The Inherent Cost of Remembering Consistently

[Technical Report]

Nachshon Cohen
EPFL
nachshon.cohen@epfl.ch

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Igor Zablotchi
EPFL
igor.zablotchi@epfl.ch

Abstract

Non-volatile memory (NVM) promises fast, byte-addressable and durable storage, with raw access latencies in the same order of magnitude as DRAM. But in order to take advantage of the durability of NVM, programmers need to design *persistent* objects which maintain consistent state across system crashes and restarts. Concurrent implementations of persistent objects typically make heavy use of expensive persistent fence instructions to ensure ordering among NVM accesses, thus negating some of the performance benefits of NVM.

This raises the question of the minimal number of persistent fence instructions required to implement a persistent object. We answer this question in the deterministic lock-free case by providing lower and upper bounds on the required number of fence instructions. We obtain our upper bound by presenting a new universal construction that implements durably any object using at most one persistent fence per update operation invoked. Our lower bound states that in the worst case, each process needs to issue at least one persistent fence per update operation invoked.

1 Introduction

Non-volatile memory (NVM) is fast, byte-addressable memory that preserves its contents even in the absence of power. Recent years have seen significant research into NVM [32, 38, 40, 42], but the technology is only now starting to become commercially available.

NVM shares similarities with both traditional stable storage and DRAM. Like stable storage, NVM allows applications to persist their state across power failures or machine restarts. Unlike stable storage, NVM is byte-addressable and fast (with access times in the same order of magnitude as DRAM [46]). In this sense, NVM promises applications that are durable and fast, for they would not need to access slow storage when persisting their state or during recovery.

Yet, the task of designing fast persistent objects (as building blocks of persistent applications) is complicated by two factors:

1. Processor registers and caches are expected to remain volatile (transient) for the foreseeable future. Therefore, simply writing to a memory location is not sufficient to ensure the persistence of its contents (even if the memory location is in NVM), because the write instruction might, for instance, be satisfied in the cache and thus lost in the case of a crash.
2. There is a priori no guarantee on the order in which cache lines are written back to NVM. However, program correctness might rely on such a guarantee, especially in a concurrent setting, which is the focus of this paper.

Due to these two factors, programming for NVM requires the use of flush instructions to force cache lines to be written back, as well as of expensive fence instructions to enforce ordering among such flushes.

This raises an interesting question: what is the minimum number of persistent fences required to implement a persistent object? In this paper, we answer this question for concurrent lock-free objects, by providing both upper and lower bounds on the number of fences required to implement them persistently.

We focus on the lock-free case because it provides an interesting trade-off. On the one hand, intuitively, lock-free objects can be implemented with a small number of fences, because they are already required to always be in a consistent state, such that progress can be made despite the failure of any number of processes. A priori, durability has the related requirement of an object state being consistent, no matter when a crash may occur. On the other hand, it is this very need for consistency that makes lock-free objects require at least a minimal number of fences for each operation invoked, as we shall see later in the paper (we discuss lock-based objects in Section 8).

The correctness (safety) property we consider in this paper is durable linearizability [28]. Durably linearizable objects satisfy the standard linearizability property: every operation seems to happen instantaneously at a *linearization point* between its invocation and response, in separation from any other process in the system. In addition, after a full-system crash, the state of the object must reflect a consistent operation subhistory that includes all operations completed by the time of the crash.

For the upper bound, we propose a new universal construction called *Order Now, Linearize Later* (ONLL) that takes any deterministic sequential specification of an object O and produces a lock-free durably linearizable implementation of O that is guaranteed to use at most one fence per operation invoked, in the worst case. Our construction in fact guarantees detectable execution [14], an even stronger property than durable linearizability, which ensures in addition that, upon recovery, processes are able to determine which operations were linearized before the crash and which operations were not.

In our universal construction, we distinguish between read-only and update operations. An update operation *op* proceeds in 3 steps, called *order*, *persist* and *linearize*, respectively. First, *op* synchronizes with other update operations to establish the linearization order of *op*. This step uses a shared lock-free execution trace data structure, based on a lock-free queue, for determining this order in a lock-free manner. Second, *op* is stored in NVM by using a per-process persistent log. Crucial to this construction is the fact that the persistent log can be implemented with only one persistent fence per append operation [12]. A helping mechanism is used to ensure that delayed processes do not create inconsistencies in the state of the object. Third, *op* announces that it has completed the persistence step. This is also the linearization point [22] of *op* if it runs solo. When setting the linearization point, care is taken to respect the linearization order computed in Step 1. A read-only operation determines its return value based on the update that most recently announced completion of the persistence step.

Since persistent fences are only performed when appending one or more updates to a process' persistent log, it is clear that our construction uses at most one persistent fence per operation. Moreover, no process can prevent the system from making progress, thus the construction is lock-free.

Our lower bound states that any lock-free implementation of a persistent object has at least one execution in which all concurrent processes need to issue one fence instruction per update operation invoked. The intuition behind this result is that processes cannot always rely on each other to persist updates and must therefore sometimes persist these updates themselves. To see this, imagine that some process p is designated to persist updates for one or more other processes but p is delayed. In order for lock-freedom to be satisfied, those other processes cannot wait indefinitely for p, and so must persist their updates themselves, thus each incurring the cost of persistent fences.

To summarize, the contributions of this paper are:

1. The ONLL universal construction, providing a lock-free durably linearizable implementation of any deterministic object. ONLL uses a single persistent fence per update operation and no persistent fences for read-only operations. ONLL also serves as upper bound on the number of persistent fences required to implement such objects.
2. A lower bound on the number of persistent fences in a lock-free durably linearizable implementation of an object.

We also discuss extensions to our universal construction for wait-freedom, improved read performance and memory reclamation.

The rest of this paper is organized as follows. Section 2 recalls useful background. In Section 3, we give a high-level overview of our universal construction. In Section 4, we describe in more details the algorithm of the universal construction and we prove its correctness in Section 5. In Section 6, we present our lower bound result. We discuss relevant related work in Section 7 and conclude with an overview of possible extensions and future directions in Section 8.

2 Background

2.1 NVM

So far, storage has been either slow but durable (e.g., SSD, hard disk, magnetic tape) or fast but volatile (e.g., DRAM). NVM promises to combine the best of both worlds through fast and durable storage. Several implementations of NVM are foreseen: Memristors [42], Phase Change Memory[32, 40] and 3D XPoint [38].

NVM is expected to be byte-addressable and attached directly to the memory bus of the CPU, accessible by standard load and store instructions. Thus, programming for NVM will probably be closer to programming for DRAM than for block-based storage. As argued in the introduction, the main difficulty in programming for NVM will likely stem from the fact that a priori there is no guarantee on when and in what order (volatile) cache lines will be written back to NVM. Therefore, programmers will need to use special instructions to ensure cache lines are written to the NVM.

