LIBPM: Simplifying Application Usage of Persistent Memory

LEONARDO MARMOL, MOHAMMAD CHOWDHURY, and RAJU RANGASWAMI,
Florida International University, USA

Persistent Memory devices present properties that are uniquely different from prior technologies for which applications have been built. Unfortunately, the conventional approach to building applications fail to either efficiently utilize these new devices or provide programmers a seamless development experience. We have built LIBPM, a Persistent Memory Library that implements an easy-to-use container abstraction for consuming PM. LibPM's *containers* are data hosting units that can store arbitrarily complex data types while preserving their integrity and consistency. Consequently, LIBPM's containers provide a generic interface to applications, allowing applications to store and manipulate arbitrarily structured data with strong durability and consistency properties, all without having to navigate all the myriad pitfalls of programming PM directly. By providing a simple and high-performing transactional update mechanism, LIBPM allows applications to manipulate persistent data at the speed of memory. The container abstraction and automatic persistent data discovery mechanisms within LIBPM also simplify porting legacy applications to PM. From a performance perspective, LIBPM closely matches and often exceeds the performance of state-of-the-art application libraries for PM. For instance, LIBPM's performance is 195× better for write intensive workloads and 2.6× better for read intensive workloads when compared with the state-of-the-art PMemIO persistent memory library.

CCS Concepts: • **Information systems** → **Storage class memory**; • **Hardware** → *Memory and dense storage*; • **Software and its engineering** → Real-time systems software;

Additional Key Words and Phrases: Persistent memory, next-generation applications, application persistence

ACM Reference format:

Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. 2018. LIBPM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4, Article 34 (December 2018), 18 pages.
<https://doi.org/10.1145/3278141>

1 INTRODUCTION

The need for high-performance data persistence is a cornerstone concern for application development. Application developers continuously find new and clever ways to move data between *slow but large persistent storage* and *fast but small volatile memory*. However, the next generation persistent memory (PM) devices present interesting opportunities to reorganize the traditional IO stack for better performance. First, the dichotomy of byte-addressability and persistence ceases to exist with PM. Second, read/write latencies of PM are comparable to those of DRAM [6, 9, 16]. The unique properties of PM make them suitable candidates for CPU bus attachment and accessibility through load and store instructions. Thus, next-generation PM devices such as ReRAM, STT-MRAM, PCM, and 3D-XPoint prompt a rethink of conventional OS interfaces to storage.

Authors' addresses: L. Marmol, M. Chowdhury, and R. Rangaswami, Florida International University, School of Computing and Information Sciences, 11200 SW 8th St, Miami, FL 33199; emails: {marmol, mchow017, raju}@cs.fiu.edu.

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

© 2018 Association for Computing Machinery.

1553-3077/2018/12-ART34 \$15.00

<https://doi.org/10.1145/3278141>

Different classes of solutions have been proposed to tackle the problem of the impedance mismatch derived from these newer type of memory. The first class of solutions provide system level support for PM usage that include building PM-aware file systems [7, 9, 25, 26], PM-optimized block devices [2, 3], RDMA-based PM file system back-ends [28], and region based atomic durability on shared persistent pages [4]. The second class of solutions focuses on providing support to the programmers to directly access persistent objects residing in PM. Multiple programming abstractions [5, 10, 23] have been proposed, and user-level PM libraries [19] are also being developed for facilitating the use persistent objects directly within code without incurring the overheads of the conventional OS and storage software stacks. In this article, we reevaluate the second class of solutions and propose a simple and easy-to-use solution for PM based persistent objects.

The main objective of persistent objects is providing programmers the means to access objects directly on PM within the process address space. Ideally, the programmers would have access to a persistent heap that would allow them to allocate, modify, and reload the object in a different application scope. While NVheaps [5], Mnemosyne [23], and PMEM.IO [19] propose robust ways to do that, in all the cases the method for handling persistent objects is not as straightforward as handling volatile objects. All of these solutions require annotating persistent objects explicitly as well as marking persistent transactions using specific keywords. HEAPO [11], unlike the other solutions, provides native heap management rather than building the heap on top of the persistent file systems. It also proposes undo-only logging by employing a key value store based logging mechanism. However, it still expects developers to annotate the transactions similar to the other solutions.

However, SoftPM [10] proposes much simpler abstraction but only for block-based storage. Porting legacy applications to a PM-based programming model becomes simpler by the introduction of orthogonal persistence (i.e., automatic persistence of objects generated from a particular known root object) supported by LIBPM. The useful notion of orthogonal persistence is not fully supported by these solutions NVheaps [5], Mnemosyne [23], and PMEM.IO [19].

In this article, we build and evaluate LIBPM. LIBPM creates the abstraction of persistent *containers*, a new way for applications to consume PM. Containers can also be identified as persistent object stores or persistent heaps. In this article, we do not discuss Linux containers (e.g., Docker Containers); rather, the LIBPM containers we refer to represent persistent object stores or heaps.

LIBPM containers make PM consumption simple for applications through the use of familiar interfaces [10] and implement consistency of related updates all while retaining the performance benefits of PM. Applications create containers and use in-memory data structures; a user-level library automatically discovers container-resident data to be made persistent using pointer chasing techniques based on a single, simple application hint. With this model, most of the conventional persistence operations are obviated. As a result, containers also simplify the porting of existing applications and creation of new applications that use PM. Container data are mapped to the process address space and data consistency is enforced through a simple transactional operation exposed to the application. The container library preserves the relationships between the application's objects (via pointers) within a container on application restart.

