Native OS Support for Persistent Memory with Regions

Mohammad Chowdhury
Florida International University
Email : mchow017@cis.fiu.edu

Raju Rangaswami
Florida International University
Email : raju@cis.fiu.edu

***Abstract*—Modern operating systems have been designed around the hypotheses that (a) memory is both byte-addressable and volatile and (b) storage is block addressable and persistent. The arrival of new Persistent Memory (PM) technologies, has made these assumptions obsolete. Despite much of the recent work in this space, the need for consistently sharing PM data across multiple applications remains an urgent, unsolved problem.**

The *Region System* is a high-performance operating system stack for PM that implements usable consistency and persistence for application data. The region system provides support for consistently mapping *and* sharing data resident in PM across user application address spaces. Its high-performance design minimizes the expensive PM ordering and durability operations by embracing a minimalistic approach to metadata construction and management. Finally, the region system creates a novel IPI based PMSYNC operation, which ensures atomic persistence of mapped pages across multiple address spaces.

I. INTRODUCTION

Memory and storage have been managed as separate entities within operating systems (OS) because of their uniquely different properties. Whereas memory is byte-addressable, volatile, and fast, storage is block addressable, persistent, and slow. The emergence of byte-addressable persistent memory (PM) hardware, such as ReRAM, STT-MRAM, PCM, and 3D-XPoint present a combination of properties of both memory and storage. The current OS software stack, which was not designed to exploit the unique properties of PM, thus requires a rethink.

Persistent memory raises two fundamental challenges for building future systems. First, entirely new approaches to device access become inevitable with persistent memory devices that offer 2-3 orders of magnitude lower latency than flash-based SSDs. The latency of PM access affects not just application latency but also system resource consumption. To minimize the overhead of using PM, making accesses purely memory-oriented become inevitable. Second, working with persistent memory is significantly different than block storage since it is directly exposed to the CPU. Working with it correctly can introduce significant complexity to development work-flow. Radically new approaches for exposing persistent memory to applications and simplifying the task of the developer become necessary.

Current OS support for PM involves reusing the abstractions and interfaces of the file or memory subsystems. Conventional file systems expose persistent storage by presenting a file abstraction to applications and use block-oriented access to persist data. For byte-addressable PM, accessing it in large blocks slows down both read and write operations significantly, owing to higher data software stack and data transfer overheads [7], [8] as well as the *read-before-write* requirements [20]. On the other hand, while memory management systems support byte-granularity access via mapping physical addresses, they do not support persistent namespaces nor the notion of consistency or durability of memory updates. What is necessary is a PM-tailored OS software stack that can address the unique needs of applications when using PM and simplify their development without sacrificing the performance benefits of using PM.

Recent work has tackled the above impedance mismatch by building PM-aware file systems [13], [14], [23], [24], programming abstractions [12], [15], [22], PM-optimized block devices [7], [8], RDMA-based PM file system back-ends [26], and user-level PM libraries [2]. The PM-specific file systems build on the well-established POSIX interface, utilize PM's byte-addressability, and provide durability guarantees, but they do not address the consistency of updates to file data – mapped and shared between applications. Consequently, applications are required to implement custom mechanisms for ensuring the consistency of their PM-resident data, a difficult task given the nuanced treatment necessary for ordering operations to PM without loss of performance.

Our thesis is that persistent memory will drive new applications — applications that use it not just for its persistence but also its memory like properties. Such applications would ideally want to map PM space within their address spaces for direct access. Further, they would require arbitrary and unordered allocation and deallocation of PM space similar to how memory is used today. Finally, they would want a simple interface that atomically persists a group of updates to in-memory state. To fill this need, we propose the *region system*, a new PM-specific OS software stack that exposes a persistent namespace with memory-like operations augmented with transactional consistency.

The *region system* is both lightweight and low-overhead; it minimizes the amount of metadata it maintains and eschews redundancy to simplify durability and consistency operations. To support mapping PM space within process address spaces for direct access, the *region system* provides a persistent *msync* operation **pmsync** which provides atomicity and gives

