A Closer Look at Detectable Objects for Persistent Memory

Mohammad Moridi
University of Waterloo
Canada
moridi@uwaterloo.ca

Erica Wang
University of Waterloo
Canada
e52wang@uwaterloo.ca

Amelia Cui
McGill University
Canada
wen.cui3@mail.mcgill.ca

Wojciech Golab
University of Waterloo
Canada
wgolab@uwaterloo.ca

ABSTRACT

Research on multi-core algorithms is adapting rapidly to the new opportunities and challenges posed by persistent memory. One of these challenges is the fundamental problem of formalizing the behaviour of concurrent objects in the presence of crash failures, and giving precise meaning to the semantics of recovery from such failures. Li and Golab (DISC'21) recently proposed a sequential specification for such recoverable objects, called the *detectable sequential specification* or DSS. Building on their work, we explore examples of how DSS-based objects can be used by a sample application, and examine more closely the division of labour between the application's environment, the application code, and the recoverable object used by the application. We also propose an alternative formal definition of correctness, called the *unified detectable sequential specification* (UDSS), that simplifies both the object's interface and the application code. Using a black box transformation, we show how a UDSS-based object can be implemented from one that conforms to Li and Golab's specification. Finally, we present experiments conducted using Intel Optane persistent memory to quantify the performance overhead of our transformation.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Concurrent algorithms**.

KEYWORDS

persistent memory; concurrency; fault tolerance; data structures; correctness

ACM Reference Format:
Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. 2022. A Closer Look at Detectable Objects for Persistent Memory. In *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED '22), July 25, 2022, Salerno, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3524053.3542749

ApPLIED '22, July 25, 2022, Salerno, Italy

ACM ISBN 978-1-4503-9280-8/22/07…$15.00
https://doi.org/10.1145/3524053.3542749

1 INTRODUCTION

Research on multi-core algorithms is adapting rapidly to the new opportunities and challenges posed by persistent memory, which offers the combined benefits of main memory and secondary storage. One of these challenges is the fundamental problem of formalizing the behaviour of concurrent objects in the presence of crash failures, and giving precise meaning to the semantics of recovery from such failures. This paper continues the study of *detectable objects*, introduced by Friedman, Herlihy, Marathe, and Petrank [16], which provide special interfaces that make it possible for an application to determine the effect of an operation that was interrupted by a failure. The forensic information obtained through such interfaces is then used to decide the correct recovery actions for a given application, for example re-invoking (or not re-invoking) the interrupted operation to ensure that it takes effect exactly once.

Li and Golab [24] recently proposed a sequential specification for detectable objects, called the *detectable sequential specification* or DSS, which conceptually follows the execution pattern described in [16]: a thread first announces its intent to apply a detectable operation (i.e., prepares it), and then executes it. The DSS explicitly separates these two phases of execution, meaning that two distinct operations and state transitions must be applied on a detectable object to apply one operation on the underlying shared object type. For example, enqueuing an element into a DSS-based queue entails calling *prep-enqueue* followed by *exec-enqueue*. A resolution procedure called *resolve* is then used to determine whether the enqueue operation was actually executed, only prepared, or neither. Using a FIFO queue as an example, Li and Golab partly bridged the gap between theory and practice by showing that the DSS is simple enough to implement on the current generation hardware [23, 24], and yet powerful enough to enable correct recovery in a simple producer-consumer synchronization application [25].

Despite initial success in solving a non-trivial and practically-motivated synchronization problem, relatively little is known about the strengths and weaknesses of the DSS as a specification for the building blocks of crash-tolerant software systems. Several specific scientific questions come to mind: (1) Can the DSS be simplified by combining the prepare and execute phases into a single operation, making it more similar to other specifications of correctness [3, 5]? (2) Is the DSS sufficiently powerful to enable recovery in more

complex applications than the one described by Li and Golab [25]? (3) What are the practical performance implications of the design choices underlying the answers to the first two questions?

This paper makes the following contributions with respect to the open research questions. To answer Question 1, we first consider variations on the producer-consumer problem discussed in [25], and exhibit a scenario where a solution is impossible unless application-managed state is saved to persistent memory between the prepare and execute phases of an operation on a DSS-based object; this proves that the prepare and execute phases of the DSS cannot always be combined. Next, we show that this impossibility result breaks down when a more elaborate form of the DSS described in Section 2.1 of [24], which we call the *augmented detectable sequential specification* (ADSS), is applied instead. In other words, an application that uses an ADSS-based object in our scenario no longer needs to record additional recovery state outside the detectable object, and this observation yields a partial answer to Question 2. We then return to Question 1 and introduce an alternative definition of correctness called the *unified detectable sequential specification* (UDSS), which collapses the prepare and execute phases into a single operation that appears to take effect instantaneously at one point, which simplifies the application code even further. We also propose a black box technique that transforms an ADSS-based object into a UDSS-based object of the same underlying type. For Question 3, we implement both DSS-based and UDSS-based versions of Li and Golab's recoverable lock-free queue (code available online [27]), and demonstrate that they offer comparable scalability by conducting experiments on a multiprocessor equipped with Intel Optane persistent memory.

2 RELATED WORK

Persistent memory has embedded itself firmly into the scientific literature. In particular, software techniques for persistent memory have been intensely researched for over a decade, and there is a substantial body of practical work in this area that deals with both concurrency control and recovery [7, 8, 10–12, 20, 22, 26, 28, 29, 33, 34]. These activities inspired a number of recent theoretical contributions that address issues of complexity, computability, and correctness [2–6, 9, 13, 16–18, 21, 24, 32]. The question of correctness is most closely related to our paper, particularly in the context of *detectability*, which was introduced recently by Friedman, Herlihy, Marathe, and Petrank [15, 16]. Informally speaking, detectability is the notion that an application can interact with a shared object to determine the outcome of an operation whose execution was (or may have been) interrupted by a crash failure, which is difficult in the absence of transactions.