Containers are designed to address several goals that are critical to the adoption of any software solution for PM. *First*, it implements an interface that is familiar to the existing systems by borrowing the simplest elements from both the memory and file interfaces. *Second*, it creates a simple yet effective transactional model that provides data consistency when using PM with low runtime overhead and without requiring CPU ISA changes. *Last*, the porting of existing applications is made simple. Containers thus fill critical software gaps needed to ensure the adoption of PM technologies in the application marketplace.

Fig. 1. LIBPM Architecture.

In this article, we demonstrate that programming with LIBPM is significantly simpler than using the state-of-the-art PMEM.IO persistent memory library. In our experience, developing similar data structures using LIBPM requires 7% to 27% less lines of code compared with PMEM.IO. What is more significant is that the eliminated code has high development complexity; we illustrate this with a real-world development use-case. Additionally, the performance of applications developed with LIBPM is up to 195× better for write intensive workloads and up to 2.6× better for read intensive workloads when compared with PMEM.IO. In what follows, we motivate the design and implementation of LIBPM, discussing its origins in the SoftPM [10] project, highlighting the design principles underlying its evolution, and comparing it both qualitatively and quantitatively against state-of-the-art contributions in the persistent memory space.

2 ARCHITECTURE

LIBPM is logically and architecturally divided into two components, core and back end. These components address functionality visible to applications and interaction with persistent memory management mechanisms within the OS respectively. All of LIBPM's functionality is available to applications via its API. LIBPM's architecture is depicted in Figure 1 and further discussed in the remainder of this section.

2.1 Application Programming Interface

We believe that a new API for persistent memory will emerge shaped by at least two guiding principles. Foremost, a majority of the application developers will expect the simplicity of the volatile memory API for persistent memory but will expect a user-level library to handle many of the subtleties related to using persistent memory correctly. Thus, the API should ideally provide capabilities to ensure that such access is done safely so as to not corrupt persistent application state after a crash. Second, the library-layer that implements the API must be sufficiently low-overhead so as to preserve the performance advantages of using persistent memory over block-based persistent storage. Guided by these design principles and our experiences from previous

Table 1. The LIBPM API

Category	API	Description
Container Management	open(...)	Open / create container.
	close(...)	Close open container.
	clear(...)	Clear container contents.
	delete(...)	Delete all container data and metadata (persistently).
Memory Allocation	pmalloc(...)	Allocate persistent object in container.
	pfree(...)	Free persistent objects.
Transactional Updates	commit(...)	Atomically apply all pending updates to the container.
	rollback(...)	Undo all updates to the container.
Annotation	setroot(...)	Set object as the root of the container.
	getroot(...)	Get address of root object.
	exclude(...)	Do not follow pointer during container closure.

work on Software Persistent Memory (SoftPM) [10], we propose an API that combines a memory like interface with the *container* abstraction.

The proposed API, summarized on Table 1, is grouped into four main categories. The first category, Container Management, handles container related operations. LIBPM library provides a flat name space for the containers. It maintains sufficient information to map the container ID with the underlying persistent files that constitute the containers. It allows programmers to create or restore a container with a simple API call – open. A container can only be opened exclusively by a single application. LIBPM checks the status of the container before allowing an application to access it.

Developers can close an opened container or delete an existing container, thereby reclaiming the PM used by the container, using the close and delete API calls respectively. The second category allows developers to create arbitrarily complex data structures within a given container using pmalloc and free them from the containers using pfree. It should be noted that in addition to hosting objects, containers also preserve the relationships (pointers) between objects by storing additional metadata. This additional information makes it possible for the container to properly restore the persistent objects at arbitrary memory locations while maintaining pointer correctness.

The third category implements the transactional model. The commit operation atomically persists all the changes made to a given container. Alternatively, all the changes could be aborted by issuing a rollback call. The last category defines annotation related operations on the container. These allow the programmer to set and fetch the container’s root using setroot and getroot respectively. The container root specifies the starting point when computing the memory closure of the container to identify all data that needs to be made persistent.

2.2 Core

The LIBPM Core manages the containers that applications create to provide transparent persistent memory allocation and transactional persistence support. Four interconnected components of the LIBPM Core—*Memory Allocator*, *Transaction Manager*, *Pointer Manager*, and *Discovery*—provide these capabilities. These components, together, implement the LIBPM API.

The *Memory Allocator* is responsible for allocating persistent objects of arbitrary size. pmalloc allocates memory from the mapped pages provided by the *Page Allocator* from the *Backend* component. It implements a variant of slab allocation [1] whereby objects are grouped together based on their size to minimize fragmentation. Besides speeding up allocation, this slab-based allocation technique allows an efficient implementation for freeing persistent memory.

