MOD: Minimally Ordered Durable Datastructures for Persistent Memory

Swapnil Haria*
University of Wisconsin-Madison
swapnilh@cs.wisc.edu

Mark D. Hill
University of Wisconsin-Madison
markhill@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

Abstract

Persistent Memory (PM) makes possible recoverable applications that can preserve application progress across system reboots and power failures. Actual recoverability requires careful ordering of cacheline flushes, currently done in two extreme ways. On one hand, expert programmers have reasoned deeply about consistency and durability to create applications centered on a single custom-crafted durable datastructure. On the other hand, less-expert programmers have used software transaction memory (STM) to make atomic one or more updates, albeit at a significant performance cost due largely to ordered log updates.

In this work, we propose the middle ground of composable persistent datastructures called Minimally Ordered Durable datastructures (MOD). We prototype MOD as a library of C++ datastructures—currently, map, set, stack, queue and vector—that often perform better than STM and yet are relatively easy to use. They allow multiple updates to one or more datastructures to be atomic with respect to failure. Moreover, we provide a recipe to create additional recoverable datastructures.

MOD is motivated by our analysis of real Intel Optane PM hardware showing that allowing unordered, overlapping flushes significantly improves performance. MOD reduces ordering by adapting existing techniques for out-of-place updates (like shadow paging) with space-reducing structural sharing (from functional programming). MOD exposes a *Basic interface* for single updates and a *Composition interface* for atomically performing multiple updates. Relative to widely used Intel PMDK v1.5 STM, MOD improves map, set, stack, queue microbenchmark performance by 40%, and speeds up application benchmark performance by 38%.

*Now at Google.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378472

CCS Concepts. • **Software and its engineering → Software libraries and repositories**; **Software fault tolerance**; **Software performance**; • **Information systems → Storage class memory**.
Keywords. crash-consistency, durability, datastructures, persistent memory.

ACM Reference Format:
Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378472

1 Introduction

Persistent Memory (PM) is here—Intel Optane DC Persistent Memory Modules (DCPMM) began shipping in 2019 [21]. Such systems expose fast, byte-addressable, non-volatile memory (NVM) devices as main memory and allow applications to access this persistent memory via regular load/-store instructions. In fact, we ran all experiments in this paper on a system with engineering samples of Optane DCPMM [22, 23].

The durability of PM enables *recoverable applications* that preserve in-memory data beyond process lifetimes and system crashes, a desirable quality for workloads like databases, key-value stores and long-running scientific computations [4, 28]. Such applications use cacheline flush instructions to move data from volatile caches to durable PM and order these flushes carefully to ensure consistency. For instance, applications must durably update data before updating a persistent pointer to the data, or atomically do both.

However, few recoverable PM applications have been developed so far, though libraries like Mnemosyne [46] and Intel Persistent Memory Development Kit (PMDK) [19] have existed for several years. Currently, there are two approaches to building such applications: single-purpose custom datastructures (e.g., persistent B-trees [6, 44, 48]) or general-purpose transactions. Both approaches have some benefits, but we believe that neither is suitable for developers building new PM applications.

Although custom datastructures are typically fast, significant effort is needed in designing these structures to ensure that updates are performed atomically with respect to failure, i.e., either all modified data is made durable in PM or none.

Designers need to ensure that modified data is logged and dirty cachelines are explicitly flushed to PM in a deliberate order enforced by the use of appropriate fence instructions for consistency. Moreover, performance optimizations useful in one datastructure may not generalize to others. Finally, custom datastructures typically do not support the composition of failure-atomic operations spanning multiple datastructures, e.g., popping an element of a durable queue and inserting it into a durable map.

At the other extreme, existing PM libraries offer software transactional memory (PM-STM) for building crash-consistent code, but with complicated interfaces and high performance overheads. Operations on existing datastructures are wrapped in transactions to facilitate consistent recovery on a crash. These transactions also allow developers to compose failure-atomic operations that update multiple datastructures. However, existing PM-STM interfaces can be hard to use correctly. For instance, PMDK transactions require programmer annotations (`TX_ADD`) to specify memory locations that may be modified within a transaction. Incorrect usage of such annotations is a common source of crash-consistency bugs in applications built with PMDK [31].

Moreover, the generality of transactions comes with performance overheads, mainly from flushes to PM. As we will show, on average 64% of the overall execution time in PM-STM based applications is spent in flushing activity. These high overheads arise from excessive ordering constraints in these transactions, with each transaction having 5-11 ordering points (i.e., `sfence` on x86-64). Our experiments on Optane DCPMM show that flushes (i.e., `clwb` on x86-64) slow execution more when they are more frequently ordered. For instance, 8 `clwbs` can be performed 75% faster when they are ordered jointly by a single `sfence` than when each `clwb` is individually ordered by an `sfence`.

To make PM application development more widely accessible, we propose a middle ground: Minimally Ordered Durable Datastructures (MOD) [16], a library of many persistent datastructures with simple abstractions and good performance (and a methodology to make more). MOD performs better than transactions in most cases and also allows the composition of updates to multiple persistent datastructures. To allow the programmer to easily build new PM applications, we encapsulate away the details of persistence such as crash-consistency, ordering and durability mechanisms in the implementation of these datastructures. Instead, MOD enables programmers to focus on core logic of their applications.

Similar efforts such as the Standard Template Library (STL) [11] in C++ have proved extremely popular, allowing programmers to develop high-performance applications using simple datastructure abstractions whose efficient and complicated implementations are hidden from the programmer. MOD offers datastructure abstractions similar to those in the STL, namely map, set, stack, queue and vector. For each datastructure, MOD offers convenient failure-atomic update and lookup operations with familiar STL-like interfaces such as `push_back` for vectors and `insert` for maps.

New abstractions get wider adoption only if they perform well. For high performance, MOD datastructures use shadow paging [15, 32] to minimize internal ordering in update operations—one `sfence` per failure-atomic operation in the common case. Specifically, we rely on out-of-place writes to create a new and updated copy (*shadow*) of each datastructure without overwriting the original data. As partial updates are no longer a correctness concern, these out-of-place writes do not need to be logged and can be flushed with overlapping flushes to minimize flushing overheads.

To reduce the memory overhead introduced by shadow paging, our datastructures use *structural sharing* optimizations found in purely functional datastructures [13, 36, 38, 41, 43]. With these optimizations, the updated shadow is built out of the unmodified data of the original datastructure plus modest new and updated state. Consequently, the shadow incurs additional space overheads of less than 0.01% over the original datastructure. On Intel Optane DCPMM, our MOD datastructures improve the performance of map, set, stack, queue microbenchmarks by 43%, hurts vector by 122%, and speeds up application benchmarks by 36% as compared to Intel PMDK v1.5. We also present a methodology to repurpose other existing purely functional datastructures into new persistent datastructures.

Finally, MOD offers the ability to compose failure-atomic updates to multiple durable datastructures. Only for such use cases, we expose the underlying out-of-place update operations to the programmer. Thus, programmers can update multiple datastructures to generate new versions of these datastructures. We provide a convenient `Commit` interface to failure-atomically replace all original datastructures with their respective updated versions.

We make the following contributions in this paper:

- We present the design, interfaces and implementation of the MOD library of high-performance durable datastructures with encapsulated persistence.
- We provide a recipe to create more MOD datastructures from existing functional datastructures.
- We develop an analytical model for estimating the latency of concurrent cacheline flushes on Optane.
- We release a C++ library of MOD datastructures, documented in the Artifact Appendix.

