NVTraverse: In NVRAM Data Structures, the Destination Is More Important Than the Journey

Michal Friedman∗
Technion, Israel
michal.f@cs.technion.ac.il

Naama Ben-David∗
CMU, USA
nbendavi@cs.cmu.edu

Yuanhao Wei∗
CMU, USA
yuanhao1@cs.cmu.edu

Guy E. Blelloch
CMU, USA
guyb@cs.cmu.edu

Erez Petrank
Technion, Israel
erez@cs.technion.ac.il

Abstract
The recent availability of fast, dense, byte-addressable non-volatile memory has led to increasing interest in the problem of designing durable data structures that can recover from system crashes. However, designing durable concurrent data structures that are correct and efficient has proven to be very difficult, leading to many inefficient or incorrect algorithms. In this paper, we present a general transformation that takes a lock-free data structure from a general class called traversal data structure (that we formally define) and automatically transforms it into an implementation of the data structure for the NVRAM setting that is provably durably linearizable and highly efficient. The transformation hinges on the observation that many data structure operations begin with a traversal phase that does not need to be persisted, and thus we only begin persisting when the traversal reaches its destination. We demonstrate the transformation’s efficiency through extensive measurements on a system with Intel’s recently released Optane DC persistent memory, showing that it can outperform competitors on many workloads.

CCS Concepts: • Computing methodologies → Concurrent algorithms; • Information systems → Data structures; • Hardware → Non-volatile memory.

Keywords: Non-volatile Memory, Concurrent Data Structures, Non-blocking, Lock-free

∗The first three authors contributed equally to this work. Their order of appearance in the paper was selected randomly.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 Introduction
Now that non-volatile random access memory (NVRAM) has finally hit the market, the question of how to best make use of it is more pressing than ever. NVRAM offers byte-addressable persistent memory at speeds comparable with DRAM. This memory technology now can co-exist with DRAM on the newest Intel machines, and may largely replace DRAM in the future. Upon a system crash in a machine with NVRAM, data stored in main memory will not be lost. However, without further technological advancements, caches and registers are expected to remain volatile, losing their contents upon a crash. Thus, NVRAM yields a new model and opportunity for programs running on such machines—how can we take advantage of persistent main memory to recover a program after a crash, despite losing values in cache and registers?

One challenge of using NVRAM is that a system crash may occur part way through a large update, leaving the memory with some, but not all, of the changes that should have been executed. In some concurrent lock-based programs, the state of memory after a partial update may not be consistent, and may be unsafe for other processes to observe. Furthermore, without knowing the entire update operation and what changes it should have made, it may be impossible to return the memory to a consistent state. This may require heavy-duty mechanisms, like logging or copying.

Interestingly, a well-studied class of programs called lock-free algorithms ensures that the memory is always in a consistent state, even during long updates. In a nutshell, lock-freedom requires that processes be able to execute operations on the shared state regardless of the slow progress (or failure) of others in the system. Thus, lock-free algorithms are a very natural fit for use in NVRAM.
However, using NVRAM introduces still more challenges. Because of their small size, caches inherently require evicting cache lines back to main memory. On modern caches, these evictions are performed automatically when needed, thus offering a fast, transparent interface for their user. Yet, when main memory is persistent, this can create complications; it is possible that values written to cache later in a program get evicted earlier than others, thus making main memory hold out-of-order values. When a system crash occurs, this can leave the memory in an inconsistent state, from which it may not be possible to recover. This problem is especially challenging when multiple processes access the same memory locations. Lock-free algorithms are generally not designed to handle such reordering of memory updates.

To fix this, explicit flushes and fences can be introduced into programs to force certain changes to appear in main memory before others. In particular, if we add a flush and a fence instruction between every two synchronized instructions of a process, main memory remains consistent. Izraelevitz et al. [27] formalized this intuition and showed that this technique indeed leads to a consistent memory state in all lock-free programs, regardless of crashes. Thus, many theoretical papers have focused on a model in which changes to the memory are assumed to be persisted immediately, and always in the order they occur [3–6]. However, fences are notoriously expensive, causing this approach to be prohibitively slow, despite guaranteeing correctness.

A lot of research has instead focused on decreasing the amount of flushing needed during regular execution to be able to recover [10, 11, 30, 32, 42]. However, the notion of “being able to recover” is flexible: do we allow the loss of some progress? Without defining this clearly, unexpected behavior can result from algorithms that are seemingly ‘correct’. Significant work has also focused on defining these goals [1, 11, 23]. Izraelevitz et al. [27] introduced the notion of durable linearizability. In a nutshell, a concurrent data structure is said to be durably linearizable if all executions on it are linearizable once crash events are removed. This disallows the state of the data structure in memory to be corrupted by a crash, and does not allow the effect of completed operations to be lost. In their paper, Izraelevitz et al. posit that durable linearizability may be prohibitively expensive.

Since then, several works presented hand-tuned algorithms that achieve durable linearizability while performing reasonably well [18, 21]. However, they only show specific data structures, and do not yield a general way of designing practical durably linearizable data structures. The difficulty of this task can be traced back to understanding the dependencies between operations in the algorithm. Only expertise about persistence, combined with a deep understanding of an algorithm and careful reasoning about its inherent dependencies, has so far been productive in finding correct and more efficient solutions.

Our contributions. In this paper, we provide a technique that achieves the best of both worlds for a large class of lock-free data structure implementations; we show an automatic transformation that can be applied to lock-free algorithms of a certain form, and makes such data structures persistent and efficient, with well-defined correctness guarantees that are provably correct. We take a substantial step in removing the need for expert familiarity with an algorithm to make an efficient durably linearizable version of it. Our key insight is that many lock-free concurrent data structure implementations begin operations with a traversal of the data structure, and that, intuitively, the values read along this traversal do not affect the operation’s behavior after the traversal.

We formalize a large class of lock-free linearizable algorithms we call traversal data structures. Traversal data structures are node-based tree data structures whose operations first traverse the data structure, and then perform modifications on nodes that descend from where their traversal stopped. The traversal is guaranteed to only make local decisions at every point in time, not relying on previous nodes to determine how to proceed from the current one. These algorithms also follow some natural rules when removing nodes from the data structure.

We show that many existing pragmatic concurrent algorithms can easily be converted into traversal data structures, without losing their efficiency. We use Harris’s linked-list [24] as a running example throughout the paper to help with exposition. Other data structures implementations, like common BST, (a,b)-tree, and hash table algorithms [7, 8, 20], can also be easily converted. Furthermore, traversal data structures capture not just set data structures, but also queues, stacks, priority queues, skiplists, augmented trees, and others. Most requirements of traversal data structures are naturally satisfied by many lock-free data structures. Thus, it is not hard to transform a new data structure implementation into a traversal data structure.