Following the publication of Friedman et al.'s detectable queue [15], several research endeavours have sought formal definitions of the interface by which applications can resolve the outcome of interrupted operations. Attiya, Ben-Baruch, and Hendler [3] defined *nesting-safe recoverable linearizability* (NRL), which is a framework that not only defines how interrupted operations are resolved, but also explains the manner in which nested objects are recovered. In an attempt to create a practical implementation of NRL, Ben-David, Blelloch, Friedman, and Wei [5] redefined the semantics of recovery to weaken the required system assumptions, and reduced NRL to a simpler sequential specification that relies on monotonic sequence numbers to identify a particular operation. Such a sequential specification is used in conjunction with an off-the-shelf correctness property, such as Aguilera and Frølund's strict linearizability [1], to define an object's correct behaviour under concurrent access. In contrast, NRL introduces its own novel variation of linearizability [19], and cannot be expressed as a sequential specification in an axiomatic style that defines states and state transitions without referring to failures. Li and Golab [24] generalized Ben-David et al.'s approach, which was formalized in [5] specifically for a recoverable Compare-And-Swap object, by introducing the *detectable sequential specification* (DSS). The DSS can be instantiated for an arbitrary object type, and is also more implementation-independent in the sense that it does not prescribe the use of monotonic sequence numbers. Following the terminology of [24], we will refer to the revised NRL definition from [5] as NRL+.

An interesting scientific question surrounding formal notions of detectability concerns assumptions regarding the detectable object's environment. Ben-Baruch, Hendler, and Rusanovsky [4] proved that objects conforming to an NRL-like specification, which assumes that execution is restarted following a failure at the most deeply nested recoverable operation, fundamentally require not only linear space but also helpful input from the environment in the form of *auxiliary state*. Such state can be provided to the object either through specialized operation arguments, or through shared memory operations that bypass the object's abstract interface and directly access the object's internal state. Li and Golab [24] showed that a DSS-based implementation of the FIFO queue can evade this requirement, which intuitively follows from their weaker system assumptions. Specifically, the recovery of detectable objects in their work is entirely application-managed, and the application is always restarted from the beginning rather than from the point of failure.[1] Thus, an application that uses DSS-based objects labours more heavily during recovery to determine the correct course of action, as compared to an NRL-based application, but also carries a lighter load in the sense that the execution does not need to restart at (or even near) the point of failure as long as user-defined requirements are ultimately met.

3 MODELLING ASSUMPTIONS AND THE PRODUCER-CONSUMER PROBLEM

We consider a collection of asynchronous threads (sometimes also called processes) that communicate through persistent shared memory. The memory hierarchy includes a volatile cache layer, and special persistence instructions are used to force updates from the cache to the persistent memory. Threads may fail by crashing, which restarts them from the beginning of their program. Crash failures are system-wide, meaning that all threads fail simultaneously. Threads may recover independently, or by the execution of a centralized recovery procedure by the system during recovery. Most of our discussion applies to both styles of recovery.

The algorithmic problem considered in this paper is the implementation of a detectable concurrent object, and its use in an application subject to some high-level user-defined requirements,

[1] In practical terms, execution resumes from the program's entry point, such as the main function in a C/C++ program.

such as "at least once" semantics of execution. We consider strict linearizability [1] the gold standard for the correctness of such an object, and treat detectability as a feature of the object's sequential specification, as prescribed by [5, 24]. Informally speaking, strict linearizability states that operations on a shared object appear to take effect instantaneously at some point between their invocation and response, and that operations interrupted by a crash either take effect before the crash event or not at all.

Examples of how detectable objects may be used in practice by applications are scarce in the research literature on this topic. Li and Golab [25] introduced a simple producer-consumer synchronization problem that demonstrates the use of a DSS-based concurrent object, and we reproduce their algorithm[2] for completeness in Figure 1. In Li and Golab's fault-tolerant synchronization problem, a pair of

Shared variables:

- Q: queue object based on DSS, as defined formally in [24]

Procedure producer

```
1  ⟨op, res⟩ := Q.resolve()
2  if op = ⊥ then
3  |  start := 1
4  else if res = ⊥ then
5  |  start := argument of op
6  else
7  |  start := argument of op + 1
8  for i from start to ∞ do
9  |  Q.prep-enqueue(i)
10 |  Q.exec-enqueue(i)
```

Procedure consumer

```
11 ⟨op, res⟩ := Q.resolve()
12 if op ≠ ⊥ ∧ res ≠ ⊥ ∧ res ≠ EMPTY then
13 |  print(res)
14 while true do
15 |  Q.prep-dequeue()
16 |  val := Q.exec-dequeue()
17 |  if val ≠ EMPTY then
18 |  |  print(val)
```

Figure 1: Producer-consumer example with a DSS-based queue and consecutive elements.

asynchronous threads communicate using a single instance of a FIFO queue, and *no other shared variables*. One thread is a producer that enqueues integer elements, and the other thread is a consumer that repeatedly dequeues elements and prints their values to the screen.[3] Enqueue and dequeue operations are applied in two phases – prepare and execute (lines 9 to 10 and lines 15 to 16) – which the DSS defines for each operation on the underlying object type (or *base type*). In the absence of failures, each positive integer is eventually enqueued exactly once, then dequeued exactly once by the producer, and also printed to the screen exactly once at line 18.