2 Background

We first provide basic knowledge of PM programming and functional programming as required for this paper.

2.1 Persistent Memory System

We consider a system in which the physical address space is partitioned into volatile DRAM and durable PM. While the contents of PM are preserved in case of a system failure,

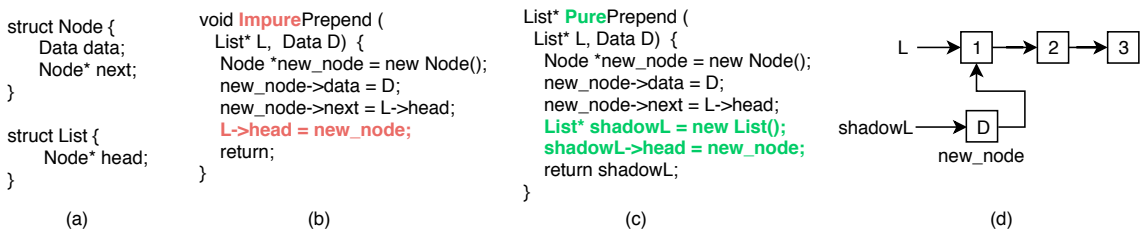

Figure 1. For linked list defined in (a), implementation of prepend as (b) impure function with original list L modified and (c) pure function where new updated shadowL is created. (d) shadowL reuses nodes of list L for efficiency.

DRAM and other structures such as CPU registers and caches are wiped clean. This system model is similar to most prior work [5, 30, 33, 46] and representative of Optane DCPMM.

Recoverable PM software rely on hardware guarantees to know when PM writes are *persisted*, i.e., when a write is guaranteed to be durable in PM. Writes are first stored in volatile caches to exploit temporal locality of accesses and written back to PM at a later time unknown to software, depending on the cache replacement policy. Hence, PM systems support two instructions for durability and/or ordering: a flush instruction to explicitly writeback a cacheline from the volatile caches to PM, and a fence instruction to order subsequent instructions after preceding flushes become durable.

2.2 Persistent Memory Programming

Here, we discuss applications that rely on the persistence of PM. Such applications are *recoverable* if they store enough state in PM to successfully recover to a recent state and without losing all progress after a system crash. There are several challenges involved in programming recoverable applications. Sufficient application data must be persisted in PM to allow successful recovery to a consistent and recent state. System crashes at inopportune moments could result in partially updated and thus inconsistent datastructures that cannot be used for recovery. As a result, programmers have to carefully reason about the ordering and durability of PM updates. Unfortunately, PM updates in program order can be reordered by hardware including write-back caches and memory controller (MC) scheduling policies.

To hide these programming challenges, researchers have developed *failure-atomic sections* (FASEs) [5]. From a programmer's point of view, any PM writes contained in a FASE are guaranteed to be atomic with respect to system failure. For example, prepending to a linked list (Figure 1b) in a FASE guarantees that either the linked list is successfully updated with its head pointing to the durable new node or that the original linked list can be reconstructed after a crash.

PM libraries [7, 19, 46] typically implement FASEs with software transactions guaranteeing failure-atomicity and durability. All updates made within a transaction are durable when the transaction commits. If a transaction gets interrupted due to a crash, write-ahead logging techniques are typically used to allow recovery code to clean up partial updates and return persistent data to a consistent state. Hence, recoverable applications can be written by allocating datastructures in PM and only updating them within PM transactions. We discuss the performance bottlenecks of PM transactions in Section 3.

2.3 Functional Programming Concepts

In this work, we leverage two basic concepts from functional programming languages: pure functions and purely functional datastructures. These ideas are briefly described below and illustrated in Figure 1.

Pure Functions. A pure function is one whose outputs are determined solely based on the input arguments and are returned explicitly. Pure functions have no externally visible effects (i.e., side effects) such as updates to any non-local variables or I/O activity. Hence, only data that is newly allocated within the pure function can be updated. Figure 1 shows how a pure and an impure function perform a prepend operation to a list. The impure function overwrites the head pointer in the original list L, which is a non-local variable and thus results in a side effect. In contrast, the pure function allocates a new list shadowL to mimic the effect of the prepend operation on the original list and explicitly returns the new list. Note that the pure function does not copy the original list to create the new list. Instead, it reuses the nodes of the original list without modifying them.

Functional Datastructures. In functional languages, purely functional or *persistent datastructures* are those that preserve previous versions of themselves when modified [13]. We refer to them only as functional datastructures to avoid confusion with persistent (i.e., durable) datastructures for PM.

Once created, purely functional datastructures can not be modified. To update such datastructures, a logically new version is created without modifying the original. Thus these datastructures are inherently multi-versioned.

To reduce space overheads and improve performance, functional datastructures (even arrays and vectors) are often implemented as trees [36, 38]. Tree-based implementations allow different versions of a datastructure to appear logically different while sharing most of the internal nodes of the tree. For example, Figure 1 shows a simple example where

the original list L and the updated list shadowL share nodes labeled 1, 2 and 3. This class of optimizations is referred to as *structural sharing*.

3 Mitigating Performance Bottlenecks

Good performance typically aids the adoption of new abstractions. Thus in this section, we seek to identify the main performance bottlenecks in PM-STM workloads and understand how to mitigate these overheads.

Overheads in PM-STM Workloads. At a high level, PM-STM implementations suffer from two main overheads: flushing (required for durability of data) and logging (required for failure-atomicity). We measured these overheads on Optane DCPMMs by running recoverable PM workloads (Table 2, Section 6) with Intel PMDK v1.5, a state-of-the-art PM-STM implementation that uses hybrid undo-redo logging. As shown in Figure 2, these applications spend on average about 64% of their execution time performing flushing and 9% performing logging operations. These PM-STM implementations flush both log entries and data updates to PM, and we consider the time spent in flushing log entries as part of flushing overheads. Clearly, flushing overheads are the biggest performance bottlenecks in these applications.

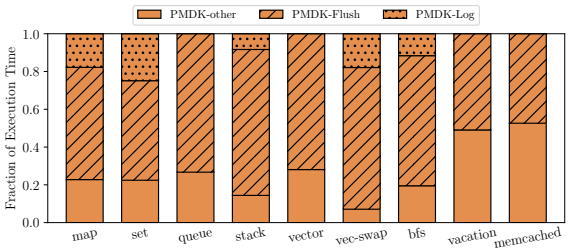

Figure 2. Fraction of execution time spent logging and flushing data in PM workloads using PMDK v1.5.

As we show in the rest of this section, the high flushing overheads in PM-STM are caused by excessive ordering constraints (`sfence`) limit the overlapping of long-latency flush instructions. Undo-logging techniques typically require 5-50 fences [33] per transaction. These fences mainly order log updates before the corresponding data updates. In some implementations, the number of fences per transaction scales with the number of modified cachelines. In our workloads with hybrid undo-redo logging, we observed 4-23 flushes and 5-11 fences per transaction (Figure 10 in Section 6). Consequently, the median number of flushes overlapped per fence is 1-2, resulting in high flush overheads.