After defining traversal data structures, we show how to automatically inject flush and fence instructions into a traversal data structure to make it durably linearizable. The key benefit of our approach is that no flushes are needed during most of the traversal. Of all memory locations that are read, only a few at the end of the traversal must be flushed. Because of the careful way in which the traversal is defined, these can be automatically identified. After the traversal, all further locations that the operation accesses must be flushed. However, in most operations, the traversal encapsulates a large majority of the work.

We formally prove that the flushes and fences that we specify are sufficient for durability for all traversal data structures. Thus, this paper presents the first practical, provably correct implementation of many durable data structures; the only previously known durable algorithm that was proven correct is the DurableQueue of [21].
Finally, we experimentally evaluate the algorithms that result from our transformation, by transforming a list, two binary search trees, a skip list, and a hash table to the traversal form and then injecting flushes automatically to obtain a durable data structure. We compare our implementations to those that result from the general transformation of Izraelevitz et al. [27], the original (non-persistent) version of each algorithm, the OneFile transactional memory [41], and the hand-tuned durable version presented by David et al. [18]. We reclaim objects using simple epoch-based memory management. Our results show that persisted traversal data structures, or NVTraverse data structures, outperform Izraelevitz et al. [27]’s construction significantly on all workloads. Thus, NVTraverse data structures are a much better alternative as a general transformation for concurrent data structures to become durable. Furthermore, NVTraverse data structures outperform David et al. [18]’s data structures on about half of the workloads; those with lower thread counts or larger data structure sizes. This provides insights on the tradeoffs between flushes, fences, and writes under contention.

Using the method proposed in this work involves two steps. The first step (which is manual) involves making sure that the target lock-free data structure is in traversal format. The second step (which is automatic) involves adding flushes and fences to make the lock-free data structure durable. The second step is the major contribution of this work, because it spares programmers the effort of reasoning about persistence. On the other hand, the definition of traversal data structures is not as simple as we would have wanted it to be. While many data structures are already in traversal form, the programmer must verify that this is the case for their data structure before using our transformation. Sometimes, small modifications are required to make a data structure a traversal one. Simplifying the definition of traversal data structures (while keeping the correctness and generality of the transformation) is an interesting open problem. We stress that the contribution of this paper is significant, as the alternative approaches available for building durable linearizable data structures are either (1) to use the simple but inefficient transformation of Izraelevitz et al. [27], or (2) to carefully reason about crash resilience to determine where to place flushes and fences for each new data structure.

We note that persistent transactions [12, 16, 28, 34, 41, 43, 46, 49, 52] resolve the problem of complexity in NVRAM programming; using a persistent transactional framework, a programmer need not be an expert in persistence to write correct programs. However, transactional memories suffer the drawbacks of lock-based programs, requiring heavy-duty logging mechanisms, and thus not yielding highly efficient solutions. In this work, we focus on issuing single operations persistently on lock-free data structures, thus giving up the full generality of transactions. Lock-free transactional data structures [19] allow executing several operations atomically on lock-free data structures. An interesting open question is how to make these transactions persistent, but this is outside the scope of this work.

In summary, our contributions are as follows.

- We define a large class of concurrent algorithms called traversal data structures. Many known lock-free data structures can be easily put in traversal form.
- We show how to automatically make any traversal data structure durable with significantly fewer flushes and fences than previously known general techniques.
- We prove that our construction is correct.
- We implement several data structures using our transformation and evaluate their performance compared to state-of-the-art constructions.

2 Model and Preliminaries

In this paper, we show how to convert a large class of algorithms designed for the standard shared-memory model into algorithms that maintain their correctness in non-volatile (persistent) memory with crashes. Thus, we first present a recap of the classic shared-memory model, and then discuss the changes that are added in our persistent memory model.

Throughout the paper, we sometimes say that a node \( n \) in a tree data structure is above (resp. below) another node \( n' \) if \( n \) is an ancestor (resp. descendant) of \( n' \).

Classical shared memory. We consider an asynchronous shared-memory system in which processes execute operations on data structures. Data structure operations can be implemented using instructions local to each process, including return statements, as well as shared atomic read, write, and compare-and-swap (CAS) instructions. We refer to write and successful CAS instructions collectively as modifying instructions, and to modifying instructions, and return statements collectively as externally visible instructions. In the experiments, we use the term threads instead of processes.

Linearizability and lock-freedom. We say that an execution history is linearizable [26] if every operation takes effect atomically at some point during its execution interval. A data structure is lock-free if it guarantees that at least one process makes progress, if processes are run sufficiently long. This means that a slow/halted process may not block others, unlike when using locks.

Persistent memory. In the persistent memory model, there are two levels of memory—volatile and persistent memory, which roughly correspond to cache and NVRAM. All memory accesses (both local and shared) are to volatile memory. Values in volatile memory can be written back to persistent memory, or persisted, in a few different ways; a value could be persisted implicitly by the system, corresponding to an automatic cache eviction, or explicitly by a process, by first executing a flush instruction, followed by a fence. We assume that when a fence is executed by a process \( p \), all locations that were flushed by \( p \) since \( p \)’s last fence instruction get persisted. We say that a value has been persisted by time \( t \) if the value reaches persistent memory by time...
Throughout the paper, we refer to the linked-list presented by Harris [24] when discussing properties of traversal data structures. We now briefly describe how this algorithm works.

Harris [24] presented a pragmatic linearizable lock-free implementation of a sorted linked-list. The linked-list is based on nodes with an immutable key field and a mutable next field, and implements three high-level operations: insert, delete, and find, which all take a key as input. Each of these operations is implemented in two stages: first, the helper function search is called with key k, and after it returns, changes to the data structure are made on the nodes that the search function returned. The search function always returns two adjacent nodes, left and right, where right is the first element in the list whose key is greater than or equal to k, and left is the node immediately before it.

To insert a node, the operation simply initializes a node with key k and next pointing to the right node returned from the search function, and then swings left’s next pointer to point to the newly initialized node (using a CAS with expected value pointing to right). If the CAS fails, the insert operation restarts. The find operation is even simpler; if right’s key is k, then it returns true, and otherwise it returns false.

The subtlety comes in in the delete operation. The next pointer of each node has one bit reserved as a special mark bit. If this bit is set, then this node is considered marked, meaning that there is a pending delete operation trying to delete this node. If a node is marked, we say that it is logically deleted.

In the persistent memory model, crashes may also occur. A crash event causes the state of volatile memory to be lost, but does not affect the state of persistent memory. Thus, all modifications that were pending at the time of the crash are lost, but all others remain. Each data structure may have a recovery operation in addition to its other operations. Processes call the recovery operation before any other operation after a crash event, and may not call the recovery operation at any other time. The recovery operation can be run concurrently with other operations on the data structure. We say that an execution history is durably linearizable [27] if, after removing all crash events, the resulting history is linearizable. In particular, this means that the effect of completed operations may not be lost, and operations that were in progress at the time of a crash must either take effect completely, or leave no effect on the data structure. Furthermore, if an operation does take effect, then all the operations it depends on must also have taken effect.