One such instruction on Intel machines is `clflush` [26], which forces a cache-line to be written back to the NVM. This instruction is strongly ordered: a call to `clflush` returns only once the cache line is written-back to the NVM and is durable. Consequently, invoking this instruction stalls the CPU for the entire duration of accessing the NVM, which is expected to be very expensive in terms of CPU cycles.

Durability can also be ensured by using asynchronous write-back instructions, such as `clflushopt` or `clwb` [24]. We adopt this approach in this paper since it can be up to an order of magnitude faster than `clflush` [25]. Multiple invocations of these instructions are not ordered, so multiple cache lines can be flushed in parallel. Since these instructions do not stall the CPU and can be processed in the background, we consider the cost of invoking such instructions to be zero. Of course, this also means that invoking write-back asynchronous instructions is not sufficient to ensure durability.

In order to ensure that an asynchronous write-back completes and data is made durable, a *fence instruction* is required, which stalls the CPU until all active asynchronous write-back instructions complete. The fence instruction stalls the CPU for the entire duration for accessing the NVM, which can be expensive. Thus, our main focus in this work is reducing the number of such fences. We emphasize that it is possible to execute a fence while no asynchronous cache line flush instructions are active, in which case the CPU does not (necessarily) access the NVM. We denote an execution of a fence while asynchronous cache line flushes are pending by *persistent fence*.

2.2 Processes and Operations

We consider a set of *n* processes that communicate through a set of shared memory access primitives. We make no assumptions on the relative speeds of the processes; in particular, at any point in time, processes may be delayed for arbitrary or even infinite amounts of time. As in previous work [28], we also assume that the whole system can crash at any point in time and potentially recover later. On such a full-system crash, we assume that the contents of NVM are preserved but that the contents of the processor's registers and caches are lost. Processes running at the time of the full-system crash also crash and are replaced by new processes after recovery. After a system recovers and before resuming normal operation, we assume that a (potentially empty) *recovery* routine is invoked in order to bring the persistent objects on the NVM back to a consistent state.

We classify operations on an object as *read-only* or *update*. Updates are operations that influence the result of subsequent operations; read-only operations do not influence later operations. Updates also read the state of the object and have return values. The *state* of the object is the sequence of update operations applied to the object; the first operation must be INITIALIZE. This definition implies that update operations are deterministic: applying the same sequence of updates on the object always results in the same state.

Our universal construction assumes the existence of a *compute* method. Given a read operation *r* and the state of the object *s* (i.e., the sequence of update operations on the object), this method computes the returned value of *r* applied to *s*. For an update operation *u*, the returned value is computed on the state of the object *s* immediately after appending *u*.

3 ONLL: a Primer

We give in this section a high-level view of ONLL, our universal construction that takes any deterministic object *O* and produces a lock-free durably linearizable implementation of *O* that requires at most one persistent fence per update operation and no persistent fence for read-only operations. Broadly, the execution of an operation under ONLL follows three stages: (1) *order* (in which the linearization order of the operations is established), (2) *persist* (in which the operation is made persistent) and (3) *linearize* (in which the operation is linearized). We first present the rationale behind these three stages and then give an overview of how ONLL works. We end the section with a concrete example, illustrating a shared counter implementation produced by ONLL.

3.1 Rationale

Not performing any persistent fences when reading is a highly desirable property. However, this imposes some constraints on the design of ONLL:

1. The linearization point of an update operation cannot coincide with the time when the operation reaches NVM. This is because NVM can only be written using simple writes (as opposed to the cache that supports CAS), so it cannot in general serve as a synchronization point. A reader cannot distinguish whether data already reached NVM or not.
2. The linearization order of update operations must be known before the operation is written to NVM. This is because NVM must contain enough information to replay this information in the correct (linearization) order.
3. The linearization point of an update operation must happen after the write to NVM.

The last constraint was derived by the following contradiction. Suppose that the linearization point of an update does not happen after the write to the NVM. Then, a reader may observe the update operation before it reaches NVM. There are three cases, each leading to a different contradiction:

- The reader finishes its operation before the dependent update is persisted. This breaks linearizability if the reader performs an external operation (e.g., print) before the dependent update reaches NVM and the system crashes afterward. It is not possible to recover the update, but the dependent read was already observed.
- The reader waits for the dependent update to be persisted. This breaks lock-freedom since the process executing the update operation may stall arbitrarily long.
- The reader helps the dependent update to persist. This breaks the property of never executing a persistent fence for read-only operations.

3.2 ONLL Design

The design of ONLL derives naturally from the above-mentioned constraints. Since the linearization order of an update operation must be known before the operation is made durable, and the operation must be made durable before it is linearized, an update operation *u* under ONLL has three stages: order, persist, linearize.

First, in the order stage, *u* creates a descriptor *d* for itself and appends *d* to the tail of a shared *execution trace*. Second, in the persist stage, *u* gathers all operations that have not been persisted yet and appends them to a per-process persistent log (residing in NVM), along with the ordering information. Third, in the linearize stage, *u* sets a flag at *d*, called the *available* flag, thus announcing that all operations in the execution trace up to *u* have been persisted. This serves as linearization point for *u* and for any pending operations that *u* may have written to NVM in the persist stage. Finally, *u* can compute its return value based on the state of