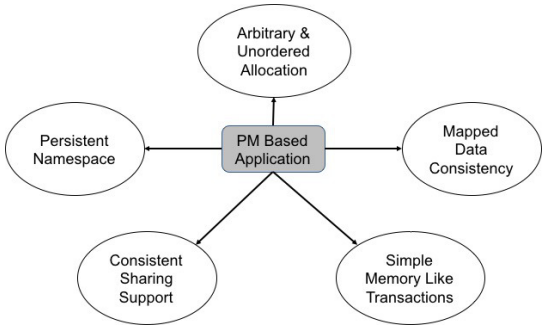

Fig. 1. Application requirements for using PM

full control on mapped data persistence to the applications. The *region system* supports mapping of PM pages within multiple application address space at the same time. To achieve transparent yet consistent sharing across the sharing processes, updates are reflected across all processes upon invocation of *pmsync* by one of the processes. The *region system* uses an inter-processor interrupt (IPI) based solution which ensures that invocation of *pmsync* by any one of the cooperating processes gets immediately reflected within the address spaces of all processes sharing the region. Synchronizing region updates and persistence operations is done by the cooperating processes which share such pages. These semantics enable simple data sharing within persistent memory without sacrificing logical functionality.

The *region system* implements a novel *dual-pointer* mechanism to manage its internal metadata that eliminates copy-on-write amplification throughout the *region system* metadata tree. The *region system* supports the creation and management of *regions* which allows for arbitrary and unordered allocation of PM space at the *page* granularity. This is a necessary requirement for applications to fully benefit from memory like usage of PM. Maintaining consistency of PM requires careful ordering of updates which involves flushing dirty cache lines to the PM adding significant cost to the overall process [14], [16]. As a remedy, the *region system* employs a non-redundant metadata architecture requiring only atomic 8-byte updates to ensure durable PM operation.

The *region system* performs better than the ext4-DAX file system when compared for persistent operations (i.e. operations which update PM resident metadata) with similar functionality. Besides that, the region-based version of *libpmem* performs competitively with the regular and ext4-DAX versions (both non-transactional) while providing strong consistency guarantees.

II. Application Usage of PM

The characteristics of PM devices blurs the conventional boundary of storage and memory, and exposes the limitations of the existing solutions in managing PM optimally. We believe that PM access interfaces within the OS should be tailored to expose the unique properties of PM devices so applications can exploit the full potential of this new technology with *ease*. In this section, we justify a minimum set of requirements (as shown in Figure 1) that the OS should meet to satisfy both the support of new PM devices and its use by applications.

A. Persistent Namespaces

As with conventional storage, applications using PM will require the ability to identify previously stored data and distinguish it from unrelated data stored by other applications.

B. Mapped Data Consistency

Unlike block-based persistent storage of today, PM devices can be accessed directly by the CPU. To utilize this new, powerful capability, it is valuable to expose PM directly within a process' virtual address space. Previous studies have shown that with PM, the current storage stack contributes to 97% of the access overhead [6] and that direct CPU loads/stores can significantly lower latency and improve the CPU efficiency of applications [1]. Thus, the conventional memory mapping approach used for volatile DRAM and files becomes very valuable with PM. However, the possibility of corruption increases as the PM can contain uncommitted data after a system failure or crash. Thus, it is necessary to have a mechanism to achieve atomic durability of mapped data and to revert back to a previously application defined consistent state in case of a failure to achieve atomic durability.

C. Consistent Sharing

Direct exposure of PM to CPU load/stores provides an unique opportunity to reflect all the changes made to a particular PM location visible to multiple applications simultaneously. The current PM-specific solutions does not support any notion of shared data consistency. The file mapping mechanism either supports private copies (MAP_PRIVATE) of the data, or shared copies (MAP_SHARED) which might not be transparent across the applications at any given time. We posit that the applications which map the same PM area have some motive to do so, and they should be able to transparently make the updates visible to all concerned parties. However, the applications may decide durability points by synchronizing amongst themselves. The OS, in this case, should provide the basic support for transparent atomic durability of shared PM areas across sharing applications.

D. Simple Memory-like Interface

Mapping the PM directly to applications address space only to update the PM areas using a complex transactional mechanism would bring little benefit to the application developers. Current transactional mechanisms [2], [12], [22] require specific set of steps to start, end, or persist a transaction. In some cases, the applications have to go through the cumbersome task of identifying each of the PM resident objects. We believe that this approach does not yield full benefit of direct PM access, and makes the development process harder for the developers. Our proposal is that applications should be able to continue their current approach of using in-memory objects and should

TABLE I
A SUMMARY OF RECENT RESEARCH ON PM-SPECIFIC SOFTWARE SOLUTIONS

	Namespace	Mapped Data Consistency	Consistent Sharing	Memory Like Transactions	Arbitrary and Unordered Allocation
File systems [13], [14], [17], [23], [27]	✓	✗	✗	✗	✗
Memory subsystem	✗	✗	✗	✓	✓
Block devices [7], [8], [10]	✓	✗	✗	✗	✗
Persistent Heaps [2], [12], [22]	✓	✓[transactional]	✗	✗	✓
NOVA [24]	✓	✓[private]	✗	✓	✗
Mojim [26]	✓	✓[replicated]	✗	✓	✗
Atomic Msync [18], [21]	✓	✓[non-PM]	✗	✓	✗

not have to worry about individually ensuring each objects durability. They should be able to make changes to objects in a PM area and persist the updates with a simple call like `msync` at a single point in time. The changes that were made durable at a certain time should be recoverable until the application issues durability for a second set of modifications to the same region.