For recovery, both the producer and the consumer determine what integer value they most recently acted on, and attempt to resume execution of their loops from the point of failure. The algorithm achieves "at least once" semantics in the following sense: each integer value is eventually enqueued and dequeued only once, but may be printed more than once if it is the last value dequeued prior to a particular crash failure. This is accomplished by careful analysis of the forensic information returned by the call to $Q.resolve()$ during recovery (line 1 and line 11), which describes the outcome of the most recently prepared operation by the recovering thread. For the producer, this call returns $(\perp, \perp)$ if no element has been enqueued, $(enqueue(i), \perp)$ if $enqueue(i)$ was prepared but not executed, and $(enqueue(i), OK)$ if $enqueue(i)$ was both prepared and executed. The producer then chooses to resume from iteration 1, i, or $i + 1$, respectively. The consumer's call to $Q.resolve()$ similarly returns $(\perp, \perp)$ if no element has been dequeued, $(dequeue(), \perp)$ if a dequeue operation was prepared but not executed, and $(dequeue(), i)$ if a dequeue was prepared and executed with response i. The consumer prints i in the third case (if the response was not EMPTY), which may be redundant if the failure occurred immediately after line 18, and resumes its loop.

The producer-consumer example demonstrates that the DSS is sufficiently powerful to enable recovery in a simple application, though not quite in the sense of always resuming execution from the exact point of failure. We considered whether similar behaviour is also achievable using NRL and NRL+, and we fell short of finding a concrete solution using either framework.

One drawback of NRL with respect to the problem at hand is the assumption that a program is structured as a collection of recoverable objects. For example, [3] states that when a process p crashes, "the system may eventually resurrect process p by invoking the recovery function of the inner-most recoverable operation that was pending when p failed." On the other hand, the semantics of recovery are less clear when a crash occurs while p is outside of any recoverable operation on a shared object, such as at the top of the loop (immediately before line 9) in the producer's mainline in Figure 1. Even if the producer's algorithm is restarted from the beginning in this case, it is not clear how the producer would determine where to resume its loop given that NRL recovery yields only the response of an operation, which is trivial for enqueue.

For NRL+, we found that the sequential specification of the recovery operation is somewhat loose. More concretely, the response of the recovery operation is stated using two cases, where only the first case uniquely identifies the last operation by process p that took effect prior to a crash [5]:

> "Each $Recover(i)$ operation R returns a sequence number seq and a flag f with the following properties:
> - If $f = 1$, then seq is the sequence number of the last successful CAS operation with process id i.
> - If $f = 0$, all successful CAS operations before R with process id i have sequence number less than seq."

Since the detectable object's implementation decides which of the two cases applies, the above specification theoretically allows

[2] Our rendition of their algorithm explicitly accounts for dequeues that return EMPTY.
[3] It is assumed for simplicity that the print statement is failure-atomic.

Recover(i) to default to the second case ($f = 0$) and return a very large *seq* that is not useful for recovery. Even if the implementation defaulted to the first case ($f = 1$), it is not obvious how to generalize NRL+ from CAS objects to FIFO queues since the sequence number returned by *Recover*(i) is insufficient to obtain the value returned by the consumer's most recent dequeue operation.

The code in Figure 1 also highlights a drawback of the DSS, namely that each enqueue and dequeue requires two separate operations on the shared queue for the prepare and execute phases, even though the natural pattern appears to be that these phases are invoked back-to-back. This observation inspired our Question 1 from Section 1. In other words, if calls to *prep-op* and *exec-op* for some operation *op* are always made in pairs, then can the prepare and execute phases be coalesced into a single operation for simplicity? In the next section, we answer Question 1 in the negative by considering a more realistic version of the producer-consumer problem in which the enqueued values may not be consecutive integers. After all, if the producer always generates values by counting from 1 onward, then Li and Golab's producer-consumer problem has a trivial solution in which the consumer does not communicate with the producer at all, and there is no need for a shared queue.[4]

4 PRODUCER-CONSUMER PROBLEM WITH ARBITRARY DATA ELEMENTS

In our version of Li and Golab's synchronization problem, the producer computes the i'th element to be enqueued by calling a special function `getData`(i) that maps $\mathbb{Z}^+$ to $\mathbb{Z}^+$, and is known a priori to the producer but not the consumer. The semantics of producer-consumer synchronization can then be formalized as stated below.

Definition 4.1. The producer and consumer must satisfy the following behaviours:

- Safety 0: The producer may enqueue but not dequeue, and the consumer may dequeue but not enqueue.
- Safety 1: For any $k \geq 1$, if the producer enqueues at least k times, then the k'th value enqueued is `getData`(k).
- Safety 2: For any $k \geq 1$, if the consumer dequeues at least k times, then the k'th value dequeued is `getData`(k).
- Liveness: In any fair infinite execution with finitely many failures, and for any $k \geq 1$, the consumer eventually dequeues successfully (i.e., removes an element from a non-empty queue) at least k times and prints each dequeued value at least once.

Our problem formulation intentionally excludes solutions where the iteration number i is embedded into the value enqueued by the producer to simplify recovery. This decision reflects a separation of concerns, which is essential for understanding the division of labour between the application and the shared object in terms of tracking recovery state. A relaxed version of the problem that admits cross-layer optimization between the consumer's algorithm and the producer's is discussed later on in this section.

We present a solution to our more elaborate version of Li and Golab's producer-consumer problem in Figure 2 using a DSS-based shared FIFO queue and two process-local persistent variables for checkpointing the producer's progress. The operation `PERSIST` is used to flush the updated values of these variables from the volatile cache to persistent memory, and includes any memory barrier required for this purpose.