The *Transactions Manager* ensures all-or-nothing persistence semantics for related updates, i.e., all updates to persistent memory that are related from the application's standpoint are made atomically persistent. At a high level, this is accomplished by maintaining two versions of persistent objects: a snapshot version that reflects the previous consistent state and a current version wherein updates are performed in place. Once the container is in a new consistent state after absorbing all necessary modifications into the current version, then the application can either (i) make the current version persistent by issuing a commit operation, or (ii) revoke all updates to the container by issuing a rollback operation, which would restore the container to its previous, i.e., snapshot, version. It can be deduced that because of maintaining two versions (i.e., *snapshot and current*) LIBPM may consume twice as much memory than absolutely necessary in the worst case. However, we do not anticipate that all the container pages are made dirty at the same time for a majority of the workloads. The working-set properties of memory and storage accesses whereby a small portion of the entire memory/storage space is typically accessed actively at any given time. Thus, we anticipate that a small subset of segments from a container would be uncommitted at any given time.

The *Pointer Manager* is responsible for ensuring that the relationships (pointers) between persistent objects get preserved at all times. To do so, it stores metadata that describes the locations and target addresses of persistent pointers so that these pointer values can be corrected when objects are moved in memory, as well as restored correctly when a container is reopened. The latter is particularly important in environments for which LIBPM can not guarantee that the container will be mapped at the same offset within the process' address space [22].

Finally, the *Discovery* component implements automatic discovery of persistent data based on application-specified annotations for the container's *root* data structure, and automatically annotated information about pointers and memory allocations using static analysis (e.g., CIL [17]). This component follows the basic approach for memory closure discovery as described in the SoftPM work [10]. This component simplifies the development of new applications as well as the porting of existing applications to persistent memory.

2.3 Backend

LIBPM's *back-end* is in charge of implementing persistence. The back-end is made up of two subcomponents: the Page Allocator and the Flusher. The Page Allocator exports an interface to the upper layers that allows them to allocate and free persistent memory pages. The specific details about how to allocate or free persistent pages are handled by Page Allocator modules. These modules are free to implement the interface whichever way they see fit, allowing them to apply different approaches and optimizations specific to the operating system in use. The containers are made up of multiple segments. Initially a 64MB segment is allocated to a container, and more are allocated on demand. However, the size of the allocation can be configured to match the system requirements.

The Flusher's primary task is guarantee that writes are durable. Modern systems use multiple levels of caches and memory controllers sitting between the CPU and the memory. Because of these, synchronous updates must be treated with care to guarantee that they make it to the persistent media. Depending on the available instructions and other specifics of the hardware, the Flusher customizes the data transfer between the volatile CPU caches and PM to optimize for performance.

3 DESIGN

In this section we describe many of the tradeoffs we made when designing LIBPM. We start by stating assumptions that shaped our design as well as predictions about hardware trends. Then we

```

1  struct container_root {
2      struct binary_search_tree *node;
3  } *c_root;
4  pmalloc(sizeof(*c_root), &c_root);    //allocate the root in PM
5  setroot(cid, c_root, sizeof(*c_root)); // cid: container id
6  c_root->node = tree_root;             // a fully populated, instantiated tree
7  commit(cid);

```

Listing 1. Migrating RB Tree into persistent memory using containers.

describe the container abstraction and our transactional model that are at the core of our library. We discuss how persistent memory pointers are managed in LIBPM and contrast it with alternate approaches for managing persistent memory pointers. Finally, we discuss the limitations of LIBPM.

3.1 Assumptions

We assume a class of persistent memories with byte addressability and sufficient performance to enable direct CPU access via the memory bus. Further, we assume that traditional CPU architectures for memory access remain fundamentally unchanged. In other words, persistent memory will continue to be managed as pages, which are then managed between the OS and the CPU complex via page tables with hardware accelerated accesses enabled by a hierarchy of volatile CPU caches. The CPU instruction set may evolve to support persistent memory in the future [8, 16, 23]. While these will likely impact the details of the mechanisms we develop, they will not fundamentally change the core research questions that we address in this dissertation. Further, while independently important, concerns such as improving the reliability and durability properties of persistent memory technologies are being addressed by other researchers [13, 20] and are outside the scope of this article.

3.2 The PM Container Abstractions

Our previous work on Software Persistent Memory (SoftPM) [10] developed a library that allows applications to create data containers for persistently storing arbitrary structured data. SoftPM *containers* present a powerful abstraction for the developer who simply identifies a single top-level *container root* data structure to persist. All data reachable from this top-level data structure (via pointers) is automatically discovered and atomically persisted by the library infrastructure.

The root structure, for instance, may contain pointers to all user-defined data structures in the persistent memory. Listing 1 depicts an example of an in-memory Binary Search Tree being migrated to persistent memory using containers in only few lines of code.

A container can be used to store a portion of an application's in-memory data persistently and guarantee that such data is self-contained. The container provides referential (e.g., pointer-based) integrity and completeness as well as atomic data durability. The primary design goal of SoftPM is to create an easy-to-use interface similar to how volatile memory is used today, transactional consistency for updates to containers, and low-overhead persistence. Application-specified persistence points represent consistent states of application data in persistent memory and allow for straightforward failure recovery by the application.