Flushes on Test Machine. In this paper, we focus on the `clwb` instruction that writes back a dirty cacheline but may not evict it from the caches. This instruction commits instantly but launches an asynchronous flush that is unordered with other flushes to different addresses, as shown in Figure 3. Ordering points (`sfence`) stall the CPU until all inflight

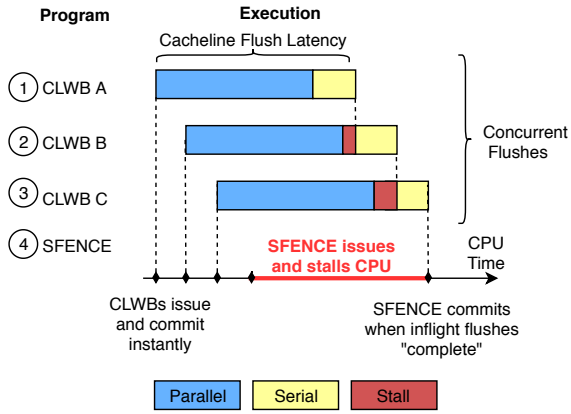

Figure 3. Execution of concurrent flushes on Optane.

flushes are completed. Thus, ordering points degrade performance by exposing the actual latency of asynchronous flushes. On Optane (system configuration in Table 1), we observed the latency of one `clwb` followed by one `sfence` to be 353 ns, for a cacheline in the L1D cache. In this paper, flushes refer to asynchronous flush instructions like `clwb`.

Effects of Ordering Points. To mitigate the high flush overheads, we must reduce the frequency of ordering points and enable the overlap of multiple flushes. We evaluated the efficacy of this approach on Optane DCPMMs via a simple microbenchmark. Our microbenchmark first allocates an array backed by PM. It issues writes to 320 random cachelines (= 20KB < 32KB L1D cache) within the array to fault in physical pages and fetch these cachelines into the private L1D cache. Next, it measures the time taken to issue `clwb` instructions to each of these cachelines. Fence instructions are performed at regular intervals, e.g., one `sfence` after every N `clwb` instructions. The total time (for 320 `clwb` + variable `sfence` instructions) is divided by 320 to get the average latency of a single cacheline flush. Figure 4 reports the average flush latency for varying flush concurrency.

The blue line in Figure 4 shows that the average flush latency can be effectively reduced by overlapping flushes, up to a limit. Compared to a single un-overlapped flush (`clwb` +`sfence`), performing 16 flushes concurrently reduces average flush latency by 75%. However, overlapping more than 16 flushes results in almost no further improvements in flush latency.

Analytical Model of Flush Latencies. As a side note, while Figure 4's blue line shows the empirical benefit of overlapping flushes, it also seems to closely follow Amdahl's law [1]. In particular, the red line shows an Amdahl's law fit using the Karp-Flatt metric [25] that has concurrent flushes acting 82% parallel and 18% serial. With the 18% serial component, it is easy to understand the diminishing returns of many concurrent flushes. As the hardware is a black box, we do not

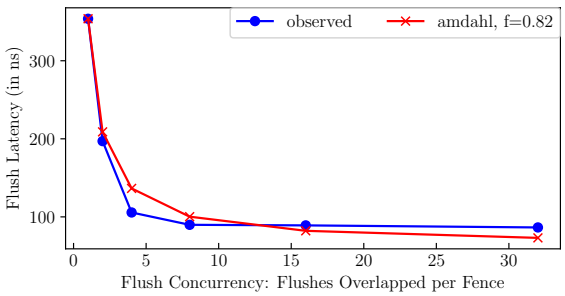

Figure 4. Average `clwb` latency observed on Optane and estimated analytically (amdahl).

yet know what features cause the appearance of serialization on Optane.

4 Minimally Ordered Durable Datastructures

We address the high flushing costs with Minimally Ordered Durable (MOD) datastructures that allow failure-atomic and durable updates to be performed with *one ordering point* in the common case. These datastructures significantly reduce flushing overheads that are the main bottleneck in recoverable PM applications. We have five goals for these datastructures:

1. *Failure-atomic updates* for recoverable applications.
2. *Minimal ordering constraints* to tolerate flush latency.
3. *Simple programming interface* that hides implementation details for handling simple use cases.
4. *Allow composition* of failure-atomic updates to multiple datastructures.
5. *Support for common datastructures* for application developers, i.e., set, map, vector, queue and stack.
6. *No hardware modifications* needed to enable high performance applications on currently available systems.

We first introduce the *Functional Shadowing* technique that is the key idea enabling MOD datastructures. Next, we show a recipe to create MOD datastructures from existing functional datastructures. Then, we describe MOD's programming interfaces.

4.1 Functional Shadowing

Functional Shadowing leverages shadow paging techniques to minimize ordering constraints in updates to PM datastructures and uses optimizations from functional programming to reduce the overheads of shadow paging. As per shadow paging techniques, we implement non-destructive and out-of-place update operations for all MOD datastructures. Accordingly, updates of MOD datastructures logically return a new version of the datastructure without any modifications to the original data. As shown in Figure 5, a `push_back` operation in a vector of size 7 would result in a new version of size

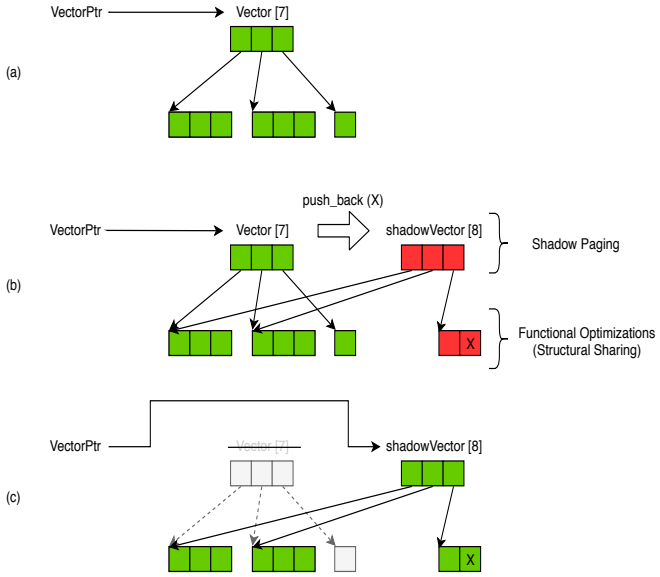

Figure 5. Functional Shadowing in action on (a) MOD vector. (b) `push_back` operation creates a shadow that reuses data from the original. (c) Application proceeds with shadow and old data is reclaimed.

8 while the original vector of size 7 remains untouched. We refer to the updated version of the datastructure as a *shadow* in accordance with conventional shadow paging techniques.

There are no ordering constraints in creating the updated shadow as it is not considered a necessary part of application state yet. We do not log these writes as they do not overwrite any useful data. In case of a crash at this point, recovery code can reclaim memory corresponding to any partially updated shadow in PM. Due to the absence of ordering constraints, we can overlap flushes to all dirty cachelines comprising the updated shadow to minimize flushing overheads. A single ordering point is sufficient to ensure the completion of all the outstanding flushes and guarantee the durability of the shadow. Subsequently, the application must atomically replace the original datastructure with the updated shadow. In case of a crash, our approach guarantees that the application points to a consistent version of the durable datastructure, albeit a stale version in some cases.

We reduce shadow paging overheads using optimizations commonly found in functional datastructures. Conventional shadow paging techniques incur high overheads as the original data must be copied completely to create the shadow. Instead, we use *structural sharing* optimizations to maximize data reuse between the original datastructure and its shadow copy. We illustrate this in Figure 5, where `shadowVector` reuses 6 of 8 internal nodes from the original `Vector` and only adds 2 internal and 3 top-level nodes. In the next subsection, we discuss a method to convert existing implementations of functional datastructures to MOD datastructures.

