Mirror: Making Lock-Free Data Structures Persistent

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

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

Pedro Ramalhete
Cisco Systems, Switzerland
pramalhe@gmail.com

Abstract

With the recent launch of the Intel Optane memory platform, non-volatile main memory in the form of fast, dense, byte-addressable non-volatile memory has now become available. Nevertheless, designing crash-resilient algorithms and data structures is complex and error-prone as caches and machine registers are still volatile and the data residing in memory after a crash might not reflect a consistent view of the program state. This complex setting has often led to durable data structures being inefficient or incorrect, especially in the concurrent setting.

In this paper, we present *Mirror*—a simple, general automatic transformation that adds durability to lock-free data structures, with a low performance overhead. Moreover, in the current non-volatile main memory configuration, where non-volatile memory operates side-by-side with a standard fast DRAM, our mechanism exploits the hybrid system to substantially improve performance. Evaluation shows a significant performance advantage over NVTraverse, which is the state-of-the-art general transformation technique, and over Intel's concurrent lock-based key-value datastore. Unlike some previous transformations, Mirror does not require any restriction on the lock-free data structure format.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Software and its engineering** → **Software libraries and repositories**.

Keywords: Non-volatile memory, lock-free, concurrent data structures

ACM Reference Format:
Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454105

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454105

1 Introduction

With the recent introduction of the Intel Optane memory platform, non-volatile random access memory has become available. Such non-volatile memory offers both large byte-addressable memory that does not lose its content upon a crash and access speeds comparable to DRAM. This form of memory, denoted non-volatile main memory (NVMM), can serve as a very large main memory, allowing persistent data structures to be manipulated by the program with no need to save their content (in an augmented format) to a secondary persistent drive. Optane DC Persistent Memory by Intel currently comes with storage capacities that can go up to 512 GB per NV-DIMM, and with read throughput that is around 3x slower than conventional DRAM.

Numerous recent studies have proposed various schemes to exploit the benefits of NVMM. A substantial fraction of these research projects attempt to design persistent data structures that in turn facilitate better storage systems, databases, key-value stores, etc. Designing data structures and algorithms for NVMM is hard because caches and registers are still volatile, and their content is lost upon a crash. Thus, the state of main memory following a crash may be inconsistent, missing recent data writes not yet flushed from the cache into the main memory. To ensure that important data is available in non-volatile memory after a crash, special instructions that flush data to the (non-volatile) memory are provided by the hardware. These instructions allow the programmer to make sure that certain data is written to memory, securing the state of the execution even when a crash occurs. Such flushing of data from cache to memory, however, is costly, and good algorithms use these instructions sparingly. The difficulty of the design increases in the presence of concurrent data structures as correctness, efficiency, and scalability become harder to achieve simultaneously, requiring the involvement of experts on concurrency and durability.

Researchers have tried to reduce the difficulty of designing durable data structures with an assortment of techniques that fall into two broad categories: persistent transactions and general transformations. Persistent transactions can take a sequential implementation of a data structure and provide a persistent and concurrent execution of the data structures' methods (e.g., [3, 4, 9, 20, 28–31, 37, 39]). The generality of persistent transactions usually comes with a significant performance price tag. Moreover, although some of these techniques are capable of guaranteeing non-blocking progress, they all serialize write operations, which severely limit scalability in write-heavy workloads.

General transformations are applied on specific data structures, making them durable using an automatic transformation that any programmer may apply. Most existing transformations (as well as the transformation presented in this paper) are applied on concurrent lock-free data structures. As indicated in previous work [13, 17, 18, 27, 41], a lock-free algorithm is resilient to thread failures, implying that lock-free data structures are a natural choice for being adapted to persistence. Izraelevitz et al. [27] presented a mechanized transformation that can be applied to any lock-free data structure, making it persistent. Even though the transformation is simple, the resulting durable data structures are slow, due to every atomic load or store needing to issue a flush and fence instructions. Subsequent work [17] has improved performance dramatically, albeit with increased complexity and with the transformation restricted to a subclass of the lock-free data structures.

In this paper we present *Mirror*, a simple automatic transformation that is capable of converting any linearizable lock-free data structure into a persistent and lock-free durably linearizable [27] data structure. The first idea underlying the Mirror transformation is to have two copies of the data. At first sight, this may seem costly because each write to the data structure needs to be executed twice, but there are two points that make these two replicas worthwhile. First, only the first write (to the first replica of the data) needs to be persisted (with a flush and a fence), making the second (non-persisted) write a great deal lighter. Second, using the second replica for reading from the data structure eliminates the need to ever persist read data, which is a great advantage for a general transformation.

The second idea is to place the second (volatile) replica in the volatile DRAM. While the first copy of the data is used for persistence and must be placed in the non-volatile memory, the second copy is only used to speed loads from the data structure and is never persisted. Placing the second copy in the volatile DRAM increases the access speed of loads substantially, in current platforms by a factor of 3x. This idea improves the performance of the durable data structures obtained by the Mirror construction even further and we get a significant throughput improvement over the state-of-the-art transformation of NVTraverse [17]. The idea of having one persistent replica with all the critical data and one volatile replica (which might be located in DRAM) was first presented in [41] for a specific construction of a set. Extending their idea to a general construction presents a challenge in preserving linearizability, which we solve in this paper.

Given that many operations on lock-free data structures spend a considerable amount of time traversing nodes, effectively executing loads, the usage of DRAM for the volatile replica significantly increases the throughput in read-dominated workloads, and can yield a non-negligible advantage even on write-intensive workloads due to traversals that precedes the updates. For example, the persistent linked-list data structure created by Mirror outperforms the persistent linked-list created by NVTraverse by a factor higher than 4x (for a list of size 128, with 8 concurrent threads, and 20% update operations). This factor goes higher than 10x for a workload with read-only operations. Furthermore, Mirror outperforms the persistent lock-based data structure Cmap by pmemkv [38] by up to 3.95x for a hash-table of size $8M$, with 8 concurrent threads. In fact, the evaluation shows that the data structures created by Mirror sometimes beat even hand-made specific data structures that were designed to execute solely on the non-volatile main memory. Moreover, because of the partial use of volatile DRAM, the persistent data structures created by Mirror can often execute faster than original (non-persistent) data structures that execute on the slower non-volatile memory.

In Section 4 we explain the details of the construction and argue why it works correctly. Loosely speaking, all read data is guaranteed to be persisted (in the persistent replica) before it is read (in the volatile replica), and, therefore, there is no need to worry about persisting read data. An invariant of the durable data structure obtained from the Mirror transformation is that the second volatile copy is at most one value behind the first copy. A version number is added to each shared data structure field to keep the modifications well ordered, in an adjacent word, using a double-word-compare-and-swap (DWCAS) on every modification. In Section 5, we prove that the resulting data structure is durably linearizable.

To make the transformation easy for the programmer, we built an implementation of the primitives that handle the two copies, i.e., the modified load, store, and CAS operations that should be called by the durable data structure. These are implemented as overloading of these operations using a modified `std::atomic` type. We also built an adequate allocator that can work with the two copies. Given these implementations, applying Mirror to a data structure is simple. It consists of type annotation to replace the usage of `std::atomic` with `patomic` and replacing the calls to the system allocator with the Mirror's allocator. In addition, the programmer needs to specify the roots of the data structure from which all nodes of the data structure are reachable, and provide a routine that, given the roots, traverses all nodes reachable from the roots. In case of a crash, this traversal is required by the recovery procedure. This can be easily implemented for all existing data structures that we are aware of. Mirror imposes no algorithmic change to the lock-free data structure code in order to make it persistent.

The closest previous work is NVTraverse [17], which translates a lock-free data structure in a traversal form into a durable data structure automatically. NVTraverse avoids flushes and fencees during the traversal part of a data structure operation, which obtains high efficiency. NVTraverse requires that the transformed data structure is lock-free in a specific traversal form, defined in the paper. Transforming a

data structure into a traversal form requires some programmer expertise and may sometimes harm performance. Mirror uses a very different transformation and it obtains significant improvements on current platforms. Additional discussion appears in Section 7.