E. Arbitrary and Unordered PM Allocation

Let's assume an use case where an application wants to construct and manipulate a persistent B-tree to store some data. The application would allocate memory for the B-tree internal nodes as well as data nodes and these could have different sizes. The allocated nodes can be deleted in arbitrary order depending on the applications requirements. The memory subsystem can easily handle the use case by allocating chunks of memory for the B-tree nodes, which can later be de-allocated irrespective of the order of allocation. The only issue here is that the memory subsystem does not support associating a persistent namespace to the allocations. This capability is also not supportable using the file interface, where files are sequential byte streams that do not support arbitrary and unordered allocation of PM. Some file systems support punching holes in a file, but the support is offset by the complexity of managing arbitrary chunk sizes. We postulate that, for future PM consuming applications, adding and removing PM areas of arbitrary sizes within a defined namespace in an unordered manner would be a primary requirement.

III. RELATED WORK

Research on PM-centric software stacks has mainly addressed two areas: *(i)* application usage of the persistent memory, and *(ii)* native OS support for PM. As we discuss below, these two classes of solutions have been developed as silos and solutions that span the concerns of both areas remain unexplored. While we discuss individual solutions in the remainder of this section, Table I summarizes the recent research in this space evaluated against the requirements introduced in Section II.

A. Application usage of PM

The work on application usage of persistent memory has focused on new programming abstractions and models [12], [15], [22]. NV-Heaps [12] and Mnemosyne [22] propose persistent heap-based solutions for PM. Both utilize the *mmap()* system call to create the abstraction of persistent heaps. The persistent memory library effort [2] provides object-based transactional support (`libpmemobj`) built on top of low level persistent memory library (`libpmem`). These solutions rely on memory transactions to make direct PM updates atomic and durable. Moreover, while these solutions support consistent updates and provide transactional consistency, they require that applications explicitly specify individual updates to PM, a cumbersome task. Finally, there is no support for sharing data stored in PM across processes in these solutions.

B. Native OS support for PM

OS stacks built for block devices [5], [7], [8], [10] forfeit the benefit of mapping PM to the application address space reducing the scope for improvement over traditional storage. Block-based file systems are unable to fully exploit the capabilities of low-latency persistent memory devices [19]. Recent file systems such as PMFS [14], BPFS [13], SCMFS [23], HVMFS [27], FCFS [17] provide PM-specific file management solutions. These solutions all provide persistent namespace and support mapping portions of files to process address spaces. However most of these file systems do not support atomic durability of mapped data and none of them support arbitrary and unordered allocation/deallocation of PM-resident data.

Solutions for consuming PM mostly target supporting the vast majority of applications that consume storage using the file `read/write/fsync` interface. This interface is not best suited for PM which can support direct `load/stores` from CPU. When using the `mmap` interface to make this possible, most existing solutions does not provide consistency and atomic durability guarantee for all possible use cases. The importance of failure-atomic msync when using file systems has been articulated well by Park and Verma *et al.* [18], [21]. Though their proposed solutions are independent of the underlying storage type, they are designed for specific file systems. Mojim [26] and Nova [24] address the issue of mapped PM data consistency when being updated by a single application with some restrictions. Thus surprisingly, despite being a very important usage model for PM, the topic of mapped data durability remains largely unexplored.