4.2 Recipe for MOD Datastructures

We provide a simple recipe for creating MOD datastructures from existing functional datastructures:

1. First, we use an off-the-shelf persistent memory allocator `nvm_malloc` [2] to manage PM allocations.
2. Next, we allocate the internal state of the datastructure on the persistent heap.
3. Finally, we extend all update operations to flush all modified PM cachelines with unordered `clwb` instructions. Ordering is performed in the Commit step described later.

While functional datastructures do not support durability by default, they offer a suitable starting point from which to generate MOD datastructures. They support non-destructive update operations which are typically implemented through pure functions. Thus, every update returns a new updated version (i.e., shadow) of the functional datastructure without modifying the original. They export simple interfaces such as map, vector, etc. that are implemented internally as highly optimized trees such as Compressed Hash-Array Mapped Prefix-trees [42] (for map, set) or Relaxed Radix Balanced Trees [43] (for vector). These implementations are designed to amortize the overheads of data copying as needed to create new versions on updates.

Optimized functional implementations also have low space overheads via structural sharing to maximize data reuse between the original data and the shadow. Tree-based implementations are particularly amenable to structural sharing. On an update, the new version creates new nodes at the upper levels of the tree, but these nodes can point to (and thus reuse) large sub-trees of unmodified nodes from the original datastructure. The number of new nodes created grows extremely slowly with the size of the datastructures, resulting in low overheads for large datastructures. As we show in our evaluation section, the additional memory required on average for an updated shadow is less than 0.01% of the memory consumption of the original datastructure of size 1 million elements.

Moreover, the trees are broad but not deep to avoid the problem of 'bubbling-up of writes' [8] that plagues conventional shadow paging techniques. This problem arises as the update of an internal node in the tree requires an update of its parent and so on all the way to the root. We find that existing implementations of such functional datastructures are commonly available in several languages, including C++ and Java.

We conjecture that the ability to create MOD datastructures from existing functional datastructures is important for three reasons. First, we benefit from significant research efforts towards lowering space overheads and improving performance of these datastructures [13, 36, 38, 41, 43]. Secondly, programmers can easily create MOD implementations of additional datastructures beyond those in this paper by using our recipe to port other functional datastructures. Finally, we forecast that this approach can help extend PM software beyond C and C++ to Python, JavaScript and Rust, which have implementations of functional datastructures.

4.3 Programming Interface

To abstract away the details of Functional Shadowing from application programmers, we provide two alternative interfaces for MOD datastructures:

- A **Basic** interface that abstracts away the internal versioning and is sufficient for simple use cases.
- A **Composition** interface that exposes multiple versions of datastructures for complex use cases, but still hides other implementation details.

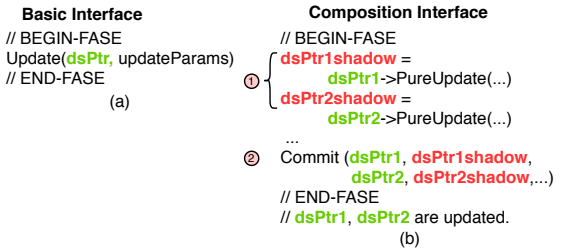

Figure 6. Failure-atomic Code Sections (FASEs) with MOD datastructures using (a) Basic interface to update one datastructure and (b) Composition interface to atomically update multiple datastructures with (1) Update and (2) Commit.

4.3.1 Basic Interface. The Basic interface to MOD datastructures (Figure 6a) allows programmers to perform individual failure-atomic update operations to a single datastructure. With this interface, MOD datastructures appear as mutable datastructures with logically in-place updates. Programmers use pointers to datastructures (e.g., `ds1` in Figure 6a), as is common in PM programming. Each update operation is implemented as a self-contained FASE with one ordering point, as described later in the next section. If the update completes successfully, the datastructure pointer points to an updated and durable datastructure. In case of crash during the update, the datastructure pointer points to the original uncorrupted datastructure. We expose common update operations for datastructures such as `push_back`, `update` for vectors, `insert` for sets/maps, `push`, `pop` for stacks and `enqueue`, `dequeue` for queues, as in C++ STL.

The Basic interface targets the common case when a FASE contains only one update operation on one datastructure. This common case applies to all our workloads (see Section 6.1) except `vacation` and `vector-swaps`. For instance, `memcached` uses one map as a cache, with FASEs comprising at most one insert.

4.3.2 Composition Interface. The Composition interface to MOD datastructures (Figure 6b) is a more general programming interface. It allows programmers to failure-atomically

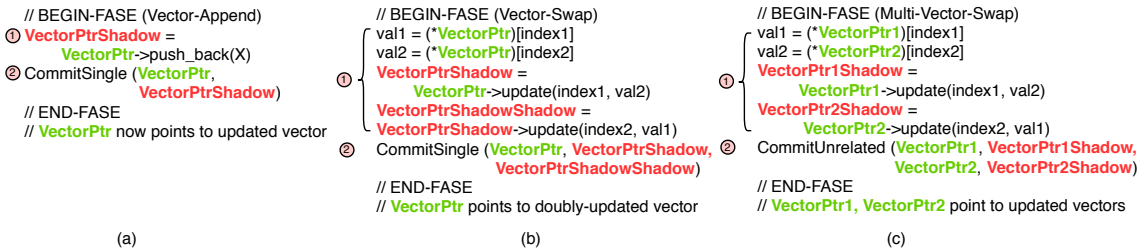

Figure 7. Using the Composition interface (1. Update, 2. Commit) for failure-atomically (a) appending an element to a vector, (a) swapping two elements of a vector and (c) swapping two elements of two different vectors.

perform updates on multiple datastructures or perform multiple updates to the same datastructure or any combination thereof. For instance, moving an element from one queue to another requires a pop operation on the first queue and a push operation on the second queue, both performed failure-atomically in one FASE. Complex operations such as swapping two elements in a vector also require two update operations on the same vector to be performed failure-atomically. In such cases, the Composition interface allows programmers to perform individual non-destructive update operations multiple datastructures to get new versions, and then atomically replace all the updated datastructures with their updated versions in a single *Commit* operation.

With this interface, programmers can build complex FASEs, each with multiple update operations affecting multiple datastructures. Each FASE must consist of two parts: Update and Commit. In Update, programmers perform updates on one or more MOD datastructures, whereby new versions of these datastructures are created that are guaranteed to be durable only after Commit. Thus, programmers are temporarily exposed to multiple versions of datastructures. Programmers use the Commit function to atomically replace all the original datastructures with their latest updated and durable versions. Our Commit implementation (described in Section 5.2) has a single ordering point in the common case. We use this interface in two workloads: `vector-swaps` and `vacation`.

Figure 7 demonstrates the following use cases:

Single Update of Single Datastructure: While this case is best handled by the Basic interface, we repeat it here to show how this can be achieved with the Composition interface. In Figure 7a, appending an element to `VectorPtr` results in (`VectorPtrShadow`). The Commit step atomically modifies `VectorPtr` to point to `VectorPtrShadow`. As a result of this FASE, a new element is failure-atomically appended to `VectorPtr`.

Multiple Update of Single Datastructure: Figure 7b shows a FASE to swap two elements of a vector. The Update step involves two vector lookups and two vector updates. The first update results in a new version `VectorPtrShadow`. Then, the new version is updated to get `VectorPtrShadowShadow`) that possesses the effects of both updates. Finally, Commit makes `VectorPtr` point to the latest version.

Single Updates of Multiple Datastructures: Figure 7c shows how we swap elements from two different vectors in one FASE. For each vector, we perform update operation to get a new version. In Commit, both vector pointers are atomically updated to point to the respective new versions.