Shared variables:

- Q: queue object based on DSS, as defined formally in [24]

Private variables (persistent):

- A: checkpoint variable, initially 0
- B: checkpoint variable, initially 0

Procedure producer

```
19 ⟨op, res⟩ := Q.resolve()
20 if op = ⊥ then
21 |  start := 1
22 else if res = ⊥ then
23 |  start := A
24 else
25 |  start := B + 1
26 for i from start to ∞ do
27 |  val := getData(i)
28 |  A := i
29 |  PERSIST(&A)
30 |  Q.prep-enqueue(val)
31 |  B := i
32 |  PERSIST(&B)
33 |  Q.exec-enqueue(val)
```

Procedure consumer

```
34 ⟨op, res⟩ := Q.resolve()
35 if op ≠ ⊥ ∧ res ≠ ⊥ ∧ res ≠ EMPTY then
36 |  print(res)
37 while true do
38 |  Q.prep-dequeue()
39 |  val := Q.exec-dequeue()
40 |  if val ≠ EMPTY then
41 |  |  print(val)
```

Figure 2: Producer-consumer example with a DSS-based queue and arbitrary data elements.

The algorithm for the consumer is identical to Figure 1, but the producer now relies crucially on the checkpoint variables A and B to determine where to restart its loop because the operation signature returned by *Q.resolve*() is no longer sufficient to distinguish any two enqueue operations. Checkpoints are taken twice in each iteration of the loop: using A at lines 28 to 29 before the prepare phase of enqueue, and using B at lines 31 to 32 between the prepare phase and execute phase. During recovery, if the failure occurred between the prepare phase and execute phase of the same iteration i, then A is used to determine i and execution resumes at iteration i

[4]The trivial solution we have in mind is that the producer prints the integers in a loop, saves the iteration number in a persistent variable, and is able to resume at the appropriate iteration without any interaction with the producer.

(line 23). Otherwise, the failure occurred between the execute phase of iteration i and the prepare phase of iteration $i+1$, B is used to determine i, and execution resumes at iteration $i+1$ (line 25).

On first impression, it may seem that the checkpoint variable A is sufficient to enable recovery, but its value cannot distinguish between a failure immediately before the call to $Q.prep\text{-}enqueue$ in iteration i and a failure immediately after the call to $Q.exec\text{-}enqueue$ of the same iteration. This is problematic because, in both cases, $Q.resolve$ may return the same response $\langle enqueue(val), OK\rangle$ where $val = \texttt{getData}(i-1) = \texttt{getData}(i)$. Similarly, checkpoint variable B by itself is not sufficient as its value cannot distinguish between a failure immediately before $Q.exec\text{-}enqueue$ in iteration i and a failure immediately after $Q.prep\text{-}enqueue$ in iteration $i+1$, which is problematic if $\texttt{getData}(i) = \texttt{getData}(i+1)$. Thus, two checkpoints per iteration are, informally speaking, necessary. We formalize this observation in Theorem 4.2 below.

Theorem 4.2. *Any algorithm that solves the producer-consumer problem formalized in this section (Definition 4.1) using a DSS-based FIFO queue, requires that the producer can access persistent variables in addition to the shared queue. Furthermore, there is at least one execution of the algorithm where such a variable is accessed between the prepare and execute phases of some enqueue operation, and at least one execution of the algorithm where such a variable is accessed between the execute phase of some enqueue operation and the prepare phase of the next enqueue operation.*

Proof. We will first show that the producer must write a persistent variable between the prepare and execute phases of some enqueue operation in some execution. Suppose for contradiction that this is not true, meaning that a solution to the problem exists such that the producer's algorithm never writes a persistent variable between the prepare and execute phases of any enqueue operation. Let the mapping $\texttt{getData}(i)$ be defined as follows:

$$\texttt{getData}(i) = \begin{cases} 1 & \text{if } i = 1 \text{ or } i = 2 \\ i & \text{otherwise} \end{cases} \quad (1)$$

We know from safety property 1 in Definition 4.1 that the producer must enqueue values in the order $1, 1, 3, 4, 5, 6, 7, \ldots$. Now consider the following two execution histories: in H, a crash occurs immediately before the prepare phase of the second enqueue operation; and in H', a crash occurs immediately after the execute phase of the second enqueue operation.[5] Now let S be an extension of H in which the producer and consumer run without crashing, and where their steps are scheduled fairly. It follows from safety property 1 and the liveness property in Definition 4.1 that the producer enqueues $1, 3, 4, 5, 6, 7, \ldots$ in this extension. Similarly, let S' be an extension of H' in which the producer and consumer run without crashing, and note that the producer enqueues $3, 4, 5, 6, 7, \ldots$ in this extension. Since we assume that the producer does not access any persistent variables between the prepare and execute phases of enqueue, and since the producer is not allowed to dequeue by safety property 0 in Definition 4.1, it follows that the states of the system at the end of H and H' are indistinguishable to the producer irrespective of the actions of the consumer. As a result, the producer must enqueue the same elements in S as in S', and this contradicts our earlier observation regarding these execution fragments.