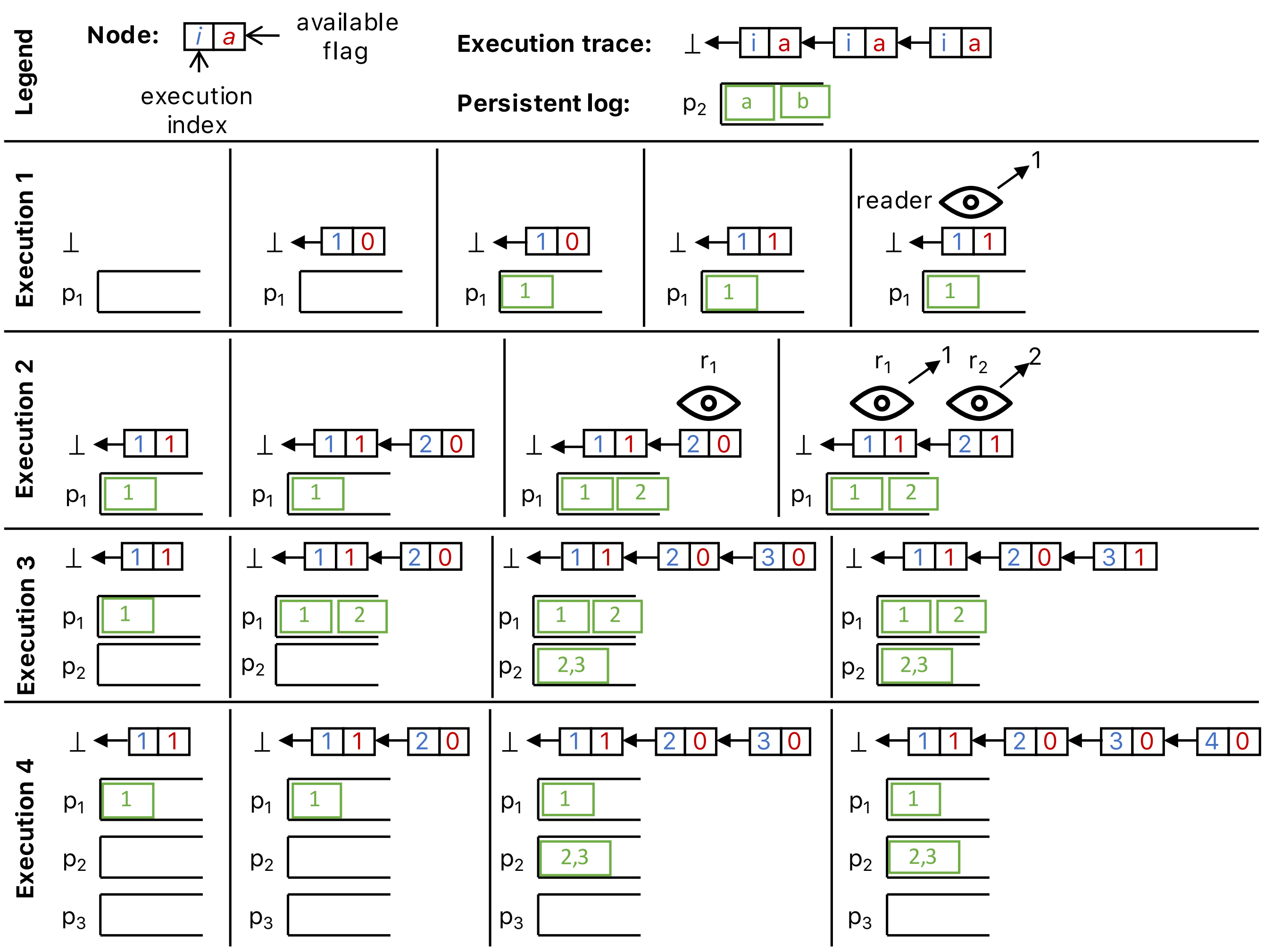

Figure 1. Executions of a counter implemented using ONLL.

the object up to d. u is linearized at the linearize stage, after the write to the NVM, but according to the order computed at the order stage. In other words, the linearize stage is the linearization point of an update operation unless another process helped; it also helps to set linearization points of all dependent update operations.

A read-only operation r simply traverses the execution trace, starting from the tail, until it encounters the first descriptor d_{first} with a set available flag. r then computes its return value based on the state of the object up to d_{first}, as recorded in the execution trace.

This design ensures that the linearization point of an operation happens after the write to the NVM, so that readers are never required to wait or help with persisting the update operation. Moreover, the design ensures that after a crash, the NVM contains enough information to recover the state of the object in same order as the linearization points.

3.3 Illustration: Shared Counter

We illustrate how our ONLL universal construction works for a concrete shared object: a counter. A counter holds an integer value and has two operations: *increment* and *read*. The first is an update operation that increments the counter's value and returns the new value. Read is a read-only operation that simply returns the value of the counter. In what follows, we walk through several increasingly complex executions of the counter, to illustrate various situations that can arise with our construction. These executions are illustrated in Figure 1.

Sequential update and read. In the first execution, a single process p_1 executes an update operation (increment), followed by a read-only operation. Initially, both the execution trace and p_1's persistent log is empty. Process p_1 creates a new node n with execution index equal to 1 and available flag unset. Then, p_1 appends to the persistent log an entry containing all operations that have not been persisted yet (just n in this case). To finalize the update, p_1 sets n's available flag and returns the new value of the counter, 1. Next, p_1 performs a read operation by traversing the execution index from tail to head, stopping at the first node with a set available flag. In our case, there is only one node n, and its available flag is set. The read thus computes its return value based on the state of the counter at n. n corresponds to a state in which one increment has been performed, so the read returns 1.

Update concurrent with reads. In the second execution, process p_1 is executing an update concurrently with two readers r_1 and r_2. The counter initially has value 1: there is already a node n_1 in the execution trace. The update appends a new node n_2 to the execution trace, appends the relevant entry to p_1's persistent log, and then pauses. r_1 traverses the execution trace from tail to head, stopping at n_1, the first node with a set available flag. p_1 resumes execution and sets the available flag of n_2. r_2 begins traversing the execution trace and stops at n_2. Finally, the three operations return: r_1 returns 1, based on the state of the counter at n_1; r_2 returns 2, based on the state at n_2; p_1's update returns 2, also based on the state at n_2.

Update helping another update. In the third execution, processes p_1 and p_2 are each executing a increment concurrently. Initially, the counter has value 1 and the execution trace contains 1 node.

Process p_1 appends a node to the execution trace, adds the corresponding entry to the persistent log, and then pauses. p_2 also appends a node to the execution trace and then adds a persistent log entry containing all operations that have not been persisted yet: both p_1's update and p_2's update. Finally, p_2 sets the available flag of n_3 and returns 3. Any reader starting its traversal after n_3's available flag has been set will return 3, even though the available flag of n_2 has not yet been set.

Crash concurrent with updates and reads. In the fourth execution, processes p_1, p_2 and p_3 are each executing an update operation. Initially, the counter has value 1 and the execution trace contains 1 node.

Process p_1 appends a node n_1 to the execution trace and then pauses for the rest of the execution. p_2 appends an execution trace node n_2, adds an entry to the persistent log corresponding to its own update and to the update of p_1, and then pauses before setting the available flag of n_2. p_3 also appends an execution trace node n_3, and starts adding a node to the persistent log corresponding to the updates of p_1, p_2 and p_3. The system crashes before any of the operations have returned.

After the crash, the state of the counter reflects the updates of p_1 and p_2. These can be reconstructed from p_2's persistent log, even though the no available flag was set during the execution. The post-crash state of the counter does not however reflect p_3's update, because p_3 did not finish adding its persistent log entry.

Since no available flag was set during the execution, any reader concurrent with the updates will return 1, the initial value of the counter. Post-crash readers will return 3.

4 ONLL: a Universal Construction

In this section, we first detail the data structures required by ONLL and then we describe the ONLL algorithm itself.

4.1 Data Structures

The ONLL algorithm depends on two basic building blocks: a single-fence persistent log and a lock-free queue.

We assume that an update operation can be stored in (persistent) memory by using the `operation` structure; input parameters are considered part of the operation and are thus also reflected in the operation structure.

4.1.1 Persistent Log Usage

ONLL uses per-process persistent logs. We leverage the log implementation of Cohen et al. [12], which uses only one persistent fence per append. Each append invocation records up to MAX-THREADS operations, the number of recorded operations, and an execution index; pseudo code is provided in Listing 1. The first operation in the operation array is the current update operation executed by the process. The rest of the operations in the operation array are used to help other processes to persist their data. The `executionIndex` is a unique index that represents the ordering of the linearization point of the first operation. Operations in the array are sequential, so that the execution index of the k-th help operation is `executionIndex` $- k$.