The rest of this paper is organized as follows. Section 2 discusses the setting and presents correctness definitions. Section 3 provides an overview of the Mirror library and the detailed implementation and its correctness appears in Section 4. The experimental evaluation for different data structures is presented in Section 6. Section 7 discusses related work and Section 8 concludes.

2 Model and Preliminaries

The recent release of new non-volatile main memory technologies enables us to use this memory as we have never before used it. These technologies guarantee durability, byte-addressability, high density and larger capacity than DRAM. In this section, we discuss assumptions regarding the underlying architecture and explain the related definitions.

2.1 Persistent Memory

We consider a system of asynchronous n processes $p_1, p_2, ..., p_n$ with a shared memory system. This shared memory may be divided into three parts: (1) A volatile cache, which is the fastest memory, but volatile, meaning that its content will not survive a potential crash; (2) a volatile DRAM, which is slower than the cache, but volatile as well and is larger than the cache; and (3) an NVMM, which is slower than the DRAM but faster than traditional non-volatile memories. The NVMM is durable, and we assume that only its content will remain after a crash, as opposed to cache and DRAM. Each process may access the DRAM and the NVMM with a byte-granularity load or store that are first written/read from the cache. These values may be written-back either implicitly by the mechanism that manages the cache or explicitly by calling specific instructions that write back data from cache to NVMM. A value that was written-back to the NVMM at time t, either implicitly or explicitly, is called *persisted* at time t.

2.2 Explicit Write-Backs

As mentioned above, explicit write-backs occur by calling two specialized instructions. The first one is a *flush*, which is not blocking and triggers a write-back for a specific memory location. The second one is a store *fence*, which orders all preceding writes and flushes executed by that process to become visible to other processes before any writes or flushes executed after the fence. Intuitively, a fence following a flush may be thought of as blocking until all previously flushed locations have reached the memory. The specific architecture instructions are presented in Intel, AMD and ARM manuals [1, 2, 23, 24], and described thoroughly in [35, 36].

2.3 Durable Linearizability

Every operation on a data structure consists of an *invocation* which is the first executed operation's instruction, and a *response* which happens after the execution of the last operation's instruction. An execution of a concurrent program is modeled by a *history* comprising a finite sequence of invocation and response events of operations by processes. Every operation's invocation and response are considered events which are related to the calling process. In our system, we consider system-wide crashes as events, which are not associated with a specific process, and resets all the volatile memories. No persistent memory location is affected.

A history is said to be *linearizable* if every operation takes effect instantaneously at some moment between its invocation and response [22]. Since this definition does not capture system-wide crash events, Izraelevitz et al. [27] introduced an extended definition of linearizability that also takes into account system-wide crashes.

A history is said to be *durably linearizable* if the removal of all crash events from the history still leaves the history linearizable [27]. This means that all operations completed before a crash must take effect, plus some of the overlapping operations. If an operation survives a crash, all the operations it depends on, must also survive the crash. To satisfy *durable linearizability*, the execution model allows a recovery operation, which is called immediately after the crash, before any other operation is made.

A property that naturally fits crash survival is *lock-freedom*. An object is *lock-free* if there is progress always being made by at least one process. In particular, even if some processes pause, other processes will be able to continue executing their operations on the object. Furthermore, even if some process halts during an operation, other processes will continue executing, as no process may block another. It also means that a lock-free object always keeps the memory in a consistent state. We exploit this property in our algorithm to guarantee that after a crash, a linearizable lock-free data structure that uses our construction will be durably linearizable.

3 The Mirror Library

The *Mirror library* is a general interface that provides durability in an automatic manner. The main concept consists of having two copies of every variable, which allows separation between operations, such as read and write. Not only does this separation enable a significant performance improvement, but the *actual* memory type on which these operations execute also affects the throughput, e.g., DRAM vs NVMM, as presented in Section 6. The interface is simple to use, and eliminates the need to understand all the complexities that emerge when using non-volatile main memory. The library uses all three components of current existing

hardware: volatile caches, non-volatile main memory and a volatile DRAM.

3.1 Replica Location

Mirror keeps two replicas of each persistent variable. One replica is used for persistence and the other one for reading. This separation allows reading a consistent value only from the fast volatile memory. To ensure persistence, however, writes still have to be done on both the persistent and volatile memories. A careful implementation is needed to guarantee correctness, as shown in Section 4.

The first replica must be located in the NVMM, which provides durability. The second replica is located in the volatile main memory, the DRAM, where reads are more efficient. If a volatile main memory is not available, for example, when there is insufficient space for a large database, the second replica from which the reads are executed might be located on the persistent memory as well, which still yields advantages in read-heavy workloads, as presented in Section 6. For simplicity, we call the first replica the persistent replica rep_p, and the second replica the volatile replica rep_v.

3.2 Interface

The *Mirror* library is simply an extension of the std::atomic library [6] with added support for persistence on non-volatile main memory, by overloading the existing operations. Its API provides the exact API of *std::atomic* from the C++ standard, and two additional operations for the allocator usage.

compare_exchange_strong (T& expected, T newVal): Compares the current field's value with the expected value and, upon success, replaces the current value with the new one. Otherwise, loads the current object's value into the expected parameter. This is all done atomically on both replicas. When successful, both memory locations will have the new value. If the operation fails, neither of the memory locations will have the new value.

fetch_add (T add): Replaces the current value with the arithmetic's addition result of the current value and the parameter, atomically. This operations always succeeds, and guarantees that both memory locations will have the same value.

load (): Returns the current value atomically.

store (T desired): Stores a new value atomically. The same value is stored on both replicas.

There are two more operations made available to the programmer:

init(): Initializes the persistent and volatile memory regions. This operation needs to be called once at the beginning of the execution, and immediately after each system-failure. This operation mmaps a persistent and volatile memory region and copies the relevant data from the persistent to the volatile memory.

alloc(T value): A wrapper for allocating an object. This operation needs to be called every time an object is allocated to guarantee it will be adequately allocated on the volatile and persistent regions.

Last, the user needs to provide a tracing operation which is able to trace all data given the persistent roots, similar to previous works [17, 27, 41].

To use this interface, a variable *T* needs to be converted to *patomic<T>*. An example is shown in Figure 1. It shows how to convert an object with all its fields so that it can use this infrastructure. The example converts a node of Harris's linked list [21] into *patomic<>*. It calls the *init()* operation at the beginning of the execution and the allocator's wrapper every time an object is dynamically allocated (instead of the allocator itself), so that the variable is allocated on both replicas.

In Section 4, we explain how using this library guarantees atomicity and persistence.

```
1 class Node {
2   patomic<unsigned int> key;
3   patomic<T> value;
4   patomic<Node*> next;
5 }
```

Figure 1. Example of using the Mirror's Library

4 Mirror Underlying Mechanism

If an object needs to be persistent, i.e., survive a crash, every field *T* within this object needs to be converted to *patomic<T>*. It must use Mirror's allocator wrapper and provide a tracing operation. Moreover, if a data structure is lock-free and linearizable [22], using *patomic* version with all its fields will automatically convert it to be durable linearizable [27]. In what follows we explain the implementation of the Mirror library. This implementation does not require any compiler or operating system modification and it may be added to a new platform by any programmer to obtain the benefits of the Mirror transformation.

4.1 Implementation

To maintain consistency, every variable *T* that is converted to *patomic<T>* has two fields: an *std::atomic* value and an *std::atomic* sequence number, which is correlated to that value. The template class is presented in Figure 2.