Multiple Updates of Multiple Datastructures: The general case is realized by combining previous use cases.

5 Implementation

We prioritize minimal ordering constraints in our implementation of the two interfaces to MOD datastructures.

5.1 Basic Interface

As shown in Figure 8a, the Basic interface is a wrapper around the Composition interface to create the illusion of a mutable datastructure. The programmer accesses the MOD datastructure indirectly via a pointer. On a failure-atomic update, we internally create an updated shadow of the datastructure by performing the non-destructive update. Then, using Commit, we ensure the durability of the shadow and atomically update the datastructure pointer to point to the updated and durable shadow. Thus, we hide FS details from the programmer.

5.2 Composition Interface

The Composition interface can be used to build complex multi-update FASEs, each with one ordering point in the common case.

To support the Update step, MOD datastructure supports non-destructive update operations. Within these update operations, all modified cachelines are flushed using (weakly ordered) `clwb` instructions and there are no ordering points or fences. However, this step results in multiple versions of the updated MOD datastructures.

The Commit step ensures the durability of the updated versions and failure-atomically updates all relevant datastructure pointers to point to the latest version of each datastructure. We provide optimized implementations of `Commit` for two common cases as well as the general implementation,

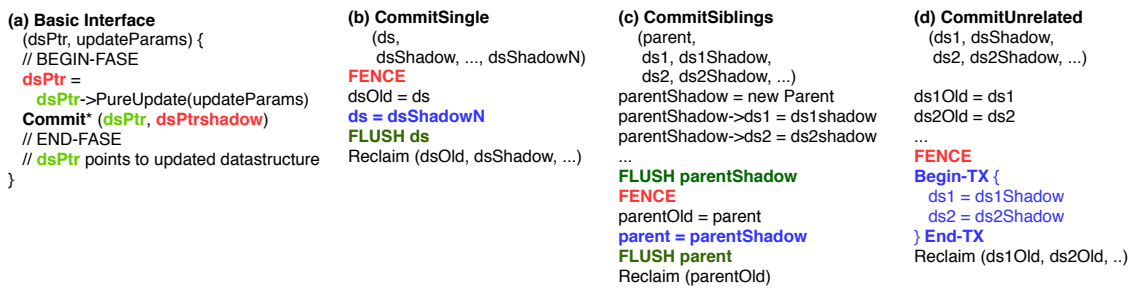

Figure 8. (a) Implementation of Basic interface as a wrapper around the Composition interface. Commit implementation shown for multi-update FASEs operating on (b) single datastructure, (c) multiple datastructures pointed to by common *parent* object, and (d) (uncommon) multiple unrelated datastructures.

as shown in Figure 8. We discuss the memory reclamation needed to free up unused memory in Section 5.4.

The first common case (`CommitSingle` in Figure 8b) occurs when one datastructure is updated one or multiple times in a FASE (e.g., Figure 7a,b). To Commit, we update the datastructure pointer to point to the latest version after all updates with a single 8B (i.e., size of a pointer) atomic write. We then reclaim the memory of the old datastructure and intermediate shadows, i.e., all but the latest shadow version.

The second common case (`CommitSiblings` in Figure 8c) occurs when the application updates two or more MOD datastructures that are pointed to by a common persistent object (*parent*) in one FASE. In this case, we create a new instance of the parent (`parentShadow`) that points to the updated shadows of the MOD datastructures. Then, we use a single pointer write to replace the old parent itself with its updated version. We used this approach in porting `vacation`, wherein a `manager` object has three separate recoverable maps as its member variables. A commonly occurring parent object in PM applications is the root pointer, one for each persistent heap, that points to all recoverable datastructures in the heap. Such root pointers allow PM applications to locate recoverable datastructures in persistent heaps across process lifetimes.

In these two common cases, our approach requires **only one ordering point per FASE**. The single ordering point is required in the commit operation to guarantee the durability of the shadow before we replace the original data. The entire FASE is a single epoch per the epoch persistency model [37]. Both of the common cases require an atomic write to a single pointer, which can be performed via an 8-byte atomic write. In contrast, PM-STM implementations require 5-11 ordering points per FASE (Section 6.4).

For the general and uncommon case (`CommitUnrelated` in Figure 8d) where two un-related datastructures get updated in the same FASE, we need to atomically update two or more pointers. For this purpose, we use a very short transaction (STM) to atomically update the multiple pointers, albeit with more ordering constraints. Even in this approach, the majority of the flushes are performed concurrently and efficiently as part of the non-destructive updates. Only the flushes to update the persistent pointers in the Commit transaction cannot be overlapped due to PM-STM ordering constraints.

Thus, the Composition interface enables efficient FASEs even when updating multiple datastructures.

5.3 Correctness

We provide a simple and intuitive argument for correct failure-atomicity of MOD datastructures. The main correctness condition is that there must not be any pointer from persistent data to any unflushed or partially flushed data. MOD datastructures support non-destructive updates that involve writes only to newly allocated data and so there is no possibility of any partial writes corrupting the datastructure. All writes performed to the new version of the datastructure are flushed to PM for durability. During Commit, one fence orders the pointer writes after all flushes are completed and updates are made durable. Finally, the pointer writes in Commit are performed atomically. If there is a crash before the atomic pointer writes in Commit, the persistent pointers point to the consistent and durable original version of the datastructure. If the atomic pointer writes complete successfully, the persistent pointers points to the durable and consistent new version of the datastructures. Thus, MOD supports correct failure-atomic updates.

5.4 Memory Reclamation

Persistent memory leaks cannot be fixed by restarting a program and thus are more harmful than leaks of volatile memory. Such PM leaks can occur on crashes during the execution of a FASE. Specifically, allocations from an incomplete FASE leak PM data that must be reclaimed by recovery code. Additionally, our MOD datastructures must also reclaim the old version of the datastructure on completion of a successful FASE.

We use reference counting for memory reclamation. Our MOD datastructures are implemented as trees. In these trees,

each internal node maintains a count of parent nodes that point to it. We increment reference counts of nodes that are reused on an update operation and decrement counts for nodes whose parents are deleted on a delete operation. Finally, we deallocate a node when its reference count hits 0.

A key optimization is that reference counts are not stored durably in PM. On a crash, all reference counts in the latest version can be scanned and set to 1 as the recovered application sees only one consistent and durable version of each datastructure.

We rely on garbage collection during recovery to clean up allocated memory from an incomplete FASE (on a crash). Our performance results include the time spent in garbage collection. As our datastructures are implemented as trees, we can perform a reachability analysis starting from the root node of each MOD datastructure to mark all memory currently referenced by the application. Any unmarked data remaining in the persistent heap is a PM leak and can be reclaimed at this point. A common solution for catching memory leaks is to log memory allocator activity. However, this approach reintroduces ordering constraints and degrades the performance of all FASEs to prevent memory leaks in case of a rare crash.

5.5 Automated Testing

While it is tricky to test the correctness of recoverable datastructures, the relaxed ordering constraints of shadow updates allow us to build a simple and automated testing framework for our MOD datastructures. We generate a trace of all PM allocations, writes, flushes, commits, and fences during program execution. Subsequently, our testing script scans the trace to ensure that all PM writes (except those in commit) are only to newly allocated PM and that all PM writes are followed by a corresponding flush before the next fence. By verifying these two invariants, we can test the correctness of recoverable applications as per our correctness argument in Section 5.3.