Next, we will show that the producer must write a persistent variable between the execute phase of some enqueue operation and the prepare phase of the next enqueue operation in some execution. Suppose for contradiction that this is not true. The analysis is similar to the first part of the proof, and we assume the same definition of $\texttt{getData}(i)$. We know from safety property 1 in Definition 4.1 that the producer must enqueue values in the order $1, 1, 3, 4, 5, 6, 7, \ldots$. Now consider the following two execution histories: in H, a crash occurs immediately before the execute phase of the first enqueue operation; and in H', a crash occurs immediately after the prepare phase of the second enqueue operation. Let S and S' be extensions of H and H', respectively, in which the producer and consumer run without crashing, and where their steps are scheduled fairly. It follows from safety property 1 and the liveness property in Definition 4.1 that the producer enqueues $1, 1, 3, 4, 5, 6, 7, \ldots$ in S, and $1, 3, 4, 5, 6, 7, \ldots$ in S'. As before, it follows that the states of the system at the end of H and H' are indistinguishable to the producer irrespective of the consumer, and so the producer must enqueue the same elements in S as in S', leading to a contradiction. □

Our analysis of Figure 2 shows that the prepare and execute phases of the DSS cannot always be combined because the application may need to save additional recovery state, external to the DSS-based object, between these phases. This, in turn, implies that the DSS is not sufficiently powerful to enable correct recovery without additional application-managed recovery state. The necessity of such additional state is reminiscent of Ben-Baruch, Hendler, and Rusanovsky's observation regarding the necessity of *auxiliary state* in NRL-like implementations [4], but is required in our case only for the correctness of application, and not for the correct behaviour of the detectable object used by the application.

The impossibility result stated in Theorem 4.2 can be defeated by relaxing the specification of the producer-consumer synchronization problem. One solution, introduced by Li and Golab in Section 2.1 of [24], is to store an additional numerical tag during the prepare phase to distinguish operations with identical signatures and argument values. We refer to this variation of the DSS as the *augmented sequential specification* (ADSS). Another solution, suggested by an anonymous reviewer, is to use the DSS without any modifications, but redefine the semantics of the elements stored in the queue. Specifically, instead of appending $\texttt{getData}(i)$ in iteration i, the producer appends the tuple $\langle i, \texttt{getData}(i)\rangle$, and the consumer discards the first component of the tuple. Both solutions make the problem solvable without application-managed checkpoints, and this observation yields a partial answer to our Question 2 from Section 1.

5 THE UNIFIED DETECTABLE SEQUENTIAL SPECIFICATION

Drawing inspiration from our earlier examples, we now propose a novel *unified detectable sequential specification* (UDSS) that combines the prepare and execute phases of the DSS/ADSS into a single operation. This design choice further simplifies application code and, as we show later on in Section 6, has little impact on performance.

[5] If the prodcuer calls *prep-enqueue* more than once between the first and second *exec-enqueue*, we treat the last such *prep-enqueue* as the prepare phase of the second *exec-enqueue*.

5.1 Formal Definition

We begin with a formal definition of the UDSS. Given a conventional object type $T = (S, s_0, OP, R, \delta, \rho)$ representing a set of states S, initial state s_0, set of operation signatures OP, set of responses R, state transition function δ, and response function ρ, the UDSS of T is a sequential specification $UD\langle T\rangle = (\bar{S}, \bar{s_0}, \bar{OP}, \bar{R}, \bar{\delta}, \bar{\rho})$ obtained from T by the following transformation:

- Each state $\bar{s} \in \bar{S}$ is a tuple $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$, as in the ADSS, where $s \in S$ and the components $\mathcal{A}/\mathcal{T}/\mathcal{R}$ map threads to operation signatures, tags, and responses, respectively. In the initial state $\bar{s_0}$, $s = s_0$ and $\mathcal{A}/\mathcal{T}/\mathcal{R}$ map each thread to $\perp$.
- $\bar{OP}$ comprises all the operations of OP, as well as new *auxiliary operations*: *detectable-op* for each $op \in OP$, and *resolve*.
- The state transition function $\bar{\delta}$, response function $\bar{\rho}$, and response set $\bar{R}$ are described in Figure 3 using an axiomatic style modelled after [19, 24].

$$\{true\}$$
$$detectable\text{-}op(t) \ / \ t_i \ / \ \rho(s, op, t_i)$$
$$\{s' = \delta(s, op, t_i) \wedge \mathcal{A}'[t_i] = op \wedge \mathcal{T}'[t_i] = t \wedge \mathcal{R}'[t_i] = \rho(s, op, t_i)\} \quad (2)$$

$$\{true\}$$
$$resolve \ / \ t_i \ / \ (\mathcal{A}[t_i], \mathcal{T}[t_i], \mathcal{R}[t_i]) \quad (3)$$
$$\{\}$$

$$\{true\}$$
$$op \ / \ t_i \ / \ \rho(s, op, t_i) \quad (4)$$
$$\{s' = \delta(s, op, t_i)\}$$

Figure 3: Unified detectable sequential specification (UDSS) of base type T, denoted $UD\langle T\rangle$.

In practical terms, the operations described axiomatically in Figure 3 behave as follows. For each $op \in OP$ of type T, *detectable-op* (Axiom 2) is used to apply op in a detectable way. The tag argument t of *detectable-op* is saved in the state of the detectable object but excluded in the computation of δ and ρ of the base type T, as in the ADSS. Operation *resolve* (Axiom 3) returns the operation signature op, tag t, and response r of the most recent *detectable-op*, or $(\perp, \perp, \perp)$ if *detectable-op* was never applied by the same thread. Finally, operation op (Axiom 4) simply applies the state transition prescribed by op in a non-detectable way with no other side-effects.

The UDSS supports detectability in the following sense: if a thread calls *resolve* immediately after applying *detectable-op*(t) with response r, then *resolve* returns (op, t, r); and if the thread did not apply *detectable-op* for any op, then *resolve* returns $(\perp, \perp, \perp)$.