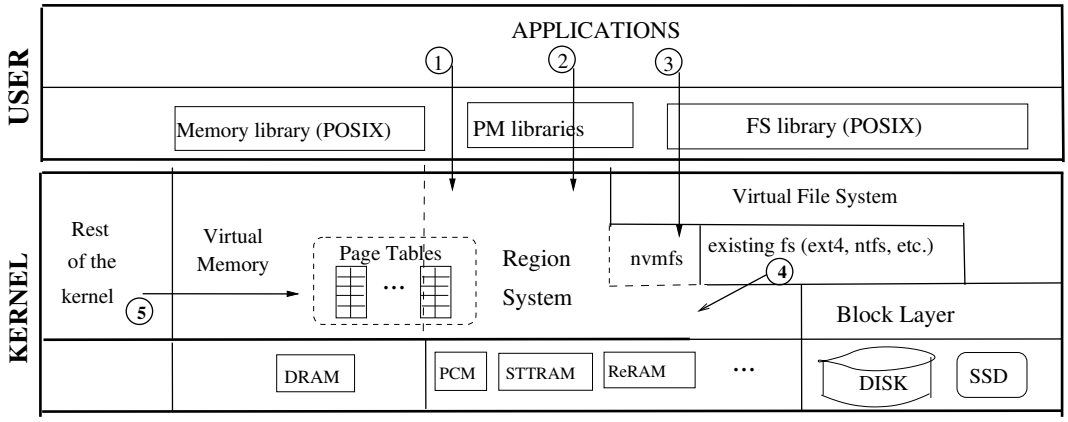

Fig. 2. A PM-specific memory/storage stack – ① Applications use PM directly using region system interfaces. ② Applications consume PM through PM libraries (e.g. pmem.io), which can use region system or PM-specific file systems to access PM. ③ Applications access PM using traditional POSIX interface through ④ Applications using PM using traditional file systems. ⑤ Rest of the kernel accessing PM using region system.

Furthermore, how such data would be shared across multiple processes or applications remains unaddressed.

The Nova [24] file system implements support for atomic-mmap with strong consistency guarantees. However, the atomic-mmap primitive does not present actual file pages to an applications address space as it only maps copies of the persistent pages. Though this version of atomic-mmap achieves consistency, it forfeits sharing of mapped pages across multiple applications.

Mojim [26] proposed a set of replication schemes with varied degree of reliability and consistency guarantees. The OS service provides the applications with simpler interfaces for creating *sync points* for the data area by calling specific system calls (such as `msync` or `gmsync`). Mojim provides availability, reliability, and consistency guarantees via replication to a peer node. However, no consistency guarantees are provided for the application-mapped data areas for the general un-replicated case.

Recent file systems have introduced new techniques for PM interaction such as epochs, short-circuit-shadow paging [13], atomic-in-place updates [14], [24], and fine grained metadata journaling [9] to optimize metadata updates. However, these solutions do not minimize the number of metadata updates necessary for a given operation. For instance, appending to or increasing the size of a file may require an addition of a new data block, which could result in multiple updates to the inode including rewriting the file size, initializing data pointers, and free-space bitmap updates. In fact, the existing PM-specific solutions all store redundant metadata for ensuring the consistency of data stored in PM, a design that has seamlessly percolated from legacy file systems designed for block storage. The more pieces of metadata a PM managing layer maintains, the greater the burden of *ordering* these updates, which in turn impacts performance [12]–[15], [22], [24]. NV-tree [25] proposed a B+ tree data structure specially designed to reduce metadata ordering by carefully separating metadata depending on their ordering requirements. However, it does store redundant metadata. NoFS [11], on the other hand, proposes a back-pointer based metadata structure which eliminates the need for ordering but increases the number of overall updates made to the persistent storage.

IV. Design Rationale

The region system is designed to fulfill the requirements outlined in Section II, and to primarily achieve the following goals described below.

A. Atomic Durability for Mapped Data

As we discuss in Section III, PM-specific solutions, either as OS optimizations or development of new programming abstractions, fall short on providing comprehensive support for mapped application data management. Agreeing upon a single, new mechanism to achieve application data consistency is difficult because of varied consistency requirements across applications. However, we can agree that applications would like to ensure that their data remains consistent in a state that the application can recover from after a system crash or a power failure. To tolerate arbitrary points of failure, data durably stored in PM must be continuously maintained in states that meet application consistency definitions. While the durability requirements of applications could vary arbitrarily, we recognize a fundamental requirement that applies across applications—the capability to make a set of changes to a set of application data reach the PM (i.e. are made durable) atomically.