6 Evaluation

In this work, we seek to provide a library of recoverable datastructures with good abstractions and good performance. We answer four questions in our evaluation:

1. **Programmability**: What was our experience programming with MOD datastructures?
2. **Performance**: Do MOD datastructures improve the performance of recoverable workloads?
3. **Ordering Constraints**: Do workloads with MOD datastructures have fewer fences than with PM-STM?
4. **Additional Overheads**: What are the additional overheads introduced by MOD datastructures?

6.1 Methodology

Test System Configuration. We ran our experiments on a machine with actual Persistent Memory—Intel Optane DCPMM [22]—and upcoming second-generation Xeon Scalable processors (codenamed Cascade Lake). We configured our test machine to be in 100% App Direct mode [17] and use the default Directory protocol. In this mode, software has direct byte-addressable access to PM. Table 1 reports relevant details of our test machine. Finally, we measured read latencies using Intel Memory Latency Checker v3.6 [45].

CPU	
Type	Intel Cascade Lake
Cores	96 cores across 2 sockets
Frequency	3.7 GHz (with Turbo Boost)
Caches	L1: 32KB Icache, 32KB Dcache L2: 1MB, L2: 33 MB (shared)
Memory System	
PM Capacity	2.9 TB (256 GB/DIMM)
PM Read Latency	302 ns (Random 8-byte read)
DRAM Capacity	376 GB
DRAM Read Latency	80 ns (Random 8-byte read)

Table 1. Test Machine Configuration.

Hardware Primitives. Our workloads use `clwb` instructions (available on Cascade Lake) for flushing cachelines and the `sfence` instructions to order flushes.

OS Interface for PM. Our test machine runs Linux v4.15.6. The DCPMMs are exposed to user-space applications via the DAX-filesystem interface [47]. Accordingly, we created an ext4-dax filesystem on each PM DIMM. Our PM allocators create files in these filesystems to back persistent heaps. We map these PM-resident files into application memory with flags `MAP_SHARED_VALIDATE` and `MAP_SYNC` [9] to allow direct user-space access to PM.

PM-STM Implementation. We use the PM-STM implementation (*libpmemobj*) from Intel's PMDK library [19] in our evaluations. We choose PMDK as it is publicly available, regularly updated, Intel-supported and optimized for Intel's PM hardware. Moreover, PMDK (v1.4 or earlier) has been used for comparison by most earlier PM proposals [10, 29, 30, 39, 40]. We evaluate both PMDK v1.5 (released October 2018), which uses hybrid undo-redo logging techniques as well as PMDK v1.4, which primarily relies on undo-logging.

Workloads. Our workloads include several microbenchmarks and two recoverable applications, consistent with recent PM works [10, 26, 29, 30, 39, 40]. As described in Table 2, our microbenchmarks involve operations on commonly used datastructures: map, set, queue, list and vector. The `vector-swaps` workload emulates the main computation in the `canneal` benchmark from the PARSEC suite [3].

Benchmark	Description	Configuration
map	Insert/Lookup random keys in map	8B key, 32B value
set	Insert/Lookup random keys in set	8B key, 32B value
stack	Push/Pop elements from top of stack	8B elements
queue	Enqueue/Dequeue elements in queue	8B elements
vector	Update/Read random indices in vector	8B element
vec-swap	Swap two random elements in vector	8B element
bfs	Breadth-First Search using recoverable queue on Flickr graph [12]	0.82M nodes, 9.84M edges, 8B elements
vacation	Travel reservation system with four recoverable maps	query range:80%, 55% user queries
memcached	In-memory key value store using one recoverable map	95% sets, 5% gets, 16B key, 512B value

Table 2. Benchmarks used in this study. Workloads performs 1 million iterations of the operations described.

The baseline map datastructure can be implemented by either hashmap or ctree from the WHISPER suite [33]. Here, we compare against hashmap which outperformed ctree on Optane DCPMM. Moreover, we also measured two recoverable applications from the WHISPER suite: `memcached` and `vacation`. We modified these applications to use the PMDK and MOD map implementations. Other WHISPER benchmarks are not applicable for our evaluation as they are either filesystem-based or do not use PM-STM. Instead, we created the `bfs` workload that uses a recoverable queue for breadth-first search on the large Flickr graph [12]. We ran all workloads to completion.

6.2 Programmability

While the rest of this section presents quantitative performance results, in this paragraph we qualitatively describe the programmability of MOD datastructures. We demonstrated the use of MOD datastructures in two existing applications: `vacation` and `memcached`. With MOD datastructures as with C++ STL, applications get access to datastructures via narrow, expressive interfaces but without access to the internal implementation. However, `memcached`, like many PM applications, uses a custom datastructure (hashmap) whose implementation is tightly coupled to the application logic. Thus our main challenge was to decouple the code (i.e., application logic) from the internal datastructure implementation. We do not expect this to be an issue when building new applications. For instance, `vacation` was easy to port as its datastructure implementations were neatly encapsulated. Also, `vacation`'s logic required composing failure-atomic updates to multiple distinct maps that were members of the same object, for which we used our Composition interface with `CommitSiblings`.

6.3 Performance

Figure 9 shows the execution time (so smaller is better) of PM workloads with PMDK transactions and MOD datastructures. We make the following observations.

First, PMDK v1.5 with hybrid undo-redo logging [18] outperforms undo-logging based PMDK v1.4 by 23%.

Second, MOD datastructures offer a speedup of 43% on average for pointer-based datastructures (map, set, queue, stack) over PMDK v1.5. The performance improvements are attributed to lower flushing overheads (50% vs 66% of PMDK v1.5 execution time) and no logging overheads (0% vs 13%).

Third, for only `vector` and `vec-swap` microbenchmarks, the abstraction benefits of MOD datastructures come with a performance cost, not benefit. This occurs due to the overhead of moving from a dense 1-D array to a tree-based implementation that functional datastructures use to facilitate incremental updates. Future work can attempt to mitigate this slowdown.

For our applications, MOD datastructures show an average speedup of 36% over PMDK v1.5. Here, the performance improvements arise from lower flushing overheads (25% vs 50% of PMDK v1.5 execution time) and no logging overheads. `vacation` shows a lower speedup of 13% as we have to copy and flush the parent object in our approach (with `CommitSiblings`), while PMDK updates in-place.

6.4 Flushing Concurrency

Figure 10 illustrates ordering (x-axis) and flushing frequency (y-axis) for datastructure update operations with PMDK v1.5 and MOD. We omit lookup operations as they do not require flushes or fences.

Lower ordering constraints enable lower flushing overheads, if the number of flushed cachelines is comparable. PMDK workloads typically exhibit a high number of fences per operation. In our evaluated workloads, MOD datastructures always have only one fence per operation. While MOD datastructures copy and flush additional data than PMDK, there are no log entries to be flushed. For map and set implementations, the amount of flushed data is comparable in both approaches. Pop operations in the MOD queue occasionally require a reversal of one of the internal linked lists resulting in greater flushing activity than PMDK on average. The reduction in flushes comes from the absence of log entries as well as implementation differences. However, writes and swaps to the MOD vector require significantly more cachelines to be flushed as compared to the PMDK vector. This explains the performance degradation observed previously.

6.5 Additional Overheads

MOD datastructures introduce two new overheads: space overheads and increased cache pressure. First, extra memory is allocated on every update for the shadow, resulting in additional space overheads. Secondly, functional datastructures (including vectors and arrays) are implemented as pointer-based datastructures with little spatial locality.