5.2 Application Examples

Figure 4 shows how to solve the producer-consumer problem with arbitrary data elements using a UDSS-based queue. Compared to the ADSS-based algorithm (omitted due to lack of space), the producer's recovery code considers two cases instead of three, and the body of the while loop is shorter by one statement. A more elaborate version of the algorithm can be derived for the case when the implementation of Q restricts the size of the tags to only k bits, in which case the algorithm checkpoints every 2^{k-1} iterations.

Shared variables:

- Q: queue object based on UDSS

Procedure producer

```
42 ⟨op, tag, res⟩ := Q.resolve()
43 if op = ⊥ then
44 |  start := 1
45 else
46 |  start := tag + 1
47 for i from start to ∞ do
48 |  val := getData(i)
49 |  Q.detectable-enq(val, i)
```

Procedure consumer

```
50 ⟨op, tag, res⟩ := Q.resolve()
51 if op ≠ ⊥ ∧ res ≠ EMPTY then
52 |  print(res)
53 while true do
54 |  val := Q.detectable-deq(⊥)
55 |  if val ≠ EMPTY then
56 |  |  print(val)
```

Figure 4: Producer-consumer example with a UDSS-based queue and arbitrary data elements.

5.3 Transformation from ADSS to UDSS

In this section, we show that an object that implements the ADSS for some base type T (i.e., $AD\langle T\rangle$) can be transformed into one that implements the UDSS for T (i.e., $UD\langle T\rangle$). The transformation (Figure 5) uses a single base object B of type $AD\langle T\rangle$, which we assume is atomic for simplicity of analysis, but can be replaced with a strictly linearizable one. The transformation preserves wait-freedom, and yields a strictly linearizable implementation of $UD\langle T\rangle$.

The core idea underlying the transformation is that thread t_i saves the result of its most recently completed detectable operation to $LastOp[i]$ in *resolve*, and calls *resolve* inside *detectable-op* before preparing and executing op on the base object B. The linearization points (LPs) of the implemented operations are as follows:

- The LP of *detectable-op* is the call to $B.exec\text{-}op()$.
- The LP of op is the call to $B.op()$.
- The LP of *resolve* is the PERSIST at line 64 or the read of $LastOp[i]$ at line 67.

Theorem 5.1. *The transformation presented in Figure 5 maintains the following invariant:*

> *Let $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$ denote the state of the base object B of type $AD\langle T\rangle$. Then the state of the implemented object of type $UD\langle T\rangle$ is a tuple $(\bar{s}, \bar{\mathcal{A}}, \bar{\mathcal{T}}, \bar{\mathcal{R}})$ such that:*

Shared variables:

- B: atomic base object of type $AD\langle T\rangle$
- $LastOp[1..n]$: array of triples, each element initially $(\perp, \perp, \perp)$

Procedure *detectable-op*(t: tag)

57 *resolve*()
58 B.*prep-op*(t)
59 **return** B.*exec-op*()

Procedure *op*()

60 **return** B.*op*()

Procedure *resolve*()

61 $(op, tag, res) := B.resolve()$
62 **if** $op \neq \perp \wedge res \neq \perp$ **then**
63 $\quad LastOp[i] := (op, tag, res)$
64 $\quad$ PERSIST(&$LastOp[i]$)
65 $\quad$ **return** (op, tag, res)
66 **else**
67 $\quad$ **return** $LastOp[i]$

Figure 5: Transformation from $AD\langle T\rangle$ to $UD\langle T\rangle$, thread t_i.

(1) $\bar{s} = s$
(2) *for each thread* t_i, *if* $\mathcal{R}[i] = \perp$ *then* $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i]) = LastOp[i]$[6]
(3) *for each thread* t_i, *if* $\mathcal{R}[i] \neq \perp$ *then* $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i]) = (\mathcal{A}[i], \mathcal{T}[i], \mathcal{R}[i])$

Theorem 5.2. *The transformation presented in Figure 5 is strictly linearizable and preserves wait-freedom.*

Proof. Preservation of wait-freedom follows easily from the structure of the transformation. For strict linearizability, first note that by our earlier definition of linearization points, each implemented operation that takes effect does so at a point between its invocation, and its response or a crash event that interrupts its execution. To complete the analysis, we must show that the implemented UDSS operations return correct responses with respect to the chosen linearization points (LPs). We proceed by case analysis. For each implemented operation, we consider the state immediately before the execution of the LP, and immediately after. We will use $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$ and $(\bar{s}, \bar{\mathcal{A}}, \bar{\mathcal{T}}, \bar{\mathcal{R}})$ to denote the state of the base object B and implemented object, as determined by our choice of LPs, immediately before the LP. Primed symbols will denote the analogous states immediately after the LP.

Case A: *detectable-op*(t) by thread t_i. The LP of this operation is B.*exec-op*() at line 59. The actual response of the implemented operation is the same as the response of B.*exec-op*(), which is $\rho(s, op, t_i)$. The correct response of the implemented operation is $\rho(\bar{s}, op, t_i)$ since $AD\langle T\rangle$ and $UD\langle T\rangle$ share the same response function ρ. By our invariant (Theorem 5.1), $s = \bar{s}$, and so $\rho(s, op, t_i) = \rho(\bar{s}, op, t_i)$. Thus, the actual response is correct.

Case B: *op*() by thread t_i. The LP of this operation is B.*op*() at line 60, and the analysis is identical to Case A.

Case C: *resolve*() by thread t_i. The LP of this operation is a PERSIST at line 64 or a read at line 67. The actual response of the implemented operation is the value persisted to $LastOp[i]$ at line 64, or the value read from $LastOp[i]$ at line 67. The correct response of the implemented operation is $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i])$.