### 2.1 Running Example: Harris’s Linked List

Throughout the paper, we refer to the linked-list presented by Harris [24] when discussing properties of traversal data structures. We now briefly describe how this algorithm works.

Harris [24] presented a pragmatic linearizable lock-free implementation of a sorted linked-list. The linked-list is based on nodes with an immutable key field and a mutable next field, and implements three high-level operations: insert, delete, and find, which all take a key as input. Each of these operations is implemented in two stages: first, the helper function search is called with key k, and after it returns, changes to the data structure are made on the nodes that the search function returned. The search function always returns two adjacent nodes, left and right, where right is the first element in the list whose key is greater than or equal to k, and left is the node immediately before it.

To insert a node, the operation simply initializes a node with key k and next pointing to the right node returned from the search function, and then swings left’s next pointer to point to the newly initialized node (using a CAS with expected value pointing to right). If the CAS fails, the insert operation restarts. The find operation is even simpler; if right’s key is k, then it returns true, and otherwise it returns false.

The subtlety comes in in the delete operation. The next pointer of each node has one bit reserved as a special mark bit. If this bit is set, then this node is considered marked, meaning that there is a pending delete operation trying to delete this node. If a node is marked, we say that it is logically deleted.

More specifically, a delete operation, after calling the search function, uses a CAS to mark the right node returned by the search, if the key of right is k. After successfully marking the right node, the delete operation then physically deletes the right node by swinging left’s next pointer from right to right.next. This two-step delete is crucial for correctness, avoiding synchronization problems that may arise when two concurrent list operations contend.

The search function guarantees that neither of the two nodes that it returns are marked, and that they are adjacent. To be able to guarantee this, the search function must help physically delete marked nodes. Thus, the search function finds the right node, which is the first unmarked node in the list whose key is greater than or equal to k, and the left node, which is the last unmarked ancestor of right. Before returning, the search function physically deletes all nodes between the two nodes it intends to return.

### 3 Traversal Data Structures

In this section, we introduce the class of data structures we call traversal data structures, and the properties that all traversal data structures must satisfy. In Section 4, we show an easy and efficient way to make any traversal data structure durable. We begin with two simple yet important properties.

**Property 1** (Correctness). A traversal data structure is linearizable and lock-free.

**Property 2** (Core Tree). A traversal data structure is a node-based tree data structure. In addition to the tree, there may be other nodes and links that are auxiliary, and are only ever used as additional entry points into the tree.

The part of the data structure that needs to be persistent and survive a crash is called its core. The other parts can be stored in volatile memory and recomputed following a crash. Property 2 says that only the core part of a traversal data structure needs to be a tree.

For example, a skiplist can be a traversal data structure, since, while the entire structure is not a tree, only a linked list at the bottom level holds all the data in the skiplist, while the rest of the nodes and edges simply serve as a way to access the linked list faster. Similarly, data structures with several entry points, like a queue with a head and a tail, can be traversal data structures as well. Of course, all tree data structures fit this requirement.

More precisely, the core of a traversal data structure must be a down-tree, meaning that all edges are directed and point away from the root. For simplicity, we use the term tree in the rest of the paper.

Property 2 is important since it simplifies reasoning about the data structure, and thus allows us to limit flush and fence instructions that need to be executed to make traversal data structures durable. Note that many pragmatic data structures, including queues, stacks, linked lists, BSTs, B+ trees,
skiplists, and hash tables have a core-tree structure. Optionally, a traversal data structure may also provide a function to reconstruct the structure around the core tree at any point in time. However, our persistent transformation maintains the correctness of the core tree regardless of whether such a function exists.

A traversal data structure is composed of three methods: `findEntry`, `traverse`, and `critical`. These are the only three methods through which a traversal data structure may access shared memory, and they are always called in this order. The operation `findEntry` takes in the input of the operation, and outputs an entry point into the core tree. That is, the `findEntry` method is used to determine which shortcuts to take. This can be the head of a linked-list, a tail of a queue, or a node of the lowest level of the skip list, from which we traverse other lowest-level nodes. Note that `findEntry`, and is allowed to simply return the root of the tree data structure, e.g., the head of a linked-list.

Once an entry point is identified, a traversal data structure operation starts a traverse from that point, at the end of which it moves to the critical part, in which it may make changes to the data structure, or determine the operation’s return value. The critical method may also determine that the operation must `restart` with the same input values as before. However, the traverse method may not modify the shared state at all. The operation execution between the beginning of the `findEntry` method and the return or restart statement in the critical method is called an `operation attempt`. Each operation attempt may only have one call to the `findEntry` method, followed by one call to the traverse method, followed by one call to a critical method. The layout of an operation of a traversal data structure is shown in Algorithm 1. Operations may not depend on information local to the process running them; an operation only has access to data provided in its arguments, one of which is the root of the data structure. The operation may traverse shared memory, so it can read anything accessible in shared memory from the root. No other argument is a pointer to shared memory. This requirement is formalized in Property 3.

**Property 3 (Operation Data).** Each operation attempt only has access to its input arguments, of which the root of the data structure is the only pointer to shared memory. Furthermore, it accesses the shared data only through the layout outlined in Algorithm 1.

By similarity to the original algorithm of Harris, the traversal version of Harris’s linked-list is linearizable and lock-free (thereby satisfying Property 1), and the data structure is a tree (thereby satisfying Property 2). Furthermore, Harris’s linked-list implementation gets the root of the list as the only entry point to the data structure and it only uses the input arguments in each operation attempt. Each operation of Harris’s linked-list can be easily modified to only use the three methods shown in Algorithm 1. The `findEntry` method simply returns the root. The `findEntry` method encompasses the `search` portion of the operation, but does not physically delete any marked nodes. Instead, the `findEntry` method returns the `left` and `right` nodes identified, as well as any marked nodes between them. The rest of the operation is executed in the critical method. Therefore, Harris’s linked-list can easily be made to satisfy these properties, and thus can be converted into a traversal data structure.

From now on, when we refer to a traversal data structure, we mean only its core tree, unless otherwise specified.

### 3.1 Traversal

Intuitively, we require the traverse method to “behave like a traversal”. It may only read shared data, but never modify it (Item 1 of Property 4), and may only use the data it reads to make a local decision on how to proceed. The traverse method starts at a given node, and has a stopping condition. After it stops, it returns a suffix of the path that it read. In most cases the nodes that it returns are a very small subset of the ones it traversed. The most common use case of this is that the traversal stops once it finds a node with a certain key that it was looking for, and returns that node. However, we do not specify what this stopping condition is, or how many nodes are returned, to retain maximum flexibility.