Space Overheads. In Table 3, we report the increase in memory consumption on doubling the capacity of datastructures, i.e., inserting an additional 1 million elements in a datastructure of size 1 million elements. On average for most

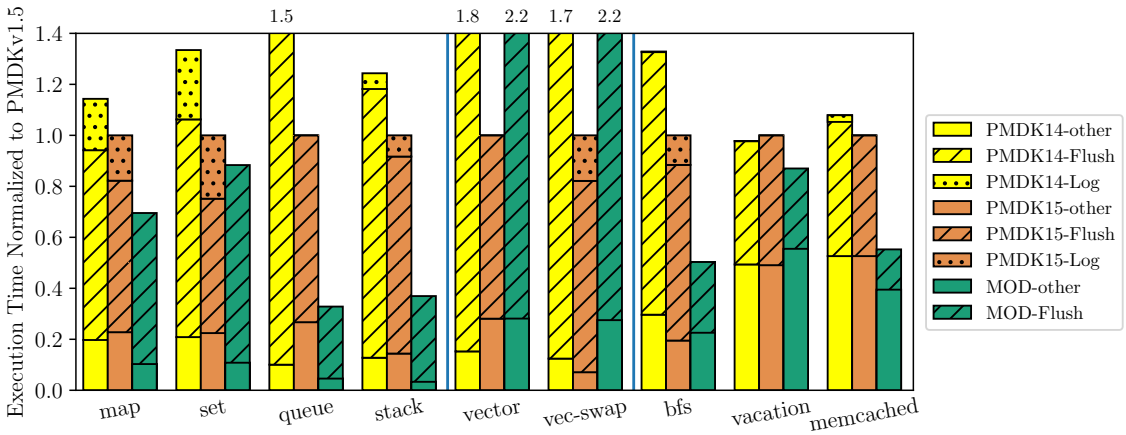

Figure 9. Execution Time of PM workloads, normalized to PMDK v1.5 implementation of each workload.

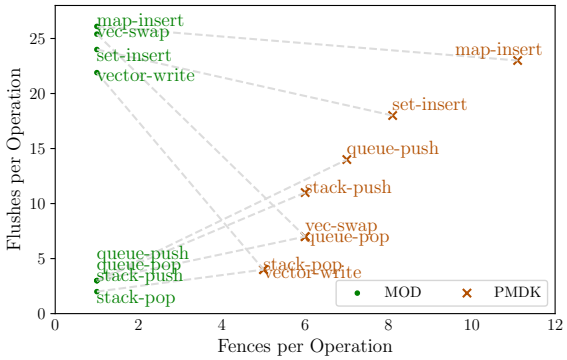

Figure 10. Flush and fence frequency in PM workloads.

of our workloads (except vector), the memory consumption of MOD datastructures only grows 21% faster than PMDK datastructures. More importantly, every individual update operation only requires 0.00002-0.00004× extra memory beyond the original version, as compared to 2× extra memory in naive shadow paging. Thus, structural sharing in our datastructure implementations minimizes the FS space overheads.

	map	set	stack	queue	vector
MOD	1.87×	2.08×	2.25×	1.67×	131×
PMDK	1.78×	1.75×	1.50×	1.50×	2×

Table 3. Increase in memory consumption when datastructure grows from 1M to 2M elements.

Cache Pressure. While our MOD datastructures typically perform better than PMDK datastructures, interestingly they also exhibit greater cache misses. Unfortunately, it was not possible to separate cache misses to PM and DRAM in our experiments on real hardware, but we expect most of the cache accesses to be for PM cachelines in our workloads.

Due to pointer-chasing accesses, MOD datastructures have more cache misses in the small L1D cache, as seen in Figure 11. This is evident in case of map, set and vector workloads, which show 2.8-4.6× the cache misses with MOD datastructures than with PMDK. The PMDK implementations of map, set and vector involve arrays contiguously laid out in memory, offering greater spatial locality by avoiding pointer-chasing patterns.

MOD implementations of stack, queue and `bfs` show low cache miss ratios, comparable to the PMDK implementations. These results to be expected as push and pop operations in these datastructures only operate on the head or the tail, resulting in high temporal locality.

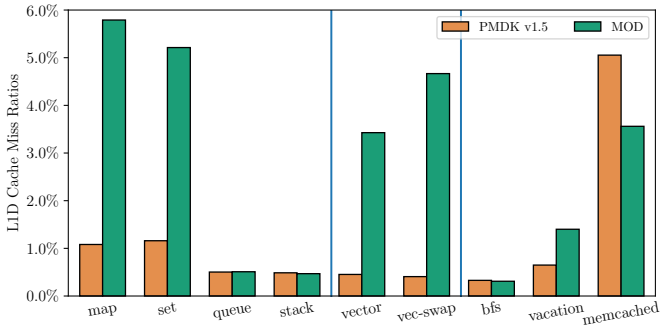

Figure 11. L1D Cache miss ratios for PM workloads.

7 Related Work

Prior research mainly consists of PM-STM optimizations and datastructure-specific optimizations for PM.

7.1 PM-STM Optimizations.

Among software proposals, Mnemosyne [46], SoftWrap [14], Romulus [10] and DudeTM [29] rely on redo logging, NV-Heaps [7] employs undo logging techniques and PMDK [19] recently switched from undo logging to hybrid undo-redo log [20]. Each of these approaches requires **4+ ordering points per FASE**. Most undo-logging implementations require ordering points proportional to the number of contiguous data ranges modified in each transaction and can have as many as 50 ordering points in a transaction [33]. In contrast, redo-logging implementations require relatively constant number of ordering points regardless of transaction size and are better for large transactions. However, redo logging requires load interposition to redirect loads to updated PM addresses, resulting in slow reads and complexity.

Romulus and DudeTM propose novel mechanisms based on redo-logging and shadow paging to reduce ordering constraints. Romulus uses a volatile redo-log with shadow data stored in PM while DudeTM uses a persistent redo-log with shadow data stored in DRAM. Both approaches double the memory consumption of the application as two copies of the data are maintained. This is a greater challenge with DudeTM as the shadow occupies DRAM capacity, which is expected to be much smaller than available PM. MOD datastructures only have two versions during an update operation, with significant data reuse between the two versions. Moreover, DudeTM and Romulus incur logging overheads and require store redirection, unlike MOD datastructures. Finally, better hardware primitives for ordering and durability have also been proposed. For instance, DPO [27] and HOPS [33] propose lightweight ordering fences that do not stall the CPU pipeline. Efficient Persist Barriers [24] move cacheline flushes out of the critical path of execution by minimizing epoch conflicts. Speculative Persist Barriers [40] allow the core to speculatively execute instructions in the shadow of an ordering point. Forced Write-Back [35] proposes cache modifications to perform efficient flushes with low software overheads. All these proposals lower the overheads of ordering points in PM applications, whereas we reduce the number of ordering points in these applications. Moreover, these proposals require core microarchitecture or cache modifications.

7.2 Recoverable Datastructures.

While our work converts existing functional datastructures into recoverable ones, the following papers demonstrate the value of handcrafting recoverable datastructures. Dali [34] is a recoverable prepend-only hashmap that is updated non-destructively while preserving the old version. Updates in both MOD and Dali are logically performed as a single epoch to minimize ordering constraints. However, our datastructures are optimized to reuse data between versions, while the Dali hashmap uses a list of historical records for each key. The CDDS B-tree [44] is a recoverable datastructure that also relies on versioning at a node-granularity for crash-consistency. Extending such fine-grained versioning to other datastructures beyond B-trees does not seem to be trivial.