4.1.1 Sequence Number. The sequence number is crucial for correctness, linearizability in the presence of two object copies. Every variable has an associated sequence number which is increased monotonically with every value change. The sequence number is important because we need to maintain two copies and avoid *ABA* problems. A sequence number, however, is needed only if the field changes so immutable fields, such as immutable keys, do not require it. The scenario shown in Figure 3 demonstrates why a value

```
1 template <typename T>
2 class patomic {
3   atomic<T> value;
4   atomic<uint64_t> sequence;
5 }
```

Figure 2. Patomic Class

change requires an additional sequence number: Let variable v start with holding the tuple $\{5, 2\}$, which represents a value 5 and a sequence number 2, both on its volatile and persistent replicas. Let process p_1 write the tuple $\{10, 3\}$, where the sequence number is increased by one in the persistent replica and pause before writing it to the volatile replica. Now, let another process p_2 write a new value, $\{5, 4\}$. Before doing it, it needs to make sure that both replicas have the same value. If this is not the case, p_2 *helps* p_1, meaning that it finishes p_1's write to the volatile replica, and only then changes the persistent value and the corresponding volatile value to be $\{5, 4\}$. If the sequence number had not existed, and the first process had continued its execution, it might be able to change the volatile replica to be 10 again, even though the sequence of the values $5 \rightarrow 10 \rightarrow 5 \rightarrow 10$ has not ever occurred. Attaching an increasing sequence number to it, however, guarantees this scenario can never happen. Now, p_1 will not be able to change the value $v = \{5, 4\}$ to be equal to $\{10, 3\}$ and the volatile replica will stay $\{5, 4\}$.

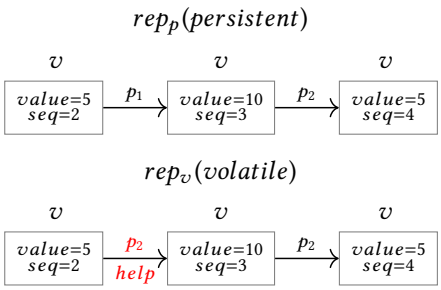

Figure 3. A scenario that could cause problems if there was no sequence number

4.1.2 CAS Instruction. As mentioned above, all reads are made on one replica, called rep_v and writes are made on both replicas to maintain consistency. The write is first made on the persistent replica, called rep_p, and then on rep_v. Reads are always wait-free and made on rep_v solely. The pseudo-code for the CAS operation is presented in Figure 4. Let v be a variable. v is located on both rep_v and rep_p. We start by checking that the variable v in rep_p, v_{rep_p}, is equal to the value of v in rep_v, v_{rep_v}, both in terms of the sequence number and the value itself. We read v's value and sequence number from the persistent and the volatile replicas in lines 5-16. Reading the value and its sequence number is not atomic, and, therefore, there is a need to assure that they are related. Accordingly, there is a read of the sequence number, followed by a read of the value, followed by another read of the sequence number. Only when the values are equal (after line 29), it is possible to write the new value to rep_p. The new value will contain the new value itself and a sequence number that is increased by 1 compared to the last written sequence number. The write is done with a *double-word-compare-and-swap (DWCAS)* instruction that swaps both the value and the sequence number atomically (line 40). Upon success, there is an attempt to write the same new value on rep_v (line 44). If this succeeds, then the write is finished after writing to the volatile replica. If there is a failure in writing to volatile memory, it means that there was a concurrent attempt to write the same value (or a more updated value) to rep_v. Either way, we can finish the operation. Note that it is not possible that the CAS fails due to an older value of v in the volatile replica, because we previously made sure that the volatile version had the expected value before updating the value on both replicas.

On the other hand, if there is a failure in writing to persistent memory, then we help the thread that succeeded by attempting to write v's value on rep_p to rep_v (line 47). Then we can return false. To maintain consistency, the helping is done only by changing the volatile replica from the value the helping thread read from the volatile replica before its failure, with the value that made its DWCAS operation in the persistent replica in line 40 to fail. The value that made it fail will be located in the before variable (the DWCAS updates this value if it fails). In addition, there is a special case where the concurrent thread might write the expected value to the persistent replica, and the DWCAS in line 40 will fail due to a different sequence number. In a regular CAS, the operation needs to succeed and therefore, we restart the operation in line 46.

The last remaining case is that the value read from rep_v is different from the value read from rep_p. In this case, there is an ongoing concurrent operation executing and we attempt to help it (line 19), by copying the value and sequence number from the persistent replica to the volatile replica. As we first write to the persistent replica and then to the volatile replica, the sequence number in the volatile memory and its correlated value, v_{rep_v}, can be at most off by one, in comparison to the persistent memory.

All other writing operations, e.g., *store*, *fetch_add*, are implemented by calling the *CAS* operation, with slight changes. As simple write and FAA instructions never fail, the implementation keeps on calling the *CAS* operation in a loop until it succeeds.

The method as described here does not support data structures that use double-word fields with a wide CAS, as we need to add a version to each field and modify the value and version atomically. In all algorithms with double-word fields that we are aware of, however, these fields contain a unique

value for each modification. In fact, most of these algorithms use on of the words for versioning. In such cases, the Mirror construction works well without adding an additional version word and can be applied as is. An ABA problem cannot occur in this case. The Mirror algorithm can be extended to also handle all other algorithms than use double-word CAS operations, but this extension is more involved and not needed in practice.

4.2 Persistent Roots

Loads and stores are used to access the non-volatile memory, which is mapped directly into the process address space. To access the NVMM, it is possible to *mmap* a direct access file, making all the data accessible through *persistent roots* [5, 8, 9, 39]. Persistent roots are simply known addresses from the mmaped file, which is located on the NVMM. Making all data accessible from these persistent roots and guaranteeing a consistent state, will assure complete recovery upon a crash. We assume that the NVMM mapping is always done to the same base address. In this way, all the pointers within that memory will remain in a consistent state.

4.3 Memory Reclamation

Every object may have two kinds of data: critical data and auxiliary data. Auxiliary data refers to data that may be recovered from the critical one, but critical data must persist for recovery. Therefore, there is a trade-off between run-time overhead and recovery time. On the one hand, we can save all data, both critical and auxiliary, and spend minimal time on recovery, but it incurs a costly overhead on the run-time itself. On the other hand, we may reduce the run-time overhead, and let recovery reconstruct all the auxiliary missing data. Exploiting the fact that there are two replicas of the data may allow us to ensure that the auxiliary data are read/written from the volatile replica, while the critical data are always written to the persistent replica. A natural fit for this will be all the metadata of the allocator. As we maintain two replicas, all allocator metadata may reside on the volatile replica due to the fact that the persistent replica behaves exactly the same, with just an offset. Moreover, having persistent roots allow us to reconstruct everything we need to continue executing upon a crash, without the need to read the metadata. Our technique guarantees that all reachable data from persistent roots is persistent. Therefore, upon a crash, it is easy to distinguish which data can be reclaimed. Moreover, if the reachable data is duplicated (by having two replicas), there is no need to manage the memory of both replicas. Thus, we manage the memory in the volatile replica, which yields more efficient reads and writes, and upon a crash, we need to traverse the persistent roots, and copy all the reachable data to the volatile memory. This, however, is possible only if a tracing operation is provided, as expected in previous works as well [13, 17, 27, 41]. For the volatile memory reclamation scheme, we use *ssmem*, an epoch based garbage collector (GC), and an object-based memory allocator [14].

Another possibility is to use a more costly technique, which persists only the allocator's core-data on the NVMM. Upon a crash, it re-constructs all the auxiliary data, and executes an offline GC. This offline GC simply traverses the persistent roots and reclaims all the data that is not reachable [5, 7].

4.3.1 Address Translation. To be able to manipulate both replicas, rep_v and rep_p, we assume that both base addresses are always mapped to the same virtual address space. As pointers are managed by the volatile memory, it is easy to translate the volatile address to the persistent address by simply adding the difference between the persistent base address and the volatile base address to the pointer itself. In other words, the delta between the two replicas is used to translate the addresses of the volatile and persistent memories. This technique is simple and the translation is efficient. If mapping to the same virtual address space is not possible after a crash, then another way of implementing the address translation might be to persist both the base address of the persistent memory and the offset between the volatile and persistent addresses from the persistent root so addresses will be calculated by using these offsets.