A relevant question then would be –*Why advocate for larger granularity mmap based access in preference to smaller granularity atomic writes?*
Several applications today mmap multiple file pages, modify those pages, and invoke msync to finally make the write durable (e.g., MongoDB). This batch durability of mapped data typically achieves higher throughput given the spatial locality of access and batched I/O operations in comparison to multiple interposed smaller PM writes to a out-of-place

TABLE II
SYSTEM CALL INTERFACE TO PERSISTENT MEMORY. *Syscalls address one or more of the following classes of functions: (1) Namespace management, (2) Allocation, (3) Mapping, (4) Consistency, and (5) Sharing.*

Class	Name
	`region_d open(char *region_name, flags f);`
	`int close(region_d rd);`
1	`int delete(char *region_name);`
	`ppage_number alloc_ppage(region_d rd);`
2	`int free_ppage(region_d rd, ppage_number ppn);`
	`vaddr pmmap(vaddr va, region_d rd, ppage_number ppn, int nbytes, flags f);`
3,5	`int pmunmap(vaddr va);`
4,5	`int pmsync(region_d rd);`

logging structure. Besides that, it empowers programmers with a simple interface to commit data when they deem fit to do so without specifying individual transactions. Furthermore, it is likely to require significantly greater modification to the application's work-flow, design, and implementation to employ smaller granularity logging of updates. The region system primitives, pmmap and pmsync, help these applications to achieve batch durability of mapped data atomically, and it will help programmers to evolve in-PM data structures consistently without the need for introducing complex transactions in application logic.

B. Minimize Cache Flush

If multiple inter-dependent objects reside in a system, the updates to those objects require to happen simultaneously to maintain data integrity. With file systems for instance, the file *inode* and free-space bitmap contain redundant information, which need to be atomically written out to maintain consistency. Conventionally, such redundant information is maintained for performance reasons; loading all file inodes to reconstruct the free-space bitmap was considered expensive and thus the redundant persistent versions. However, the redundancy also adds complexity to file system design and overhead during runtime. Furthermore, updates to PM requires careful ordering of instructions, and the problems of not doing so are well-described in contemporary PM research literature [12]–[15], [22], [24]. The absence of correlated metadata significantly lowers the ordering requirements for PM updates. The *region system* design uses no metadata redundancy which facilitates implementing the atomicity of metadata related operations using atomic single word PM updates.

V. REGION SYSTEM

A. Interface and Usage

Persistent Memory can be accessed using a variety of mechanisms as depicted in Figure 2. Our main concern is to present persistent memory within an applications address space for direct access without burdening the application developers with the complexities of maintaining transactional consistency of their data. We propose an interface that empowers the PM application developer with namespace support, atomic metadata operations, and shared mapped data consistency. Table II lists the region system call interface.

Listing 1. Region System usage illustration

```
#define PAGE_SIZE 4096

int rd = open("__region_1");
int ppage_no = alloc_ppage(rd);
void *log = pmmap(NULL, rd, ppage_no, PAGE_SIZE,
      MAP_SHARED);

/* write to log */

pmsync(rd);
unmap(log);
close(rd);
```

The region system creates the `region` abstraction for using persistent memory. A region is an unordered collection of persistent pages (`ppages`) identified by an unique region descriptor. `Ppages`, which are distinguished by page numbers, can be added to a region and deleted later in any sequence irrespective of the sequence in which they were allocated. Unlike conventional file systems, there is no `read/write` access to regions; memory-mapping of `ppages` is the only mechanism to consume PM. The `pmmap` system call allows the application to map `ppages` to the process address space, while `pmunmap` reverses the mapping. By invoking `pmsync`, applications can make all the changes to a regions' mapped ppages atomically durable. The code snippet in Listing 1 illustrates how an application would consume PM through the region system interface.

B. Non-redundant Metadata Structure

To support low-latency operations, the region system adopts a lightweight and low-overhead approach to managing PM. First, it carefully avoids keeping any redundant data persistently. Avoiding the use of redundant metadata eliminates the need for atomic updates to inter-dependent metadata, simplifying the task of keeping metadata consistent. Only a single version of the metadata necessary to reconstruct the region system state is kept up-to-date on the PM device. Free-space bitmaps are only persisted on clean shutdowns and not during normal operation. During crash recovery, to reconstruct free space information, the region system scans through region metadata in the background. The process is initiated during region system mount time. The volatile and persistent metadata for the region system and their relations are depicted and described in Figure 3.

System calls that update metadata are designed to maintain consistency. The same general principle holds for all the system calls—all region metadata get updated atomically to implement atomicity of the system calls themselves. For all the persistent metadata operations, we identify the durability point, and make sure that all updates preceding the durability point are made durable using a persistent barrier (CLWB+SFENCE) followed by the final durability operation. The durability point of each operation is the last 8-byte update which marks the completion of the operation. For an example,