Item 2 and Item 3 of Property 4 formalize the intuition that a traversal does not depend on everything it read, but only on the local node’s information. The traverse method proceeds through nodes one at a time, deciding whether to stop at the current node, using only fields of that node, and, if not, which child pointer to follow. The child pointer decision is made only based on immutable values of this node; intuitively, if a node has an immutable key and a mutable value then keys can be compared, but the node’s value cannot be used. The stopping condition can make use of both mutable and immutable fields of the current node.

If the traverse method does stop at the current node, it then determines which nodes to return. This decision may only depend on the nodes returned; no information from earlier in the traversal can be taken into account (Item 4 of Property 4). Intuitively, we allow the traverse method to return multiple

```java
381

**Algorithm 1. Operation in a traversal data structure**

```
nodes since some lock-free data structures make changes on a neighborhood of the node that their operation ultimately modifies. Examples of such a data structure include Harris’s linked-list [24], Brown’s general tree construction [8], Ellen et al.’s BST [20], Herlihy et al.’s skip-list [25], and others. All of these data structures make changes on the parent or linked list algorithm can satisfy Property 4. Recall that the traversal data structure. An instruction is considered a legal instruction that can be executed to atomically disconnect a given marked, connected subset of nodes S from a traversal data structure. An instruction is considered legal if it is performed in some extension of the current execution. We further require that at each configuration, there is at most one legal disconnect instruction for a contiguous set of marked nodes S. This in effect means that the marks themselves must have enough information encoded in them to uniquely identify the disconnection instruction that may be executed. Some data structures, like Harris’s linked list, achieve this trivially, since there is only ever one way to

the search function, which begins by finding the first node in the list that is unmarked and whose key is greater than or equal to the search’s key input (this node is called the right node). It then finds the closest preceding unmarked node, called the left node. We define the search function up to the right node as the traversal method. The traversal method then returns all nodes from left to right. At every point along the traversal, it uses only fields of the current node to decide whether or not to stop. If not, it always reads the next field and follows that pointer; its decision of how to continue its traversal does not depend on any mutable value that it reads. The returned nodes depend only on values between left and right. Thus, Items 1, 2, 3, and 4 of Property 4 are satisfied.

To see that Item 5 is satisfied, note that the only valueChange in Harris’s algorithm is the marking of a node for deletion. So, if a traversal T1 stops at a node n (i.e., n is the right node of the search), and a traversal T2 with the same input sees n marked, then T2 would stop after n, but would return a node above n (T2’s left and right nodes must both be unmarked, and n must be in between them).

3.2 Critical Method: Node Disconnection

The only restrictions we place on the critical method’s behavior are on how nodes are disconnected from the data structure. Disconnections may be executed to delete a node from the data structure, but some implementations may disconnect nodes to replace them with a more updated version, or to maintain some invariant about the structure of a tree.

Many lock-free data structure algorithms [8, 20, 24, 25, 37] first logically delete nodes by marking them for deletion before physically disconnecting them from the data structure. This technique prevents the logically deleted nodes from being further modified by any process, thus avoiding data loss upon their removal. We begin by defining marking.

Definition 1 (Mark). The mark method takes a node as input. A node is marked if the mark method has been called on it. Once a node is marked, no field in it can be modified.

We require that before any node is disconnected, it is marked (Item 1 of Property 5). Furthermore, marking is intended only for nodes to be removed from the data structure. To formalize that, Item 2 of Property 5 states that there is always a legal instruction that can be executed to atomically disconnect a given marked, connected subset of nodes S from a traversal data structure. An instruction is considered legal if it is performed in some extension of the current execution. We further require that at each configuration, there is at most one legal disconnect instruction for a contiguous set of marked nodes S. This in effect means that the marks themselves must have enough information encoded in them to uniquely identify the disconnection instruction that may be executed. Some data structures, like Harris’s linked list, achieve this trivially, since there is only ever one way to
We now present two additional requirements for traversal
A node in Harris’s linked list is considered marked if the
the following properties:
property that marked nodes can be removed from the data
structure in any order. This property is formalized in Item 3.

Property 5 (Disconnection Behavior). In a traversal data
structure, node disconnections satisfy the following properties:
1. Mark Before Delete: Before any node is disconnected
from a traversal data structure, it must be marked.
2. Unique Disconnection: Consider a configuration C
and let S be a connected subset of marked nodes in the
core tree of a traversal data structure. Let P(S) be the
parent of the root of S. If P(S) is unmarked, then there
is exactly one legal instruction on P(S) that atomically
disconnects exactly the nodes in S.
3. Irrelevant Disconnection Order: Let N = \{n_1 \ldots n_k\}
be the set of nodes that were marked at configuration C.
Let E_1 and E_2 be two executions that both start from the
same configuration and only perform legal disconnection
operations. If all marked nodes are disconnected after E_1
and after E_2, then the state of the nodes in the data structure
after E_1 and after E_2 is the same.

We now argue that Harris’s linked list satisfies Property 5.
A node in Harris’s linked list is considered marked if the
lowest bit on its next pointer is set. Once this bit is set, the
node becomes immutable. Item 1 of Property 5 is satisfied
because a node can only be disconnected if it is marked. If S
is a set of marked nodes with an unmarked parent P, S can
be disconnected by a CAS that swings P.next from pointing
to the first node of S to pointing to the node after the last
node of S. This is the only legal instruction on P that is able
to disconnect exactly the nodes in S, so Item 2 of Property 5
is satisfied. If several nodes are marked, removing them in
any order yields the same result: a list with all of the marked
nodes removed, and all of the rest of the nodes still connected
in the same order. Thus, all items of Property 5 are satisfied.

3.3 Algorithmic Supplements
We now present two additional requirements for traversal
data structures. These requirements are imposed so that a
traversal data structure can go through the transformation
to being persistent. We do not expect these properties to
naturally appear in a lock-free algorithm and we therefore
call them 'supplements'. They should be added to a data
structure for it to become a traversal data structure. Both
supplements are easy to implement.

Supplement 1. There is a function disconnect(root) which
takes in the root of the traversal data structure and satisfies
the following properties:
1. If no traversal data structure operation takes a step dur-
ing an execution of disconnect(root), then there will be
no marked nodes at the end of the disconnect(root).

Algorithm 2. Operation in an NVTraverse data structure

2. disconnect(root) can only perform disconnect instruc-
tions defined in Item 2 of Property 5 and no other modi-
yfying instructions.
3. disconnect(root) can be run at any time during an exe-
cution of the traversal data structure (without affecting
the linearizability of the traversal data structure).

The disconnect(root) operation can be implemented by
traversing the data structure and using the the unique atomic
disconnection instruction for the marked nodes. For Harris’s
linked list, we can supply a function that traverses the linked
list from the root pointer and trims all the marked nodes.