4.3.2 Init and Allocation. When a program begins to run, we mmap a direct access file to maintain the replica on the NVMM. First, we allocate the roots of the data structure itself in the persistent roots' region. Afterwards, for every allocated location we perform the allocation on the volatile space and then copy the variable with its sequence number to its matching persistent memory location according to the address translator. From that point, the memory is updated in both replicas, but metadata are managed only on the volatile replica.

Our underlying allocator's wrapper operation is responsible for constructing an object which is first constructed on the DRAM. We use the object-based memory allocator provided by David et al. [14], but any allocator can be used. After that, the object is constructed on the NVMM as well, without the metadata that are related to the allocator.

4.3.3 Recovery. After a crash, a recovery operation should be invoked before re-executing the program. We require the data structure to supply a tracing operation, which just traces all the reachable data from a set of roots. Once we resurrect the roots, we can use this routine to trace all the reachable objects on persistent space and re-allocate them. Since data is allocated on the NVMM as a copy of the volatile space, without persisting any metadata, it is not always easy to re-allocate the objects back on the volatile memory in the specific addresses dictated by the non-volatile locations. Practically, if the volatile allocator allows specifying the allocation addresses, then we can allocate the volatile objects

in the adequate locations and we are done. Otherwise, we can re-allocate all reachable objects on both memories from scratch. First we mmap a new file on the NVMM. Once objects are traced, we allocate every node on the volatile and also on the persistent memory with correlated addresses, and then the original mmapped file (used before the crash) can be deleted.

5 Correctness

We now describe briefly the correctness of our construction, showing that any linearizable [22] and lock-free data structure that uses our construction is persistent and satisfies durable linearizability [27], in particular.

A data structure is considered durably linearizable if all the operations that completed, survive upon a crash, plus some overlapping ones. The operations that were concurrent with the crash must survive if their effect has impacted other operations. By reading only from the volatile region, we make sure that any value read was already persisted, meaning that if that process executed a durable change, it already read the persisted values that influenced that change. This occurs because the Mirror's write persists the value right before writing it to rep_v either by the writer itself or by a helping one.

Theorem 5.1. *A linearizable and lock-free data structure that uses the Mirror construction provides a durably linearizable data structure.*

To prove Theorem 5.1, we first need to claim that our implementation for the load and store instructions yields the expected behavior. As mentioned above, a load reads a copy from the volatile memory, rep_v, and a store first writes the value to the persistent memory, rep_p and only then to rep_v. As all the store operations are implemented with the help of our Mirror' CAS implementation, presented in Figure 4, we describe only the relation between loads and CASes.

We start by describing the linearization points which are the actual moments where these operations take effect. Usually, these instructions are atomic, but in our implementation they are more complex. The load returns the value that is located in the volatile memory, atomically, according to Figure 5, even though the variable actually consists of an *atomic<T>* value and an *atomic<int64_t>* sequence number. Therefore, the linearization point of this operation is the actual load of the value itself. According to Figure 4, the linearization point of a successful CAS operation is the moment when the new value and the corresponding sequence number is written by a *DWCAS* instruction to the volatile memory, rep_v, after a successful write of the same value and sequence number to the persistent memory in line 40. The linearization happens between lines 40–44. The linearization point of an unsuccessful operation can occur in line 32 or in line 47.

```
1  template<typename T>
2  bool patomic<T>::compare_exchange_strong (T&
       expected, T newVal) {
3    patomic<T>* rep_p_addr = REP_V_2_REP_P(this);
4    while (true) {
5      rep_p_seq = rep_p_addr->seq; // Read rep_p
6      rep_p_val = rep_p_addr->val;
7      rep_p_seq_again = rep_p_addr->seq;
8
9      rep_v_seq = this->seq;      // Read rep_v
10     rep_v_val = this->val;
11     rep_v_seq_again = this->seq;
12
13     // Restart if seq and val inconsistent
14     if (rep_p_seq_again!=rep_p_seq ||
15         rep_v_seq_again!=rep_v_seq)
16       continue;
17
18     // Help to complete another ongoing write
19     if (rep_p_seq == rep_v_seq+1) {
20       FLUSH(rep_p_addr);
21       FENCE();
22       before = {rep_v_val, rep_v_seq};
23       after = {rep_p_val, rep_p_seq};
24       DWCAS(this, before, after);
25       continue;
26     }
27
28     // Make sure we have the same versions
29     if (rep_p_seq != rep_v_seq) continue;
30
31     // If value on rep_p is not expected, fail
32     if (rep_p_val != expected) {
33       expected = rep_p_val;
34       return false;
35     }
36
37     // Update rep_p
38     before = {rep_p_val, rep_p_seq};
39     after = {newVal, rep_p_seq+1};
40     bool res = DWCAS(rep_p_addr, before, after);
41     FLUSH(rep_p_addr);
42     FENCE();
43     if (res) {
44       DWCAS(this, before, after);
45     } else {
46       if (before.val == expected) continue;
47       DWCAS(this, {rep_v_val,rep_v_seq}, before);
48     }
49     return res;
50   }
51 }
```

Figure 4. Patomic Compare_exchange_strong Implementation

```
1 template<typename T>
2 T patomic<T>::load () {
3     return this->value.load();
4 }
```

Figure 5. Patomic Load Implementation

Lemma 5.2. *The implementation of the load and CAS operations is a linearizable implementation of an atomic variable.*

Before we prove this lemma, we prove some helping ones.

Lemma 5.3. *The persistent value can be changed at time t only if the sequence numbers on persistent and volatile memories match right before t.*

Proof. In line 29, a writing process checks whether the sequence numbers on the volatile and persistent memory are the same. Only after it has done so, will it try to change the persistent memory. As the change is done by a DWCAS and first done to persistent memory, it will succeed only if the current value on the persistent memory equals the expected value, which was equal on both the persistent and volatile memories. □

Lemma 5.4. *The sequence number in volatile memory is always lower by one or equal to the sequence number in persistent memory.*

Proof. According to Lemma 5.3, only if both the sequences on the persistent and volatile memories match, will there first be an attempt to change the persistent memory. If this attempt is successful, then after being equal, the sequences are at a distance of one, as the DWCAS changes the persistent memory to contain the current sequence raised by one due to lines 39–40. At this point, there is an attempt either by the same thread in line 44 or by others in line 24 or line 47 to make the sequence number on the volatile memory match the persistent memory. Moreover, to change the volatile sequence, the expected sequence is always lower by one than the current sequence in the persistent memory. The first that succeeds, match the sequence in volatile and persistent memories again. In addition, once the volatile sequence has changed, it might never get the same sequence number again, as sequence numbers on persistent memory are always monotonic, and as a consequence, on volatile memory as well. □

Lemma 5.5. *If the sequence numbers on volatile memory and persistent memory match, then the values must also match.*

Proof. DWCAS always updates the sequence number atomically with a related value, and there is only one value attached to a sequence number that is successfully written to the persistent memory. If the DWCAS fails, the next attempt will get a larger sequence number. Correspondingly, as values are first written to the persistent memory and only then to volatile memory, the process that succeeded in writing to the persistent memory will attempt to write the same value and sequence number to the volatile memory. According to lines 19–26, other processes can write to the volatile memory as well, but only the value that currently exists in the persistent memory, and will succeed only if the current sequence number of the volatile memory is lower by one than the current sequence number of the persistent memory. Therefore, there is always a match between the value and sequence number on the volatile and persistent memories. □

Let us proceed to the proof of Lemma 5.2.