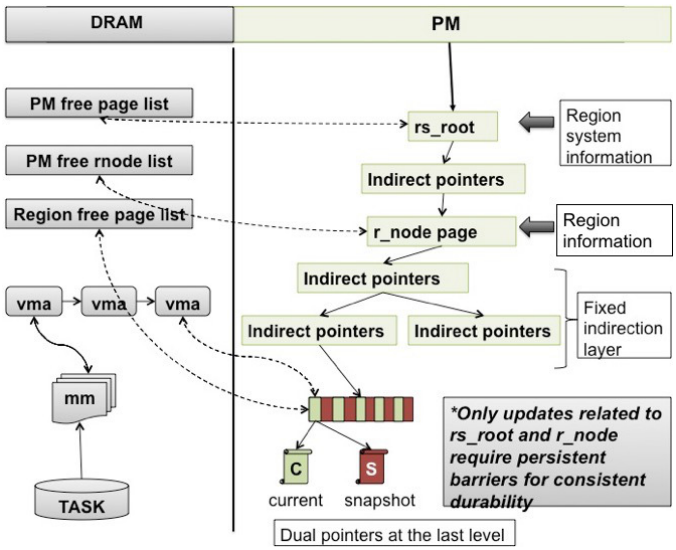

Fig. 3. Region system layout and separation of volatile and persistent metadata.

updating the r_node status to RNODE_VALID is the durability point for creating a region. All internal metadata updates are done before this step, and the status update protected by the persistent barrier finalizes the operation.

C. Consistency of Mapped Data

The `pmsync` interface simplifies the consistent management of durable data mapped into an application's address space. With pmsync, an application can choose when it wants to make any of its in-memory data durable. Initially, when a group of `ppages` from a `region` is mapped to an application's address space, updates to the `ppages` are not made durable until a `pmsync` is invoked. On `pmsync`, all updates to the `ppages` since the previous `pmsync` are made atomically durable. Any updates to `ppages` after the latest `pmsync` get discarded in case of a system crash or failure. Regardless of the number of `ppages` mapped to a process' address space, all updates to a region are always made atomically durable.

To accomplish atomic durability, CPU load/store capability has to be revoked immediately for all pages of the region after the pmsync invocation until all the pages are in a protected state. However, CPUs may not deprive other tasks from executing for more than a short period of time to preserve system responsiveness. To achieve these goals the `region system` issues an IPI to all CPUs and puts them in a non-maskable interrupt state immediately after acquiring the region-wide lock which protects a region from modifications. One of the CPUs IPI-handler write-protects all the dirty pages, while other CPUs wait for the completion of write protection so that they may not alter the contents of the pages.

The second step is to flush all the dirty cache lines to the PM. Upon returning from the IPI, the active CPU also flushes the dirty cache lines for the region referenced by the kernel virtual address. At this point region system relies on CPU cache snooping to make the pages consistently visible across all the processors. After cache flush, an idempotent execution of pmsync is initiated. First, the region r_node status is updated to `pmsync_in_progress`. The region system metadata architecture, as shown in Figure 3, is designed to have two pointers per `ppage`, one for the current mapped version, and other for the snapshot version. Dual pointers, used in combination with the `r_node` flags - `pmsync_in_progress and pmsync_complete`, ensure an `idempotent` pmsync. After the r_node is set to `pmsync_in_progress`, pmsync is guaranteed to finish successfully, even after a system crash. Finally, the snapshot pointer is modified to point to the current page and delete the previous snapshot if there was any, before setting the region status to `pmsync_complete`. The modifications to the `r_node status` are protected by the (`clwb+sfence`) persistent barrier to enforce proper ordering of updates.

CoW Optimization: During normal operation, any access to a mapped page within a pmsync'ed region results in a copy of the current page, and the snapshot pointer is changed to point to the copied page while the current (old) page is made accessible to the user. Existing snapshot mechanisms allow the `copy-on-write` (CoW) update to propagate to intermediate layers of metadata, even up to the root. The region systems novel `dual pointer` technique eliminates such CoW amplification by limiting recursive updates to the lowest metadata layer in the region system metadata hierarchy.

Recovery: Region system can be brought back to a consistent state by traversing the metadata tree and informed by the rs_root and r_node flags. Due to space limitations, we avoid the discussion here and intend to provide a detailed description of these in a separate article.

VI. Implementation

The region system is implemented as kernel module for Linux kernel 3.10.14. A small (<100 LOC) kernel patch has to be applied before the module can be installed. The kernel patch involves the modification to task_struct and other minor kernel data structures to support the region system. Once the module is installed, the region system can be mounted by specifying the start address (physical) and size of the region that falls within the PM's physical address range. The module has been successfully deployed and tested using Intel's hardware based PM emulator [14] which emulates PM latency according to the administrator-defined PM configuration.