Listing 1. recordEntry

```
struct recordEntry{
    operation ops[MAX_THREADS];
    int num_ops; //between 1 and MAX_THREADS
    long executionIndex;
}
```

4.1.2 Transient Execution Trace

The second data structure used by ONLL is a *transient* (i.e., not necessarily stored in NVM) execution trace of the object. This represents the sequence of all update operations applied to the object. Recall that the execution trace is equivalent to the state of the object. We emphasize that read-only operations do not appear in the execution trace of an object since they do not influence the state of the object; in our design, a read-only operation never writes to shared memory or to NVM.

The sequence of the update operations in the execution trace is partitioned into a *non-fuzzy* prefix and a *fuzzy window* postfix. The fuzzy window represents a set of currently

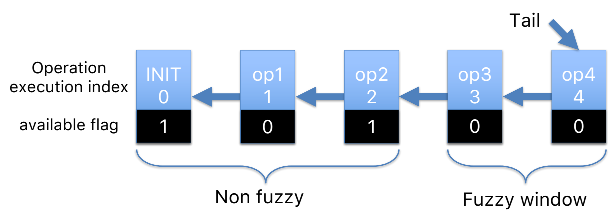

Figure 2. Illustrating the execution trace data structure and the fuzzy window. Since op2 has a set available flag, all operations preceding it, including op1, are considered part of the non-fuzzy window. op3 and op4 have no later operation with a set available flag and so are the fuzzy window of the execution trace.

executing operations that are not yet guaranteed to reside on NVM and their linearization point has not yet occurred. The non-fuzzy prefix consists of all other operations, which are guaranteed to reside on NVM and their linearization point has already occured. The fuzzy window is implemented by assigning an *available* flag for each operation in the execution trace. The fuzzy window spans from from the latest operation in the execution trace up to (but not including) the latest operation with available flag set. Available flags can be set in any order, depending on the speed of the relevant process. A set available flag is never cleared.

It is important to note that the fuzzy window is continuous: if an operation *op* has its available flag unset, but a later operation has its available flag set, then *op* is not part of the fuzzy window. An illustration appears in Figure 2.

The execution trace is implemented in a lock-free manner, based on the lock-free queue algorithm [37]. A slight difference from a traditional lock-free algorithm is the need to compute the *execution index* of each operation, which counts the number of update operations in the execution trace before the current operation. Pseudo code for the queue operation appears in Listing 2. Computing the execution index of a new operation is done at Line 34.

The execution trace data structure supports an operation for reading the latest operation with its available flag set, which represents the latest operation in the non-fuzzy part of the execution trace. Pseudo code appears at Listing 2 Line 38. However, the implementation does *not* return the latest operation in the non-fuzzy part of the execution trace since it does not guarantee an accurate snapshot of the execution trace. It is possible that while traversing the operations with available flag unset, the availability of a later operation was set. In fact, it is possible that the returned node was never the latest operation in the non-fuzzy part. We do not attempt to compute a snapshot of the non-fuzzy window since ONLL is correct despite this anomaly, as is described later.

Listing 2. Execution trace

```
1  struct queueNode{
2      operation op;
3      long idx;
4      bool available;
5      queueNode *next;
6      set<operation> getFuzzyOps(){
7          queueNode *curr=tail;
8          set<operation> ops;
9          while(curr−>available==false){
10             ops.add(curr−>op);
11             curr=curr−>next;
12         }
13         return ops;
14     }
15     queueNode *latestAvailable(){
16         queueNode *curr=tail;
17         while(curr−>available==false){
18             curr=curr−>next;
19         }
20         return curr;
21 }}
22
23 class executionTrace{
24     queueNode *tail;
25     executionTrace(){
26         tail=new operation(INITIALIZE, 0, true, null);
27         //also serves as a sentinel
28     }
29     void insert(queueNode *node){
30         queueNode *ltail;
31         do{
32             ltail = tail;
33             node−>available = false;
34             node−>idx = ltail−>idx+1;
35             node−>next = tail;
36         }while(CAS(&tail, ltail, node)==false);
37     }
38     queueNode *latestAvailable(){
39         queueNode *curr=tail;
40         return curr−>latestAvailable();
41 }}
```

4.2 ONLL Algorithm

Our algorithms for updating the object and reading the object are presented in Listing 3 and Listing 4, respectively.

An update operation starts by adding a new node to the execution trace at Line 3. This corresponds to setting the linearization order of the update operation without making it visible to read-only operations and without linearizing it. At Line 5, the fuzzy window of the operation is computed. This corresponds to the set of operations preceding the current operation but are not yet guaranteed to be persistent. (We later show — Proposition 5.2 — that this operation is finite since there are at most MAX-THREADS nodes in the fuzzy

Listing 3. ONLL-update

```
1  Update(operation op){
2      queueNode *node = new queueNode(op);
3      executionTrace.insert(node);
4      operation fuzzyOps[MAX_THREADS] =
5                                  node−>getFuzzyOps();
6      persistentLog.append(fuzzyOps, node−>idx);
7      node−>available=true;
8      //write should not interleave with instructions
9      //executed after the operation finishes
10     memory_fence(sequential_consistency);
11     return compute(node, op);
12 }
```

window). Helping these operations to persist on the NVM prevents waiting for an unresponsive process. Then, the current operation and the helped operations are persisted by appending them to the private, persistent log (Line 6). The update part finishes by writing the available flag and executing a memory fence (standard concurrency memory fence, not a persistent fence). This corresponds to the linearization point of the operation (unless another thread helped it) and makes the operation visible to read-only operations. Finally, if the update operation also returns a value, this value is computed and returned to the caller.

Listing 4. ONLL-read

```
Read(operation op){
    queueNode *node = executionTrace.latestAvailable();
    return compute(node, op);
}
```

A read-only operation gets the latest node with available flag set. This node correspond to the latest node in the non-fuzzy prefix of the object state. Then, the return value is computed based on this state and returned to the caller.

After a system crash, the transient execution trace is reconstructed from the persistent logs of all processes.

Listing 5. ONLL-recovery

```
    executionTrace.push(INIT, 0).setAvailable();
    for(i=1; true; i++){
        Find log entry E with lowest execution index $j : j \geq i$.
        if(E does not exist)
            break;
        operation op=E.ops[j−i];
        executionTrace.push(op).setAvailable();
    }
```

The recovery process starts by adding the initialization operation to the execution trace, which serves as a sentinel node in the execution trace. Then, it iteratively searches for the next operation in the execution trace by looking into the persistent logs of all processes. If the operation *op* was not stored into any persistent log, the recovery process looks for an operation with a higher execution index and finds *op* by looking into the helped operations by the later operation. Finally, the found operation is pushed to the execution trace and the available flag is set.

5 ONLL: Correctness

In this section, we prove the following theorem.

Theorem 5.1. *For any deterministic object O, there exists a lock-free durably linearizable implementation of O that requires at most one persistent fence per update operation and no persistent fence per read-only operation.*

We prove the theorem by first showing that ONLL is lock-free and then that it is durably linearizable.