Sketch Proof. Let v_a be an atomic variable with the value v, which currently exists in the persistent memory, implemented by the Mirror's load and CAS operations. Let us assume, w.l.o.g. that the sequence number that is related to the value v is s at time t. Let p_1 be the first process that attempts to change v to v' after t, and let p_2 be a process that loads the current value. We show that a load always reads the last successful CAS. According to Lemma 5.4, the sequence number of the volatile memory is distant from the sequence number of the persistent memory by at most one. Moreover, according to Lemma 5.5, the value corresponds to the sequence number. Thus, we have two possible scenarios:

1. Both replicas of the value and the sequence number are the same in the volatile and persistent memories at time t. If p_2 reads before p_1's attempt, i.e., when the values of the persistent and volatile memories have not changed, by the semantics of the std::atomic library, p_2 will return the value v as written in the volatile memory, as expected. If, however, p_2 reads after t, this means that p_1 has already attempted to change v_a. p_1 first tries to change the persistent memory, and succeeds (since it is the first one after t, no other process manages to change the persistent replica before this and the values remain the same). Afterwards, this process, and all the other concurrent processes that try to CAS as well help (or fail) to change the value in the volatile memory so that they match. In other words, there was a point in time where both values on persistent and volatile memories matched because if they did not, no other process could have changed the persistent memory again due to Lemma 5.3. This happens before line 44. If p_2 reads before that linearization point, it still reads the value v. Otherwise, it will read the value v', which is after the linearization point, which is the expected value. In both cases, p_2 reads the correct value. Any other process that tries to change v as well will fail if p_1 is the first one that attempts to change v_a. A special case exists where the expected value and the new value are the same. In this case, a process might still fail in line 40 as the sequence number might have changed but a regular CAS should not fail. If the former case occurs, this process will try

again (line 46). If another process fails due to p_1's success, it helps to update the volatile memory to match both replicas before its return.

2. The persistent replica is off by one from the volatile replica. In this case, p_1 will recognize this state in line 19 and help to update the volatile value so that it is equal to the persistent value. If p_1 is the first process that manages to update v to v', this means it reached line 40, i.e., it passed line 29 where both replicas are the same, and then we go back to the first case. If p_2 reads before the values match, i.e., before v reaches the volatile memory, it means that the operation that wrote value v was not linearized yet, and p_2 would return the current value of the volatile memory, which is the previous one, as expected. If it reads after v was written to rep_v, then the operation that wrote v was linearized and p_2 would return that value. If, however, p_1 fails to change the value in the persistent memory, one of the following might have happened: (1) the values did not match, meaning that another process has managed to change rep_p in contradiction to the fact that p_1 was the first one that attempted to change v_a or (2) the expected value was different than v. In this case, we expect the operation to fail, exactly as occurs in line 32.

□

We now outline the proof for Theorem 5.1.

Sketch Proof. To prove a data structure is durably linearizable we need to show that if we remove all crash events, the history remains linearizable. Let us consider a history H with one crash event c at the end (if there is more than one crash, the theorem could be proved by induction). We construct an equivalent history H' without a crash that contains all the persisted writes. By showing that history H' is linearizable, we prove that the original history H is durably linearizable. As mentioned above, all linearization points were already persisted before their occurrence, thus, all linearized operations survive a crash. Consider all running processes p_1, p_2, ..., p_i in H whose last instruction before the crash was a successful CAS on the persistent memory. As there might be only one successful CAS on a single location, there are different such locations. Our recovery operation simply copies the content of the persistent memory to the volatile memory, and linearizes those i successful CASes, as every location has only one successful CAS instruction. All other processes are paused and do not continue executing. The paused processes have not written anything to persistent memory (otherwise they would have been considered as among the mentioned i processes). Therefore, these processes have not changed the data structure, and have no effect. We get that H' is equivalent to H and linearizable. Since our data structure is lock-free, new processes will be able to continue executing from that state. □

6 Evaluation

6.1 Experimental Setup

We ran our measurements on an an Intel machine with two Xeon Gold 6234 processors, each with 8 cores, 3.3GHz max frequency and 2-way hyper-threading, which were disabled during the experiments to increase stability. The machine has 366GB of DRAM and 1.5TB of NVMM (Intel OptaneTM DC memory), organized as 12 × 128GB DIMMS (6 per processor). Each core has an L1 cache of 32KB and an L2 cache of 1MB. The L3 cache is 25MB per processor (8 cores). The operating system is Ubuntu 18.04.1, and code was written in c++ compiled using g++ (GCC) version 9.3.0. with -O3 optimization. We used an *App-Direct Mode Interleaved* in our configuration. For persisting objects, we called the *clwb* and *sfence* instructions for flush and fence, respectively. We use the *clwb(address)* instruction followed by an *sfence* to allow different write-backs to occur in parallel. To measure the influence of different flush instructions on our construction, we tried to use also *clflush* and *clflushopt* instead of *clwb* and got the same results up to a statistical error. We believe it happens due to the way our algorithm works, as there is DWCAS right after every flush instruction, which acts as a fence on Intel platform. In addition, current NVRAM platforms invalidate cache lines after they are flushed (even by clwb), which implies the same cache misses. Therefore, if clwb does not invalidate cache-lines in future platforms, it may reduce some of the advantage Mirror has today. To implement Mirror on ARM, the analogue instructions are *DC CVAP* and a full system *DSB* instruction for flush and fence execution.

To fix the memory reclamation and make it similar in all compared algorithms, we used the *libvmmalloc* implementation of the PMDK library [25] and the durable version of the *ssmem* memory manager for all compared algorithms. ssmem is the same allocation method used in related work [17, 41]. We work with key-value pairs, both of size 8B. Nodes are cache-aligned to 128B. The reported results are averages of 10 repetitions, each ran for 5 seconds. We used a uniform random key distribution from the range of $[0, r-1]$ for varying r's. Every data structure was initialized with $r/2$ keys before the run, and measured with varying percentages of reads that cover the standard YCSB benchmark [11]: A (50% reads), B (95% reads), and C (100% reads). In other experiments, we also ran the frequently used workload of 10% inserts, 10% deletes and 80% read operations.

We evaluated the performance of our construction on four different linearizable and lock-free data structures: A Linked-List [21], A Hash-Table, (based on Harris et al.'s [21] with a linked-list in every bucket), a lock-free BST by Aravind et al. [34] and a lock-free Skip-List [16]. We compared our general construction with two other general constructions: (1) Izraelevitz et al.'s [27] construction that adds a flush and a fence for every shared read/write operation, and (2) the

NVTraverse [17] construction that can be applied to traversal data structures (defined in [17]) and removes the need for persisting traversals. Data structures generated by NVTraverse apply flushes and fences only to reads and writrs of fields in the nodes around the "destination" of the operation, where operations actually take effect. We also added the state-of-the-art ad hoc construction for sets: *SOFT* and *LinkFree* by Zuriel et al.'s [41]. This allowed testing the performance of data structures automatically output by Mirror to highly optimized hand made data structures whose design requires high expertise. In addition, we also tested our construction against an Intel's engine, Cmap in pmemkv [38], which is a persistent lock-based key-value datastore.

6.2 One Replica on DRAM

We first checked the performance where the volatile replica of the data structure output by the Mirror transformation is placed in the volatile memory (DRAM), and the other (persistent) replica is placed in the non-volatile memory. This configuration enabled both advantages of the Mirror transformation: no persisting of reads, and fast DRAM read executions.

6.2.1 List Scalability. The results for the Linked-List's scalability are shown in Figure 6 (a). We ran $1 - 16$ threads, where above 8 threads, we cross the NUMA-node boundaries, which leads to some threads reading from further and slower memory. The list is initialized with 128 keys over a key range of $0 - 255$, and the workload executes randomized 80% look-ups, 10% inserts and 10% deletes. Since the list is small, it resides on the cache, and therefore sometimes the original non-persistent list *ListOriginalDRAM* from [21], has the same throughput as its version *ListOriginalNVMM* that executes on the NVMM, up until crossing the node boundaries. (The non-persistent version executes no flushes or fences). Crossing the NUMA-node boundaries makes data go through the memory and then the use of DRAM implies faster execution.