LIBPM implements SoftPM's container abstraction for a hybrid memory system that is composed of both volatile DRAM and other CPU-addressable persistent memory. However LIBPM is only designed for data structures that would eventually reside entirely in the persistent storage. Hence, systems that provide support for hybrid memory data structures [18, 27] are not comparable to LIBPM. Furthermore, persistence requirements for byte-addressable PM pose requirements that are distinct from those of block-based storage for which SoftPM was built. The ordered

durability of data spread across the volatile caches spread across multiple cores of the system needs to be designed in a manner that allows easy developer control over these operations but with minimal developer effort, and optimized for performance. This requires careful use of CPU instructions for enforcing cache line flushes and write barriers, and the use of newer PM-optimized instructions introduced by processor manufacturers (e.g., Intel's recent `clflushopt`, `pcommit`, and `clwb` instructions [12]).

LIBPM, though heavily influenced by the design principles of SoftPM, differs significantly in terms of its design and implementation. LIBPM's design is focused on optimal use of PM whereas SoftPM was designed exclusively for block-based storage such as flash and disk drives. LIBPM introduces the concept of segments that can be directly mapped to a process' virtual address space. As we discuss in Section 3.2.1, the data segments hold the persistent objects and allow direct access to the objects by the applications and thereby claim the benefits of byte-addressability. SoftPM uses volatile memory as temporary placeholder for objects before they get written out to block-based storage. Instead, LIBPM minimizes cache flushes to PM (the destination for persistent data) by using page level logging (Section 5.4.3). Finally, as discussed in Section 3.3, LIBPM implements transactions differently than SoftPM by making use of the write ahead metadata log and copy-on-write data log segments.

3.2.1 Segments. A LIBPM container is a collection of segments. Each segment is a set of contiguous pages that get mmaped into the process address space for direct access. There are three types of segments: Metadata, Data, and DLog. Every container has exactly one metadata segment, in which all of the container's self-describing information is stored. The metadata segment also hosts (i) the root pointer, from which all other objects in the container can be found and (ii) the data log, used to make all updates in the container atomic. The data segments are responsible for hosting all persistent objects. Hence, allocation sizes are limited by the size of data segments; this is a configurable parameter that can be changed at container creation time. As more objects get allocated, more data segments are created, growing the container as needed.

Each data segment hosts objects of a given size, making memory management simple and fast. To reduce the number of data segments, memory allocations are rounded up to next multiple of 16 bytes. This is also a configurable property of the container. Finally, there are the DLog segments, which are used to store consistent versions of objects stored in data segments for crash recovery purposes. Just like data segments, Dlog segments are allocated on demand. The more changes a transaction makes, the more Dlog segments become necessary and get created. When a transaction is completed successfully, its Dlog segments can be reused or garbage collected.

3.3 Transactions

When designing a transactional mechanism for LIBPM we wanted to strike a good balance between performance and ease-of-use. The result is a transaction model that combines memory protection mechanisms with undo logging and operates at a page granularity. Most significantly, this approach eliminates the need for the programmer to be explicit about what data is about to be modified before modifying data, thus vastly simplifying development in comparison to recent efforts in this space [5, 24]. LIBPM simply maps segments into the process address space without write permissions and registers a fault handler to track updates. Thus, memory protection allows updates to be logged by LIBPM without developer involvement. LIBPM does not support container access by multiple applications; rather, it keeps a local version of the snapshot per container. However, LIBPM can handle the case where another thread in the same application tries to update data in parallel. Our assumption is that the threads would communicate between themselves to access/commit data structures at specific execution points-in-time when data is consistent.

Figure 2(a) illustrates LIBPM handling of updates to containers starting from a recently opened container using segments of size three pages. When a container is opened, all data segments are write protected. First, the application tries to update an object allocated in the third segment (Figure 2(b)). Since the page hosting the object is write protected, LIBPM's fault handler gets triggered. Then the entire page gets copied to a non-full Dlog segment (Figure 2(c)) and a log entry is added (Figure 2(d)). Finally, the page is made write-able (Figure 2(e)) and the update proceeds. Subsequent updates to the same page do not trigger additional faults and can proceed without any overhead (Figure 2(g) and (h)).

3.4 Persistent Memory Pointers

In addition to guaranteeing the consistency and durability of persistent objects, LIBPM's containers also need to preserve the relationships (pointers) across the objects they host.

To better understand what this means, let us consider a container hosting a linked list of m nodes $N_1, N_2, \dots, N_m$ such that N_i points to N_{i+1} . When this container is opened, there is no guarantee that the nodes will be mmap'ed at the same addresses they were created. Modern systems implement address space layout randomization (ASLR) by default and therefore it is likely that *next* pointers of the list would be *broken*. To solve this problem, LIBPM stores enough metadata in the container about the objects it hosts so that when a container is opened, all pointers are *fixed* before control is handed to the application. Fixing a persistent pointer simply means updating the target address of the pointer. These updates can be performed in $O(n)$ time in the worst case, where n is the number of persistent pointers in the container. In practice, however, we have seen that this can be avoided by hinting the mmap system call about where container segments should be mapped at runtime. Since this is a dynamic operation performed by the LIBPM at runtime, ASLR constraints do not apply for mmap hinting in as much the same way as they would to statically mapped persistent objects.

Previous works have proposed other solutions to the persistent pointer management problem. Most of these can be categorized into three groups:

Fixed-map. These solutions simply force all persistent objects to be mapped into the same address space locations, accomplished by disabling ASLR. Disabling ASLR, however, is not recommended, since it makes the OS more vulnerable to various types of security attacks [21].