The second supplement that we require for a traversal data
structure is that it keeps an extra field in each node, which
stores the original parent of this node in the data structure.
Since the data structure is a tree, a node can only have a
single parent when it joins the data structure. We require
the address of the pointer field that was changed to link in
the new node to be recorded in the extra field. Note that it is
possible that a sub-tree is added as a whole by linking it to a
single (parent) pointer in the data structure. In this case, that
same parent pointer should be stored in all the nodes of the
inserted sub-tree. The location of this pointer must be stored
in the original parent field before the node is linked to the
data structure to ensure that this field is always populated.

Supplement 2. A designated field in each node n, called
the original parent (OP) of n, must store the location of the pointer
that was used to connect n to the data structure.

In Section 4 we specify how this field is used. Adding a
field to the data structure may be space consuming, so we
also propose an optimization that can avoid storing this field.
In our running example, before inserting a node to Harris’s
linked list we put the address of the next field of the preceding
node in the original parent field of the new node.

4 NVTraverse Data Structures
In this section we show how to apply flush and fence instruc-
tions to any traversal data structure to create an efficient
and provably correct durably linearizable version of it. These
flush and fence instructions can be applied automatically.

At a high-level, no persisting is done during the traverse
method, whereas, in the critical method, every field accessed
must be persisted before the next externally visible instruc-
tion is executed. Furthermore, we add another phase between
the traverse and critical methods, in which we ensure that the nodes returned by the traverse method are persisted.

**Recovery.** The recovery phase executes the disconnection function guaranteed by Supplement 1 in Section 3.3. No additional action is required.

### 4.1 Before the Critical Method

We now specify the fields that must be persisted before the critical method begins.

**Protocol 1.** Let \( n_1 \ldots n_k \) be the nodes that were returned by the traverse method of some operation \( op \), where \( n_1 \) is the topmost node returned. Before the beginning of the critical method of \( op \), the following fields must be persisted.

- The original parent pointer of \( n_1 \).
- All fields that the traverse method read in \( n_1 \ldots n_k \).

We flush these fields in two functions, called *ensureReachable* and *makePersistent*, corresponding to the first and second items, respectively. We briefly describe how we implement each of these functions. Note that these functions are the same for all traversal data structures, and can simply be inserted as black boxes between the traverse and critical methods of a given traversal data structure.

**ensureReachable.** The *ensureReachable* function’s goal is simply to flush one field: the original parent (OP) pointer of the topmost node returned by the traverse method. Note that the original parent of a node might not be the current parent of that node, since other nodes may have been inserted in between. By Supplement 2 (from Section 3.3) the OP field is available in the node. The function simply takes this topmost node and flushes the location indicated by its OP field.

**An optimization for ensureReachable.** While the proposed original parent mechanism is simple, it can also be costly, since it requires an extra word on each node, and may also delay garbage collection. We therefore present an alternative solution. For the common case where the insert operation always connects a single node to the structure, *ensureReachable* may simply flush the current parent of its input node. For this, the traversal phase must return the current parent of the first node returned from the traversal.

This method can also be used if the insert operation links at most \( k > 1 \) nodes to the structure simultaneously, but becomes less efficient; the traversal must return the last \( k \) nodes on the traversal path before the first node that the traversal procedure returns. These nodes are then flushed by *ensureReachable*. In the full version [22], we prove this approach correct. We summarize this in the following lemma.

**Lemma 4.1.** In an NVTraverse data structure implementation in which the deepest tree ever atomically inserted is of depth \( k \), the *ensureReachable* method can be implemented as follows.

- If \( n \) has an OP field, flush the location in this field.
- Otherwise, flush a path of length \( k \) back from \( n \).

**makePersistent.** The purpose of *makePersistent* is to persist all fields read by the traversal on the nodes that it returned. This can be done by simply flushing these fields, and then executing a single fence instruction (which also ensures the completion of the flush of *ensureReachable*).

### 4.2 During the Critical Method

Simply put, all fields accessed by some process \( p \) during the critical method must be persisted before the next externally visible instruction is executed by \( p \). However, this can be relaxed for fields that cannot be accessed by any process other than \( p \). Intuitively, the idea is that \( p \) must ensure, before executing an instruction \( e \) that may affect other processes, that the values that \( p \) relied on to determine \( e \)’s parameters must be persisted. To achieve this, we use the following rules.

**Protocol 2.** In any critical method, the following flush and fence instructions must be injected.

- Flush after every read of a shared variable.
- Flush after every write/CAS instruction.
- Fence before every write/CAS on a shared variable.
- Fence before every return statements.

Note that this means that local work requires less flush and fence instructions than shared work. In particular, when initializing a node, a process executes flushes after initializing each field, but only needs to fence once before atomically inserting the new node into the data structure. Furthermore, there is no need to flush after reading an immutable field.

### 4.3 Correctness

We say that the algorithm resulting from applying Protocol 1, Protocol 2 and the specified recovery procedure to a traversal data structure form an NVTraverse data structure. The algorithm is presented in Algorithm 2. In the full version [22], we prove the following theorem.

**Theorem 4.2.** Every NVTraverse data structure is durably linearizable.

Intuitively, a data structure is durable linearizability if a crash cannot make the effect of any operation be lost. NVTraverse data structures achieve this by ensuring that every modification to shared memory is persisted before any process uses that modification’s value. That is, every value read by \( p \) is persisted before \( p \) makes its next shared memory modification. The only exception is during the traversal, where a process may read values without persisting them. However, due to the restrictions on how the traverse method may behave, no modification that \( p \) does can depend on a value it read (but did not return) during its traversal. Other than persisting all values that may affect \( p \)’s shared modifications, we also need to make sure that \( p \)’s writes cannot disappear from the data structure at a later point. This can happen if \( p \) wrote on a part of the data structure that might not be reachable from the root upon a crash. To prevent this from happening,
the ensureReachable function persists the pointer that connects the relevant subtree to the rest of the structure. Thus, the flush and fence instructions we prescribe are necessary; removing any of them could violate the correctness of some NVTraverse data structure. However, hand-tuned data structure implementations could still save on flushes and fences by reasoning more carefully about dependencies in the data.

4.4 Example
In Algorithm 3 and Algorithm 4, we present pseudocode showing Harris’s linked-list (HLL) as an NVTraverse data structure. Note that the traverse method ends in the middle of the search function, since the search also executes some physical deletions, which are part of the critical method of a traversal data structure. The traverse method returns the set of nodes for the ensureReachable (using the ensureReachable optimization) and makePersistent functions to flush. In Algorithm 3, in lines 6-14, we show how every operation is executed. Every operation starts with finding an entry to the core tree structure. In a linked-list, the entry point is the root of the list. Therefore, findEntry returns the root. After that, the traverse function from Algorithm 4 is called. This function returns exactly three nodes. The right node, which is the first unmarked node in the list whose key is greater than or equal to k, and left node, which is the last unmarked ancestor of right. In addition, it returns the current parent of left, as described in the optimization in Section 4.1. In line 11, the current parent node is flushed in order to make sure that the left node is reachable from the head, followed by line 12, where makePersistent flushes the left and right nodes. After that, the critical part is executed; depending on the operation, we go to either insertCritical in line 16, deleteCritical in line 34 or findCritical in line 1 in Algorithm 4.