5.1 Lock-freedom

An implementation is *lock-free* if it guarantees that infinitely often, some operation returns in a finite number of steps. More specifically, if any process is permanently taking steps, some operation will eventually return.

The lock-freedom proof uses the following proposition, showing that traversing the size of a fuzzy window is bounded regardless of the initial node.

Proposition 5.2. *At any time during any execution of* ONLL *and any MAX-THREADS+1 consecutive nodes in the execution trace, at least one has an available flag that is set.*

Proof. Let t be any time point during the execution and let $S = \{n_i, \ldots, n_{i+MAX-THREADS}\}$ be MAX-THREADS+1 consecutive nodes in the execution trace. Clearly, there are at least two nodes $n_j, n_k \in S$ that correspond to two operations by the same process.

In our model, two operations cannot be executed by the same process at the same time: the first operation must return before the second operation can be invoked. Thus, let n_j be the earlier operation and the t' be the time it finished. Clearly, $t' < t$ since at time t the operation corresponding to n_k already appended itself to the execution trace, implying that it was already invoked.

According to Listing 3, an operation does not finish before setting the available flag and executing a memory fence. Thus, at least n_j has a set available flag, as required. □

Recall that a set available flag is never unset; this property together with Proposition 5.2 imply that `getFuzzyOps` and `latestAvailable` are wait-free operations since they always finish in O(MAX-THREADS) steps and cannot be interfered with by other processes.

Lemma 5.3. *Suppose that compute — the function for computing the return value of an operation — always finishes in a finite time. Then* ONLL *is lock-free.*

Proof. Reads first find the `latestAvailable` node and then execute *compute* on the resulting node. `latestAvailable` finishes in a bounded number of steps and is thus wait-free. *Compute* operates on a prefix of the execution trace starting with the `latestAvailable` node. This prefix is never modified by any thread (except for the available flag, which is ignored by *compute*). Since we assume *compute* finishes in a bounded number of steps, it is also wait-free. Thus, reads are wait-free.

Next we consider updates. Appending to the execution trace is a lock-free operation. Getting the fuzzy window of an operation is wait-free since it always finishes in a bounded time. Appending to the thread's private persistent log is also wait-free since an append is never interrupted by other processes and it finishes in a bounded number of steps. Finally, setting the available flag of a node is a wait-free operation. Thus, we conclude that updates are lock-free.

□

5.2 Durable linearizability

We first recall the concept of durable linearizability and then proceed with the proof of durable linearizability in ONLL.

5.2.1 Technical Preliminaries

The execution of a concurrent system is modeled by a *history*, a sequence of events. Events can be operation *invocations* and *responses*. Each event is labeled with the process and with the object to which it pertains. A *subhistory* of a history H is a subsequence of the events in H.

A response *matches* an invocation if they are performed by the same process on the same object. An operation in a history H consists of an invocation and the next matching response. An invocation is *pending* in H if no matching response follows it in H. An *extension* of H is a history obtained by appending responses to zero or more pending invocations in H. $complete(H)$ denotes the subhistory of H containing all matching invocations and responses.

For a process p, the *process subhistory* $H|p$ is the subhistory of H containing all events labeled with p. The object subhistory $H|O$ is similarly defined for an object O. Two histories H and H' are equivalent if for every process p, $H|p = H'|p$.

A history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. A history is *well-formed* if each process subhistory is sequential.

A *sequential specification* of an object O is a set of sequential histories called *legal* histories of O. A sequential history H is *legal* if for each object O appearing in H, $H|O$ is legal.

An operation op_1 *precedes* op_2 in H (denoted $op_1 \rightarrow_H op_2$) if op_1's response event appears in H before op_2's invocation event. Precedence defines a partial order on the operations of H.

Definition 5.4 (Linearizability)**.** A history H is *linearizable* if H has an extension H' and there is a legal sequential subhistory S such that

L1 $complete(H')$ is equivalent to S

L2 if an operation op_1 precedes an operation op_2 in H, then the same holds in S.

Informally, this definition is equivalent to saying that an object is linearizable if every operation appears to take effect instantaneously at some point (the *linearization point*) between invocation and response. Incomplete operations (invocations without matching responses) may or may not have a linearization point.

Durable linearizability [28] captures the fact that an object's state should remain consistent even across crashes and recoveries, without "erasing" any completed operations.

Definition 5.5 (Consistent cut)**.** Given a history H, a consistent cut of H is a subhistory P of H such that if $op_2 \in P$ and $op_1 \rightarrow_H op_2$, then $op_1 \in P$ and $op_1 \rightarrow_P op_2$.

Definition 5.6 (Durable linearizability)**.** An object O is *durably linearizable* if its states immediately before and immediately after a crash and recovery reflect histories H and H' respectively such that (1) H and H' are linearizable and (2) H' is a consistent cut of H for which every complete operation op in H is also in H'.

In other words, all operations that completed before the crash must be included in H', but some operations that had not yet completed may be excluded and thus not be reflected in the post-recovery state of the object. However, the operations in H' must constitute a consistent cut of H, meaning that if some operation op is included in H', then so must all operations on which op depends.

Informally, an object O is durably linearizable if, in any history H produced by O, every operation appears to take effect instantaneously at some point (the *linearization point*) between invocation and response. Incomplete operations (invocations without matching responses, either due to delayed processes or to system crashes) may or may not have linearization points. Operations concurrent with a crash may be reflected in the post-crash state of the object (in which case these operations have linearization points before the crash) or may not be reflected (in which case these operations have no linearization points).

5.2.2 Durable Linearizability in ONLL

Lemma 5.7. ONLL *is durably linearizable.*

The proof of durably linearizable for ONLL proceeds in 4 steps. First, we define linearization points for all update operations and define a time point for a crash (if a crash occurs). Second, we prove that the linearization point of an update falls between its invocation and response. Third, we show that a read is linearizable. Fourth, we show that the state of the object after crash and recovery matches the state of the object before recovery, according to the update operations linearized before the crash.

We start by defining a point in time where each update operation linearizes. When defining linearization points, we use the following convention. An integral time, denoted by $t_\square$, corresponds to a specific instruction executed on the durable shared object; a non-integral (i.e., fractional) time, denoted by $t_\square - a \cdot \epsilon$ denote a linearization point that happens before time $t_\square$ but does not relate to a specific event during execution. We assume ϵ is sufficiently small and a is a positive finite number so that $t_\square - 1 < t_\square - a \cdot \epsilon < t_\square$.

The linearization point of an update operation op with execution index i is the earlier between (1) the time t_i the i-th available flag was set at the end of op or (2) immediately before the time t_j when the j-th available flag was set for $j > i$. To avoid distinguishing between these two cases, we consider the first j-th available flag that was set such that $j \geq i$. The linearization point of op_i is $t_i = t_j - (j - i) \cdot \epsilon$ for a sufficiently small $\epsilon > 0$.