VII. Experimental Evaluation

We evaluated region system performance using microbenchmarks and several real-world applications. Due to space limitations, we only present microbenchmark evaluation and demonstrate a case where regular PM applications/libraries achieve stronger consistency guarantees by using the region system.

A. Microbenchmarks

The region system exports a minimal interface to the applications for mmap based usage of PM. The system calls

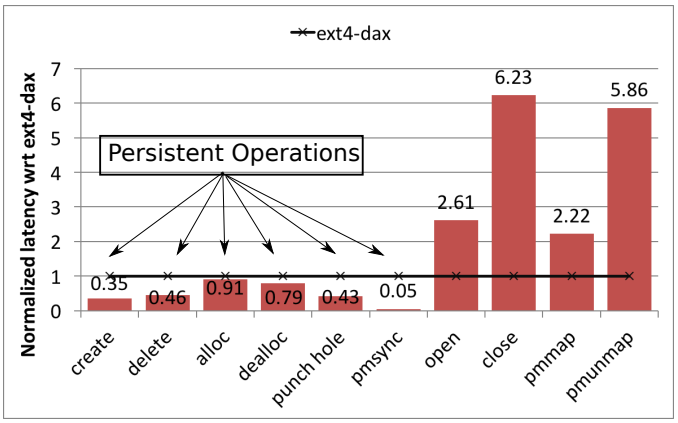

Fig. 4. Average latency of region system operations relative to ext4-DAX

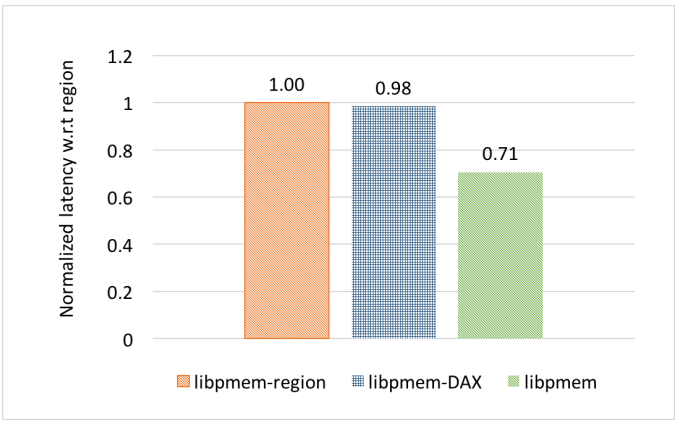

Fig. 5. Normalized average libpmem and libpmem-DAX latency with respect to libpmem-region

in Table II are designed based on the POSIX standard file system calls in terms of functionality. We wrote several simple applications to compare the performance of similar operations when using the region system and the ext4-DAX file system. However, ext4-DAX does not provide any of the atomic durability for data as provided by the region system. The main purpose of this evaluation is thus an evaluation of the cost of the additional features provided by the region system relative to the state-of-the-art ext4-DAX performance features. Ext4-DAX also supports punching holes via the `fallocate` system call. To evaluate this, we compare the performance of deallocating every alternate pages after a file/region is initially allocated. As we can see from Figure 4, for the majority of persistent operations, region system performs better than ext4-DAX. This is attributable to the non-redundant metadata design leading to reduced updates to PM.

B. PMEM.IO with Region System

PMEM.IO [2] is a suite of persistent memory libraries developed for making persistent memory programming easier. PMEM.IO's libpmem provides an interface to map PM via the underlying PM specific file system (e.g., ext4-DAX). However, it implements PM specific functions such as `pmem_flush`, `pmem_drain`, etc. in user space without involving the underlying file system. Libpmem can also be configured to use ext4-DAX's msync instead of the user level flushing functions. We call the former variation as LIBPMEM and the latter variation as LIBPMEM-DAX. Neither variant provides support for transactional consistency of the data stored using libpmem. We created a variant of libpmem which is built to consume regions implemented by the region system and supports mapping ppages to the user address space. We call this variant LIBPMEM-REGION. By using LIBPMEM-REGION, pending data updates are made transactionally durable when using pmsync. We compared the LIBPMEM-REGION which provides atomic durability of PM changes with the non transactional LIBPMEM and LIBPMEM-DAX. We ran the libpmem library's `pmem_flush` benchmark [3] for all three systems. This benchmark writes a single byte to several pages of the mapped file/region and flushes the contents later. We ran the tests for different sizes of files/regions (12MB to 786MB) while pmembench executed multi-threaded (1 to 16). To determine how the region system performs on an average relative to these two variants we calculated their normalized average latency over the same dataset. Pmsync performs within 2% of LIBPMEM-DAX msync, and 30% of LIBPMEM pmem_flush. The normalized average latency of LIBPMEM and LIBPMEM-DAX are shown in Figure 5. For an application that dirties a small number of pages between consecutive syncs, the performance of LIBPMEM-REGION is competitive relative to its less transactional variants.