The critical function of an insert operation, in lines 16-32 in Algorithm 3 starts by deleting marked nodes. This deletion occurs only if the left and right nodes that were returned from traverse are not adjacent. If left and right were not adjacent and the deletion from lines 37-53 in Algorithm 4 fails, the insertion operation restarts. If the key already exists, the operation returns false (lines 21-22). As the key is an immutable field, we do not flush after reading the key. Note that deleteMarkedNodes executes a fence before returning. Therefore, there is no need to re-execute that fence in lines 19 and 22. Afterwards, a new node is allocated, followed by a flush after write. In line 25 there is a fence before the CAS in line 26 to insert the newly allocated node. This CAS is followed by a flush after CAS and a fence before the return. If the insertion has failed due to concurrent activity, the operation will be re-executed (line 32). The critical functions of the delete and find operations follow the same rules.

The traverse function is presented in Algorithm 4 in lines 8-35. The inner while loop, from line 16 to line 27, traverses the list from the root and tries to find the first node which is unmarked with a key equal to or greater than k. The marked nodes before k are saved in the nodes variable. After the right node is inserted to nodes in line 29, the nodes variable contains the left node which was unmarked at the moment it was inserted (in line 21), followed by all the marked nodes until the right node (line 29). If the right node is marked by the time line 30 is executed, the traversal restarts. If not, we proceed to line 33 where we insert left’s parent to the parent variable and return both the parent and nodes variables (to allow us to persist them later on). Note that by the given properties, no modification is ever done in the traverse.

The last function we present here is called deleteMarkedNodes (lines 37-53 of Algorithm 4). This function gets the nodes from the traverse as an input and checks whether there are more than two nodes (more than the left and right ones). If it finds more than two nodes, then there is a need to trim all the marked nodes by executing a CAS in line 44. The key observation here that the CAS will be successful only if the current left.next pointer is still the pointer that was read during the traverse. If this is the case, the marked nodes will be trimmed successfully and the changed field will be flushed afterwards. In line 47 we make sure again that the right node is not marked. If the node is marked, or the trimming was unsuccessful (line 53), then the function will return false and the traversal will need to be re-executed. Otherwise it returns true. Before every return, we make sure that a fence is executed. Moreover, in line 48 there is a flush which is done due to the read of the shared variable in line 47.

Some further optimizations can be done, but we omit them from the pseudocode for readability.

5 Experimental Evaluation
We implement five traversal data structures: an ordered Harris linked-list [24], two binary search trees (BST) based on the algorithm of Ellen et al. [20] and Natarajan and Mittal [37], a hash table implemented by David et al. [18] based on Harris’s linked-list, and a skiplist based on the algorithm of Michael [35]. We compare the performance of the original, non-durable version of the algorithms to four ways of making it durable: our NVTraverse data structure (Traverse), Izraelevitz et al. [27]’s construction (Izraelevitz), the implementation of David et al. [18] (Log Free) and Ramalhete al.’s implementation for durable transactions [41] (Onefile).

5.1 Setup
We run our experiments on two machines; one with two Xeon Gold 6252 processors (24 cores, 3.7GHz max frequency, 33MB L3 cache, with 2-way hyperthreading), and the other with 64-cores, featuring 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores.

The first machine has 375GB of DRAM and 3TB of NVRAM (Intel Optane DC memory), organized as 12 × 256GB DIMMS (6 per processor). The processors are based on the new Cascade Lake SP microarchitecture, which supports the clwb instruction for flushing cache lines. We use the sfence instruction for fences. We use libvmmalloc from the PMDK
Algorithm 3. HLL Persistent Insert and Delete

```java
class Node<T,V> {
    T key; // immutable field
    V value;
    Node *next; }

bool operation (T key) {
    bool restart, val = true, false;
    while (restart) {
        Node* entry = findEntry(root, input);
        List <Node> parent, nodes = traverse(root, key);
        flush(&(parent.next)); // ensureReachable ()
        makePersistent(nodes);
        restart, val = opCritical(nodes, key);
        if (!restart) return val; }

bool, bool insertCritical (List <Node> nodes, T key)
    bool succDelete = deleteMarkedNodes(nodes);
    if (succDelete == false) {
        return true, false; // retry
    Node* left, right = nodes.front(), nodes.back();
    if (right.key == key) {
        return false, false; // no flush - immutable
    } else {
        return false, true; // node inserted
    }

bool, bool deleteCritical (List <Node> nodes, T key)
    bool succDelete = deleteMarkedNodes(nodes);
    if (succDelete == false) {
        return true, false; // retry
    Node* left, right = nodes.front(), nodes.back();
    if (right.key != key) {
        return false, false; // no key
    } else {
        Node* rNext = right.next;
        flush(&(right.next));
        if (!isMarked(rNext)) {
            flush(&(right.next) & fence); // before CAS
            bool res = CAS(&(right.next), rNext, mark(rNext));
            flush(&(right.next) & fence); // before CAS
            if (res) {
                // before return
                flush(&rNext); // before return
                return false; } }
        flush(&(left.next) & fence); // before return
        return false, true; // retry
    }

bool, bool findCritical (List <Node> nodes, T key)
    Node* right = nodes.back();
    fence; // before return
    if (right.key != key) {
        return false, false; // no flush - immutable
    } else {
        return false, true; // key exists }

List <Node> traverse(Node head, T k)
    List <Node> parent, nodes;
    Node* leftParent, right;
    while (true) {
        leftParent, left, right = head, heaf, null;
        nodes.clear();
        Node* pred, curr = head, head;
        Node* succ = curr.next();
        while (true) {
            nodes.append(curr);
            if (curr.key < k) {
                nodes.append(curr);
                break;
            } else {
                parent.append(leftParent);
                return parent, nodes; }

bool deleteMarkedNodes(List <Node> nodes)
    if (nodes.size() == 2) {
        fence; // before return
        return false; } }
    Node* left, right = nodes.front(), nodes.back();
    Node* leftNext = nodes[1];
    fence; // before CAS
    bool res = CAS(&(left.next), leftNext, mark(leftNext));
    flush(&(left.next) & fence); // before return
    if (res) {
        if (right && isMarked(right.next)) {
            flush(&right.next) & fence; // before return
            return false; } }
    fence; // before return
    return true; }
    fence; // before return
    return false; }
```
library to place all dynamically allocated objects in NVRAM, which is configured to app-direct mode. All other objects are stored in RAM. The operating system is Fedora 27 (Server Edition), and the code was written in C++ and compiled using g++ (GCC) version 7.3.1. The second machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). The operating system is Ubuntu 14.04 (kernel version 3.16.0). clwb is not supported, so we used the synchronized clflush instruction instead. The code was written in C++ 11 and compiled using g++ version 8.3.0 with -O3.