Other proposals have targeted recoverable B+-trees used in storage systems. NV-Tree [48] achieves significant performance improvement by storing internal tree nodes in volatile DRAM and reconstructing them on a crash. wB+-Trees [6] uses atomic writes and bitmap-based layout to reduce the number of PM writes and flushes for higher performance. These optimizations cannot be directly extended to other datastructures. MOD datastructures are all implemented as trees, and could allow these optimizations to apply across datastructures with further research.

8 Conclusion

MOD provides programmers of PM recoverable applications with a middle ground between the expert work of handcrafting recoverable datastructures and performance overhead of STM. Future work includes implementing additional MOD datastructures, incorporating non-temporal writes, and handling concurrency.

Acknowledgments

We thank the Wisconsin Multifacet group for their feedback. We thank Sanjay Kumar for his support with the Intel AEP early access program. This work was supported by the National Science Foundation (CCF-1617824 and CNS-1815656) and Hill's John P. Morgridge Chair. Haria and Swift have significant financial interests in Google and Microsoft respectively.

References

[1] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS)*.

[2] Tim Berning. 2017. nvm malloc: Memory Allocation for NVRAM. https://github.com/hyrise/nvm_malloc.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[4] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. 2010. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[5] 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 (OOPSLA)*.

[6] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment* 8 (February 2015).

[7] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*.

[9] Jonathan Corbet. 2017. Two more approaches to persistent-memory writes. https://lwn.net/Articles/731706/.

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

[11] cppreference. 2018. Containers Library. https://en.cppreference.com/w/cpp/container.

[12] Tim Davis. [n.d.]. The university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices.

[13] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38 (1989).

[14] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*.

[15] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys (CUSR)* 13, 2 (June 1981).

[16] Swapnil Haria. 2019. *Architecture and Software Support for Persistent and Vast Memory*. Ph.D. Dissertation. University of Wisconsin-Madison.

[17] Alper Ilkbahar. 2018. Intel Optane DC Persistent Memory Operating Modes Explained. https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/.

[18] Intel. [n.d.]. New release of PMDK. https://pmem.io/2018/10/22/release-1-5.html.

[19] Intel. [n.d.]. Persistent Memory Development Kit. http://pmem.io/pmdk.

[20] Intel. [n.d.]. PMDK issues: introduce hybrid transactions. https://github.com/pmem/pmdk/pull/2716.

[21] Intel. 2018. Intel Optane DC Persistent Memory Readies for Widespread Deployment. https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment.

[22] Intel. 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019). http://arxiv.org/abs/1903.05714

[24] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*.

[25] Alan H. Karp and Horace P. Flatt. 1990. Measuring Parallel Processor Performance. *Communications of the ACM (CACM)* 33 (May 1990).

[26] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[27] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[28] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*.

[29] 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 (ASPLOS)*.

[30] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[32] Raymond A. Lorie. 1977. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems (TODS)* 2, 1 (March 1977).

[33] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[34] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC)*.

[35] M. A. Ogleari, E. L. Miller, and J. Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[36] Chris Okasaki. 1998. *Purely Functional Data Structures*. Ph.D. Dissertation. Carnegie Mellon University.

[37] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*.

[38] Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-vectors in a Systems Language. *Proceedings of the ACM on Programming Languages* 1 (September 2017).

[39] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[40] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[41] Michael Steindorfer. 2017. *Efficient Immutable Collections*. Ph.D. Dissertation. University of Amsterdam.

[42] Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[43] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[44] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures

for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*.

[45] Vish Viswanathan. 2018. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[46] Haris Volos, Andres J. 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 (ASPLOS)*.

[47] Matthew Wilcox. 2014. DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/.

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*.

A Artifact Appendix

A.1 Abstract

This artifact description provides information to recreate the experiments from Section 6 of this paper. Accordingly, six microbenchmarks and three workloads (described in Table 2 in the main paper) are provided that use either our library of Minimally Ordered Durable (MOD) datastructures or the Intel PMDK library. In this appendix, we describe the compilation and execution process for these libraries and benchmarks. Note that while the experiments in Section 6 were performed on a system with Intel Optane DC Persistent Memory Modules (DCPMM), the publicly released artifacts do not depend on DCPMMs to enable functional reproduction and use with simulators. Moreover, we provide a list of changes required to accurately reproduce the experiments on DCPMM (see Notes).

A.2 Artifact check-list (meta-information)

- **Program: MOD library and benchmarks.**
- **Compilation: GCC with support for C++14.**
- **Data set: Provided.**
- **Hardware: Intel system with Optane DCPMM.**
- **Metrics: Execution Time.**
- **Output: Execution Time in log files.**
- **How much disk space required? 16 GB.**
- **How much time is needed to complete experiments? 60 mins.**
- **Publicly available? Yes.**
- **Archived (provide DOI)?: 10.5281/zenodo.3563186**

A.3 Description

A.3.1 How delivered. We have uploaded the software as a compressed archive at https://doi.org/10.5281/zenodo.3563186.

A.3.2 Hardware dependencies. The libraries and benchmarks provided by us can only be compiled and executed on modern Intel architectures that support either `clwb` or `clflushopt` instructions.

A.3.3 Software dependencies. The main software dependency is Intel's Persistent Memory Development Kit v1.5. A version of PMDK v1.5 is provided in the available artifact with modifications to optionally disable logging activities. Other dependencies include:

- GCC compiler with support for C++14 and -mclflushopt compile flag (tested with GCC v7.4.0).
- PMDK-specific: autoconf, pkg-config, libndctl-devel, libdaxctl-devel.
- memcached-specific: libevent-dev, memslap driver.
- msr-tools (optional) for controlling hardware prefetchers.

A.3.4 Data sets. The microbenchmarks and `vacation` generate their input datasets randomly. `Memcached` requires the publicly available `memslap` driver to generate the inputs and a sample input graph is provided for `bfs`.

A.4 Installation

Decompress the tar.gz archive preserving the directory structure.
Compile pmdk:
```
$ cd pmdk; bash compile.sh
```
Compile nvm_malloc:
```
$ cd nvm_malloc; bash compile.sh
```
Compile vacation with pmdk:
```
$ cd vacation-pmdk; bash compile.sh
```
Compile memcached with pmdk:
```
$ cd memcached-pmdk; bash compile.sh
```
Compile vacation, memcached with mod:
```
$ cd Immutable-Datastructure-c++; bash compile.sh
```
Compile bfs with pmdk and mod:
```
$ cd graph-algo; bash compile.sh
```
Compile microbenchmarks:
```
$ cd benchmarks; bash compile.sh
```
Copy the memslap binary from the libmemcached dependency into the memslap folder.

A.5 Experiment workflow

We have provided a script '`run.sh`' that executes all microbenchmarks and applications assuming that the original directory structure is maintained and above installation steps are followed.

A.6 Evaluation and expected result

From the top-level directory, execute the run script with one parameter, a path to a new folder to be created on PM or disk (for functional evaluation).
```
$ bash run.sh <PATH-to-new-folder>
```
The run script will display progress markers and the names of log files containing the results. For each benchmark, configuration details and execution time (latency) are reported:
```
Backing file:temp/bench
Implementation:immer
Datastructure:map
Percent writes:50Number of elements:1000000
Number of operations:1000000
...
Latency: 1208305587 ns
```

A.7 Notes

To accurately reproduce our results on Intel Cascade Lake (or newer) systems with Optane DCPMMs, instructions are provided in the text file 'PMEM_NOTES' in the archive.