If the condition evaluated earlier at line 62 is true, then the actual response is equal to the value written to $LastOp[i]$ at lines 63 to 64, which was returned earlier by the call to $B.resolve()$ at line 61, and which is therefore equal to $(\mathcal{A}[i], \mathcal{T}[i], \mathcal{R}[i])$. Since clause 3 of the invariant (Theorem 5.1) applies and states that $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i]) = (\mathcal{A}[i], \mathcal{T}[i], \mathcal{R}[i])$, the actual response is correct.

On the other hand, if the condition evaluated earlier at line 62 is false, then the actual response is equal to the value read from $LastOp[i]$ at line 67, which was written and persisted during an earlier execution of *resolve*() on the implement object. In this case, $\mathcal{R}[i] = \perp$, and so clause 2 of the invariant (Theorem 5.1) applies and states that $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i]) = LastOp[i]$. Thus, the actual response is $(\bar{\mathcal{A}}[i], \bar{\mathcal{T}}[i], \bar{\mathcal{R}}[i])$, and this is the correct response. □

6 EXPERIMENTAL EVALUATION

This section presents an experimental performance evaluation comparing two strictly linearizable implementations of Li and Golab's detectable lock-free FIFO queue [24]: one based on the DSS, and another obtained by transforming the DSS-based version first to ADSS and then to UDSS using our transformation from Section 5.

We conduct scalability experiments using an Intel multiprocessor with genuine Optane persistent memory. The queue prototypes are implemented using C++ and the Intel Persistent Memory Development Kit (PMDK) [31]. Our code is loosely based on Li's open-source implementation [23] of the DSS-based queue [24], which itself is embedded into a fork of the open-source implementation of Wang et al.'s Persistent Multi-word Compare-And-Swap (PMwCAS) [35]. We have rewritten most of Li's code to isolate the queue implementation from PMwCAS, and also transitioned to using the persistent pointer class from the libpmemobj library to link persistent objects in a manner that compensates for address space layout randomization (ASLR).[7] Despite the overhead of using the persistent pointer class, which introduces additional method calls, our DSS-based queue is actually faster than Li's, mainly due to improved memory alignment. Our code is publicly accessible on an institutional git server [27].

One complication we encountered with respect to persistent pointers is that they cannot be updated in a failure-atomic manner using basic machine instructions. This is because such pointers comprise two 64-bit fields – the memory pool identifier and offset – whereas Intel supports only 64-bit failure-atomic writes [30]. Since we avoid the transactions provided by the libpmemobj library due to their high performance overhead, except during the one-time initialization phase, our workaround is to restrict the memory layout

[6]The notation $LastOp[i]$ refers to the value stored in persistent memory, and not the potentially newer value in the volatile cache.

[7]Each time the code is restarted, a persistent object is mapped to a potentially different virtual address because of ASLR. Thus, pointers between objects must be relative, and virtual addresses must be calculated by adding the offset to the base address of the memory-mapped file containing the persistent objects.

of the data structure to a single memory pool and manipulate only the offset field of the persistent pointer using 64-bit loads, stores, and Compare-And-Swap operations.

Our results, presented in Figure 6, evaluate the performance of the DSS-based and UDSS-based queue implementations in failure-free runs with up to 20 threads. The workload in our experiments is similar to [24]. Multiple threads apply detectable enqueue and dequeue operations repeatedly to a single shared queue. The elements enqueued by each thread are integers. Since the queue structures are implemented as linked lists and queue nodes must be dynamically allocated, we use a memory management scheme similar to [23, 24], where each thread pre-allocates a pool of such nodes and a variation of epoch-based reclamation (EBR) [14] is used to recycle nodes. We also include data for a non-detectable DSS-based queue, where queue operations are applied directly (Axiom 4 in Figure 1 of [24]) rather than via *prep-op* and *exec-op*. In each experiment, the queue is initialized with 4 queue nodes. Each point plotted is the mean throughput value (millions of operations per second) computed over a sample of 6 runs, and in all cases, the sample standard deviation is less than 2% of the sample mean. The results in Figure 6a show that the UDSS-based queue performs comparably to DSS despite differences in the interface.

Figure 6b breaks down the execution latency for the UDSS-based queue into three components: overhead due to memory management, enqueue processing, and dequeue processing. The stacked bar chart shows that our EBR-based memory management is quite efficient, and that dequeue operations are slower than enqueues. The latter observation is consistent with the structure of Li and Golab's algorithm [24], where both enqueues and dequeues update the linked list of nodes, and dequeues additionally mark removed nodes with the caller's thread ID.

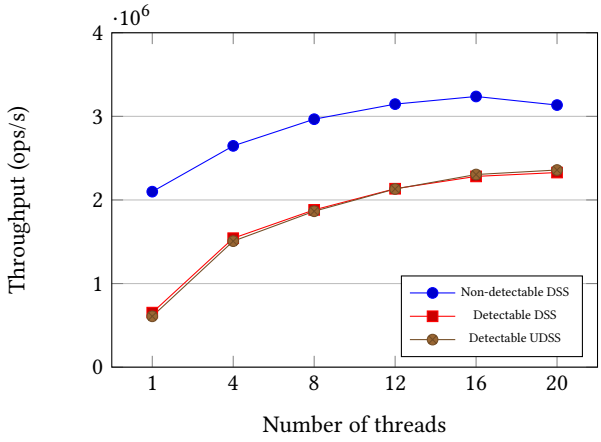
Next, we consider the producer-consumer application with one producer and one consumer. Table 1 shows the throughput and the proportion of time spent by the producer in *exec-enqueue*. The DSS column corresponds to the code in Figure 2, and UDSS corresponds to a variation of Figure 4 that uses 16-bit tags. Each number is the average of a sample of 6 runs, and the sample standard deviation is less than 2% of the sample mean. The results presented in the table demonstrate that the DSS-based producer-consumer implementation has slightly better performance than the UDSS-based one. Moreover, the proportion of time spent in *exec-enqueue* suggests that the overhead of checkpointing in the DSS-based implementation of the producer is lower than the overhead introduced by our UDSS transformation (e.g., line 57, line 61, and line 64 in Figure 5).

Table 1: Analysis of producer-consumer application.

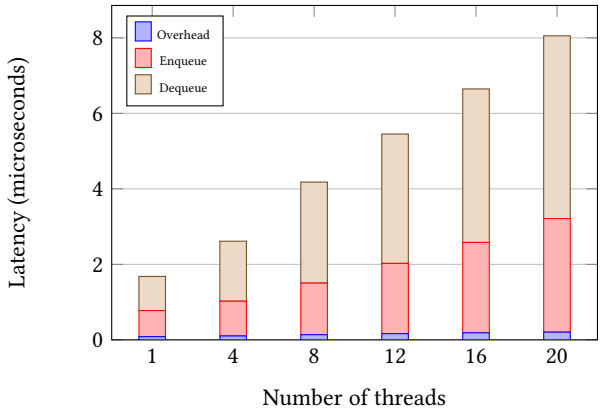
Implementation type	DSS	UDSS
Throughput (queue ops/s)	641k	618k
Time spent by producer in *exec-enqueue*	84%	63%

7 CONCLUSION

This paper posed and partially answered a number of fundamental questions regarding specifications of correctness for detectable objects, focusing on the detectable sequential specification (DSS)



(a) Throughput of DSS-based and UDSS-based queue implementations.



(b) Latency analysis of UDSS-based queue implementation.

Figure 6: Throughput and latency experiments.

formalized recently by Li and Golab [24]. We first asked whether the separation of a detectable operation into a prepare phase and execute phase in the DSS is fundamentally necessary, and found that the answer is affirmative in a simple producer-consumer application. On the other hand, we reached the opposite conclusion with respect to Li and Golab's augmented detectable sequential specification (ADSS), which uses auxiliary operation arguments in the form of numerical tags to distinguish operations with identical signatures. Next, we introduced the *unified detectable sequential specification* (UDSS), which combines the prepare and execute phases of the ADSS into a single operation to simplify application code. Finally, we proposed a black box transformation from ADSS to UDSS, and evaluated its practical performance overhead on Li and Golab's detectable queue.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This research was 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.

REFERENCES

[1] Marcos K. Aguilera and S. Frølund. 2003. *Strict linearizability and the power of aborting.* Technical Report HPL-2003-241. Hewlett-Packard Labs.

[2] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2020. Tracking in Order to Recover - Detectable Recovery of Lock-Free Data Structures. In *Proc. of the 32nd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 503–505.

[3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*. 7–16.

[4] Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. 2020. Upper and Lower Bounds on the Space Complexity of Detectable Objects. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*. 11–20.

[5] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[6] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust Shared Objects for Non-Volatile Main Memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*. 20:1–20:17.

[7] Trevor Brown and Hillel Avni. 2016. PHyTM: Persistent Hybrid Transactional Memory. *Proc. VLDB Endow.* 10, 4 (2016), 409–420.

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

[9] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–269.

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

[11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 271–282.

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

[13] Carole Delporte-Gallet, Panagiota Fatourou, Hugues Fauconnier, and Eric Ruppert. 2022. When is Recoverable Consensus Harder Than Consensus?. In *Proc. of the 41st ACM Symposium on Principles of Distributed Computing (PODC)*.

[14] Keir Fraser. 2004. *Practical Lock-Freedom.* Ph.D. Dissertation. University of Cambridge. Computer Laboratory.

[15] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2017. Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, Vol. 91. 50:1–50:4.

[16] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. 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)*. 28–40.

[17] Wojciech Golab. 2020. The Recoverable Consensus Hierarchy. In *Proc. of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 281–291.

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

[19] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[20] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. 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)*. 427–442.

[21] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*. 313–327.

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

[23] Nan Li. 2021. *GitHub fork of* Persistent Multi-Word Compare-and-Swap (PMwCAS) for NVRAM. https://github.com/lnnn1982/pmwcas

[24] Nan Li and Wojciech Golab. 2021. Detectable Sequential Specifications for Recoverable Shared Objects. In *Proc. of the 35th International Symposium on Distributed Computing (DISC, Vol. 209)*. 29:1–29:19.

[25] Nan Li and Wojciech Golab. 2021. Detectable Sequential Specifications for Recoverable Shared Objects. (2021). https://www.youtube.com/watch?v=fFVwNZZphMI Presentation at the 35th International Symposium on Distributed Computing (DISC'21).

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

[27] Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. 2022. DSSQueue. https://git.uwaterloo.ca/wgolab/DSSQueue.git

[28] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. In *Proc. of the 18th International Conference on Extending Database Technology (EDBT)*. 689–694.

[29] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131.

[30] Andy Rudoff. 2017. Persistent Memory Programming. *login Usenix Mag.* 42, 2 (2017).

[31] Andy Rudoff and the Intel PMDK Team. 2020. Persistent Memory Development Kit. https://pmem.io/pmdk/ [last accessed 2/11/2021].

[32] Gal Sela and Erez Petrank. 2021. Durable Queues: The Second Amendment. In *Proc. of the 33rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 385–397.

[33] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*, Vol. 11. 61–75.

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

[35] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proc. of the 34th IEEE International Conference on Data Engineering (ICDE)*. 461–472.