We found that David et al. [18]’s code could not run on this NVRAM architecture for executions using more than 3 threads, so we only show comparisons to David et al’s log-free data structures on the AMD machine.

On the NVRAM machine, we avoid crossing NUMA-node boundaries, since unexpected effects have been observed when allocating across NUMA nodes on the NVRAM. Hyper-threading is used for experiments with more than 24 threads on the NVRAM machine. No hyperthreading is used on the DRAM machine. All experiments were run for 5 seconds and an average of 10 runs is reported.

On all the data structures, we use a uniform random key from a range $0, ... , r-1$. We start by prefilling the data structure with $r/2$ keys. Keys and values are both 8 bytes. Unless indicated otherwise, all experiments use an insert-delete-lookup percentage of 10-10-80. For all data structures, we measured different read distributions, covering workloads A, B, and C of the standard YCSB [15].

For volatile data structures, memory management was handled with ssmem [17], that has an epoch-based garbage collection and an object-based memory allocator. Allocators are thread-local, causing threads to communicate very little. For the durable versions, we used a durable variant of the same memory management scheme [53].

5.2 Results on NVRAM

We begin our evaluation by examining the performance of various data structure implementations on the NVRAM Intel machine. We first examine the NVTraverse version of Harris’s linked-list [24].

List Scalability. We test the scalability as the number of threads increases, showing the results in Figure 1 (a). We initialize the list to have 512 keys, and insert and delete keys within a range of 1024. We note that while the non-durable version of the list outperforms the NVTraverse data structure by 2.1x, the latter outperforms Izraelevitz et al. [27]’s construction by 25.4x and OneFile by 7.3x on 48 threads. While OneFile performs better than Izraelevitz et al. [27]’s construction, they scale similarly. The dramatic differences between NVTraverse data structures and the other approaches hold true throughout all of the experiments that we’ve tried, highlighting the significant advantage of our approach.

The non-durable version and the NVTraverse data structure have a similar throughput up to 16 threads. However, the non-durable version scales better than the NVTraverse data structure. Note that as the thread count increases, there are more flushes in the NVTraverse data structure, since each thread flushes a constant number of nodes. Each flush invalidates that cache line, meaning that as the number of threads increases, and cache misses become more likely.

List Size. Next we test how lists of different sizes affect performance. We expect that the size of the data structure may have a significant effect because a larger data structure means operations spend a larger fraction of their time traversing. The results are shown in Figure 1 (b). The first thing to note is that the original, non-durable version of the list greatly outperforms the NVTraverse data structure version for smaller lists. The non-durable version is better by 2.9x for a short list of 128 nodes and by 1.5x for the size of 256 nodes. However, the difference becomes less pronounced, and even inverts, as the list grows. This phenomenon can be explained by considering the traversal phase of operations on the list, and their role in determining the performance of a given implementation. Recall that the NVTraverse data structure construction only executes a constant number of flushes and fences per traversal, and the non-durable version never executes flushes or fences. As the size of the data structures increases, the cost of the traversal outweighs the cost of persistence. For the original list, the traversal is the primary source of delay, so the effect of increasing the traversal length in this implementation is starker than the effect of the same phenomenon in the durable version, which also spends significant time persisting. For the durable competitors, we observe the same trend as we saw in the list scalability test. The NVTraverse data structure construction outperforms Izraelevitz et al. [27]’s construction by 13.5x-39.6x and OneFile by 4.25x-2.2x on a range of 256-8192 respectively.

List Update Percentage. We now consider the effect of varying update ratios (Figure 1 (c)). These tests were run on a relatively small list (500 nodes), so the non-durable version outperforms the NVTraverse data structure. Interestingly, the non-durable list’s throughput sharply drops between 0% and 5% updates whereas the NVTraverse data structure version stays relatively stable across all update percentages. Since the list is less than 12Kb, it is small enough to fit in L1 cache, so there are virtually no cache misses on read-only workloads on the non-durable list. The NVTraverse data structure version still experiences cache misses because lookup operations perform clwb, invalidating the cache line. In our experience, it seems that in the current architecture, clwb and clflushopt yield the same throughput. In future architectures, we believe clwb will no longer invalidate cache lines and will perform better. OneFile does extremely well in read-only workloads (for which it is optimized).

BST, Hash Table and Skiplist. We study how the Hash Table, BSTs and Skiplist behave under different YCSB-like...
workloads. The results are shown in Figures 1 (d), (e), and (f) respectively. We only show OneFile’s performance in the BST, since the patterns seen on OneFile are similar in all cases, and similar to the list. We implemented two versions of the BST; one based on Natarajan and Mittal [37]’s tree, and the other on Ellen et al. [20]’s tree. We saw that the amount of flushes and fences Ellen et al. [20]’s tree executes is less than in Natarajan and Mittal [37]. However, Ellen et al. [20] performs worse than Natarajan and Mittal [37] in their volatile version, and the gap remains in the durable version.

We see that in the hash table, the non-recoverable version degrades twice as fast as the NVTraverse data structure as the number of updates grows. This is because allocating and writing nodes is more expensive than just reading. However, in the NVTraverse data structure, these costs do not form a bottleneck, because of the additional flush and fence instructions. Interestingly, the skiplist and BSTs do not exhibit this behavior; in fact, the NVTraverse data structure version degrades faster than the non-recoverable version as the update percentage increases. This can happen due to the fact that as the number of updates increases, the likelihood of failed CASes increases, which is more meaningful than in the hash table. For the NVTraverse data structure, this means executing extra flush and fence instructions, which slows it down more in comparison to the non-recoverable version.

5.3 Results on DRAM
We ran experiments on a machine with classic DRAM to compare with the algorithms of David et al. [18]. We ran David et al. [18]’s algorithms in the link-and-persist mode. Link-and-persist, suggested by David et al. [18] and Wang et al. [48], is an optimization that allows avoiding flushing clean cache lines by tagging flushed words, but is not completely general, so we did not apply it to the NVTraverse data structure constructions. David et al. [18] actually present two optimizations in their paper, but the second one they present, called the link-cache, does not provide durable linearizability [27]. At least one optimization must be selected.

List. We ran with an initial size of 8192 nodes in the list with a range of 16,384 keys, varying the thread count. The results are shown in Figure 2 (g). We notice that the linked-list algorithm of David et al. [18] outperforms ours by 15%-50%, from 32 to 64 thread counts; this is due to the fact that the link-and-persist technique avoids some flushes. With more threads, it is more likely that two threads get the same key, meaning that only one of them will have to flush. On the other hand, the NVTraverse data structure outperforms David et al. [18] by 40%-16%, for thread counts of 1-8. We believe this happens because of the same optimization. In link-and-persist, there is an extra CAS for each flush executed (to tag the word), and on a lower thread count, this optimization is less beneficial. So, our automatic construction does have a cost, but this cost is much smaller than Izraelevitz et al. [27]’s (by up to 56×).

We now test various update percentages, with 64 threads and same list size as above (Figure 2 (h)). As previously noted, the linked-list algorithm of David et al. [18] outperforms the NVTraverse data structure by at most 1.37x at 20% updates, as the flush-and-persist technique avoids some flushes.
When there are read-only operations, our list is faster by 1.7x, again, as David et al. [18] executes some CASes to avoid extra flushes. We see the same trend in Figure 2 (i), that shows 16 threads with varying key ranges. The bigger the list, the smaller the advantage of link-and-persist. OneFile [41] performs worse than our construction, as expected, by 1.1x-25x for 0%-100% update percentages respectively (Figure 2 (h)).

**Hash table.** We observe the opposite trend in the hash table. In Figure 2 (j), (k) and (l) we can see the scalability, various update percentages and ranges of keys respectively. The first two are filled with 8M nodes. For a fair comparison, due to the anomaly that we observe in Figure 2 (j) on 32 threads, the update percentages are shown on 16 threads. With 0% updates, the algorithm of David et al. [18] outperforms the NVTraverse data structure. This is due to the hash function; David et al. [18] use a bit-mask, assuming the table size is a power of 2. This is faster than the more general modulo function that we use. In all the other comparisons, the NVTraverse data structure outperforms David et al. [18] by up to 30% on 16 threads and 230% on 64 threads. In a hash table with 8M nodes, the contention on every bucket is low, so the price of the link-and-persist outweighs the benefit. This is clearer in Figure 2 (l), which shows various ranges of keys with 16 threads and 20% updates.

**Binary Search Tree.** Figure 2 (m) shows the results of various update percentages of the BST with 8M initial size and 16 running threads. We compare our two NVTraverse BST implementations with the implementation of David et al. [18], as well as the two BST versions for Izraelevitz et al. [27]’s algorithm. David et al. [18] implements the durable version of Natarajan and Mittal [37] as well. As the contention is low, same as in the hash table, the CASes for marking the flushed nodes downgrades the performance of the same BST implementation by 4%-83% for 0%-100% of updates.
Skiplist. The scalability and varying update percentages of the skiplist are depicted in Figure 2 (n) and (o). Figure 2 (n) shows the scalability for an initial size of 8M nodes and 20% updates. As it executes one less flush in every search operation, the implementation of David et al. [18] performs better than the NVTraverse data structure in a read dominated workload. For 64 threads and 20% updates, David et al. [18] is 1.3x better, and it reaches the maximum difference of 1.63x in 0% of updates. However, as seen in Figure 2 (o), as the workload becomes more write dominant, the performance degrades; it benefits less from the flush that was saved in the search operation. The NVTraverse data structure gets better throughput by 1.68x and 2.4x on 50% and 100% updates.

5.4 Other Architectures
We showed the evaluation of our transformation on two different architectures. We believe that our persistent transformation is hardware-agnostic and relevant to other frameworks that satisfy other memory models. However, the instructions that are used for executing flushes and fences should be adjusted accordingly. For instance, in an ARM architecture, a flush instruction may translate to the DC CVA P instruction and a fence instruction may be executed by calling to a full DSB instruction [2, 40].

6 Related Work
NVRAM has garnered a lot of attention in the last decade, as its byte-addressability and low latency offer an exciting alternative to traditional persistent storage. Several papers addressed implementing data structures for file systems on non-volatile main memory [11, 29, 31, 45, 50, 51].

Recipe [30] provides a principled approach to making some index data structures persistent. While on the surface, their original approach does not always yield correct persistent algorithms, even for data structures that fit their prescribed conditions. The ArXiv version of their paper has updated Conditions 1 and 2 to account for this. The new formal conditions require flush and fence instructions for every read and write, similarly to the requirement of Izraelevitz et al. [27]. While they note that in some situations, one can leave out some of these flushes and fences, it is left for the user to do so. Recipe thus exemplifies the difficulty of providing a correct, general and efficient solution, and the importance of having one. In our work, we provide a general and automatic way to reduce the required flush and fence instructions for traversal data structures that yields efficient persistent data structures, and we provide a proof that this transformation is correct.

Mnemosyne [46] provides a clean programming interface for using persistent memory, through persistent regions. Atlas [9] provides durability guarantees for general lock-based programs, but does not capture lock-free algorithms. Some general constructions for persistent algorithms have been proposed [14, 27]. Another approach for general constructions for persistent concurrent algorithms uses transactional memory. This involves creating persistent logs to either undo or redo partial transactions [13, 16, 33, 47]. While these approaches are general, they suffer from the usual performance setbacks associated with transactional memory.

Friedman et al. [21] presented a hand-tuned implementation of a durable lock-free queue, based on the queue of Michael and Scott [36], and presented informal guidelines for converting linearizable data structures into durable ones. Based on these guidelines, David et al. [18] implement several durable data structures. David et al. [18] achieve this by carefully understanding each data structure to find its dependencies, and only intuitively argue about correctness. Our definition of traversal data structures formalizes some dependencies in a large class of algorithms and removes the need for expert understanding of persistence and concurrency.

Other general classes of lock-free algorithms have been defined. Brown et al. [8] defined a general technique for lock-free trees, and Timnat and Petrank [44] defined normalized data structures. These classes were defined with different goals in mind and do not aid in finding dependencies that are critical for efficient persistence. Another line of work focuses on formally defining persistence semantics for different architectures [38, 39].

7 Conclusions
Recent NVRAM offers the opportunity to make programs resilient to power failures. However this requires persistent memory to be kept in a consistent state. In general this can be very expensive since it can require flushing and fencing between every read or write. This renders caching almost useless. In this paper we considered a broad class of linearizable concurrent algorithms that spend much of their time traversing a data structure before doing a relatively small update. The goal is to avoid any flushes and fences during the traversal (read-only portion). For a balanced tree, for example, this can mean traversing \(O(\log n)\) nodes without flushing, followed by \(O(1)\) flushes and fences. We describe conditions under which this is safe. Although the conditions require some formalism, we believe that in practice they are quite natural and true for many if not most concurrent algorithms. We study several algorithms under this framework and experimentally compare their performance to state-of-the-art competitors. We run the experiments on the recently available Intel Optane DC NVRAM. The experiments show a significant performance improvement using our approach, even beating hand-tuned algorithms on many workloads.

Acknowledgments
We thank Michael Bond for his helpful comments on this paper. This work is supported by the United States - Israel BSF grant No. 2018655, by NSF grants CCF-1910030 and CCF-1919223, by an Azrieli PhD Fellowship, a Microsoft PhD Fellowship, and an NSERC PGSD Scholarship.
References


[38] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model.