Pointer Swizzling. This class of solutions make the programmer responsible for providing serialization methods for every data type that requires persistence [14]. On the downside, these serialization methods must be updated when data types are updated.

Fat Pointers. With this class of solutions, pointers do not contain the target address. Instead, they contain enough information to compute the target address at runtime [19]. This method has the overhead of runtime pointer dereferencing but it allows persistent objects to be mapped anywhere in the address space.

3.5 Limitations

LIBPM's simple design comes with two key limitations.

3.5.1 Single Atomic Transaction. LIBPM's transaction model only allows for a single active transaction per container. Whereas this may be enough for many applications, some concurrent applications may require more control on when the updates made by a specific thread of execution are made durable.

Fig. 2. LIBPM page logging: (a) State of a container after a successful open. (b) The application tries to update object *D*, which is mapped read-only. (c) The update triggers the fault handler, which initiates the logging mechanism by making a copy of the page containing the faulting address. (d) The update is logged. (e) The page is made writable. (f) Since the page is writable, the update takes place as intended, all without user intervention. (g) Subsequent update to the same page proceed without any overhead. (h) Final state.

Fig. 3. **Container Closure:** (a) Initial state. (b) Moving all reachable nodes to the container. (c) All nodes moved. (d) Fixing back references.

3.5.2 Sharing Containers across Processes. A direct consequence of LIBPM's persistent memory model is that containers cannot be shared across multiple applications. Since the same container could be mapped at different addresses for sharing processes, there is currently no simple way to track and fix pointer locations and pointer values in a manner that is correct for both applications. In ongoing work, we rely on new mechanism provided by the *region system* [4] to support sharing LIBPM containers.

4 PORTING LEGACY APPLICATION TO PM

LIBPM provides a mechanism to incrementally port legacy applications to PM, without having to change the core application logic. To illustrate this process we refer to Figure 3(a), depicting the migration of a Binary Search Tree (BST) from volatile memory to PM managed by LIBPM. First, the developer creates a new container and allocates within it a root object pointing to the head of the tree. Then, on the next commit operation, LIBPM will add to the container all the objects that are reachable from the root R' hosted in the container. Since node 4 is the first to be reached, it gets moved into the container first, as depicted in Figure 3(b). Following a simple Breadth First Traversal, all nodes reachable from the head of the tree get moved into the container eventually, making each one persistent. It is important to note that, other graph traversal algorithms can also be employed to traverse the tree to achieve the same outcome. As nodes get moved, their volatile allocations are freed so that the total memory consumption remains roughly the same within a constant factor. Figure 3(c) depicts the state in which the entire tree has been made persistent. At this point, the entire tree is in persistent memory and ready to use. However, the old volatile root and iterator (depicted as Itr in Figure 3(c)) are no longer valid, since they point to freed memory. The last step of the container discovery process takes care of fixing these dangling pointers so their target addresses are updated. Figure 3(d) depicts the state of the memory after the commit

operation is completed. Migrating other data structures can be done in a similar fashion in about the same number of lines of code. See Listing 1 for details.

5 EVALUATION

The evaluation of LIBPM has three goals: (1) verify the correctness of our implementation, (2) measure the ease-of-use of LIBPM when compared to previous solutions, and (3) evaluate the performance of LIBPM and compare the results with previous solutions.

5.1 Methodology

Given that persistent memory hardware is currently unavailable for purchase in the marketplace, we rely on both hardware and software emulation. All our experiments were conducted on Intel's persistent memory emulation platform (PMEP [16]), which allows realistic evaluation of software that are intended to run on top of CPU-addressable and byte-addressable PM devices. The PMEP environment also allows for fine-grained latency control over PM access times and supports Intel's advanced x86 instructions specifically developed for PM. The machine was equipped with a Intel(R) Xeon CPU E5-4620 v2 @ 2.60GHz, 32GB of volatile memory, 8GB of persistent memory, and a custom version of the Linux kernel v4.1.18. We compared our system with the state-of-the-art persistent object library PMEM.IO (www.pmem.io) developed by Intel. This is the most recent and widely used PM library that we are aware of. Our comparison against the PMEM.IO library highlights the core features of LIBPM which includes significant reduction in development complexity as well as performance gains.

5.2 Correctness

To verify the correctness of LIBPM we used two well known techniques: (1) test coverage analysis and (2) fault injection. Along with the implementation of LIBPM, we also developed a test suite that exercises 100% of the LIBPM code, as reported by Clang/LLVM code coverage tools. This test suite consists of many unit tests as well as several system tests. The unit tests verify the correctness of each individual component whereas the system tests verify inter-dependencies among closely related components.

In addition to code coverage, we also performed fault injection at several faulting points contained within critical sections of the transaction and recovery logic. The faults at various faulting points could be triggered both on demand or at random points in time, allowing us to crash the application under examination and verify that the state of the container is always equal to the version found at the most recent consistency point. In every case, LIBPM preserved the consistency of the containers, giving us confidence that our implementation is sound.