If a crash happens, let t_{crash} be the time of the crash; we assume this time is higher by at least one than the last instruction executed before the crash. Clearly, operations that returned before the crash are included in the post-crash state of the object. We now examine operations that were ongoing at t_{crash}. Let op_i be such an ongoing operation, with execution index i. We examine several cases:

1. Some operation $op_j : j \geq i$ persisted op_i and op_j set its available flag at time t_j. Then op_i is linearized at time $t_i = t_j - (j - i) \cdot \epsilon$ as discussed above.
2. op_i either (a) persisted itself, but did not set its flag, or (b) was persisted by one or more other operations (Listing 3 Line 6), but none of these operations set their flag. To establish the linearization point of op_i, let op_l be the operation with highest execution index (= l) that finished persisting before t_{crash}. op_i is linearized at $t_i = t_{crash} - (l - i) \cdot \epsilon$.
3. op_i was not persisted at all. That is, no operation $op_j : j \geq i$ finished appending to the persistent log at Listing 3 Line 6. Then, op_i is not linearized and is lost in the crash.

Proposition 5.8. *The linearization point of an update operation falls between the invocation and the response of this operation.*

Proof. Consider an update operation op_i and let $j \geq i$ be the first index such that the j-th available flag that was set. The linearization point of op_i is $t_j - (j - i) \cdot \epsilon$.

When op_i is inserted in the execution trace, the latest operation in the trace is $i - 1$; clearly, op_j is inserted in the execution trace not earlier than op_i was inserted in the execution trace (note that i can be equal to j, in which case the times are equal). Setting the j-th available flag is done at time t_j. Inserting the j-th node to the execution trace happens earlier and is related to the execution of an actual instruction; thus, its time is at most $t_j - 1$. Recall that ϵ is small enough so that $t_j - 1 < t_j - (j - i) \cdot \epsilon$. Thus, op_i is inserted in the execution trace before time $t_j - (j - i) \cdot \epsilon$. op_i is invoked before is it inserted in the execution trace, establishing that the linearization point of op_i happens after its invocation.

Operation i does not finish before setting the i-th available flag. Clearly, the i-th available flag is set not earlier than the first j-th available flag is set, $j \geq i$. By definition of the linearization points, setting the j-th available flag happens at time t_j. Thus, the response to op_i happens after time $t_j \geq t_j - (j - i) \cdot \epsilon$. □

Proposition 5.9. *A read-only operation has a linearization point between the invocation and the response of the operation, such that the return value of the operation corresponds to the state of the object at the linearization point.*

Proof. A read-only operation traverses the execution trace from the tail until it reaches an entry with a set available flag. The resulting entry is the state on which the returned value is computed. Suppose that the tail pointed to op_j and the highest index operation with available flag set is $op_i : i \leq j$. Consider the time t the read-only operation reads the tail; either the i-th available flag was set at time t or not. If the i-th available flag was set at time t, then the linearization point of the read is set to time t. This clearly falls between the invocation and the response. The state of the object at time t contains op_i since its available flag is set, but not any operation in the range $[i + 1, j]$ since their available flag is unset at time t. The latter is true since otherwise the traversal would find a later node $k \geq i + 1$ with a set available flag.

Next, consider the case that the i-th available flag was not set at time t. Denote by t_e the time the read-only operation finds that the i-th available flag is set. At time t all operations in the range $[i, j]$ have unset available flag and at time t_e the i-th available flag is set. Thus, the linearization point of op_i falls between time t_e and t. We set the linearization time of the read-only operation to immediately after the linearization time of op_i and before the linearization point of any other update operation. Since the linearization point is between time t_e and t, it is after the invocation of the read-only operation and before the response, as required. □

Proposition 5.10. *The state of the* ONLL *object after a crash includes all the operations that were linearized before the crash, executed in linearization order, and none of the operations that were not linearized before the crash.*

Proof. By definition of the linearization points before a crash, the state of the object just before t_{crash} corresponds to the last operation that was written to the persistent log. After a crash, the recovery reconstructs the execution trace by traversing all persistent logs. Thus, the last entry in the execution trace after recovery is the last operation that was written to the persistent log. The order of operations follows the `executionIndex`, which is stored on the persistent logs. Thus, the order of operations is equal before the crash and after recovery. It remains to show that all operations appearing in the execution trace before the crash also appear after recovery.

Suppose, by a way of contradiction, that an operation i that appeared before the crash does not appear after the crash. Since the latest operation is the same, there exists an operation $op_j : j > i$ that appears both before the crash and after recovery. We pick the operation with smallest j (that is larger than i). Operations that have a set available flag clearly appear in the log. Thus, all operations in the range $[i, j-1]$ must have their available flag unset until t_{crash}. But since op_j persisted in the persistent log before t_{crash}, it must have been added to the execution trace before t_{crash}. According to Proposition 5.2, there are at most MAX-THREADS - 1 operations between op_i and op_{j-1} since the available flag of op_j is unset when it is appended to the execution trace.

But op_j appended to the persistent log entry all operations in the range $[i, j]$, including op_i, so it appears in a persistent log. This contradict our assumption that op_i is not available on the persistent log. ◻

The proof of Lemma 5.7 follows from Propositions 5.8, Proposition 5.9 and Proposition 5.10. The proof of Theorem 5.1 follows from Lemma 5.3 and Lemma 5.7.

Interestingly, ONLL also provides detectable execution [14]. After recovery, it is possible to check whether a given operation appears in the execution trace or not. The operation was linearized before the crash if and only if it appears in the execution trace after recovery.

6 Lower Bound

We show in this section that in any lock-free implementation of a durably linearizable object, there exists some execution that forces every update operation to issue at least one persistent fence. This is trivially true if operations are executed sequentially (otherwise a crash immediately after an operation *op* would mean that *op* is not reflected in the state of the object after recovery). Intuitively, the need for processes to persist their operations also manifests in some concurrent executions: if some process p were to always rely on other processes to persist their operations, then p might need to wait indefinitely if those other processes are delayed, thus violating lock-freedom.

We prove this intuition below, through several intermediate results, after defining relevant terminology.

Terminology. We say that a process p *runs solo* between events A and B in an execution if p is the only process taking steps between A and B in that execution. A *state* is defined by a sequence of update operations, starting with INITIALIZE. Two states H_1 and H_2 are *equivalent* (denoted $H_1 \equiv H_2$) if any possible execution, when started from H_1 produces the same results (operation return values) as when started from H_2. We denote by $H \cdot op$ the state obtained by executing operation *op* starting from state H. An operation *op* is an *update* if there exists a state H such that $H \cdot op \not\equiv H$.