Looking at the three general techniques for durability, we see that the persistent list output by the Mirror transformation outperforms the list output by *NVTraverse* [17] by $2.88x - 8.7x$ on $1 - 16$ threads respectively. The list output by NVTraverse outperforms the Izraelevitz et al. [27] list by $29x$ for one thread, $7.7x$ for 8 threads and $5.6x$ for 16 threads. Throughout the experiments, this advantage of the data structures' output by Mirror over their competitors is evident. It remains to compare to the hand made version of the linked list of Zuriel et al. [41], we notice that our list and Zuriel's list are comparable up to 4 threads, but when contention becomes higher, meaning that more writes occur because there are more running threads, Mirror's list outperforms Zuriel's list by up to 35% on 8 running threads. More threads means more writes, more write-backs, and more cache misses that are served faster on DRAM for the Mirror data structure. The NUMA effects are more chaotic, making the Mirror list extremely successful, but we leave the study of NUMA behavior on non-volatile memory to future work.

6.2.2 List Varying Size. Next, we measured the throughput for varying list sizes, with 8 running threads, and a workload of 80% reads, 10% inserts, and 10% deletes. The results appear in Figure 6 (b). Notice that as lists become larger with dominating traversal times, the differences between the various implementations become less noticeable. When traversals dominate performance, the different number of flushes/fences become less crucial. Long lists are not used in practice as it does not make sense to traverse a thousand nodes in order to locate the desired key.

We mostly see the same trends that we saw earlier. Up until the range of 8192, we notice that the Mirror construction outperforms NVTraverse by $3.6x - 1.25x$ on ranges of $256 - 2048$ keys. For larger ranges of keys, we get comparable performance.

6.2.3 List Updates. Figure 6 (c) studies the performance of workloads with various percent of updates with 8 running threads, and a list of size 128. On the read-only workload with 0% updates, loads are obtained from the cache so we expect all algorithms that do not execute any flush/fence instructions to perform excellently. This includes both original non-durable versions (either executing on DRAM or on NVMM), the Mirror list, and Zuriel's hand-made Link-Free algorithm. The 95% confidence intervals for all these algorithms overlap. Zuriel's SOFT behaves differently because it consumes more space with its split nodes. As the update percentage grows, the performances of all implementations decrease, as there is more contention of writes on the data structure, but the trend among the compared algorithms remains the same.

6.2.4 Hash-Table, BST and Skip-List Scalability. Moving to much larger data structures that do not fit into the cache, we now examine the hash-table, the binary search tree (BST) and the skip-list scalability, presented in Figures 6 (d), (g) and (j). We ran $80\% - 10\% - 10\%$ look-ups-inserts-deletes with a structure size of $8M$ nodes. Here, memory accesses (reads and writes) from the main memory, whether the DRAM or the NVMM, dominate the performance. Accessing the DRAM is much cheaper, and on a single thread execution we see Mirror's hash-table outperforming the hash-table generated by NVTraverse, and Zuriel's hash-tables by a factor of 1.8x. These three competitors behave similarly to each others. With 8 threads the difference grows to 2x and with 16 threads to 2.5x. For the BST, we see Mirror's BST outperforming NVTraverse's BST by a factor of 1.84x-2.33x on $1 - 16$ threads. For the skip-list, the difference even grows to a factor of 2.1x-2.65x on $1 - 16$ threads. The fact that reads are never persisted, in addition to the fact that reads access

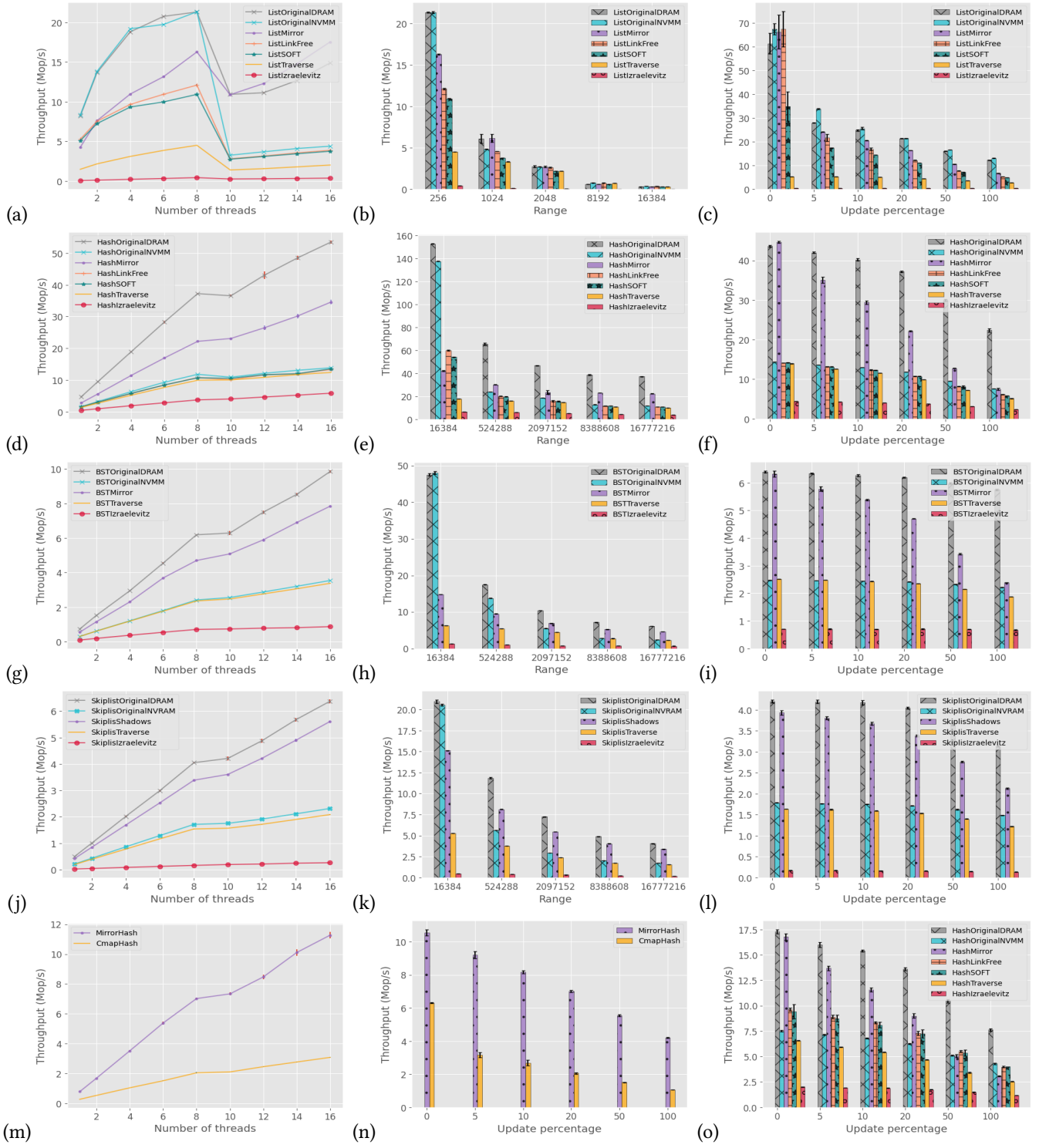

Figure 6. NVMM throughput results with one replica of Mirror placed on DRAM (a). Linked-List, varying number of threads, 80% lookups, 128 nodes. (b). Linked-List, varying size, 8 threads, 80% lookups. (c). Linked-List, varying update percentage, 8 threads, 128 nodes. (d). Hash-Table, varying number of threads, 80% lookups, 8M nodes. (e). Hash-Table, varying size, 8 threads, 80% lookups. (f). Hash-Table, varying update percentage, 8 threads, 8M nodes. (g). BST, varying number of threads, 80% lookups, 8M nodes. (h). BST, varying size, 8 threads, 80% lookups. (i). BST, varying update percentage, 8 threads, 8M nodes. (j). Skip-List, varying number of threads, 80% lookups, 8M nodes. (k). Skip-List, varying size, 8 threads, 80% lookups. (l). Skip-List, varying update percentage, 8 threads, 8M nodes. (m). Hash-Table, varying number of threads, 80% lookups, 8M nodes. (n). Hash-Table, varying update percentage, 8 threads, 8M nodes. (o). Hash-Table, varying update percentage, 8 threads, 32M nodes.