5.3 Ease of Use

Directly programming PM is challenging. And because of this, previous solutions have all proposed higher level abstractions to hide all the quirks and complexities associated with PM. However, all of the previous solutions implement a transactional model that makes the user *responsible* for logging updates manually [5, 19, 24]. This model offers adequate control over persistence and performance, but it is also difficult to use correctly, which makes it error prone and difficult to debug. For this reason, we steered clear of conventional logging approaches and instead explored an automatic logging technique that requires no user intervention. We quantify the ease of use of our approach by counting the lines of code (LOC) that it takes to implement popular data structures when compared to previous approaches.

As we can see from Table 2, implementing persistent data structures using PMEM.IO requires more LOC than when using LIBPM. For the data structures that we considered, the overhead varied

Table 2. Counting LOC Need to Persist
Popular Data Structures

Data Structure	LIBPM	PMEM.IO
Skiplist	192	264 (+27%)
Hash Table	285	337 (+15%)
RB Tree	387	417 (+7%)
B Tree	428	511 (+16%)

Smaller values are desirable.

Fig. 4. Tree rotation.

```

1  static void rbtree_rotate(struct rbtree *tree,
2                          struct rbtree_node *node,
3                          enum rb_children c)
4  {
5      struct rbtree_node *child = node->slots[!c];
6      struct rbtree_node *s = tree->sentinel;
7      node->slots[!c] = child->slots[c];
8      if (child->slots[c] != s)
9          child->slots[c]->parent = node;
10     child->parent = node->parent;
11     node->parent->slots[NODE_LOCATION(node)] = child;
12     child->slots[c] = node;
13     node->parent = child;
14 }

```

Listing 2. Implementing RBTree rotate method using LIBPM.

from 7% to 27%. To fully understand why this is the case, let us take a closer look at the code. As an example, we examine the `rbtree_rotate` function from the RB Tree implementation using LIBPM and PMEM.IO. This function serves as a good input for a qualitative comparison, because it takes about a dozen lines of non trivial code to create a functional prototype. Furthermore, it illustrates how a PMEM.IO implementation not just requires additional lines-of-code, but also that this additional code has significant complexity, making the development error-prone.

The formal description of the `rbtree_rotate` operation requirements can be stated as follows: *Given a node of the tree, and a type of rotation (see Figure 4), perform said rotation to adjust the height of the RB Tree in $O(1)$.*

Listing 2 presents the code that rotates the persistent RB Tree when using LIBPM. The keen reader will notice that this code would require no modifications if it were used to rotate a volatile RB Tree. This is because LIBPM seamlessly takes care of logging updates transparently and pointer dereferencing integrity without developer involvement. In other words, converting an existing

```

1  static void rbtree_rotate(TOID(struct rbtree) tree,
2                          TOID(struct rbtree_node) node,
3                          enum rb_children c)
4  {
5      TOID(struct rbtree_node) child = D_RO(node)->slots[!c];
6      TOID(struct rbtree_node) s = D_RO(tree)->sentinel;
7      TX_ADD(node);
8      TX_ADD(child);
9      D_RW(node)->slots[!c] = D_RO(child)->slots[c];
10     if (!TOID_EQUALS(D_RO(child)->slots[c], s))
11         TX_SET(D_RW(child)->slots[c], parent, node);
12     child->parent = node->parent;
13     TX_SET(node->parent, slots[NODE_LOCATION(node)], child);
14     D_RW(child)->slots[c] = node;
15     D_RW(node)->parent = child;
16 }

```

Listing 3. Implementing RBTtree rotate method using PMEM.IO.

application to use LIBPM as well as writing new applications to use LIBPM is very straightforward given that the manipulation of volatile data structures is a required step for any application.

Listing 3 shows the same function now implemented using PMEM.IO. The first thing to notice is that the signature of the function (lines 2 and 3) has changed. The developer needs to be aware of persistent data types and use this knowledge continuously during program creation. The use of the macro `TOID` is required to create persistent pointers. Persistent pointers, unlike regular pointers, do not store the value of an absolute offset in the process address space. Instead, they store metadata and a *relative* offset that are used to compute the target of the pointer at runtime. Throughout the rest of the function, other persistent pointers are declared in a similar fashion.

In lines 7 and 8, we see the use of another macro `TX_ADD`, which logs the values of `node` and `child` as part of the ongoing transaction to handle recovery in case the transaction fails to commit. These steps are necessary, because both of these variables eventually need to be modified. The use of PMEM.IO thus needs to be done carefully, keeping track of persistent data as it gets modified and making sure that the original values are stored safely in case there is a crash. In line 9 we encounter a new macro `D_RW` used to dereference the persistent pointer. At this point, it is safe to make the assignment, because the original values have been safely logged. At line 10, we see another dereference used within a comparison operation for two persistent pointers. If the condition is satisfied, then the parent pointer gets updated after its current value is logged using the `TX_SET` macro. For the remainder of the function more assignments and pointer dereferences are made using the same macros.

The complexity of Listing 3 is not unique to PMEM.IO. It is simply a direct consequence of the persistent pointer and transactional model used by PMEM.IO and shared by other previous work [5, 24]. While these models do provide greater control for the advanced user, it also make is very difficult for non experts to use PM. For these reasons, we believe that LIBPM strikes valuable balance between performance and programming complexity.

5.4 Performance

We evaluated the performance of LIBPM and compared it against PMEM.IO, a state of the art persistent memory library from Intel. PMEM.IO provides a set of persistent data structures with transactional support, which we ported to LIBPM as well and compared the latency of each major operation. The data structures we considered were the following:

Fig. 5. (a) PMEM.IO vs LIBPM (Inserts). (b) PMEM.IO vs LIBPM (Lookups). (c) PMEM.IO vs LIBPM (Deletes).

Hashmap A hashtable that solves collisions by linking entries together in the same bucket.

BTree An in-memory version of a regular BTree with branching factor of eight.

RBTree A standard Red-black Tree (RBTree).

Skiplist A skiplist with a single value per node and eight levels.

These data structures were modified so that each operation exposed to the users is atomic and durable. Special care was taken so each implementation had the same level of consistency and durability with both libraries: LIBPM and PMEM.IO. Each one of these data structures can create a mapping between a 64bit integer and a value of an arbitrary size. For these experiments we simply stored a NULL value, to better measure the overheads of utilizing the data structures' core logic.

Figure 5(a) depicts the results of comparing the insert operations for all persistent data structures for both LIBPM and PMEM.IO. We see that LIBPM outperforms PMEM.IO for every data structure, from 51× in case of the Skiplist up to 195× for the BTree. For lookups, on the other hand, the difference ranges from 1.3× in case of the Skiplist to 2.6× with the BTree. And finally, the performance difference for delete operations varies from 47× to 206× for the Skiplist and Hashmap, respectively, both in favor of LIBPM.

At a first glance, these results seem counter intuitive. *How is it possible that LIBPM outperform PMEM.IO when LIBPM must log much more data for the average transaction?* The answer to this question has three parts: persistent pointer model, shadow data structures, and logging granularity.

5.4.1 Persistent Pointer Model. PMEM.IO and LIBPM have very different models for persistent pointers. In the case of PMEM.IO, a persistent pointer is defined as a type that contains metadata and offset. See details on Listing 4. This means that for PMEM.IO, a persistent pointer is not simply an offset in the process address space but rather an object with enough information to compute the target address at runtime. Having this layer of indirection allows PMEM.IO to mmap a persistent object anywhere in the address space and more importantly easy sharing of persistent objects across multiple processes. For this reason, any workload that involves dereferencing several pointer will experience runtime degradation.

LIBPM, on the other hand, has a much simpler approach to persistent pointers. For LIBPM, a persistent pointer is no different from a regular pointer, with the exception that persistent pointers must be fixed before they are used. The overhead of fixing the pointer is only paid when the container is opened. After that point, persistent pointers are used no differently than regular pointers.

To measure the overhead of persistent pointer dereferencing, we created a large circular list, several orders of magnitude larger than the CPU cache, and linked the nodes in a way that consecutive nodes are far apart in memory. Then we traverse the list many times while measuring the time that it takes to move from one node to the next. For PMEM.IO, latencies were, on average, 2× higher than LIBPM, as PMEM.IO requires pointer dereferencing to obtain the memory address of the objects.

Fig. 6. Allocation Latency.

Fig. 7. Atomic update latency.

```

1 struct pmemoid {
2     uint64_t pool_uuid_lo;
3     uint64_t off;
4 };

```

Listing 4. PMEM.IO's definition of persistent pointer.

5.4.2 Shadow Data-structures. A memory allocation on persistent memory *must* be performed atomically, so that in case of a crash no memory is leaked. It is important to notice that when persistent memory is leaked, it is leaked persistently. Unlike volatile memory, restarting the application is not sufficient to reclaim a persistent memory leak. For this reason, both PMEM.IO and LIBPM implement atomic memory allocations. PMEM.IO makes its memory allocations atomic in the same way it makes memory updates atomic, by logging changes on state before the changes are made. But these log entries must be durable to guarantee that the state of the of the memory can be rolled back if need be. To make log entries durable, PMEM.IO must flush the cache lines containing the entries followed by a memory fence to preserve order. Both the flush and the memory fence operation take place in the application execution path, increasing the latency of transactions.

LIBPM implements atomic memory allocation – `pmalloc()` using *shadow bitmaps* and slab allocators. LIBPM packs together persistent object of similar sizes into a slab, and keeps track of slab availability using a persistent bitmap. Additionally, a shadow copy of the bitmap created in volatile memory, keeps track of all pending allocations and deletions. When a `pmalloc` is invoked, LIBPM finds an available slab in the shadow bitmap, and returns the correct address for that slab. After that point, LIBPM can freely use the given memory address. The shadow bitmap keeps track of the cache lines containing dirty bits with respect to the persistent bitmap, and flushes these cache lines when a `commit` is issued. In the case of a crash before the commit is completed, nothing needs to be done, since the persistent bitmap has not been modified. This approach effectively delays and thus batches cache line flushing and memory fencing until the time of commit, reducing the runtime overhead to a minimum (as shown in Figure 6).

5.4.3 Logging at a Page Granularity. Compared to PMEM.IO, LIBPM logs and flushes much more data, but on average creates much fewer log entries. This is a direct consequence of logging at a page granularity. However, logging and flushing cache lines are not as expensive as one may think. The new Intel CLWB instruction has similar semantics as CLFLUSH and CLFLUSHOPT with the significant difference that it does not invalidate the cache line it flushes. This has the advantage that subsequent accesses to the same cache line do not incur a cache miss. Both PMEM.IO and LIBPM make use of this optimized instruction but the difference lies in the number of memory fences

they issue. To make log entries durable, in addition to flushing the cache line, one must also issue a memory fence to force the flush to take place and preserve the order of operations. The more memory fences get used, the more negative the impact on application performance. As described by Kim et al. [15], the number of fences can be minimized by only issuing a fence after a set of updates belonging to the same transaction. Since PMEM.IO logs many more smaller entries for transactions, and aggressively issues memory fences for flushing each log entry, it issues greater number of memory fences compared to LIBPM. This is the reason for the high transaction latency with PMEM.IO when compared to LIBPM. LIBPM combines several techniques that minimize the number of memory fences for both updates and memory allocations and deletions.

To better understand this phenomenon, we designed an experiment that measures the latency of transactions while varying the number and size of updates per transaction. Figure 7 summarizes the results. As a base line, we used the latency of transactions logging a single entry of one page (4KB) in size. Then we measured the latency of transactions performing multiple updates (1 to 10 updates per transaction) and of multiple sizes (from 8 Bytes to 1KB). As we can see, transactions logging a single entry of small updates (e.g., 8 bytes) do have lower latency than those logging an entire page. However, as the number of update per transaction increases, the latency of the transaction also increases, quickly exceeding that of a single whole-page update. We note that for as few as two updates, the latencies of a single multi-update transaction with small update sizes exceeds that of the single whole page update transaction.

Logging at a page granularity has the advantage that updates to nearby addresses (within the same page) are very efficient, because only the first update triggers the logging logic. Subsequent updates to the same page take place at no extra cost. This is particularly efficient for applications with access patterns characterized by temporal locality. However, transactional systems based on log-on-every-change do not benefit from temporal locality.

6 CONCLUSION

In this article, we presented a new approach for consuming persistent memory based on the LIBPM *container* abstraction. This new approach enables a simple development process for new applications as well as porting existing applications so that they may use PM devices. Containers allow applications to program persistent memory very much the same way as they would program volatile memory. This simplicity is achieved through a combination of design and a simple-to-use API. First, containers are mapped directly to the process address space, so that applications access persistent memory without any operating system supervision. Second, the proposed transactional model for manipulating containers uses memory protection mechanisms to automatically track the pages that are updated at runtime. This approach frees the programmer from the error prone task of explicitly logging every byte before it is updated, as has been used in the recent open-source project, PMEM.IO. Third, containers are fully self-describing entities; they maintain enough information about the data they host so that they can be easily migrated to a different machine. We also describe in detail the architecture and design of our solution, LIBPM, and compare its performance against PMEM.IO, a state-of-the-art persistent memory library from Intel. The results show an improvement of performance for persistent data structures of up to 195× for write intensive workloads and 2.6× for read intensive workloads.

ACKNOWLEDGMENTS

We thank the members of the Intel persistent memory research group and the anonymous reviewers of this article for their insightful feedback. This work was supported in part by NSF awards CCF-1718335, CSR-1563883, and an Intel ISRA award.

REFERENCES

- [1] Jeff Bonwick. 1994. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*. 87–98.
- [2] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. IEEE Computer Society, Los Alamitos, CA, 385–395. DOI : <https://doi.org/10.1109/MICRO.2010.33>
- [3] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.
- [4] Mohammad Chowdhury and Raju Rangaswami. 2017. Native OS support for persistent memory with regions. In *Proceedings of 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*.
- [5] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.
- [6] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09)*.
- [7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09)*.
- [8] Intel Corporation. 2014. Intel Architecture Instruction Set Extensions Programming Reference. Retrieved from Retrieved from <https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf>.
- [9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems (EUROSYS'14)*.
- [10] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*.
- [11] Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-based persistent object store. *Trans. Stor.* 11, 1, Article 3 (Dec. 2014), 21 pages. Retrieved from DOI : <https://doi.org/10.1145/2629619>.
- [12] Intel Corporation. 2015. Intel Architecture Instruction Set Extensions Programming Reference.
- [13] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. 2010. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.
- [14] Alfons Kemper and Donald Kossmann. 1995. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *VLDB J.* 4, 3 (Jul. 1995), 519–567.
- [15] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. Retrieved from DOI : <https://doi.org/10.1145/2954679.2872392>
- [16] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A validation framework for persistent memory software. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*.
- [17] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, Lecture Notes in Computer Science*.
- [18] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York, NY, 371–386. Retrieved from DOI : <https://doi.org/10.1145/2882903.2915251>
- [19] Andy Rudoff. 2014. NVM Library. Retrieved from <http://pmem.io>.
- [20] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. 2010. Use ECP, Not ECC, for hard failures in resistive memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*.
- [21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, 25–25. <http://dl.acm.org/citation.cfm?id=2028067.2028092>
- [22] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*. ACM, New York, NY, 298–307. Retrieved from DOI : <https://doi.org/10.1145/1030083.1030124>
- [23] Haris Volos, Andres Jaan Tack, and Michael Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

- [24] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.
- [25] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of the SC*.
- [26] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Santa Clara, CA, 323–338.
- [27] 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'15)*. USENIX Association, Berkeley, CA, 167–181.
- [28] Yiyang Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

Received October 2017; revised July 2018; accepted September 2018