Lemma 6.1. *Let op be any update and H be any state such that $H \cdot op \not\equiv H$. Then, if for some $n \geq 2$, $H \cdot op^{n-1} \not\equiv H \cdot op^n$. Then, for all $0 \leq i \leq n-1$, $H \cdot op^i \not\equiv H \cdot op^{i+1}$.*

Proof. Assume towards a contradiction that there is an i, $0 \leq i \leq n-1$ such that $H \cdot op^i \equiv H \cdot op^{i+1}$. Applying *op* $n-i-1$ times to each side of the equivalence we obtain $H \cdot op^{n-1} \equiv H \cdot op^n$, a contradiction. ◻

Lemma 6.2. *Let op be any update and H be any state such that $H \cdot op \not\equiv H$. Then, if for some $n \geq 2$, $H \cdot op^{n-1} \equiv H \cdot op^n$, the following property holds: $\forall j \geq 1, H \not\equiv H \cdot op^j$.*

Proof. Assume by contradiction that $\exists j \geq 1 : H \equiv H \cdot op^j$. Then for all $k \in \mathbb{N}, H \cdot op^{kj} \equiv H$. Let k_n be an integer such that $(k_n - 1)j \leq n-1 < k_n j$. Then (1) $H \cdot op^{n-1} \cdot op^{k_n j-(n-1)} \equiv H$ and (2) $H \cdot op^n \cdot op^{k_n j-(n-1)} \equiv H \cdot op$. The left sides of (1) and (2) are equivalent (because $H \cdot op^{n-1} \equiv H \cdot op^n$), but the right sides are not equivalent. We have reached a contradiction. ◻

Theorem 6.3. *In a n-process system, for any lock-free durably linearizable implementation of an update operation op, there exists an execution in which (1) all processes call op concurrently and (2) each process performs at least one persistent fence during its call to op.*

Proof. By definition of an update operation, there exists a state H such that *op* applied from H produces a different state $H \cdot op \not\equiv H$.

We now consider two cases: (1) $H \cdot op^{n-1} \not\equiv H \cdot op^n$ and (2) $H \cdot op^{n-1} \equiv H \cdot op^n$. For each case, we construct an execution in which the n processes $p_1, ..., p_n$ call *op* concurrently and all necessarily perform persistent fences.

Case 1: $H \cdot op^{n-1} \not\equiv H \cdot op^n$. We construct the following execution:

- Starting from H, let p_1 call *op* and run solo until just before the response of *op*. p_1 will eventually reach this point, due to the lock-freedom of the implementation. p_1 will perform at least one persistent fence before being preempted. Otherwise, let p_1 resume and perform the very next step of returning from *op*; if a crash occurs after this response, after recovery the contents of persistent memory will be identical to that at H, which is inconsistent with the only possible linearization $H \cdot op \not\equiv H$.
- Let p_2 call *op* and run solo until just before *op* returns (p_2 eventually returns due to the lock-freedom of the implementation). p_2 performs at least one persistent fence during its call to *op*. Otherwise, let p_2 return from *op* and let p_1 resume and perform the step of returning from *op*. If a crash occurs immediately after, at recovery the contents of memory will be identical to $H \cdot op \not\equiv H$, which is incompatible to the only possible linearization $H \cdot op \cdot op$, due to Lemma 6.1.

- Continue in this way with $p_3, ..., p_n$, each time calling *op* and preempting the process just before returning. As with p_2, each process will perform at least one persistent fence before being preempted.

Case 2: $H \cdot op^{n-1} \equiv H \cdot op^n$. We construct the following execution:

- Starting from H, let p_1 call *op* and run solo. If left to run solo long enough, p_1 will eventually perform a persistent fence. Otherwise, p_1 either never returns from *op*, violating lock-freedom, or returns from *op* without having performed a persistent fence. In the latter case, a crash may occur immediately after the response of *op*; upon recovery, the contents of persistent memory will be identical to those at H, which is inconsistent with the fact that $H \cdot op \not\equiv H$.
- Preempt p_1 just before the first persistent fence.
- Let p_2 call *op* and run solo. If left to run solo long enough, p_2 will eventually perform a fence. Otherwise, if p_2 returns without a fence and the system crashes afterwards, the contents of persistent memory are identical to H which is inconsistent with all the possible linearizations H, $H \cdot op$, $H \cdot op \cdot op$ due to Lemma 6.2.
- Continue in this way with processes $p_3, ..p_n$.
- For each process $p_n, ...p_1$, resume the process for one step—the persistent fence it was about to perform—then preempt it and move to the next process.

□

Our lower bound result holds for detectable execution [14] as well, since it is a stronger criterion than durable linearizability (an implementation satisfying the former requires at least as many persistent fences as an implementation satisfying the latter).

7 Related Work

Safety criteria. Several safety criteria have been proposed in the crash-recovery model. Persistent atomicity [17] requires any operation interrupted by a crash to be linearized or aborted before any later invocation by the pending process. In the same situation, recoverable linearizability [4] requires the operation to be linearized or aborted before any later invocation by the pending process on the same object. These two criteria assume that processes may crash and recover independently. However, it can be argued that this model is unnecessarily general, since processes typically crash together in a full-system crash (e.g., restart). In a more restricted model that only allows such full-system crashes, the two criteria become indistinguishable, and equivalent to durable linearizability [28], the safety condition adopted in this paper.

Work has been done on verifying linearizability for traditional transient objects [15, 34, 41]. It would be interesting to see if such verification techniques could be extended to verify durable linearizability for persistent objects.

Our upper bound algorithm relates in an interesting way to Section 4.1 of Izraelevitz et al. [28]. In that section, the authors state that the linearization point of an operation must happen before it is persisted. However, our ONLL construction linearizes an operation *op after* the time when *op* persisted and generates the linearization order before persisting, so that it knows what to persist. In fact, as discussed in Section 3, we argue that having the persist point earlier than the linearization point is necessary for any lock-free algorithm where readers never execute a persistent fence.

The same section of [28] contains a theorem (Theorem 2), which provides a set of sufficient conditions for an object to be durably linearizable. While at first glance it may seem that our ONLL algorithm contradicts this theorem, this is not the case: ONLL is durably linearizable, without satisfying the condition that operations are linearized before they are persisted.

General transformations. Related to the generality of our upper bound construction, there has been work [7, 9, 23] on generating correct persistent applications from existing code (designed for DRAM). However, in contrast to our work, these approaches assume the application is already multi-threaded and generally also assume lock-based code. Moreover, the focus in this work [7, 9, 23] is on lessening the programming effort necessary to transform applications, not on achieving optimality in terms of the number of persistent fences used.

In the same vein of generality, our work shares similarities with the universal construction of Herlihy [19, 20]. In both cases, the construction yields a correct concurrent implementation of an object from its sequential specification.

Transactions. Significant work has been done on transactions as a means of interacting with NVM [5, 8, 11, 13, 16, 27, 29, 31, 35, 36, 44]. These efforts share similarities to our work in the following sense: they strive for generality, they aim to reduce the cost of interacting with NVM, and they often use logging. Yet, these works do not consider lock-freedom as a progress guarantee. Also, whereas in transactions logging is used to help maintain the consistency of application state, in our construction, the log *is* the state.

Persistent data structures. A specific class of shared objects are concurrent data structures [21]. There has been some work on designing efficient data structures for NVM. This work mostly focuses on indexing trees [10, 33, 39, 45]. This is natural, given that indexing trees are used extensively in data structures and file systems. Recently, Friedman et al. [14] have proposed three lock-free durable queue algorithms. These are specific approaches, not easily generalizable to other data structures or to other shared objects.

Lower bounds. Related to our lower bound result, Attiya et al. [1] have shown that linearizable implementations of strongly non-commutative operations cannot completely eliminate the use of expensive synchronization primitives such as memory barriers and atomic instructions (whose effects also include the effects of memory fences). This seems to imply that any implementation of a durably linearizable update operation requires (at least in some executions) two fences: one to account for the cited lower bound and one to account for our lower bound. However, since the effects of a memory fence also include stalling until pending cache line flushes have completed, a memory fence can also count as a persistent fence if flushes to NVM are pending. Thus, it might be possible in some cases to implement a persistent object using only one (memory) fence per update operation, accounting for both our lower bound result and that of Attiya et al. We leave open the questions of when and if such one-fence updates are indeed achievable.

8 Concluding Remarks

In the NVM era, programmers will need persistent and concurrent data structures. The performance of these is significantly influenced by the number of persistent fences executed for each operation. This paper shows that lock-free implementations require exactly one persistent fence for any update operation to ensure correctness. Our upper bound uses a novel ordering scheme to persist operations before their linearization points. Our lower bound captures the very fact that processes cannot rely on each other to persist updates and thus shows that one cannot hope to reduce the number of persistent fences while still guaranteeing durable linearizability and lock-freedom.

Below, we discuss possible extensions of our results and future directions left open by our work.

Wait-freedom. According to the proof of Lemma 5.3, the only operation in our ONLL construction that is not wait-free is the execution trace transient data structure. Since this data structure is transient, standard techniques such as the wait-free construction of Timnat and Petrank [43] can be used to derive a wait-free execution trace data structure. Alternatively, a wait-free execution trace can be based on the wait-free queue of Kogan et al. [30]. ONLL can thus easily be made wait-free.

Compressing the execution trace. An ONLL object stores its state as a sequence of all operations applied to this object. This representation could be improved for specific cases, as most practical objects have an object-specific representation of their state. For example, for the shared persistent counter discussed in Section 3, an object-specific representation would be an integer field corresponding to the current value of the counter.

If such a representation exists, one could consider a hybrid approach that combines a small ONLL execution trace for correctness with an object-specific representation for efficiency. As explained below, this approach would have the double benefit of (1) allowing better read performance and (2) enabling memory reclamation, thus reducing memory consumption.

Readers in ONLL traverse the entire execution trace; thus, reading an object's state implies traversing all update operations in the history of the object. ONLL read performance can be significantly improved by storing a local view per process, similarly to log-based systems [2, 3, 6]. The local view of process p includes (1) a representation r_p of the object up to some operation op_p and (2) the execution index of op.

A read by process p begins as before by finding the first execution trace node n with a set available flag. Then, p applies to its local representation r_p all updates between op_p and n. Then, p updates its local execution index to that of n. Finally, the read is served directly from r_p.

In this way, the overhead of a read is the difference between the execution index of the local view and the execution index of the shared object, which is expected to be significantly smaller than the number of nodes in the execution trace.

Another effect of storing the entire execution trace in ONLL is the inability to reclaim memory. Both execution trace nodes and persistent log entries have to be kept forever. If the state can be stored as a small object-specific representation, however, then there is no need to remember all update operations and log entries, thus significantly reducing memory consumption.

Execution trace nodes can be reclaimed if we use the following scheme. As before, each process p has a local transient representation of the object r_p, which p brings up to date periodically. Note that once a process p has applied an operation op from the execution trace to r_p, p will never need to read op again. Thus, once all processes have updated their local representations past op, the execution trace prefix up to op can be safely reclaimed.

We can go one step further and also reclaim persistent log entries. Each process p periodically records its local representation r_p in its persistent log, along with the execution index n of op_p. Afterwards, p can reclaim the memory of all persistent log entries with execution indexes smaller than n.

Lock-based implementations. At first glance, it might seem that by allowing implementations of persistent objects to be blocking, one could reduce the number of persistent fence instructions. For instance, the work of Cohen et al. [12] enables an implementation in which each process announces its operation and one of the processes applies all announced operations (similarly to flat combining [18]) using a single persistent fence. This implementation might seem to use only one persistent fence for every batch of concurrent operations. However, upon closer inspection, it is easy to realize that all pending operations pay the price of a persistent fence (by waiting while the combiner performs the fence), even without actually performing the fence.

References

[1] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. POPL 2011.

[2] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A Distributed Shared Log. TOCS 2013.

[3] Mahesh Balakrishnan, Aviad Zuck, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, and Tao Zou. Tango: Distributed Data Structures Over a Shared Log. SOSP 2013.

[4] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. OPODIS 2015.

[5] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence Programming Models for Non-volatile Memory. ISMM 2016.

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. ASPLOS 2017.

[7] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. OOPSLA 2014.

[8] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery Write-ahead System for In-memory Non-volatile Data-structures. VLDB 2015.

[9] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping Programmers Move to Byte-Based Persistence. INFLOW 2016.

[10] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. VLDB 2015.

[11] 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. ASPLOS 2011.

[12] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient Logging in Non-volatile Memory by Exploiting Coherency Protocols. OOPSLA 2017.

[13] Joel Edward Denny, Seyong Lee, and Jeffrey S. Vetter. Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems. IPDPS 2017.

[14] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory. DISC 2017.

[15] Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic Scalable Atomicity via Semantic Locking. *ACM Transactions on Parallel Computing*, 3(4), 2017.

[16] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-memory Database. SYSTOR 2016.

[17] Rachid Guerraoui and Ron R Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. ICDCS 2004.

[18] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA 2010.

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

[20] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 1993.

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

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

[23] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. EuroSys 2017.

[24] Intel. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[25] Intel. Intel64 and IA-32 Architectures Optimization Reference Manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[26] Intel. Intel64 and IA-32 Architectures Software Developers Manuals Combined. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[27] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic Persistent Memory Updates via JUSTDO Logging. ASPLOS 2016.

[28] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. DISC 2016.

[29] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-ahead Logging. ASPLOS 2016.

[30] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. PPoPP 2011.

[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance Transactions for Persistent Memories. ASPLOS 2016.

[32] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[33] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. FAST 2017.

[34] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Automatic Atomicity Verification for Clients of Concurrent Data Structures. CAV 2014.

[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. ASPLOS 2017.

[36] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys 2017.

[37] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC 1996.

[38] Micron. 3D XPoint Technbology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[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. SIGMOD 2016.

[40] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[41] Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. Verifying Atomicity via Data Independence. ISSTA 2014.

[42] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80, 2008.

[43] Shahar Timnat and Erez Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. PPoPP 2014.

[44] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. ASPLOS 2011.

[45] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. FAST 2015.

[46] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. MSSR 2015.