability

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.29

Funding *Wojciech Golab*: Author supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

1 Introduction

The past several years have witnessed an eruption of research into software techniques for harnessing the power of persistent memory, a byte-addressable medium that combines the performance of conventional DRAM with the data durability of secondary storage devices. Prior to the emergence of persistent memory, traditional memory hierarchies forced applications to maintain state redundantly using both in-memory structures for performance and on-disk structures for durability, which imposes a performance overhead during failure-

free operation, and also slows down recovery after a failure. The convergence of primary and secondary storage in persistent memory creates a new avenue for eliminating the inefficiency of rebuilding in-memory state from zero after each system-wide failure, and calls for thoughtful reconsideration of established software design principles.

Harnessing the performance of persistent memory while maintaining correctness to preserve application data integrity poses a number of technical challenges. To begin with, well-studied legacy techniques based on database-style recovery logging provide insufficient parallelism when applied to high-performance concurrent data structures. Novel techniques are needed to remove this bottleneck, and these often intertwine tightly with concurrency control and memory management mechanisms. The implementation task is further complicated on current generation hardware by volatile caching, which necessitates the use of explicit persistence instructions to flush updates to the persistent medium, and by the lack of support in the processor's instruction set for multi-word failure-atomic writes [13, 41]. Hardware transactional memory (HTM) [28] does not solve the latter problem on current generation multiprocessors because flushing instructions abort transactions before they can be committed.

Rising eagerly to the new challenge, researchers have devised a variety of techniques and idioms for both implementing persistent data structures and specifying their correct behavior. Early practical contributions in this space include a variety of low-level durability mechanisms and APIs for building fault-tolerant applications [10, 12, 29, 40, 44]. This work eventually spurred a wave of theoretical research on concurrent objects for persistent memory, starting with different perspectives on formalizing the behavior of such objects under concurrent access, and continuing with provably correct implementation techniques [4, 5, 6, 7, 8, 11, 20, 23, 30]. One of the fundamental scientific questions arising from this work is how to imbue persistent data structures with *detectability* – the ability to resolve the outcome of operations that were interrupted by a failure [19, 20]. This type of forensic capability is especially important in systems that lack transactions, because the application is directly responsible for deciding the correct redo and undo actions.

Friedman et al.'s practitioner-oriented definition of detectability [20] conveys clearly the high-level intention, and also explains some of the implementation details, in a model of computation where crash failures are system-wide and recovery actions occur during a *recovery phase* that is initiated by the system and precedes resumption of ordinary activity by application threads. Specifically, threads announce their intent to apply an operation by writing special shared variables, and the (single-threaded) recovery code analyzes the state of announced operations carefully after a failure to resolve their status. The outcome of this analysis indicates whether an operation took effect, including the operation's response if available, and is delivered back to the application also through shared variables. Since the application identifies each operation using a unique numerical ID, it is possible to ensure "exactly once" semantics of execution by retrying an operation, if needed, after a failure.

One of the open questions arising from [20] is how to formalize detectability as a correctness property of concurrent objects – a critical foundation for studying the complexity and computability of algorithmic problems related to such objects. We propose that a formal definition of detectability should address several desiderata: (D1) Detectability should be supported through the object's abstract interface, for example through specialized procedures. (D2) The definition should be independent of any particular model of computation or implementation style. (D3) In the context of shared memory, the definition should admit implementations on current generation hardware, which uses coherent but volatile caches (see *shared cache* model in [7]). The nesting-safe recoverable linearizability (NRL) framework of Attiya, Ben-Baruch, and Hendler [5], which we consider the current state of the art in

this area of research, meets these goals only partially. In terms of properties (D1) and (D2), NRL mandates the use of concrete program variables for certain types of interaction with a detectable object, for example to receive helpful *auxiliary state* [6] from the system during recovery, and to record the response returned by a detectable operation. Thus, the object's interface is not entirely abstract, and NRL is somewhat specific to shared memory. Regarding property (D3), the modelling assumptions of the NRL framework do not account for the volatile cache, and impose the stringent requirement that a system recovering from failure can determine "the inner-most recoverable operation that was pending" for each process [5]. We are not aware of any practical implementation of NRL that realizes the latter behaviour.

The main contribution of this paper is a novel formal definition of detectability. Concretely, we introduce the *detectable sequential specification* (DSS), which augments the traditional method of specifying typed objects under sequential access with auxiliary procedures by which an application announces its intent to execute an operation that requires detectability, and resolves the outcome of this operation. The relationship between the DSS and prior work, specifically NRL [5] and its alternative definition (herein referred to as *NRL+*) proposed by Ben-David, Blelloch, Friedman and Wei in [7], can be summarized as follows:

1. The DSS specifies the behavior of detectable objects under sequential access only, and is used in tandem with an off-the-shelf correctness condition for concurrent objects (e.g., [2, 8, 24]). NRL instead uses conventional sequential specifications, and defines correct behavior under concurrent access in a unique way to simplify correct nesting of objects. Overall, our DSS-based approach comes closer to achieving properties (D1) and (D2), but delegates responsibility for nesting objects correctly to the application.
2. Although all three techniques use specialized recovery procedures to resolve the status of operations that may have been interrupted by failures, the semantics of these procedures are shaped by different goals. In DSS and NRL+, the recovery procedure allows a thread to *determine* whether or not an operation it intended to invoke prior to a failure took effect, and if so, what response it returned. In NRL, the purpose of the recovery procedure is to *ensure* that an invoked operation took effect, and determine its response.
3. Detectability in DSS is *declarative* in that the application calls a special *prepare procedure* to indicate which operations on a concurrent object must be detectable. Later on, an application may exercise its right to detectability by calling the recovery procedure, or not. In both NRL and NRL+, all operations are detectable. Furthermore, the recovery procedure in NRL is always invoked for an operation that was interrupted by a failure.
4. DSS-based objects require minimal system assumptions, and can be implemented using standard software tools on a current generation multiprocessor with persistent main memory and a volatile cache, thus achieving property (D3). NRL is based on a simplified *private cache* model [7] and relies fundamentally on auxiliary state [6]. NRL+ is similar to DSS in terms of system assumptions, but is formalized using unbounded sequence numbers to identify different operations, which complicates implementation.[1]

In summary, our DSS-based approach embeds detectability in a sequential specification, is implementable on today's multiprocessors without relying on a persistent call stack or multi-word failure-atomic writes, gives applications the unique ability to request detectability on demand, and leaves correct nesting of objects up to application code.

[1] In practice, sequence numbers are embedded in program variables, which reduces the number of bits available to store other state (e.g., a process ID and a data value in Algorithm 1 of [7]). This is especially problematic on current generation hardware, which supports only 64-bit failure-atomic writes [13, 41].

In addition to formalizing detectability, we also propose a novel detectable queue algorithm called the *DSS queue*. Our algorithm builds on the Michael and Scott queue [36] and its recoverable but non-detectable extension, the *durable queue* of Friedman, Herlihy, Marathe, and Petrank [20]. We compare the DSS queue experimentally (see Section 4) against several alternatives using a 20-core Intel multiprocessor with Optane persistent memory.

2 Detectable Sequential Specifications

Specifying and implementing detectability for shared objects is difficult in most models of computation precisely because modern computing systems store state using a combination of volatile and persistent media. As a result, a recovering process suffers from a mild case of amnesia that hinders forensic analysis – it cannot in general determine exactly what step it was about to perform at the point of failure, or what value was returned by its last operation on a given object. In the specific case of shared memory models with persistent memory, the use of explicit persistence instructions does not cure this ailment because the processor cannot both update a memory location and flush the new value to the "persistence domain" [41] in one atomic step. Following the approach of [5, 7], we treat the amnesia by prescribing the addition of specialized operations to the object's abstract interface.

Our goal in formalizing detectability in this section is to establish rigor, which is necessary for analyzing the complexity and computability of detectable shared objects, while defining the interface to such objects in an implementation-independent manner. We approach this task by extending the traditional approach of composing a sequential specification for an object's type T, which defines the object's correct behaviour under sequential access using data structure semantics (i.e., it prescribes a set of abstract states and state transitions), with a correctness property that describes correct behaviour under concurrent access [26]. The core idea is to augment a given type T with auxiliary operations by which processes declare their intent to execute a detectable operation, and then optionally resolve the status of this operation. This yields a detectable embodiment of T, which we denote as $D\langle T\rangle$. The sequential specification of type $D\langle T\rangle$, called the *detectable sequential specification (DSS)* of type T, is obtained automatically by a transformation of the original sequential specification of T, as explained shortly in Section 2.1. Finally, the correct behavior of a detectable object under concurrent access is formalized by composing $D\langle T\rangle$ with a correctness property suitable for a given model of computation (e.g., [2, 8, 24] in the shared memory context).

We present the formal definition of the DSS without a specific model of computation to emphasize that the DSS is largely model-agnostic. Sequential specifications in general are compatible with message passing, shared memory, and "m&m" [1] models. They accommodate both system-wide and individual process failure assumptions, and are orthogonal to assumptions regarding the volatility or persistence of storage media. The main modelling assumption required at this stage is the existence of a set Π of processes (or threads), where each process p_i has a distinct ID i. We assume implicitly that a process recovers under the *same* ID so that it can refer to its earlier actions that may have been interrupted by a failure.

2.1 Formal Definition

Consider an arbitrary object type T. Formally, T is a sequential specification denoted by a tuple $(S, s_0, OP, R, \delta, \rho)$ where S is a set of abstract states, $s_0 \in S$ is an initial state, OP is a set of operations (e.g., read(), write(1)), R is a set of possible operation responses, $\delta : S \times OP \times \Pi \to S$ is a state transition function indicating the effect of each operation by a process on the abstract state, and $\rho : S \times OP \times \Pi \to R$ is a response function

indicating the operation's correct response.[2] A *detectable sequential specification (DSS)* for type $T = (S, s_0, OP, R, \delta, \rho)$, denoted $D\langle T\rangle$, is a sequential specification $(\bar{S}, \bar{s_0}, \bar{OP}, \bar{R}, \bar{\delta}, \bar{\rho})$ obtained from T by the following transformation:

- Each state $\bar{s} \in \bar{S}$ is a tuple $(s, \mathcal{A}, \mathcal{R})$ where $s \in S$, $\mathcal{A}$ is a mapping $\Pi \to OP \cup \{\bot\}$, and $\mathcal{R}$ is a mapping $\Pi \to R \cup \{\bot\}$, where $\bot \notin OP \cup R$.
- The initial state $\bar{s_o}$ is a tuple $(s_0, \mathcal{A}, \mathcal{R})$ where $\mathcal{A}$ and $\mathcal{R}$ map each element of Π to $\bot$.
- $\bar{OP}$ comprises all the operations of OP, as well as new *auxiliary operations*: *prep-op* and *exec-op* for each $op \in OP$, as well as *resolve*.
- $\bar{R} = R \cup \{(op, r) \mid op \in OP \cup \{\bot\} \wedge r \in R \cup \{\bot\}\}$
- The state transition function $\bar{\delta}$ and response function $\bar{\rho}$ for each operation $\bar{op} \in \bar{OP}$ are presented in Figure 1 using an axiomatic style modeled after [25, 26]. Each axiom indicates a pre-condition (first line), an operation / process ID / response (middle line), and a side-effect (third line). The latter indicates the new state using primed symbols; any component of the abstract state that is not explicitly referenced remains unchanged (e.g., Axiom 1 implies $s' = s$). Three of four axioms are parameterized by an operation $op \in OP$ of T, and yield a distinct operation $\bar{op} \in \bar{OP}$ of $D\langle T\rangle$ for each $op \in OP$.

$$\begin{array}{c} \{true\} \\ prep\text{-}op \ / \ p_i \ / \ \bot \\ \{\mathcal{A}'[p_i] = op \wedge \mathcal{R}'[p_i] = \bot\} \end{array} \quad (1) \qquad \begin{array}{c} \{\mathcal{A}[p_i] = op \wedge \mathcal{R}[p_i] = \bot\} \\ exec\text{-}op \ / \ p_i \ / \ \rho(s, op, p_i) \\ \{s' = \delta(s, op, p_i) \wedge \mathcal{R}'[p_i] = \rho(s, op, p_i)\} \end{array} \quad (2)$$

$$\begin{array}{c} \{true\} \\ resolve \ / \ p_i \ / \ (\mathcal{A}[p_i], \mathcal{R}[p_i]) \\ \{\} \end{array} \quad (3) \qquad \begin{array}{c} \{true\} \\ op \ / \ p_i \ / \ \rho(s, op, p_i) \\ \{s' = \delta(s, op, p_i)\} \end{array} \quad (4)$$

Figure 1 Detectable sequential specification (DSS) of type T, also denoted $D\langle T\rangle$.

In practical terms, the operations described axiomatically in Figure 1 behave as follows. For each $op \in OP$ of type T, *prep-op* (Axiom 1) and *exec-op* (Axiom 2) are used to declare the intention of a process p_i to apply op in a detectable way, and then apply it, respectively. Operation *prep-op* "remembers" op, and defines the context for a future call to *resolve* (Axiom 3), which determines the status of the most recently prepared operation. This *resolve* operation is somewhat similar to the recovery function introduced in NRL [5], but serves a different purpose: *resolve* is used to analyze the status of an operation that may have been left pending by a crash, whereas the recovery function always completes such an operation and returns its response. Finally, operation op (Axiom 4) simply applies the state transition prescribed by op in a non-detectable way with no other side-effects.

The DSS supports detectability in the following sense: after a call to *prep-op*, if *exec-op* took effect then *resolve* returns (op, r) where r is the response of op, otherwise it returns $(op, \bot)$. Since we assume that $\bot \notin R$, the response of *resolve* indicates to a process whether or not its execution of op via *exec-op* took effect. The *prep-op* and *resolve* operations are *total*, meaning that they can be called from any state, and *idempotent*, meaning that they can be called repeatedly (e.g., for example when their executions are interrupted by failures). If *prep-op* was never called for any op, then *resolve* returns $(\bot, \bot)$.

[2] The inclusion of the process ID in the arguments of δ and ρ is necessary since a detectable type encodes special recovery state for each process, and some of the operations query this state directly.

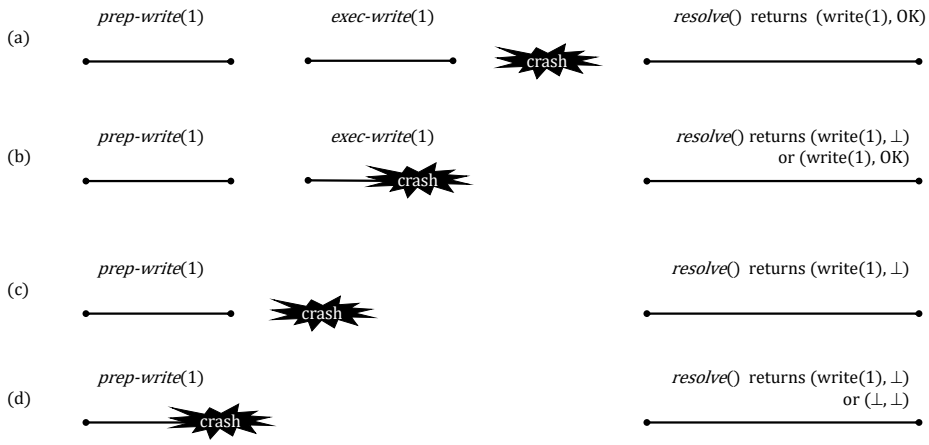


Figure 2 Informal examples of executions over an object that implements the DSS of a read/write register. The initial value of the register is 0, and time increases from left to right. Barbell symbols represent the time intervals of operation executions.

The state transitions of the DSS are illustrated in Figure 2, which presents four possible executions that can be generated by a detectable read/write register object. In example (a), a process p_i prepares a $write(1)$ operation, executes it, crashes, and resolves the operation as completed upon recovering. The mapping $\mathcal{A}$ records $write(1)$ as the prepared operation for the calling process p_i after *prep-write*(1) takes effect, and $\mathcal{R}$ records OK as the response after *exec-write*(1) takes effect. In (b), the crash occurs during *exec-write*(1), and so the $write(1)$ state transition may or may not take effect. Thus, *resolve* returns either $(write(1), \perp)$ or $(write(1), \text{OK})$, and in both cases $\mathcal{A}[p_i]$ records $write(1)$ as the prepared operation. In (c), the crash occurs before the process invokes *exec-write*, and hence *resolve* must return $(write(1), \perp)$. In (d), the crash occurs during *prep-write*(1), and hence *resolve* must return either $(\perp, \perp)$ to indicate that no operation was prepared, or $(write(1), \perp)$.

One special case deserving further attention occurs when a process p_i applies the same operation *op* repeatedly via *prep-op* and *exec-op*, which makes the response of *resolve* ambiguous. This problem can be remedied by augmenting the signature of *op* with an auxiliary argument that is saved in the state component $\mathcal{A}[p_i]$ but ignored in the computation of the state transition $\delta(s, op, p_i)$. For example, if the application is using a monotonic counter to record the number of detectable operations executed on the DSS-based object, then a single bit (i.e., the parity of the counter) is sufficient.

2.2 Discussion

The DSS-based approach marries the practical quality of Friedman, Herlihy, Marathe and Petrank's work [20] with the rigorous tone introduced by Attiya, Ben-Baruch, and Hendler's formalism [5]. More concretely, the DSS defines a purely object-oriented interface by which processes detect that status of past operations. Instead of relying explicitly on system support (as in NRL [5]) or on sequence numbers (as in NRL+ [7]) to identify an operation that may have been interrupted by a failure, the DSS internally records state regarding the last operation of each process via the auxiliary operation *prep-op*. The detection operation *resolve* helps a recovering process identify the approximate position within its program

where it crashed, for example distinguishing between cases (a) and (c) in Figure 2, and can replace the checkpointing mechanisms used in [5, 6] to some extent. A process may call the idempotent *resolve* operation arbitrarily many times to recover an earlier operation's response if its recovery efforts are hampered by additional crash failures. Such flexibility avoids having to save the response in a concrete program variable, which is how NRL deals with the problematic situation where a crash occurs immediately after an operation returns and before its response can be persisted.

For shared objects, the DSS must be combined with a suitable linearizability-like [26] correctness condition, and is compatible with several such conditions. In order from strongest to weakest, these include strict linearizability [2], persistent atomicity [24], and recoverable linearizability [8]. Note that the "program order inversion" anomaly in [8] only applies to operations on distinct objects, and cannot for example reorder an *exec-op* with a *resolve* on the same object. Our approach is inherently incompatible with the model underlying durable linearizability [30] because a crashed process in our framework must recover under the same ID to obtain meaningful output from operation *resolve*.[3] An analogous assumption is present in [5, 7].

A common misconception surrounding our work is that the DSS "does not support nesting." Indeed we do not prescribe a specific manner of recovering operations on nested objects (i.e., there is no "N" in DSS) because the DSS merely defines the abstract states and state transitions for a single object, but DSS-based objects can be nested, especially when they provide strict linearizability [2]. As an example, Section 3 of this paper describes an implementation of a DSS-based detectable queue from read/write register and Compare-And-Swap base objects in a shared memory model with persistent memory and volatile cache. Any base object of type T in this algorithm can be replaced with a strictly linearizable implementation of either T or $D\langle T\rangle$, since $D\langle T\rangle$ provides all the non-detectable operations of T. Thus, $D\langle\text{queue}\rangle$ can be constructed using implementations of $D\langle\text{read/write register}\rangle$ and $D\langle\text{CAS}\rangle$, and this demonstrates application-managed nesting of DSS-based objects. Finally, we point out that NRL [5] does not quite solve the problem of recovering nested objects either because much of the complexity associated with invoking recovery operations in the correct order is encapsulated in a crucial and difficult to implement system assumption.[4]

In terms of computability, the DSS-based approach is conveniently compatible with existing universal constructions of shared objects. For example, a wait-free recoverable implementation of $D\langle T\rangle$ for any conventional type T can be obtained in the shared memory model using Herlihy's universal construction [27], which was shown by Berryhill, Golab, and Tripunitara to yield recoverable linearizability in the presence of crash-recovery failures [8]. We believe that this construction can be extended easily from the "private cache" [7] model of persistent memory, where memory operations are assumed to persist immediately, to the more general model with volatile cache and explicit persistence instructions.

Little is known at this point regarding the complexity of DSS-based recoverable objects, but it is straightforward to show that such objects require linear space. Intuitively, this result holds across a variety of concrete models because the abstract state space of a DSS-based object encodes recovery information (via $\mathcal{A}$ and $\mathcal{R}$) for each process. NRL-like implementations also require linear space in some cases, as proved recently by Ben-Baruch, Hendler, and

[3] Durable linearizability [30] permits reuse of process IDs once the pending operations of crashed threads have "completed." We interpret this restriction to mean that both the pending operation and any detectability actions applied in connection with the operation have concluded.

[4] Section 2 of [5] states that "the system may eventually resurrect process p by invoking the recovery function of the inner-most recoverable operation that was pending when p failed."

Rusanovsky [6] for obstruction-free Compare-And-Swap. On the other hand, DSS-based objects behave very differently from NRL-like objects with respect to "auxiliary state," which some NRL-like objects receive from the application or from the system via specialized operation arguments (e.g., sequence numbers) or special shared variables. Whereas [6] proves that such external state must be provided to any NRL-like implementation of a "doubly-perturbing" type (which includes the FIFO queue), even with very weak non-blocking progress guarantees, one variation of the lock-free DSS queue algorithm presented in Section 3 requires no such state at all. Intuitively, this contrast follows from the fundamentally different semantics of recovery in DSS-based and NRL-like objects, where the former recover the most recently prepared (via a call to *prep-op*) operation and the latter recover the most recently invoked operation. Identifying the most recently invoked operation is inherently more difficult, and auxiliary state in NRL compensates for this difficulty.

3 A lock-free strictly linearizable detectable queue

As a proof of concept, we present in this section a DSS-based detectable queue implementation, called *DSS queue*, for the asynchronous shared memory model with persistent memory, volatile cache, and system-wide crash failures. The algorithm is based on Michael and Scott's venerable lock-free queue (called the *MS queue*) [36], as well as its recoverable variant for persistent memory (called the *durable queue*) published recently by Friedman, Herlihy, Marathe, and Petrank [20]. The original MS Queue uses a singly-linked list of nodes, referenced by *head* and *tail* pointers, to implement a FIFO queue, and is used heavily in practice (e.g., in the Java package java.util.concurrent). The durable queue adds the necessary flush instructions to cope with the volatile cache, and also augments the queue node structure by adding a $deqThreadID$ field, initially -1, to identify the thread who dequeues the value stored in the node. A queue node for which $deqThreadID \neq -1$ is called a *marked* node. The implementation assumes that a centralized recovery procedure is executed after each crash to complete pending operations and report their status to application threads using an array of shared variables called *returnedValues*.

We transform the n-thread durable queue into a DSS-based data structure by removing the *returnedValues* array, adding an array $X[1..n]$ to represent the state components $\mathcal{A}$ and $\mathcal{R}$ of $D\langle queue\rangle$, and adding the auxiliary operations described in Section 2: *prep-op* and *exec-op* for each $op \in \{enqueue, dequeue\}$, as well as *resolve*. In the initial state, the *head* and *tail* pointers refer to the same sentinel node that is not marked, all entries of X are NULL, and every queue node has $next =$ NULL and $deqThreadID = -1$. The access procedures for the operations of $D\langle queue\rangle$ are presented using a syntax similar to C++, where & and $\rightarrow$ denote the usual reference and dereference operators. The keyword TID represents the identifier i of the calling thread t_i, where $1 \leq i \leq n$. Logical and bitwise AND, OR, and XOR are denoted exactly as in C++.

3.1 Enqueue and supporting operations

Enqueuing operations, presented in Figure 3, generally follow the code of durable queue [20], with the addition of operations to update the array X, which stores a pointer to a queue node. We borrow the most significant bits of this pointer to record tags that indicate whether or not the detectable *enqueue* operation was prepared and then took effect.[5]

[5] Modern x86-64 processors implement 48 address bits, which leaves 16 bits available for special tags.

Procedure *prep-enqueue*(*val*: value to be enqueued).

```
1  Node* node := new Node(val) // init:  next = NULL, deqThreadID = −1
2  FLUSH (node)
3  X[TID] := node | ENQ_PREP_TAG
4  FLUSH (&X[TID])
```

Procedure *exec-enqueue*().

```
5  Node* node := X[TID]
6  while true do
7      Node* last := tail
8      Node* next := last→next
9      if last == tail then
10         if next == NULL then // at tail
11             if CompareAndSwap(&last→next, NULL, node) then
12                 FLUSH (&last→next)
13                 X[TID] := X[TID] | ENQ_COMPL_TAG
14                 FLUSH (&X[TID])
15                 CompareAndSwap (&tail, last, node)
16                 return
17         else // help another enqueuing thread
18             FLUSH (&last→next)
19             CAS(&tail, last, next)
```

Procedure *resolve*.

```
20 if X[TID] & ENQ_PREP_TAG then
21     (arg, ret) := resolve-enqueue()
22     return (⟨enqueue, arg⟩, ret)
23 else if X[TID] & DEQ_PREP_TAG then
24     ret := resolve-dequeue()
25     return (⟨dequeue, NO_ARG⟩, ret)
26 else // no operation was prepared
27     return (⊥, ⊥)
```

Procedure *resolve-enqueue*().

```
28 if X[TID] & ENQ_COMPL_TAG then
       // enqueue was prepared and took effect
29     return ((X[TID] ^ ENQ_PREP_TAG ^ ENQ_COMPL_TAG)→value, OK)
30 else
       // enqueue was prepared and did not take effect
31     return ((X[TID] ^ ENQ_PREP_TAG)→value, ⊥)
```

Figure 3 The *prep-enqueue*, *exec-enqueue*, *resolve*, and *resolve-enqueue* operations of DSS queue.

The *prep-enqueue* operation creates a queue node that holds the value to be enqueued, and saves a pointer to the node along with a tag in X. The *exec-enqueue* operation follows closely the durable queue [20] algorithm by locating the tail of the linked list and swinging the next pointer of the tail node at line 11. The code at lines 13–14 updates the node pointer saved previously in X by setting the ENQ_COMPL_TAG and flushing the updated value, and is needed for detectability. The tail pointer is then updated at line 15. The code at lines 18–19 is a helping mechanism required for lock-freedom. The non-detectable *enqueue* operation is equivalent to calling *prep-enqueue* followed by *exec-enqueue*, except that any lines accessing X (3–4 and 13–14) are omitted.

The detection function *resolve* checks whether an *enqueue* was prepared at line 20, then calls a helper routine *resolve-enqueue* that considers two cases based on the presence of the ENQ_COMPL_TAG in X, which is added by *exec-enqueue* at lines 13–14, and also by the recovery procedure described later on. If the *enqueue* operation was prepared with value *val* and took effect, (*enqueue*(*val*), OK) is returned via lines 29 and 22. If the *enqueue* operation was prepared with value *val* and did not take effect, (*enqueue*(*val*), $\perp$) is returned via lines 31 and 22. Finally, if the *enqueue* operation was never prepared then either ($\perp$, $\perp$) is returned at line 27 or a *dequeue* is resolved at lines 23–25.

3.2 Dequeue and supporting operations

The implementation of dequeuing operations is presented in Figure 4. Similarly to enqueuing operations, the code generally follows [20] with the addition of operations to update the array X, which stores a tagged pointer to a queue node. Two of the most significant bits of this pointer are repurposed to record a DEQ_PREP_TAG, which indicates whether or not the detectable *dequeue* was prepared, and an EMPTY_TAG, which indicates that a *dequeue* took effect on an empty queue.

The *prep-dequeue* operation initializes X using NULL tagged with DEQ_PREP_TAG. The *exec-dequeue* operation proceeds in two cases, the first of which follows closely the durable queue [20] algorithm. If the queue is found to be empty, meaning that the head and tail point to the same sentinel node where the *next* pointer is NULL at lines 38–40, then EMPTY_TAG is added to X at lines 41–42 and a special EMPTY value is returned at line 43. Otherwise the queue is not empty, and the correct return value is stored in the successor of the sentinel node at the head of the linked list. As in the durable queue, a thread tries to claim this value by using `CompareAndSwap` to write its ID into the *next* node at line 49, and if it succeeds, the updated $deqThreadID$ field is then flushed at line 50. Next, the head pointer is advanced at line 51. On the other hand, if a competing thread causes the `CompareAndSwap` to fail, a helping mechanism is executed at lines 54–55 to persist the updated *deqThreadID* variable, and the code repeats. The main differences from the durable queue algorithm are two-fold. First, instead of returning the dequeued value using a dynamically allocated object, DSS queue returns it directly at line 52. Second, for detectability, the pointer to the head node is written to X with DEQ_PREP_TAG and flushed at lines 47–48. Overall, DSS queue performs one less flush in the helping mechanism due to the simplified method of returning the dequeued value, one more flush prior to each `CompareAndSwap` at line 48 due to detectability, and one less memory allocation.

The non-detectable *dequeue* operation is equivalent to calling *prep-dequeue* followed by *exec-dequeue*, with two differences. First, any lines accessing X (32–33, 41–42, and 47–48) are omitted. Second, to avoid confusion between a partially executed *exec-dequeue* and a *dequeue* during subsequent detection, the analog of line 49 in *dequeue* marks the $deqThreadID$ field in a slightly different way: instead of using the caller's TID directly, it combines the TID with another special tag. Details are omitted due to lack of space.

Procedure *prep-dequeue*().

```
32 X[TID] := DEQ_PREP_TAG
33 FLUSH (&X[TID])
```

Procedure *exec-dequeue*().

```
34 while true do
35     Node* first := head
36     Node* last := tail
37     Node* next := first→next
38     if first == head then
39         if first == last then // empty queue
40             if next == NULL then // nothing new appended at tail
41                 X[TID] := X[TID] | EMPTY_TAG
42                 FLUSH (&X[TID])
43                 return EMPTY
44             FLUSH (&last→next)
45             CompareAndSwap (&tail, last, next)
46         else // non-empty queue
47             X[TID] = first | DEQ_PREP_TAG // save predecessor of node to
                be dequeued
48             FLUSH (&X[TID])
49             if CompareAndSwap (&next→deqThreadID, −1, TID) then
50                 FLUSH (&next→deqThreadID)
51                 CompareAndSwap (&head, first, next)
52                 return next→value
53             else if head == first then // help another dequeuing thread
54                 FLUSH (&next→deqThreadID)
55                 CompareAndSwap (&head, first, next)
```

Procedure *resolve-dequeue*().

```
56 if X[TID] == DEQ_PREP_TAG then
       // dequeue was prepared but did not take effect
57     return ⊥
58 else if X[TID] == (DEQ_PREP_TAG | EMPTY_TAG) then
       // empty queue
59     return EMPTY
60 else if (X[TID] ^ DEQ_PREP_TAG)→next→deqThreadID == TID then
       // non-empty queue
61     return (X[TID] ^ DEQ_PREP_TAG)→next→value
62 else
       // X holds a node pointer, crashed before completing dequeue
63     return ⊥
```

Figure 4 The *prep-dequeue*, *exec-dequeue*, and *resolve-dequeue* operations of DSS queue.

The detection function *resolve* checks whether a *dequeue* was prepared at line 23, then calls a helper routine *resolve-dequeue* that determines the correct response by considering four cases based in part on the presence of a node pointer, DEQ_PREP_TAG and EMPTY_TAG in X. The latter is updated by *exec-dequeue* at lines 41–42, and 47–48, and also by the recovery procedure described later on. If X holds a NULL pointer with PREPARED_TAG only (line 56), this indicates that a *dequeue* was prepared but did not take effect, and so $\perp$ is returned at line 57. A NULL pointer with DEQ_PREP_TAG and EMPTY_TAG (line 58) indicates that a *dequeue* was prepared and took effect on an empty queue, and so EMPTY is returned at line 59. Otherwise, X stores a non-NULL node pointer with DEQ_PREP_TAG. If the $deqThreadID$ field of the successor node matches the caller's TID (line 60), this indicates that a *dequeue* was prepared and took effect on a non-empty queue, and so *val* is returned at line 61 where *val* is the dequeued value obtained from the successor node. The final case (line 62) indicates a crash between line 47 and a successful `CompareAndSwap` at line 49, and so $\perp$ is returned at line 63 since no value was dequeued. In this case the successor node could have been marked by the same thread but in a non-detectable *dequeue*, by a different thread, or by no one.

3.3 Recovery

Discussion of recovery following a system crash is deferred to Appendix A due to lack of space. In summary, a single-threaded recovery procedure in the style of [20] adjusts $X[t_i]$ for each thread t_i after scanning the linked list of queue nodes to identify pending operations. The algorithm can be adapted straightforwardly to allow threads to recover independently, without relying on a centralized recovery phase, and this transformation eliminates the last trace of auxiliary state [6] from the DSS queue.

3.4 Analysis

The correctness properties of the DSS queue algorithm are stated formally in Theorem 1. The analysis, including a detailed description of the model, is omitted due to lack of space.

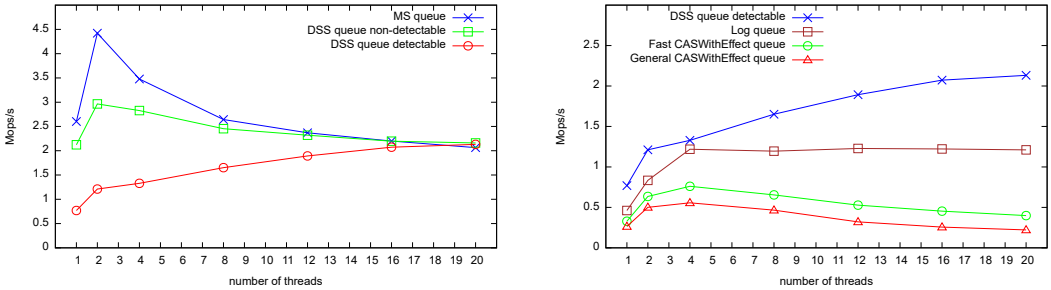
▶ **Theorem 1.** *The DSS queue is lock-free and strictly linearizable with respect to* $D\langle queue\rangle$.

4 Evaluation

In this section, we present an empirical evaluation of the DSS queue algorithm from Section 3. The experiments are conduced using a 20-core 2.1GHz Intel Xeon processor equipped with genuine Intel Optane Data Center Persistent Memory Modules (DCPMM) provisioned in App Direct mode. Turbo boost is disabled to reduce variation in running times. The software environment includes Ubuntu Linux 20.04, g++ 9.3.0, and version 1.8 of the Intel Persistent Memory Development Kit (PMDK) library [42]. The code is compiled in debug mode with optimization level O0. The Optane memory is accessed using standard C++ atomic operations configured with sequentially consistent ordering. We flush data from the volatile cache to the Optane device using the PMDK *pmem_persist* function, which internally uses Intel's cache line write back (CLWB) instruction and also includes a store fence. For memory management, each thread pre-allocates a fixed size pool of queue nodes at initialization, and dequeued nodes are returned to the free pool using epoch-based reclamation (EBR) [17]. The EBR code is borrowed from the open-source implementation

(`https://github.com/microsoft/pmwcas`) of Wang et al.'s Persistent Multi-word Compare-And-Swap (PMwCAS) [45]. The recovery procedure described in Appendix A is extended straightforwardly to prevent memory leaks, such as due to a crash in *prep-enqueue*.

Our experiments, presented in Figure 5, evaluate the scalability of different queue implementations in failure-free runs with up to 20 threads. In each experiment, the queue is initialized with 16 queue nodes, and each thread executes alternating pairs of *enqueue* and *dequeue* operations for 30 seconds. Each point plotted in the graphs is the mean throughput value (millions of operations per second) computed over a sample of ten runs, and in all cases the sample standard deviation is less than 2% of the sample mean.



(a) Different levels of detectability and persistence **(b)** Different detectable queue implementations

Figure 5 Scalability experiments.

Figure 5a investigates the cost of detectability, which threads can request on demand in our flexible DSS-based approach. The three points of comparison in Figure 5a are the following queue implementations:

- *DSS queue detectable*: the DSS queue algorithm as described in Section 3, where *enqueue* and *dequeue* are applied in a detectable manner via calls to *prep-enqueue*/*exec-enqueue* and *prep-dequeue*/*exec-dequeue*. Procedures *resolve* and *resolve* are not invoked since we consider only failure-free runs.
- *DSS queue non-detectable*: the DSS queue algorithm where *enqueue* and *dequeue* are applied in a non-detectable manner.
- *MS queue*: an implementation of the classic MS queue [36] obtained from the non-detectable DSS queue by removing flushes in *enqueue* and *dequeue*.

The scalability plot shows that non-detectable implementation is measurably faster than detectable, offering nearly 3× higher throughput at low levels of parallelism. This is primarily due to the cost of the memory operations at lines 3-4, 13–14, 32–33, and 47–48 in the detectable execution path. The additional latency due to these operations has a smaller effect on throughput at higher levels of parallelism, and performance is ultimately limited by the synchronization bottleneck at the head and tail of the queue. The MS queue has the best performance overall and beats DSS non-detectable by up to 1.5× at 2 threads, and similarly suffers from contention at higher levels of parallelism. Indeed, all three scalability curves nearly converge at 20 threads.

Figure 5b compares the DSS queue with several other detectable queue algorithms:

- *DSS queue*: our DSS queue algorithm, identical to "DSS queue detectable" in Figure 5a.
- *General CASWithEffect queue*: a simple queue algorithm where the linked list and detectability state (analogous to X in DSS queue) are manipulated using the Persistent Multi-word Compare-And-Swap (PMwCAS) algorithm of Wang et al. [45].

- *Fast CASWithEffect queue*: similar to General CASWithEffect queue, except that PMwCAS is optimized for multi-word operations that access a combination of shared variables (queue head, tail, and next pointers) and private variables (detectability state).
- *Log queue*: our own implementation of Friedman et al.'s detectable *log queue* algorithm [20], which uses per-thread logs. Operation arguments and return values are stored directly in the logs, and are accessed by other threads via helping mechanisms.

The results show that DSS queue provides superior scalability to the two algorithms based on powerful PMwCAS, which simplifies the implementation greatly but becomes a performance bottleneck as contention rises. Furthermore, Fast CASWithEffect queue outperforms General CASWithEffect queue by up to 1.5×, showing the benefit of optimizing multi-word operations that access private variables atomically with shared variables. Our implementation of the log queue also outperforms both Fast and General CASWithEffect queue, but trails behind DSS queue by up to 1.7×. One reason why DSS queue is faster is that it stores detectability state (array X) using variables that are statically allocated and effectively private as they are shared only with the single-threaded recovery procedure. In comparison, the log queue dynamically allocates log objects in addition to queue nodes, and these objects are shared during concurrent execution of *dequeue*. We plan in future work to compare DSS queue against a more heavily optimized implementation of the log queue, which may narrow the observed performance gap.

5 Related Work

Research on software techniques for persistent memory, also known as non-volatile RAM (NVRAM), erupted roughly a decade ago as solid-state secondary storage devices began to flood the market and hardware vendors began to plan for production of high-capacity byte-addressable persistent main memory modules. Several software frameworks were proposed around that time to provide low-level access to persistent state through file systems, persistent heaps, and lightweight transactions based on redo and undo logging [10, 12, 44]. In parallel with these efforts, early designs of durable data structures for persistent memory began to emerge (e.g., [43]). Subsequent practical work introduced a variety of performance optimizations based on judicious use of memory fences, persistence instructions, and hybrid memory hierarchies that combine both volatile and non-volatile main memories [9, 14, 15, 29, 31, 35, 37, 40].

More recent publications describe the design of specific in-memory data structures and synchronization objects for persistent memory, and give up the convenience of transaction-based implementations for the sake of better performance through specialized concurrency control and persistence mechanisms. Much of this work focuses on practical scalable index structures for databases, such as hash maps and search trees, some of which exploit hybrid memory hierarchies [3, 16, 34, 38, 39, 45, 46]. Due to the inherent difficulty of designing and verifying the correctness such structures, researchers have also sought efficient (i.e., non-transactional) transformations of conventional in-memory data structures to durable structures for persistent memory [18, 33], which are applicable only when the original structure follows certain common design patterns, and hence not universal. In terms of synchronization objects, several non-blocking implementations of queues, read/write registers, Compare-And-Swap, and consensus objects are known [5, 6, 7, 8, 20, 21]. Wait-free [8] and lock-free [11] universal constructions have also been proposed for such objects.

Several attempts have been made to formalize the correctness properties of concurrent objects for persistent memory through variations on Herlihy and Wing's widely-adopted linearizability property [26] and Lamport's atomic register [32]. In one line of work, which

assumes that processes or threads recover after a crash and continue execution under the same identifier, the problem was already solved earlier in the context of message passing systems by the strict linearizability property of Aguilera and Frølund [2] and persistent atomicity property of Guerraoui and Levy [24]. Berryhill, Tripunitara, and Golab [8] later relaxed these conditions by proposing recoverable linearizability, and formalized recoverable objects in the shared memory model. Izraelevitz, Mendes and Scott proposed an alternative model where thread identifiers are not reused (at least not immediately) after a crash, which makes persistent atomicity, recoverable linearizability, and ordinary linearizability indistinguishable, and defined durable linearizability as a synonym for this merged correctness condition [30]. They also introduce an innovative property called buffered durable linearizability that allows applications to trade durability guarantees for better performance due to reduced use of costly persistence instructions.

The question of whether process or thread identifiers are reused across crashes has been debated in the community. On one hand, it is true that from the point of view of the operating system, a thread resurrected after a crash is distinct from any thread that ran prior to the failure. On the other hand, applications tend to number their threads internally using simple schemes (e.g., 1 to n), and this establishes a secondary identity that survives crash failures. Such internal identifiers are used extensively in the implementations of synchronization objects, for example to index arrays. Curiously, we observe this pattern even in some durably linearizable implementations [6, 20].

Detectability was introduced informally by Friedman, Herlihy, Marathe and Petrank [19, 20], and formalized by Attiya, Ben-Baruch, and Hendler as nesting-safe recoverable linearizability (NRL) [5]. NRL is stronger than our DSS (Section 2) in the sense that it specifies precisely how to recover nested implementations, whereas DSS embodies detectability only. DSS-based objects can be nested in principle, and strictly linearizable implementations of such objects can be used in place of atomic base objects if preservation of probabilistic properties [22] is not a concern. Both NRL and the DSS augment the interface of concurrent objects with special recovery procedures, which allows a recovering thread to resolve its own pending operation after a failure, and in that sense implies the reuse of thread identifiers in the spirit of [2, 8, 24]. In contrast, pending operations are resolved by a separate recovery process in [20], which makes their approach to detectability compatible with the durable linearizability.

In terms of complexity, prior work has established bounds on the number of persistent fence instructions required by deterministic lock-free durably linearizable implementations [11], and the space required by NRL-like implementations [6]. It is also known that NRL-like implementations require some form of auxiliary state from the system [6]. Our DSS queue algorithm is exempt from the latter impossibility result, but one of its variations relies on auxiliary state in the sense that the system must initiate the centralized recovery phase (as in [20]), which updates some of the queue's base objects. Another variation of the DSS queue avoids the centralized recovery phase and has no dependence at all on auxiliary state.

6 Conclusion

In this paper, we introduced the detectable sequential specification (DSS) as a formal definition of detectability for recoverable objects. As a proof of concept, we presented a DSS-based recoverable queue algorithm, and evaluated its performance on a multiprocessor with Intel Optane persistent memory. We hope that the DSS opens a new avenue for both rigorous analysis and practical implementation of recoverable concurrent objects.

References

1 Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, 2018.

2 Marcos K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

3 Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, 2018.

4 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover: Recoverable lock-free data structures. *CoRR*, abs/1905.13600, 2019. `arXiv:1905.13600`.

5 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018.

6 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, 2020.

7 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

8 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

9 Trevor Brown and Hillel Avni. Phytm: Persistent hybrid transactional memory. *Proc. VLDB Endow.*, 10(4):409–420, 2016.

10 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

11 Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 259–269, 2018.

12 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

13 Intel Corporation. Persistent memory faq, 2020. [last accessed 2/17/2021]. URL: `https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html`.

14 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, 2018.

15 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proc. of the 15th EuroSys Conference*, pages 5:1–5:15, 2020.

16 Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)*, pages 373–386, 2018.

17 Keir Fraser. *Practical Lock-Freedom.* PhD thesis, University of Cambridge, 2004. Computer Laboratory.

18 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: in NVRAM data structures, the destination is more important than the journey. In *Proc. of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 377–392, 2020.

19 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 50:1–50:4, 2017.

20 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, 2018.

21 Wojciech Golab. The recoverable consensus hierarchy. In *Proc. of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 281–291, 2020.

22 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *In Proc. of the 43rd ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011.

23 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

24 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

25 John V. Guttag, James J. Horning, and Jeannette M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, 1985.

26 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

28 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

29 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of the 21s International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, 2016.

30 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*, pages 313–327, 2016.

31 Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGPLAN Notices*, 51(4):385–398, 2016.

32 L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

33 Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 462–477, 2019.

34 Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, 2019.

35 Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

36 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

37 Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proc. of the 18th International Conference on Extending Database Technology (EDBT)*, pages 689–694, 2015.

38 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, pages 37:1–37:16, 2017.

39 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proc. of the 2016 International Conference on Management of Data (SIGMOD)*, pages 371–386. ACM, 2016.

40 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015.

41 Andy Rudoff. Persistent memory programming. *login Usenix Mag.*, 42(2), 2017.

42 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit, 2020. [last accessed 2/11/2021]. URL: `https://pmem.io/pmdk/`.

43 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

44 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.

45 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *Proc. of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

46 Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Trans. Computers*, 65(7):2169–2183, 2016.

A Recovery of DSS queue

Following [20], we assume that a recovery phase is executed after each crash, before the application threads are revived. This is an implementation detail, and is not required in general for correctness of a DSS-based implementation. In practical terms, the recovery phase is the execution of a recovery procedure by the main thread of the application. As in [20], the recovery code scans the linked list of queue nodes from the head pointer and checks the value of the $deqThreadID$ field. For any marked node, the original algorithm ensures completion of the *dequeue* operation by recording the dequeued value in a special array of shared variables, but this is not required in our algorithm. This is because we do not use shared variables to return dequeued values, and the state required for detectability is already established at lines 47–48 of *exec-dequeue* prior to calling `CompareAndSwap` at line 49. Next, the recovery code advances the *head* pointer, if needed, to point to the last marked node reachable from the current head pointer, and sets the tail pointer to the last node in the linked list. Finally, we extend the recovery algorithm to detect any pending *enqueue* operations that took effect in the sense of persisting an updated *next* pointer at line 12, but did not record this state change for detectability (in our case at lines 13–14). During the traversal of the linked list, if a node is encountered that is referenced by some element of X and where the pointer is tagged with ENQ_PREP_TAG, the code tags the pointer with ENQ_COMPL_TAG as well; this takes care of nodes that were enqueued and had not yet been dequeueud at the point of failure. For nodes that were first enqeued and then dequeued, the code checks the pointers in X that are tagged with ENQ_PREP_TAG, and sets ENQ_COMPL_TAG for any marked node. The complete algorithm is presented in Figure 6.

Procedure *recovery*().

```
64 AllNodes := set of queue nodes reachable from head
65 tail := last queue node reachable from head
66 FLUSH (&tail)
67 oldHead := head
68 head := last marked node reachable from oldHead
69 FLUSH (&head)
70 for i ∈ 1..n do
71     if X[i] is a pointer to a node d ∈ AllNodes and is tagged with ENQ_PREP_TAG
        but not ENQ_COMPL_TAG then
           // enqueued and still in the linked list
72         X[i] := X[i] | ENQ_COMPL_TAG
73         FLUSH (&X[i])
74     else if X[i] is a pointer to a node d ∉ AllNodes and is tagged with
        ENQ_PREP_TAG but not ENQ_COMPL_TAG, and d→deqThreadID ≠ −1
        then
           // enqueued and no longer in the linked list, already marked
75         X[i] := X[i] | ENQ_COMPL_TAG
76         FLUSH (&X[i])
```

Figure 6 Recovery procedure of DSS queue.