VIII. Conclusions

The region system is a new kernel subsystem for managing PM consistently and exposing it directly into process address spaces. A fundamental driving principle behind the region system is to reduce development complexity significantly when using PM to develop powerful stateful applications. Applications benefit from a simple durability interface that makes a group of application updates to PM-resident data atomically durable. This paper highlights the ease of use, sharing capabilities, and strong consistency and data durability guarantees for mmap based usage. The region system also supports arbitrary and unordered allocation/deallocation of data within regions at the page granularity.

The design of region system opens a new window for the existing mmap based applications to migrate to using PM effectively and easily. Applications like Kyoto Cabinet, SQLite, and other key-value stores which predominantly rely on msync can be ported to achieve atomic durability by switching the underlying storage manager from existing file systems to region system. Persistent heap based solutions can also be built by utilizing the region system's atomic API calls. Most importantly, applications can be developed without worrying about complex transactional mechanisms. We anticipate that the new design points exposed by the region system pave the way for future implementations of PM-optimized operating system and application-level software.

REFERENCES

[1] Fusion-io directFS and ACM. http://www.fusionio.com/blog/blurring-the-line-between-memory-and-storage-introducing-filesystem-support-for-persistent-memory/.

[2] Persistent Memory Programming. http://pmem.io.

[3] Pmem.io Benchmarks. https://github.com/pmem/nvml/tree/master/src/benchmarks.

[4] CAMPELLO, D., LOPEZ, H., KOLLER, R., RANGASWAMI, R., AND USECHE, L. Non-blocking writes to files. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015).

[5] CAMPELLO, D., LOPEZ, H., USECHE, L., KOLLER, R., AND RANGASWAMI, R. Non-blocking writes to files. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2015).

[6] CAULFIELD, A., AND SWANSON, S. QuickSAN: A Storage Area Network for Fast, Distributed Solid State Disks, March 2013.

[7] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. 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* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 385–395.

[8] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *ASPLOS* (2012).

[9] CHEN, C., YANG, J., WEI, Q., WANG, C., AND XUE, M. Fine-grained metadata journaling on nvm. In *IEEE 32nd International Conference on Massive Storage Systems and Technology* (2016), MSST '16.

[10] CHEN, F., MESNIER, M., AND HAHN, S. A protected block device for persistent memory. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (2014).

[11] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 9–9.

[12] COBURN, J., CAULFIELD, A., AKEL, A., GRUPP, L., GUPTA, R., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS* (2011).

[13] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., BURGER, D., LEE, B., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *SOSP* (2009).

[14] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 15:1–15:15.

[15] GUERRA, J., MARMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *USENIX ATC* (2012).

[16] LANTZ, P., DULLOOR, S., KUMAR, S., SANKARAN, R., AND JACKSON, J. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014).

[17] OU, J., AND SHU, J. Fast and failure-consistent updates of application data in non-volatile main memory file system. In *IEEE 32nd International Conference on Massive Storage Systems and Technology* (2016), MSST '16.

[18] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13.

[19] SANTANA, R., RANGASWAMI, R., TARASOV, V., AND HILDEBRAND, D. A fast and slippery slope for file systems. *ACM Operating Systems Review 49*, 2 (December 2015).

[20] USECHE, L., KOLLER, R., RANGASWAMI, R., AND VERMA, A. Truly non-blocking writes. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems* (2011), HotStorage'11.

[21] VERMA, R., MENDEZ, A. A., PARK, S., MANNARSWAMY, S. S., KELLY, T. P., AND III, C. B. M. Failure-atomic updates of application data in a linux file system. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST 15)* (Feb. 2015).

[22] VOLOS, H., TACK, A. J., AND SWIFT, M. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS* (2011).

[23] WU, X., AND REDDY, A. L. N. Scmfs: a file system for storage class memory. In *Proc. of SC* (2011).

[24] XU, J., AND SWANSON, S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 323–338.

[25] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 167–181.

[26] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15.

[27] ZHENG, S., HUANG, L., LIU, H., WU, L., AND ZHA, J. Hmvfs: A hybrid memory versioning file system. In *IEEE 32nd International Conference on Massive Storage Systems and Technology* (2016), MSST '16.