DRAM, benefits the Mirror BST and skip-list considerably, especially with data structures that do not fit in the cache.

6.2.5 Hash-Table, BST and Skip-List Varying Sizes. We now consider the effect of running varying sizes of hash-tables BTSs and skip-lists with 8 threads, and 80% − 10% − 10% look-ups-inserts-deletes. The results are depicted in Figures 6 (e), (h) and (k). Excluding the smallest 8*K* size, the data structures do not fit in the cache and need to be read from the memory. Since Mirror's data structures use two replicas (and double the memory consumption) they suffer more cache misses. This is why they do not perform as well as the original (non-persistent) versions of the data structures and Zuriel's hand-made (persistent) data structure on 8K structures' size. Nevertheless, the data structures generated by the Mirror transformation always outperform the data structures output by NVTraverse and Izraelevitz's et al. When the size of the data structure grows to 256K keys, no implementation is small enough to fit in the Last Level Cache (LLC) and we see the same trends as before: reading from the NVMM and persisting the reads downgrades the performance. The Mirror's hash-table outperforms Zuriel et al.'s link-free hash-table by a factor of 1.5x–3x, and it outperforms NVTraverse by a factor of 2.8x–3.4x, for the larger (realistic) structure sizes of 255K–8M nodes.

6.2.6 Hash-Table, BST and Skip-List Updates. We also evaluated the hash-tables, BST and skip-lists to examine the impact of various update workloads. We ran 8 threads with 8*M* nodes. The related graphs are presented in Figures 6 (f), (i) and (l). We also tested a larger data structure, a hash-table with 8 threads and 32*M* nodes ,to see if it makes any difference. The results are depicted in Figure 6 (o). On a read-only workload, the data structures generated by the Mirror transformation perform similarly to the non-persistent original data structure executing on the DRAM, simply because they both access the DRAM only. But as the percentage of writes increase, the throughput of the Mirror data structures degrades, as writes are executed on the non-volatile memory as well. Nevertheless, Mirror's structures beats all other persistent data structure significantly, in both sizes. The only place where SOFT and Link-Free results are better than Mirror's results are in 32*M* nodes and above 50% updates.

6.2.7 Lock-Based Key-Value Datastore. To compare our construction with a lock-based data structure, we used the concurrent key-value store from Intel's pmemkv library [38], which is optimized for persistent memory. We tested Cmap, which was the only concurrent non-experimental engine that was provided at the time we tested. Since it was based on a hash-table, we ran it against Mirror's hash-table. We used the pmemkv-bench [26] which is a testing framework based on db-bench from LevelDB [19] and RocksDB [15].

We ran the tests with 8*M* keys of a size 8*B*. The value size was 8*B* as well and the benchmark executes randomized reads and writes. The scalability of both of those key-value stores are shown in Figure 6 (m) with 80% reads and 20% writes. We notice that the Mirror's construction outperforms Cmap significantly as Mirror is lock-free and takes advantage of its DRAM copy. Mirror's hash-table outperforms Cmap by 2.85x-3.65x on 1 − 16 threads.

The same trend is shown for various update workloads with 8 running threads, depicted in Figure 6 (n). Mirror performs better by 1.67x-3.95x on 0% − 100% writes respectively.

We conclude this part of the evaluation, where the volatile replica is placed on the DRAM, that even though the Mirror construction executes two writes, it still outperforms other durable constructions due to its usage of DRAM. Next, we check how Mirror's data structures behave when both replicas are placed on non-volatile memory. It may be important for possible future architectures in which DRAM will not be incorporated.

6.3 Mirror with Both Replicas on NVMM

Since volatile memory may not be available in some future platforms, it is interesting to check how Mirror's data structures performs when we allocate both of the replicas on the persistent memory. We still expect to see benefits from never persisting a read value, but we also expect reduced performance due to slower read accesses to the volatile replica (which is now on non-volatile memory), and also writing twice is more costly. In the evaluation that follows, we see that when the two replicas reside on non-volatile memory, Mirror is competitive with NVTraverse, sometimes better and sometimes worse, yet Mirror is somewhat easier to implement.

6.3.1 The Linked List. Figures 7 (a) - (c) show the performance of the linked list with the same workloads as in Figure 6. As expected, writing to NVMM has a much higher cost than writing to the DRAM, as we write twice to the two replicas. Mirror's data structures still perform better than state-of-the-art general constructions (NVTraverse and Izraelevitz). Nevertheless, Zuriel's hand-made lists, the SOFT and Link-Free, perform better on longer lists and on workloads with more than 20% writes. Link-Free and SOFT use an optimization that eliminates repeated redundant persisting operations. This optimization helps most with a low percentage of updates and low contention. But managing this optimization with more than 20% updates is costly and not as useful. This is why Mirror's data structures compete well with the Zuriel's manually optimized data structures in workloads with higher update percentage. As before, when the list grows, the traversals dominate the performance, and the differences between implementations reduce.

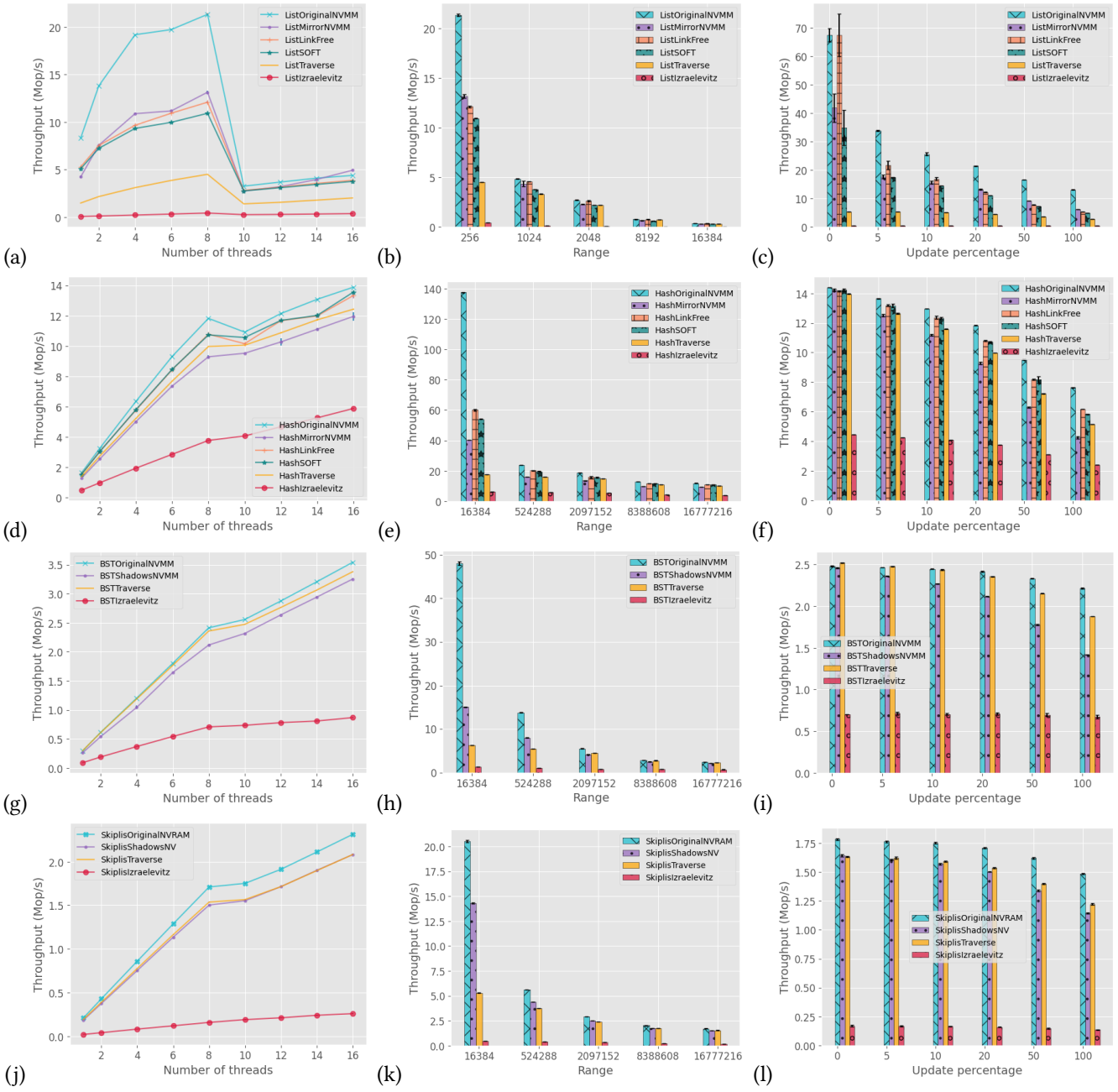

Figure 7. NVMM throughput results. Both copies of Mirrors on NVMM (a). Linked-List, varying number of threads, 80% lookups, 128 nodes. (b). Linked-List, varying size, 8 threads, 80% lookups. (c). Linked-List, varying update percentage, 8 threads, 128 nodes. (d). Hash-Table, varying number of threads, 80% lookups, 8M nodes. (e). Hash-Table, varying size, 8 threads, 80% lookups. (f). Hash-Table, varying update percentage, 8 threads, 8M nodes. (g). BST, varying number of threads, 80% lookups, 8M nodes. (h). BST, varying size, 8 threads, 80% lookups. (i). BST, varying update percentage, 8 threads, 8M nodes. (j). Skip-List, varying number of threads, 80% lookups, 8M nodes. (k). Skip-List, varying size, 8 threads, 80% lookups. (l). Skip-List, varying update percentage, 8 threads, 8M nodes.

6.3.2 Hash-Table, BST and Skip-List. The Hash-Table, the BST and the Skip-List results are shown in Figures 7 (d) - (l). Interestingly, we see that the extra cost of writing to two different memory locations has a significant impact in this case, and Mirror does not come out best. In workloads of at most 10% updates, Mirror's data structures are comparable to those of NVTraverse, because the extra writing cost is not as significant as the benefit of eliminating the need to persist the reads. In contrast, for workloads that require more than 20% writes, NVTraverse becomes better. Moreover, the

optimization for the hand-made implementations, and the fact that they do not persist pointers, avoids extra flushes and fences, which gives such implementations an advantage over the general techniques. In terms of data structure size, for smaller data structures we see an advantage for Mirror, but NVTraverse becomes faster as the size grows.

7 Related Work

There are several general techniques for obtaining durable data structures from lock-free ones. Izraelevitz et al. [27] presented a general technique that provides durable linearizability and can be applied to any lock-free data structure. This generality comes at the cost of efficiency: a *fence* must be inserted before every write operation followed by a *flush*, while a *flush* and a *fence* must be issued after every read operation. The second insertion is particularly inefficient, given that most data structure operations perform multiple loads on different memory locations and executing one *flush* followed by a *fence* for each of them is costly. To address this hight overhead, Zuriel et al. [41] presented *SOFT* and Link-Free, a technique applicable to set-like data structures, which eliminates the need to persist pointers.

Friedman et al. [17] introduced NVTraverse, which is a general technique for constructing durable data structures. NVTraverse reduces the cost of most of the loads for *traversal data structures*, defined in [17]. *Traversal data structures* start with a read-only phase, called *traversal* followed by a *critical* phase that may perform updates. Using the NVTraverse construction, most of the read values from the traversal phase are not persisted, reducing the persistency cost. NVTraverse provides an automatic way to insert flushes and fences but it requires the lock-free data structure to be in a special traversal form. Transforming a data structure into a traversal form may require some efforts and proving that it satisfies all the traversal conditions require some expertise. Moreover, in some cases conversion to the traversal form may reduce efficiency, e.g. trimming virtually deleted elements during a traversal in a linked-list. In terms of applicability, we are not aware of any known data structure that cannot be converted to the traversal form. Unlike Mirror, NVTraverse does not require underlying services such as a specific allocator and it has a smaller footprint.

Many generic techniques exist that can take a sequential implementation of a data structure and make it concurrent and durable [20, 28–32, 40]. Some of these constructions are capable of generating durable data structures with lock-free progress [4, 10, 12, 37]; however, these transformations serialize update operations, putting them at a severe scalability disadvantage when it comes to write-intensive workloads, as shown in [17].

Pronto [33] is a recent technique that combined DRAM with NVMM to yield high throughput for lock-based concurrency. In Pronto, two replicas of the data structure exist, one in DRAM and the other in NVMM. The data structure code must be modified using a set of specified rules.

8 Conclusions

In this paper we presented a simple and effective automatic transformation from lock-free linearizable data structures to persistent lock-free data structures that satisfy durable linearizability. Our transformation can make use of a hybrid system where non-volatile main memory co-exists with conventional DRAM and generate data structures that are extremely efficient in this environment. Evaluation of a linked-list, a hash table, and a binary search tree demonstrates that our method always beats state-of-the-art transformations for lock-free data structures, sometimes by a factor of up to 10x, depending on the percentage of read operation, the size of the structure, and the number of executing threads. Applying the transformation on a given lock-free data structure is easy, involving type annotation to replace the usage of `std::atomic` with `patomic`, replacing the calls to the system allocator with the Mirror allocator, and providing a traversal function for the nodes in the data structure.

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, and by an Azrieli PhD Fellowship.

References

[1] AMD. [n.d.]. *AMD64 Architecture Programmer's Manual.* https://www.amd.com/system/files/TechDocs/24594.pdf

[2] ARM. 2018. *ARM Architecture Reference Manual ARMv8.* https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf

[3] Hillel Avni and Trevor Brown. 2016. Persistent Hybrid Transactional Memory for Databases. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 409–420. https://doi.org/10.14778/3025111.3025122

[4] H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. 2020. Nonblocking Persistent Software Transactional Memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 429–430. https://doi.org/10.1145/3332466.3374506

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 677–694. https://doi.org/10.1145/2983990.2984019

[6] C++. [n.d.]. *Std::atomic Library.* https://en.cppreference.com/w/cpp/atomic

[7] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM 2020)*. 60–73.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon,

USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

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

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB.

[12] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent Memory and the Rise of Universal Constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/3342195.3387515

[13] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures.

[15] Facebook. [n.d.]. *RocksDB*. https://github.com/facebook/rocksdb

[16] Keir Fraser. 2003. Practical lock-freedom.

[17] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[18] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 28–40. https://doi.org/10.1145/3178487.3178490

[19] Google. [n.d.]. *LevelDB*. https://github.com/google/leveldb

[20] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 913–928.

[21] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. Springer, 300–314.

[22] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. 12, 3 (1990), 463–492.

[23] Intel. [n.d.]. *Developers Intel64 and IA-32 Architectures Software Manuals Combined*. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[24] Intel. [n.d.]. *Intel Architecture Instruction Set Extensions Programming Reference*. https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[25] Intel. [n.d.]. *Persistent Memory Library*. https://pmem.io.

[26] Intel. [n.d.]. *pmemkv-bench*. https://github.com/pmem/pmemkv-bench

[27] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[28] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 452–465. https://doi.org/10.1109/ISCA.2018.00045

[29] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[30] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 329–343.

[31] Virendra J. Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent Memory Transactions. *CoRR* abs/1804.00701 (2018). arXiv:1804.00701 http://arxiv.org/abs/1804.00701

[32] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[33] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 789–806. https://doi.org/10.1145/3373376.3378456

[34] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. ACM.

[35] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (2019), 31 pages.

[36] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (2019).

[37] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 151–163.

[38] Steve Scargall. 2020. *pmemkv: A Persistent In-Memory Key-Value Store*. Apress, Berkeley, CA, 141–153. https://doi.org/10.1007/978-1-4842-4932-1_9

[39] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[40] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 46, 3 (March 2011), 91–104. https://doi.org/10.1145/1961296.1950379

[41] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets.