pVM - Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage

Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan
sudarsun@gatech.edu, ada@cc.gatech.edu
Georgia Institute of Technology

Abstract

Next-generation byte-addressable nonvolatile memories (NVMs), such as phase change memory (PCM) and Memristors, promise fast data storage, and more importantly, address DRAM scalability issues. State-of-the-art OS mechanisms for NVMs have focused on improving the block-based virtual file system (VFS) to manage both persistence and the memory capacity scaling needs of applications. However, using the VFS for capacity scaling has several limitations, such as the lack of automatic memory capacity scaling across DRAM and NVM, inefficient use of the processor cache and TLB, and high page access costs. These limitations reduce application performance and also impact applications that use NVM for persistent object storage with flat namespaces, such as photo stores, NoSQL databases, and others.

To address such limitations, we propose persistent virtual memory (pVM), a system software abstraction that provides applications with (1) automatic OS-level memory capacity scaling, (2) flexible memory placement policies across NVM, and (3) fast object storage. pVM extends the OS virtual memory (VM) instead of building on the VFS and abstracts NVM as a NUMA node with support for NVM-based memory placement mechanisms. pVM inherits benefits from the cache and TLB-efficient VM subsystem and augments these further by distinguishing between persistent and nonpersistent capacity use of NVM. Additionally, pVM achieves fast persistent storage by further extending the VM subsystem with consistent and durable OS-level persistent metadata. Our evaluation of pVM with memory capacity-intensive applications shows a 2.5x speedup and up to 80% lower TLB and cache misses compared to VFS-based systems. pVM's object store provides 2x higher throughput compared to the block-based approach of the state-of-the art solution and up to a 4x reduction in the time spent in the OS.

EuroSys '16, April 18-21, 2016, London, United Kingdom

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901325

1. Introduction

The volume of data generated by servers and sensor-rich client devices is dramatically increasing, and this data volume poses a significant challenge in scaling existing memory and data storage technologies. At one end of the spectrum, scaling DRAM density without increasing the cost or energy is becoming more challenging [8]. On the other hand, block-based storage with larger capacities, such as NAND flash, suffer from significantly lower bandwidth. The widening gap between persistent storage versus volatile memory access is driving researchers to look beyond NAND for alternative non-volatile memory (NVM) technologies such as PCM [30, 34] and 3D XPoint [1]. NVMs have certain limitations compared to DRAM – writes are slow and require higher energy [30], but they also have advantages over DRAM such as higher density (4x-8x higher compared to DRAM), performance (100x faster access than SSDs), and cost effectiveness. As a result, NVMs will not only be used as fast storage devices but also as DRAM replacement when augmenting memory capacity.

However, the state-of-the-art NVM software and OS solutions [17, 20, 40] have been primarily designed to provide fast persistence by extending the virtual file system (VFS) with memory capacity scaling via NVM as a secondary feature. Current solutions use the memory mapping capability of VFS to provide additional capacity and have three shortcomings. First, these methods do not support automatic memory capacity scaling. Automatic memory capacity scaling allows for the seamless use of one memory (NVM) when resources of another memory (DRAM) are exhausted and also supports flexible memory placement mechanisms between DRAM and NVM that are not supported by VFS techniques. Second, existing VFS-based solutions do not distinguish between persistent and capacity (nonpersistent) use, thereby adding filesystem metadata bookkeeping overheads even when applications use NVM for additional capacity. This limitation coupled with cache- and TLB-inefficient VFS data structures, seriously restricts

NVM use for the scaling of memory capacity. Finally, the use of a block-based VFS in NVMs for persistent, non-hierarchical object stores such as photo stores or mail servers fails to exploit optimal NVM capabilities due to filesystem metadata and concurrency overheads.

To address the limitations of state-of-the-art solutions, this paper, **persistent Virtual Memory (pVM)**, explores an alternate approach that extends the virtual memory (VM) subsystem, instead of the VFS, for achieving the dual benefits of memory capacity scaling and persistent storage using NVMs. pVM's design is based on the principle that byte-addressable NVMs resemble 'slower' memory placed in parallel to DRAM with the memory bus as a hardware interface, rather than a faster disk. Hence, extending the virtual memory (VM) subsystem is better suited to more efficiently exploit NVMs for both capacity and persistent object storage. pVM is not a replacement for traditional filesystems and has been specifically designed for the use of NVM for scaling memory and providing a persistent object store. To the best of our knowledge, pVM is the first OS-based design that extends the VM subsystem to provide such dual-use.

pVM's OS hardware abstraction treats NVM as a NUMA node to which applications can transparently or explicitly allocate additional heap memory, as well as store persistent objects by using user space memory persistence libraries [39, 40]. This generic NUMA node-based design can support several NVM nodes, permit seamless scaling of memory capacity across the nodes, and provide applications with flexible NVM-specific memory placement policies that can be easily integrated with existing memory placement libraries. Unlike the VFS-based approach that maps both persistent and non-persistent allocations to a file, pVM's OS virtual memory framework distinguishes between persistent and non-persistent memory (page) allocations and manages them independently without adding any persistent metadata management cost for capacity (heap) allocations. This significantly reduces page access time and cache and TLB misses, which is critical for applications' performance and helps mitigate NVMs' higher device access cost when compared to DRAM (see Table 1).

pVM also extends the VM subsystem with persistence management. This avoids the metadata complexity of a block-based filesystem and the overheads of kernel mediation from using a POSIX interface, which leads to reduced time spent in the OS. With pVM, persistent objects are mapped to a region of NVM pages that can be retrieved across application sessions. Performance of the persistent object store is improved with pVM due to fast on-demand page allocations and efficient TLB and cache use. pVM is designed for object stores with flat namespaces, such as NoSQL databases, personal user data, search engines, key-value stores, etc., and thus may not be a good fit for applications with significant dependence on the hierarchical filesystem. Hence, we envision that pVM's object store will coexist with a standard block-based NVM filesystem.

Attributes	DRAM	PCM	NAND
Byte-Addressable	Yes	Yes	No
Density	1x	2x-4x	4x
Read,Write latency	1x (60ns),1x	1x, 2x-5x	400x
Endurance	10^{16}	10^{8}	10^{4}

Table 1: Comparison of NVM technologies [10, 20] with DRAM as the baseline for values with 1x.

In summary, with goals of achieving both fast persistence and efficient memory capacity scaling, pVM differs from prior efforts [14, 17, 18, 40, 43] that are optimized for one dimension – either persistence or capacity. pVM makes the following technical contributions:

- *End-to-end persistent virtual memory design* – We propose persistent virtual memory (pVM), an end-to-end virtual memory-based design that treats NVM as a memory node (similar to NUMA) and extends the OS VM subsystem for memory capacity scaling, and fast and persistent object stores. (Section 3)
- *Memory capacity scaling and placement support* – We demonstrate several benefits of pVM's VM-based design, such as automatic memory capacity scaling, support for flexible NVM data placement policies that are compatible with existing NUMA-based libraries, fast page access, and improved processor cache and TLB efficiency when compared to state-of-the-art VFS-based designs. (Section 4)
- *Fast persistent object storage* – We extend the VM data structures with support for persistence, and thereby provide fast, consistent, and durable object storage. This mechanism significantly reduces the time spent in the kernel and improves the object store throughput. (Section 4)
- *Evaluation and performance* – Our evaluation shows a 2.5x reduction in application runtime with automatic memory capacity scaling, up to a 28% reduction in cache and TLB misses for benchmarks, 84% cache and TLB miss reductions for applications, and 60% reductions in page access cost. Further, pVM-based object store improves application throughput by 52%-100% relative to a VFS block-based design, with up to 4x reduction in the time spent in the OS. (Section 5)

2. Background and Motivation

We first review the current solutions and their memory capacity scaling and persistent storage limitations, and then motivate the need for VM-based support for managing NVMs.

2.1 Background and state-of-the-art

Byte-addressable NVMs such as PCM [30, 34], Memristor, and 3D XPoint [1], are expected to provide 100x faster read and write performance compared to current SSDs [14, 18, 20]. Table 1 shows their hardware characteristics compared

against DRAM and NAND devices. These NVMs are expected to scale 2-4x more than DRAM because they can store multiple bits per cell without using refresh power and have known limitations imposed by an endurance of a few million writes per cell. Further, they can be connected to the memory bus, and accessed using load and store operations, which use the processor cache to reduce access latency [32]. Researchers have explored different usage models for the capacity and persistence use of NVM. Early NVM system software and hardware research [30, 34] studied the feasibility of using NVMs as a data cache for additional memory capacity that is transparent to the application. These studies do not use NVMs for persistent storage. Recent software solutions such as [14, 18, 20] have redesigned the block-based filesystem to suit the memory-based storage by extending the VFS data structures. In contrast, research proposals such as Mnemosyne [40], NVHeaps [17], and Aerie [41] have extended the ideas of the well-known LRVM work [36] to support heap-based persistence. Interestingly, all current heap-based proposals are managed by the VFS. In contrast, pVM manages the NVM by extending the VM subsystem instead of the VFS. Our solution lets the system and applications use NVM both for capacity and for heap-based object storage [17, 40, 41]. For comparison, we use Intel's PMFS [20] as the state-of-the-art solution due to several OS-level optimizations and its wide acceptance in the NVM research community. We also evaluate other approaches such as Mnemosyne and ramFS.

2.2 NVM capacity use issues

OS capacity scaling limitations. The benefits of using NVM for additional memory capacity has been explored in the past [17, 30]. However, prior proposals are primarily designed to support data persistence. Hence, they rely on the VFS for gaining additional memory capacity by using a library allocator that maps an NVM file into an application address space (e.g., via. *mmap()* with MAP_PRIVATE in Linux). The VFS manages the mapped NVM regions, whereas the DRAM regions are managed by the VM subsystem. Applications that do not fit in DRAM (or NVM) cannot seamlessly switch to NVM for additional capacity usage when all the memory resources are exhausted because the OS virtual memory views NVM as a filesystem rather than memory. These methods suffer from the inability to transparently scale over multiple memory types and do not have flexible policies in the OS that can guide an application's memory placement across hybrid memory types, therefore impacting application performance. Figure 1(A) shows the impact of running well-known memory capacity intensive applications with a standard malloc allocator on two configurations: (a) 6GB split across 2 DRAM sockets (3GB each) versus (b) 3GB DRAM - 3GB NVM running with PMFS. The applications are described in Table 2. While in configuration (a), the OS automatically balances memory allocation across the socket, in (b), the OS can only use 3GB of DRAM

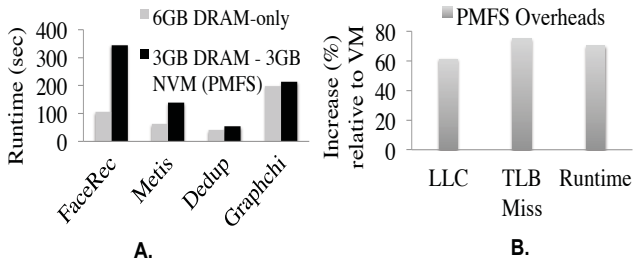

Figure 1: Capacity use. (A) Performance under limited DRAM capacity, (B) Metis software overheads when using PMFS exclusively for capacity.

Filesystem time: 60.5%	
Metadata Lock	5.16%
Metadata update	12.46%
copy_from_user	9.58%
copy_user_nocache	9.63%
Page fault	6.11%
Allocation	25.63%

Figure 2: Percentage of time spent in the filesystem for Snappy application.

because NVM is managed by the filesystem. Hence, under memory pressure, frequent swapping results in the application slowing down.

Cache, TLB inefficiency of VFS. Apart from memory capacity scaling, when the capacity requirements of an application can be satisfied with VFS-managed NVM, using the VFS induces significant software overheads due to high cache and TLB miss rates, and therefore increases the application execution time. The increase in the TLB and cache miss rates occurs first of all because current VFS-based mechanisms do not distinguish the persistent use of NVM from non-persistent use just as for additional capacity. As a result, for every allocated page, several filesystem metadata structures are updated and journaled. Second, using metadata structures like inodes and superblocks originally designed for supporting block-based devices is highly cache- and TLB inefficient. Figure 1(B) shows 80% increase in cache and TLB misses for the Metis application even when its capacity requirement is less than NVM capacity. *These results motivate the need for a VM-based design of the NVM OS software stack.*

2.3 Persistent storage limitations

NVM filesystem software overheads. Prior research has extensively studied the overheads of the block-based filesystem when using NVMs [17, 20, 40, 41]. The sources of overheads include the frequent kernel mediation required for handling I/O system calls, metadata updates (superblocks, inode bitmaps, inodes, and other data structures are updated before a block is modified), metadata concurrency issues from locking, and finally namespace management. Recent

```
Image **imgdb = nvroot_alloc("img_root", size);

START_TRANS (imgdb )
  for each new image:
   Image *imgdb[cnt]= nvmalloc("imgname", size, imgdb)
   cnt++
  ……
   /*Commit by flushing and logging */
   nvcommit("img_root", size, log=true)
END_TRANS (imgdb )

  /* persistent read, implicit load of all child ptrs*/
  img = nvread ("img_root", &size);
  /* non persistent NVM memory allocation */
  tmp = npmalloc(size)
```

Figure 3: Persistent object store programming model.

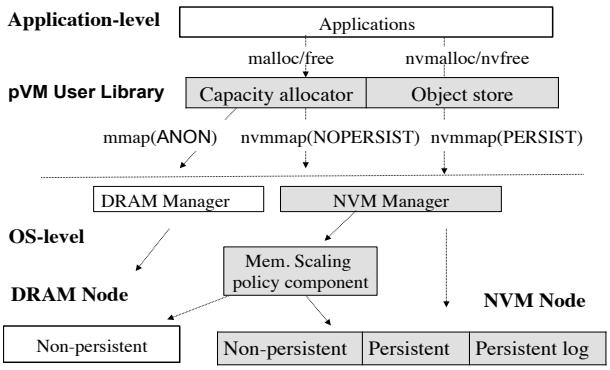

Figure 4: pVM – High level design. Shaded blocks indicate pVMs software-level stack changes.

industry proposals have redesigned the OS filesystem metadata to track pages instead of blocks, remove the buffer caches, and provide cache-line size atomic updates [20]. Other research, such as Aerie [41] has proposed a decentralized approach by separating the trusted filesystem operations (e.g., permission check, metadata update, and integrity) from the untrusted operations (e.g., dentry cache and inode namespace management) and moving the untrusted operations to the user-level. This reduces the kernel mediation cost and cache pollution.

However, these optimizations do not eliminate the software overheads associated with frequent operations on small files. To analyze an I/O intensive application, we use a file compression service – Snappy [5] as a case study in Figure 2. In this example, Snappy compresses around 2GB of image, video, email, and document files. The input shows high variation in file sizes (kilobytes to gigabytes) and requires frequent user-to-kernel transitions in the form of system calls. The transitions are required mainly because a new output file is created for each input file, which stresses the filesystem metadata updates. Figure 2 shows that close to 60% of the time is spent in the OS filesystem. The table also shows a breakdown of the cost of the filesystem components, with significant portions attributed to metadata updates and locks, kernel-to-user buffer data copying, and kernel mediation. Even using the *mmap()* and *munmap()* interface does not solve the problem for small files as every *mmap()* call is supported by other system calls (*open()*, *close()*, *stat()*) and can aggravate the issue. *These results show that the OS filesystem overhead can be significant even in the state-of-the-art NVM filesystem.*

Object storage challenges. Existing Linux object-based filesystems such as Ceph [16] and S3fuse [4] offer applications an object-based put/get interface at the user-level but map objects to files internally at the filesystem level. Hence, frequent operations on small files or objects suffer from filesystem overheads. Prior work [17, 27, 40, 41] has proposed nonvolatile heap objects that reside in some larger mapped (using *mmap()*) persistent region, and can be accessed and modified with a load and store interface. Figure 3 shows the programming model for allocating, modifying, and committing a simple persistent image object using a persistent heap-based NVM library [27]. Although the persistent storage interface avoids direct filesystem interaction and mediation, the mapped regions encapsulating persistent objects are still managed by the VFS. As discussed earlier, this results in high page access latency as well as cache and TLB inefficiency. Another minor drawback of existing persistent object stores is that all persistent objects of an application are mapped to a single large region [27]. As a result, in order to retrieve even one object, the object store has to search the entire mapped region, thus increasing the average retrieval time. *Extending the VM subsystem instead of the VFS for persistent object stores that do not depend on hierarchical filesystems can reduce page access latency and improve TLB and cache efficiency. Additionally, the VM-based approach provides a direct mapping between a persistent object and its related pages, thereby reducing the access time.* These observations motivate the design and implementation of pVM, which we describe in the following sections.

3. pVM Overview

Setting. pVM is designed for byte-addressable NVM technologies such as PCM [7, 33, 34] and 3D Xpoint [1] that are connected with a memory controller and are placed at the same level in the memory hierarchy as DRAM. NVMs can be accessed by the CPU/applications with load-store instructions to NVM pages through a hardware supported page table [9], and they exhibit higher access latency and provide lower bandwidth compared to DRAM. Because byte-addressable NVMs are not yet commercially available, we emulate them using a dedicated NUMA socket and apply memory thermal throttling to reduce the bandwidth by up to 10x relative to DRAM. To account for the differing read versus write latencies, we inject software delays using a model that accounts for LLC load and store misses. We use the model proposed by [18, 20, 38]. These emulation methods are not required for commercially available NVDIMM technology (DRAM in the front backed by an SSD in the back), but NVDIMMS have a PCIe interface rather than a memory controller-based interface, and more importantly, NVDIMMS are limited by the DRAM scalability limits.

Approach overview. We design and develop pVM to provide applications with the following important capabilities: (1) *memory capacity scaling* – OS support to achieve application-transparent memory capacity scaling similar to a multi-socket NUMA machine, (2) *efficient persistence* – fast heap-based persistent object store for applications not dependent on the filesystem hierarchy, and (3) *improved performance* – reduced cache and TLB misses when using NVMs, and maximized application performance. Figure 4 illustrates a high-level overview of the pVM design.

For scaling capacity, pVM treats NVM as a separate NUMA node and manages the NVM nodes independently from the DRAM nodes. It also provides on-demand page allocation and free space management support. Applications can allocate to NVM explicitly using library allocators for persistent (*nvmalloc()*), or nonpersistent (*npmalloc()*) use. In addition, pVM offers a set of NVM-specific NUMA policies such as $nvmrevert$ and $nvmpreferred$ (discussed in Section 4) by extending the OS and adding support to the existing user space NUMA libraries. Applications and allocators can directly map NVM memory with a *nvmmap()* interface. We describe the design and implementation details in Section 4.

For fast persistent storage, applications use a persistent library allocator similar to the ideas proposed in Mnemosyne [40], and NV-heaps [17]. Figure 3 shows the programming model for persisting an image object. pVM extends the VM subsystem by providing an object-based persistence that maps each persistent object to a set of NVM pages with ACID guarantees for both OS and application data. Every object is identified by an object identifier (objID) which is a combination of a globally unique identifier (gUID) per application plus the hash of an object name. Hence, applications sharing an object should use the gUID and objID as a capability, similar to a shared memory implementation in Linux. Prior research [17, 40] and other user space libraries [27] also use similar object naming schemes. Our current object store implementation provides basic functionality such as object creation, updates, deletes, renaming, write protection, history of object updates, logging and durability guarantees. For the object logging, we use the NVML implementation. Because most applications today use a file format, when using pVM, we use a simple file-to-object converter to convert input files to an object format.

Finally, pVM achieves improved performance due to its design approach which extends the virtual memory subsystem and its cache-efficient data structures and mechanisms.

Limitations. pVM's object store, since it only supports a flat namespace, is a good fit for applications that are not reliant on the hierarchical filesystem. Examples of such applications include persistent key-value and photo stores, and NoSQL databases. Hence, **pVM is not a replacement for filesystems**, and does not support comprehensive filesystem security policies such as link/unlink, aliases, group permissions. pVM uses unique gUID and objUID identifiers to prevent object name collisions and is similar in approach to prior systems that use flat namespaces [17, 40]. In pVM, applications that share objects can implement their own concurrency and transactional mechanism. Our ongoing work focuses on addressing these challenges with a shared memory consistency protocol.

4. Design and Implementation

The pVM system is composed of (1) a VM-based OS NVM manager (pVM-OS), and (2) a user-library (pVM-lib) that provides an interface for applications to access the NVM.

4.1 pVM-OS support

We first discuss the hardware abstraction of NVM from the view of an OS, followed by the OS-level extension required for supporting additional capacity and persistent storage in NVM. pVM provides a cleaner abstraction of NVM memory and more importantly improves the processor cache and TLB use by extending the VM subsystem. Note that the VM subsystem has undergone decades of cache and TLB optimizations, unlike the VFS subsystem that has mainly focused in improving the storage performance.

4.1.1 pVM-OS hardware abstraction

Using NVM as a NUMA node. pVM-OS treats NVM as another NUMA node, and extends the OS to support NVM-specific memory allocation and management policies. Figure 4 shows a high-level design. By using a node-based abstraction, pVM provides a cleaner hardware abstraction to the OS and the user space applications without requiring significant changes to the current system stack or the data placement mechanisms and tools, including user space NUMA libraries. With this node-based abstraction, pVM can easily extend the DRAM-based NUMA management and data placement policies to support NVM-specific policies. Importantly, this allows applications to scale seamlessly across DRAM and NVM nodes. pVM with its VM-based design can easily support additional features such as memory hot-plugging. Because byte-addressable NVMs are not commercially available, in order to evaluate the benefits of the VM-based design, we emulate NVM by treating a DRAM socket as an NVM node, and we add special flags to the node data structures to prevent general purpose allocation to the NVM node. Pages are allocated from NVM only when the pVM-OS manager requests them. Because pVM supports both additional capacity and storage, it divides NVM into three regions (1) the capacity region, (2) the persistent storage region, and (3) the OS-level persistent metadata and log region for object stores. Each region corresponds to a new Linux memory zone, and we discuss shortly how pages are allocated from each of these zones.

pVM NUMA policies. The benefits of providing fine-grained memory placement policies for the application, library, and OS in multi-socket NUMA platforms have been extensively evaluated, along with performance gains from increasing data locality. Support for flexible memory placement policies are important when NVM is used as a memory because NVMs have lower bandwidth and higher write latency, but larger capacity (2-4x more than DRAM). Prior VFS-based solutions that allow the use of NVM for capacity [17, 20, 40] do not provide such placement flexibility. In contrast, by extending the VM-subsystem and by using a NUMA-based NVM design, pVM can easily add such policies. For instance, we added three new policies – $nvmrevert$, $nvmpreferred$, and $nvmbind$ – to the OS and the existing user-level Linux NUMA library. As the names suggest, $nvmrevert$ reverts to using NVM when all DRAM pages are exhausted, $nvmpreferred$ refers to using NVM as a default memory allocation node, and $nvmbind$ restricts the application to only use NVM. We evaluate the benefits of such policies for memory capacity-intensive applications in Section 5.

4.1.2 pVM-OS software abstraction

While a NUMA-based configuration provides a cleaner hardware abstraction, it is important to extend the NVM OS data structures across different layers of the VM subsystem. Figure 5 shows multiple layers of virtual memory starting with the OS interface memory-access layer at the user level that provides the *mmap()* or *nvmmap()* interface, followed by the memory mapping layer that maps an NVM region into the user address space, and finally, the low-level allocator responsible for allocating an NVM page. We first review existing VM data structures and then discuss the changes required at each of the OS layers to enable NVM support.

VM background. In OSes such as Linux, each process has an OS context and an associated memory structure (*task mm_struct*) that encapsulates the entire user address space (heap, code, data and stack segments). A process address space can contain one or more contiguous memory regions known as virtual memory area regions (VMA). All pages in a VMA have the same access permissions, and they are used for the same purpose (heap, code pages, etc.). Note that the VMAs are either created and/or merged when applications map memory to their address space using *mmap()* or *sbrk()*. The pages are lazily added to a VMA only on the first touch (a minor page fault). These VM structures (process context, VMA, and pages) are also necessary for building the OS-level persistent state of an application.

Persistent and nonpersistent OS interface. pVM exposes a *nvmmap()* interface to user space applications and libraries, which allows them to map an NVM region explicitly into an application's address space for capacity or persistent use, as shown in Figure 5. The system call signature is similar to

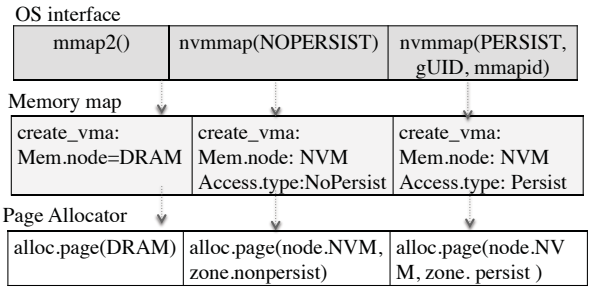

Figure 5: Extension of VM layers to support pVM.

the *mmap()* interface, except that the additional flags NOPERSIST and PERSIST are used to distinguish between the capacity and persistent uses, respectively. In addition, for persistent use, application and VMA naming arguments are supplied by the user-level object store library.

Mapping NVM into the address space. The memory mapping layer creates and updates the process memory and VMA structures. Therefore, for cleaner partitioning of NVM and DRAM data structures, the memory mapping layer is extended to distinguish between NVM and DRAM nonpersistent data structures and also between the NVM persistent and nonpersistent data structures. As shown in Figure 5, pVM introduces a special VMA memory type flag to differentiate between DRAM and NVM regions and an additional access type flag to further distinguish persistent from non-persistent NVM regions. In contrast, VFS-based NVM mechanisms do not differentiate between persistent and non-persistent NVM regions, thus imposing metadata bookkeeping penalties for both capacity and persistent storage page access, as shown in our evaluation (Section 5).

NVM page allocation and per-CPU list. The OS allocator is the heart of any VM-based management. OSes such as Linux use the buddy allocator that has undergone decades of research and optimization. pVM aims to reuse and exploit most of the logic and optimizations, without sacrificing the goal of cleaner separation between DRAM and NVM structures. Hence, we use the VMA memory type and access type to differentiate between page allocation requests, and redirect allocations to the persistent or nonpersistent NVM zones. For persistent allocation, the allocator is also responsible for checking whether the requested page already exists but is not mapped to a process address space, in which case it adds it to the page table. Furthermore, to reduce the overhead of allocator complexity, OS allocators maintain per-CPU free page lists as a fast path for allocation and reclamation, before requesting the buddy allocator. Currently, however, this is limited to homogeneous memory only. We extend the per-CPU lists with an array of per-CPU lists, containing a separate list for DRAM, NVM persistent, and NVM non-persistent allocations. Note that, after a page is allocated from a persistent list/zone, the allocator does not have

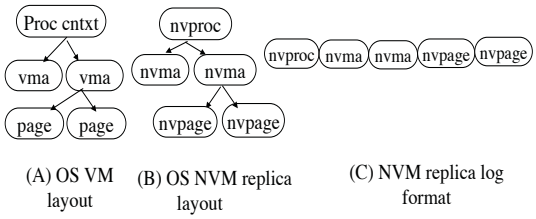

Figure 6: pVM-OS persistent structure layout.

control over the pages, and such pages are reserved, non-swappable, and managed by the OS persistence manager, until released by an application with appropriate permissions.

Configuring NVM persistence and capacity use size. For a user-space application to make the best use of NVM, it has to know the space utilization of the capacity or persistence use, and the available options to configure them. Hence, pVM provides users with an option to reserve the capacity of persistent and non-persistent zones and provides dynamic information (via Linux' 'meminfo' tool). However, reserving the capacity does not require allocation and addition of pages to the page table. In the future, we plan to explore dynamic partitioning mechanisms.

4.1.3 pVM-OS persistence support

Providing support for persistent object stores requires OS and user-level library support. The OS is responsible for persisting all data structures such as process context, VMAs and pages that form a mapped region, whereas the persistence of the actual content, i.e., application data, is the responsibility of a user-level library. In other words, the OS provides metadata persistence for mapped regions, whereas the user library manages the region to offer applications a persistent object store. We first describe the OS support for the persistent mapped region, and then discuss the user-level object store support.

Complexity of persisting VM structures. When using the VFS, the filesystem and its metadata already support persistent regions. In pVM's VM-based design, three important OS structures that form a mapped region require persistence: the process context, all VMAs inside an application, and finally all page structures inside each VMA. However, persisting these in-memory data structures is significantly complex. Each of these structures dereferences other OS structures, as they are continuously updated by different OS subsystems. Moreover, the VM data structures are designed for volatile DRAM, and hence are not persistence-friendly.

Key Idea: Replica log structures. To make the process context, VMA, and the page structure persistent, we propose a log-based approach, where we create simplified persistence-friendly replicas of these data structures. Each replica contains only the necessary information that is required to locate and load all persistent pages of a process. We term the replicas nvproc, nvma, and nvpage, corresponding to their in-memory process context, VMA, and page structures, respectively. Figure 6(A), shows the in-memory state of a process context with VMAs and related pages. Figure 6(B) in the middle shows the corresponding NVM replica structures (nvproc, nvma, nvpage) in a tree representation. Figure 6(C) represents the persistent storage log format of the replica structures.

Each process context replica, nvproc, can have a tree of nvmas, and each nvma can have a tree of nvpages. Each nvproc is identified by a globally unique identifier (gUID) supplied by the user library that maps a region of NVM using the OS interface *nvmmap()*. Each nvma contains the start and end physical addresses corresponding to the virtual address range of the VMA, and a locally unique VMAID supplied by the user library (generally an incremental *mmap()* counter). The VMAID is used for indexing the process nvma tree. Similarly, the nvpage has a physical address of the VM page and an offset set to the starting address of the VMA to which it is associated.

Creating and reloading persistent mapped regions. A persistent mapped region is created by an application or an object store library using the OS *nvmmap()* system call (see steps ① to ⑦ in Figure 7(A)). The library also provides a persistent map flag and a globally unique identifier (gUID) for each application. pVM's OS-level persistence manager uses the gUID to locate the persistent state of the replica process context in the log, and if it cannot find one, it creates a new context. Next, a VMA with appropriate persistent flags is created along with its VM, and it is then added to the replica nvproc's nvma tree, as shown in Figure 6(B). For new persistent pages allocated and added to the in-memory VMA structure, a corresponding nvpage is created and added to the corresponding nvma RB-tree, indexed by the page offset.

To reload or read a persistent region (see Figure 7(B)), the application/user-library provides the gUID and the persistent region VMAID maintained in its user-level persistent metadata. The gUID and VMAID are used to locate the state of the corresponding regions in the persistent replica logs and load them into a tree structure. The pages are loaded into an application address space only after the application accesses a persistent page in the mapped persistent region. During the first touch, a page fault is generated, and by finding the offset of the faulting address with the starting address of a VMA, a corresponding nvpage structure is identified. This contains the physical address of the page to be loaded and added to applications page table. We favor this lazy approach as it reduces restart and read times and limits TLB pollution.

Consistency and durability. The OS layer is only responsible for maintaining consistency and durability of

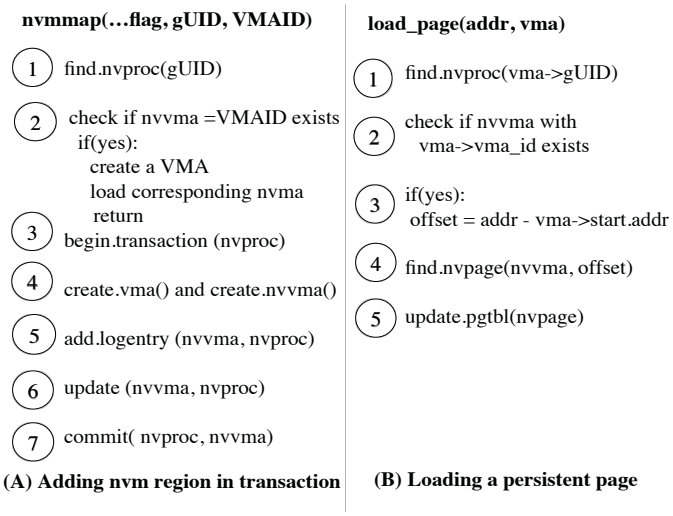

Figure 7: pVM persistent region creation, and page load.

the OS-level persistent state, i.e., the replica structures required to construct the persistent region. The consistency and the durability of actual application content are managed by the user-level object store. pVM borrows the ideas and code from PMFS's [20] optimized OS-level UNDO journaling with atomic commits for cacheline-sized updates. While PMFS maintains the consistency and durability of the filesystem metadata, pVM maintains consistency and durability for the VMA replica structures. Furthermore, a key difference between the PMFS filesystem and pVM design is that, rather than keeping a single journal of the entire filesystem, pVM maintains a separate journal for each gUID, or each process replica structure, with a global master journal bookkeeping the location of the individual process replica journals. Having multiple journals avoids contention for single journal lock across applications, specifically during frequent page updates. However, pVM currently requires applications to handle object sharing and to deal with concurrent updates. We plan to extend pVM with a more transparent mechanism, similar to the one presented in [42].

Figure 7(A) shows the high-level transactional journaling code for creating a new nvma structure. For updating an nvma, pVM first logs the nvma and its corresponding nvproc structure, and then logs the nvpage and nvma when adding or updating a new page. Each log entry in a journal is 64 bytes and the header and data of a log entry are committed first, followed by the log tail. Log updates are ordered with optimized write barriers (WB_BARRIER) and cache flush (CL_FLUSH) instructions discussed in detail by prior work [20]. Recovery happens by reloading first the master journal, followed by the log data in the journal. pVM follows an all-or-nothing model to guaranteeing ACID properties for OS persistent state. A failure to load any one persistent structure of an application makes an entire application state unusable.

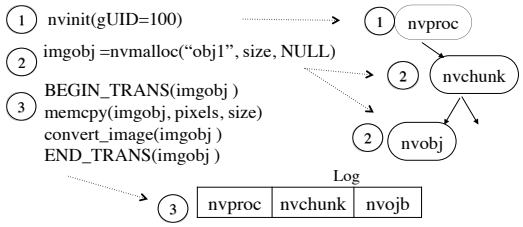

Figure 8: pVM object creation & updates.

4.2 pVM-lib allocator and object store

We next discuss pVM's user-level library that provides the application with a NVM allocator for capacity use, and object store support.

pVM capacity allocator and NUMA policies. Modern allocators map large regions of memory from the OS and manage them for subsequent allocations. For pVM, we extend the scalable jemalloc [6] library allocator with *nvmmap()* support for allocations to NVM. We use jemalloc for both nonpersistent capacity and persistent allocation with appropriate flags to distinguish them. By using a library allocator, no application-level changes are required for nonpersistent applications. Applications can also use the NVM NUMA policies.

pVM object store interface. pVM's object store interface and ACID mechanisms rely on named objects as inspired by prior NVM-as-heap research [17, 40]. We first briefly describe the application interface, followed by its object store metadata management customized for VM-based OS design. The object store provides applications with a *nvmalloc()* interface to create named, uniquely identified object (objID) formed by combining the per-application gUID and the unique object name. Figure 8 shows a sample code to allocate an image object and update it using the load and store interface. To provide consistency and durability, pVM uses Intel's persistent memory library [27] to wrap updates to an object inside transactions. We discuss the details next.

Persistent object store metadata. The persistent memory object store creates objects using a persistent memory allocator and maintains persistent metadata (state) about the objects, including their location in the mapped regions, size, objID, and information about consistency. Modern allocators are complex and maintaining the entire state of the allocators in the NVM can be expensive. Hence, pVM creates simplified replica structures in a log, similar to the OS-level replica structures discussed earlier. During application initialization, a unique gUID is generated, and a corresponding user space nvproc structure is created in a user-level persistent log, as shown in step ① in Figure 8. Next, the allocator creates a large persistent region with a gUID, and an incrementing MMAPID (used as VMAID by the OS). Upon successful creation of an NVM region, a corresponding nvchunk structure is added to the NVM metadata, as

shown in step ②. Each application (nvproc) can have several mapped regions (nvchunks), and for each object in the persistent region, a corresponding nvobject is added to the nvchunk. For consistency and durability, the objects and the object store metadata are updated within transactions, using an UNDO log ③. Note that the UNDO log is a journal located separately from the persistent metadata. Our consistency and durability mechanism is borrowed from object-based logs used in prior work [17]. For reading or recovering a persistent object, applications use the *nvread()* interface, with the gUID and object name. The object identifier generated from the combination of gUID and object name, is used to locate the corresponding persistent region and to instruct the pVM's OS manager to load the region into the application's address space. Furthermore, the consistency and durability of the persistent allocator is important. After a restart or recovery from a failure, the persistent allocator first recovers all nvproc, nvchunk, and nvobject data structures from the allocator log, rebuilds the allocator state, and garbage collects all uncommitted and unused objects [28].

Discussion. Several research proposals have focused on optimizing NVM object stores that rely on the VFS. However, pVM's main contribution is its generic VM-based OS (pVM-OS) design that addresses capacity bottlenecks and improves object storage performance. To understand the effort required to adapt other open source object stores to the pVM-OS design, we extended Intel's SNIA-based NVML library [27] with 75 lines of code. NVML maintains its own object store metadata. Hence, to create pVM-based persistent regions, we replaced the VFS-based *mmap()* with *nvmmap()*, letting us map the objects to the corresponding NVM regions using the unique ID for the mapped region similar to pVM's gUID.

5. Experimental Evaluation

We evaluate pVM and other NVM mechanisms to: (1) understand their memory capacity scaling capability, (2) analyze the cache and TLB usage efficiency and page allocation and access time, (3) quantify the object storage performance and its implications on the time spent in the OS I/O stack, and (4) measure the cost of consistency and durability guarantees.

5.1 Methodology

Experimental Setup and NVM Emulation. For our evaluation, we use a 2.4 GHz, 8 core Intel Nehalem platform with 12MB LLC, dual NUMA socket with each socket containing 3GB DDR3 memory, and a Intel-510 Series SSD. pVM is based on the Linux 3.9 kernel, whereas for PMFS and ramFS we use Linux 3.11. Mnemosyne uses a legacy 2.6.33 kernel. We use one of the NUMA socket as an NVM node. To emulate NVMs with 10x lower bandwidth and 2x or 5x slower read/write speeds relative to DRAM, we first use thermal throttling of a NUMA socket to reduce the bandwidth. To emulate latency, we dynamically inject delays into the application runtime by finding the number of load/store LLC misses after a fixed interval of time (100 ms for our experiments). While Dulloor et al. [20] use a similar technique by modifying the processor microcode-level on a specialized server-class machine, we emulate this technique via software since we lack access to such microcode. For our experiments (capacity and object store), we consider five configurations, ① DRAM only – a 6GB DRAM only configuration, ② PMFS – a split 3GB DRAM and 3GB PMFS managed approach, ③ Mnemosyne with 3GB DRAM and a 3GB NVM managed by the storage class memory map (scmmap) driver, ④ ramFS - a memory-based filesystem with 3GB DRAM and a 3GB ramFS filesystem that does not offer any ACID-based persistence guarantees (unlike PMFS), and ⑤ pVM - 3GB DRAM and a 3GB NUMA node managed by pVM. Table 3 compares the functionality and the design for each of the approaches.

Applications	Description	Workload	NVM usage type
FaceRec [3]	OpenCV face recognition on Gallagher dataset [21]	1.2GB input DB (2K images, each 800KB)	Capacity
Metis [13]	Uses Metis with 4 mappers-reducers	2GB crime data set	Capacity
GraphChi[29]	Graph pagerank algorithm	Orkut graph, 117 million edges	Capacity
Dedup [12]	Parsec Dedup benchmark	4GB OS image file	Capacity
Snappy [5]	Fast data compression used in several Google products	Image, video, audio, document files -2GB	Object store
LevelDB [22]	Google's DB used from browser to datacenter	SQLite benchmark 500K operations	Object store
Phoenix [44]	Shared-memory MapReduce running word-count	same as Metis	Object store

Table 2: Applications.

Approach	Memory scaling	Mgmt. subsystem	NUMA awareness	Consistency & durability
Mnemosyne	No	Filesystem	No	Yes
ramFS	Yes	Filesystem	No	No
PMFS	No	Filesystem	No	Yes
pVM	Yes	Virtual memory	Yes	Yes

Table 3: Comparison of approaches.

Applications' use of NVM. We use a set of benchmarks (discussed along with the results) and applications that are shown in Table 2 to evaluate the capacity scaling and object store benefits and implications of pVM. In Table 2, we categorize the applications into two types based on their NVM usage (last column). Applications marked as 'capacity' use NVM only for memory capacity scaling and keep input data in the SSD. Applications marked as 'object store' use NVM for both persistence and additional capacity. For all approaches, (pVM, ramFS, PMFS, Mnemosyne) we use the jemalloc allocator [6] from Intel's NVML library [27]

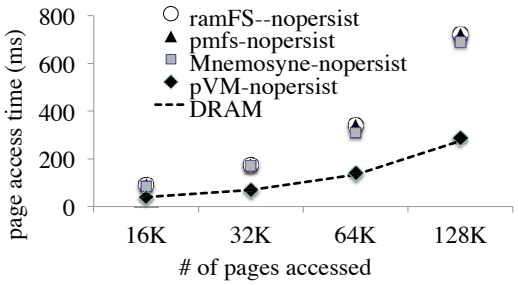

Figure 9: Non-persistent page access time for capacity use.

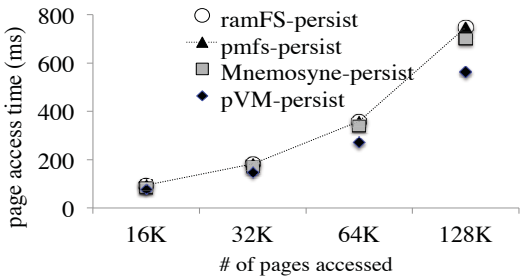

Figure 10: Persistent page access time.

because of its multithreaded scalability. PMFS and ramFS map memory from their respective persistent store using the MAP_PRIVATE flag, whereas Mnemosyne uses the MAP_SCM flag. For pVM, we replace the *mmap()* interface in jemalloc with the *nvmmap()* interface. For memory scaling analysis, we use pVM's Linux NUMA library extensions that support NVM-based NUMA policies. For persistent storage analysis, we use applications that are a good fit for object-based storage (LevelDB, Snappy, Phoenix), as discussed by Harter et al. [26]. When using the object interface, we replace the POSIX interface with NVM's heap-based object store. Modifying Snappy [5] required less than one man-day.

Limitations of memory-based filesystems. ramFS and Mnemosyne extend the Linux page cache to provide persistence using a simplified filesystem/persistence driver. Both of these approaches rely on an application-level allocator that first maps a large file with required flags. The flags and the file descriptors are used by the VFS-based component to allocate from any NUMA node that the current process is scheduled to. VFS uses the allocator managed by the VM subsystem, but persistent and non-persistent use are not differentiated. The allocated/used pages are marked dirty, which prevents the VM from recycling or swapping them to SSD. As a result, the application can non-deterministically terminate when memory usage exceeds the available free memory. We discuss these issues in more detail when evaluating the memory capacity scaling of applications.

5.2 NVM capacity use analysis

To understand the effectiveness of pVM's VM-based design for using NVM for additional capacity, we first use a set of benchmarks to evaluate page access cost and cache and TLB efficiency, and then we use real-world capacity-intensive applications to understand the implications of pVM's NUMA-based policies. We compare pVM with the other VFS-based approaches.

Page allocation cost is an important metric for understanding OS and application performance. Applications and allocators map regions of memory or NVM using the *mmap()* (or *nvmmap()* for pVM) interface for capacity and persistence needs. However, the actual allocation of a page happens only on its first touch. To measure the page allocation cost, we use the Linux memory mapping scalability benchmark [2], which first maps a large region of memory and then randomly touches pages across their page boundaries. Figures 9 and 10 show the performance of non-persistent and persistent page accesses. The x-axis represents the total pages accessed, and the y-axis represents the page access cost, including page allocation time and the time to update the page table. We analyze this for PMFS, ramFS, Mnemosyne, DRAM, and pVM.

Analysis. pVM distinguishes between non-persistent (capacity) and persistent allocation using the flags provided to *nvmmap()*, and hence does not add filesystem metadata/-bookkeeping or journal overheads for nonpersistent allocations. In contrast, the VFS-based PMFS lacks the ability to distinguish between persistent and non-persistent use and therefore adds filesystem overheads even for pmfs-nopersist access. Therefore, pVM-nopersist reduces the page access cost by 2.5x relative to pmfs-nopersist. ramFS and Mnemosyne also suffer from high page access costs, both similar to that of PMFS. Mnemosyne provides a heap-based interface to applications, but page allocation is handled by the VFS subsystem, which is implemented over the filesystem buffer cache and requires filesystem metadata access. We also observe that Mnemosyne provides less than a 10% benefit over PMFS, and benefits mainly result from avoiding the strict transactional updates of PMFS.

Persistent page access requires loading the filesystem B-tree or the dentry caches and updating several complex data structures such as an inode, an inode bitmap, blocks, and a superblock. While pVM also has to locate pages from a VMA-level (nvma) RB-tree, only three simple replica structures, nvproc, nvma, and nvpage, are updated and journaled. Hence, pVM-persist shows a maximum performance gain of up to 20% compared to pmfs-persist and Mnemosyne and around 8% over ramFS, which does not include persistence guarantees. The small improvement of pmfs-nopersist over pmfs-persist comes from the use of the MAP_PRIVATE flags that instruct the filesystem to skip persistence operations on the page content. Using a B-tree instead of an RB-tree in pVM can improve these gains further. *This set of results shows the importance of using the VM-based design to distinguish between persistent and nonpersistent page allocation/access.*

We next look at how efficiently pVM uses the processor cache and TLB with applications from the well known

App	Canneal	Facesim	Ferret	Stream cluster	Swaptions	x264
Type	Enginee--ring	Animation	Similarity Search	Data Mining	Financial Analysis	Media processing
WSS	2GB	256MB	128MB	256MB	512KB	16MB

Table 4: Parsec benchmark application characteristics.

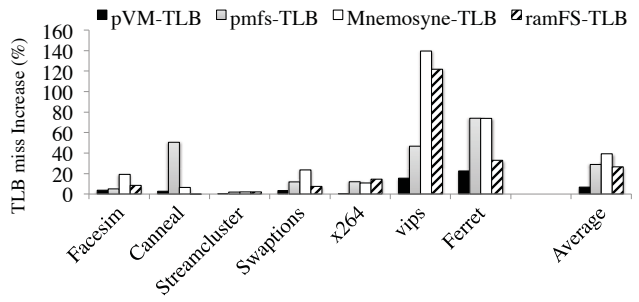

Figure 11: TLB miss analysis. y-axis denotes TLB increase (%) relative to DRAM. The bars corresponding to 'Average' represent the average over all benchmarks.

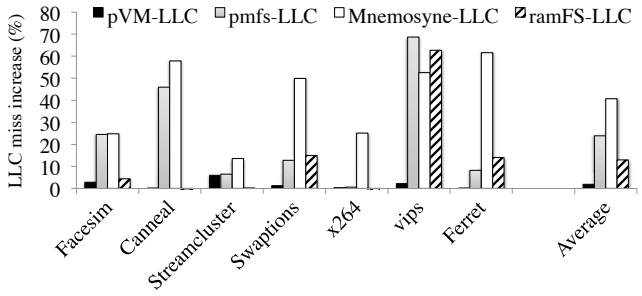

Figure 12: Cache (LLC) miss analysis. Y-axis denotes increase in (%) relative to DRAM.

PARSEC benchmark suite [12]. We use a broad range of real-world applications from the suite (Table 4) with different working set size (WSS) and capacity requirements [11]. Figures 11 and 12 show the percentage increase in TLB and cache (LLC) misses (on the y-axis) relative to using only DRAM (baseline).

Analysis. As evident from the figures, pVM achieves a significant reduction in TLB and cache misses by extending the VM subsystem. It reduces the average TLB and LLC misses by up to 29% and 24%, respectively, when compared to the VFS-based PMFS approach, and even more when compared to Mnemosyne. By reusing most of Linux's virtual memory management support, pVM incurs a negligible increase in TLB and LLC miss rates relative to the DRAM baseline. For highly memory intensive applications such as vips and ferret, the 10-22% increase for pVM is mainly due to higher remote node (NVM node) misses. In contrast, for PMFS, we notice substantially higher TLB miss rates (up to 60%), particularly when the working set size is large (facesim, canneal). A closer analysis reveals that these misses for PMFS are large because VFS does not distinguish between persistent and nonpersistent pages, which adds transactional bookkeeping costs to the filesystem metadata. Note that filesystem data structures are not cache/TLB friendly when compared to VM-based data structures, which further increases the cache and TLB pollution for PMFS.

With Mnemosyne, we noticed a similar significant increase in both TLB and cache misses for memory intensive benchmarks while its VFS-based cache mechanism performed reasonably better for applications with lower memory intensity. Overall, the average TLB and LLC misses increased by 40% and 39%, respectively. For ramFS, the overheads were relatively lower when compared to Mnemosyne. We attribute the difference to the following reasons: (1) Mnemosyne uses only an MAP_SCM flag and does not distinguish between persistent and non-persistent NVM use. Although both ramFS and Mnemosyne extend the VFS caching mechanism, Mnemosyne also provides persistence guarantees for a mapped region. These added overheads lead to a higher cost for persistence when compared to ramFS. (2) Another minor reason for the increase in TLB and cache miss rates is that Mnemosyne uses a legacy Linux kernel, 2.6.33, and does not incorporate the VFS and VM subsystem optimizations of newer Linux kernels. *These results demonstrate that a VM-based design leads to lower overheads with respect to TLB and cache misses.*

5.2.1 Memory scaling and placement impact

To evaluate the effectiveness of pVM's memory placement policies in addressing memory capacity scaling and performance issues, we use the applications in Table 2. For pVM, we evaluate two policies – 'pVM-nvmrevert' (revert to NVM if DRAM is exhausted) and 'pVM-nvmpreferred' (allocate first to NVM). The input data for the applications is placed in the SSD rather than in NVM, so that we can investigate only the capacity scaling benefits. As explained earlier, when using ramFS, PMFS and Mnemosyne for additional capacity, the user-level allocator has to map one or more large files to the NVM. However, the application's memory requirement can also sometimes exceed the available NVM space. Both ramFS and Mnemosyne use the VFS page cache that can dynamically scale over DRAM, whereas PMFS requires a reservation during boot. On the other hand, ramFS and Mnemosyne cannot release pages that have been allocated and written because they do not support swapping. Lack of swapping results in significant memory pressure, which can slow down the overall system and even cause termination of the applications.

Analysis. We first compare pVM against DRAM, PMFS, and ramFS in Figure 13. Clearly, DRAM (the optimal case) and pVM-nvmrevert outperform PMFS. In PMFS, when DRAM is exhausted, the OS lacks support to allocate transparently from a PMFS backed NVM because it uses an entirely different subsystem (VM versus VFS). With the increase in DRAM pressure, swapping is initiated, which results in the application slowing down. However,

pVM-nvmrevert provides a seamless memory scaling capability to switch automatically to NVM when DRAM is exhausted. While using NVM reduces performance compared to DRAM, it still provides a 2.5x and 2x speedup relative to PMFS for applications such as FaceRec and Metis. Interestingly, for Dedup with 3GB peak memory usage and 3GB NVM (a corner case), the PMFS approach (54 sec) marginally improves performance compared to pVM-nvmrevert (61 sec). This additional overhead occurs because pVM-nvmrevert starts allocating pages from the slow NVM after reaching a non-critical free page threshold for DRAM [24]. Normally, this logic prevents all DRAM pages from being exhausted and as such is also common in current Linux NUMA policies, which pVM builds on. In the PMFS case, it is not NUMA-aware, so it exhausts all 3GB without using slow NVM pages. A stricter, NVM-specific policy can be used to avoid these overheads. As expected, pVM-nvmpreferred has the highest cost. We expect the pVM-nvmpreferred policy to be useful only for low priority or memory bandwidth and latency-insensitive applications. pVM-nvmrevert achieves close to DRAM performance and significantly reduces cache and TLB (80%) miss rates versus the VFS-based approach discussed earlier in Section 2 (see Figure 2).

Unlike pVM, ramFS has no flexible policies. Hence, we map a large region of memory (6GB, which includes DRAM and NVM nodes) as ramFS and run the application with a DRAM preferred NUMA policy that would first allocate pages from DRAM and then from NVM. ramFS performs better than PMFS as it can dynamically allocate pages using the OS allocator, and it does not provide any journaling or ACID guarantees. As a result, for some applications, ramFS performance is similar to pVM (e.g., FaceRec and Graphchi). The performance of ramFS drops with memory capacity-intensive applications such as Metis and Dedup because dirty pages cannot be swapped, which results in the application slowing down due to memory pressure.

Figure 14 shows the comparison against Mnemosyne without the NVM latency and bandwidth emulation. Mnemosyne's legacy PCI driver does not export memory controller information required for DRAM throttling, so we just measure the software bottlenecks. As expected, Mnemosyne's performance trends are similar to those of ramFS, except with higher overhead for memory-intensive applications (Dedup and Metis), because it reuses most of the ramFS page cache code. *pVM's memory scaling capability and flexible NVM memory placement policies allow for better performance with memory-intensive applications that scale across DRAM and NVM.*

5.2.2 Database benchmark

To analyze the impact of pVM's VM-based object storage performance, we first analyze the overall throughput and the OS-level overheads, such as journaling, for the popular NoSQL LevelDB key-value store [22, 23]. We evaluate:

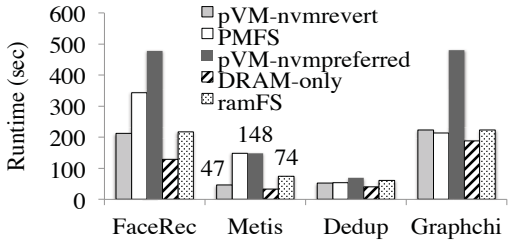

Figure 13: Memory scalability impact.
(pVM-nvmrevert, pVM-nvmpreferred are NUMA policies)

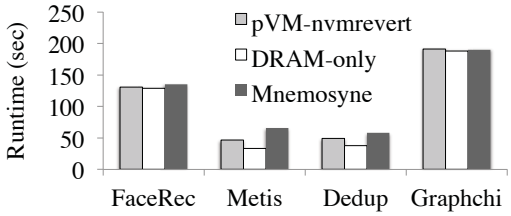

Figure 14: Memory scalability impact without NVM latency and bandwidth emulation.

(1) pVM-obj, (2) pmfs-obj – pVM's user-lib for object store with memory mapped from VFS-based PMFS, (3) pmfs-mmap – a mmap instead of block-I/O version of the applications, (4) pmfs-block (baseline) – block-based file I/O with a traditional POSIX interface, and (5) ramFS – an approach with no consistency or durability guarantees. We do not evaluate Mnemosyne due to the lack of support for full NVM emulation.

LevelDB offers to applications a key-value interface and is a perfect fit for an object-based store. LevelDB internally uses a POSIX block interface for updates and a memory map interface for reads. Figure 15 compares the throughput for pVM and pmfs-obj relative to pmfs-block for 500K transactions of the LevelDB benchmark [23] where the x-axis shows the four types of access patterns. For Write-Sync, updates even within a transaction are committed (logged), unlike Write-Seq and Write-Random where commits happen only outside the transaction. For read operations, the entire database is mapped to memory without significant performance differences between different approaches. Hence, we only study the write access patterns. Figure 16 shows the reduction in throughput (%) as a result of OS-journaling.

Analysis. Clearly, both pmfs-obj and pVM provide higher throughput relative to pmfs-block. pmfs-obj improves throughput for sequential and synchronous writes by 15-20%, respectively, whereas pVM-obj achieves a 30-53% increase. Both pmfs-obj and pVM-obj reduce the OS-filesystem and the user-kernel switch cost, unlike with pmfs-block and pmfs-mmap. Note that pmfs-mmap also requires several supporting I/O calls every time a file is mapped into memory. Other overheads include user-space-to-kernel data copy costs discussed in (Section 2), and most importantly, higher OS journaling costs, as shown in Figure 16.

pVM, pmfs-obj comparison. Although pmfs-obj reduces the OS filesystem and context switching costs because ob-

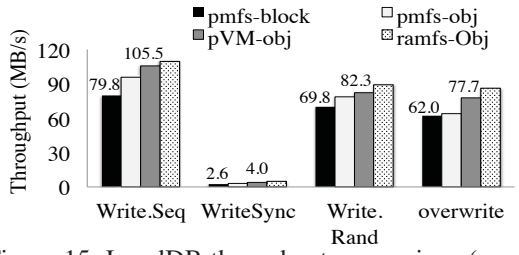

Figure 15: LevelDB throughput comparison (numbers over the bar in MB/sec).

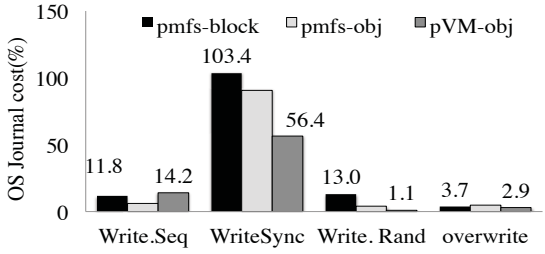

Figure 16: LevelDB: OS journaling cost (%) on throughput. ramFS does not support journaling.

jects are mapped to a region managed by the VFS (i.e., PMFS), we see higher OS-journaling overhead due to complex filesystem metadata, higher page access costs for frequent allocations and release operations for small transactions, as well as higher cache and TLB misses. Instead, pVM's persistent storage is specialized for object stores, and it needs to maintain a consistent state by using only its replica structures with some bookkeeping discussed in Section 4. As a result, the OS journaling cost with pVM and its impact on database throughput are reduced, thereby providing better performance. ramFS, a hypothetical case without persistence guarantees, does not have an OS journaling cost or related cache and TLB misses. However, this approach is useful when an application does not require any durability guarantees.

5.2.3 Object store analysis for applications

Finally, we study the impact of pVM's VM-based design for improving the overall application performance and reducing the time spent in OS operations. We compare the functionally similar PMFS and pVM approaches.

Impact of object-based interface. In Figure 17, the left y-axis shows the file compression throughput for Snappy (MB/sec), and the right y-axis shows the time spent inside the OS and filesystem. Figure 18 displays the runtime comparison (left y-axis) for a word count in Phoenix with a 2GB data set, and the right y-axis shows the corresponding OS time. For Snappy, both pVM-obj and pmfs-object provide significant gains over the traditional pmfs-mmap and pmfs-block approaches. As discussed in Section 2, applications like Snappy show high variability in I/O sizes and perform frequent I/O calls. Using an object-based interface (pVM-obj or pmfs-obj) significantly reduces the cost associated with context switches, filesystem metadata, and the synchronization overheads of pmfs-block and pmfs-mmap. As a result, the time spent inside the OS (and specifically in the filesystem) is reduced by up to 4x with this interface. Interestingly, using the mmap interface with pmfs-block causes the performance to deteriorate when compared to pmfs-block. This performance difference is primarily due to the increased kernel mediation from several supporting system calls for files which have low processing time.

The second application, Phoenix MapReduce [44] maps the input data into the application address space, and then computes over the mapped input which is either backed by the VFS or pVM. Since no I/O calls are made after the initial mapping, the object-based interface provides less than an 11% improvement over pmfs-block. The memory map mode performs better compared to the block-mode by avoiding an extra copy from the filesystem to the DRAM buffer and has almost similar performance as pmfs-mmap.

Extending VM for persistence. When comparing pVM and pmfs-obj, pVM reduces the kernel time for Snappy by 67% relative to pmfs-obj, leading to an 18% improvement in throughput. Similarly, the throughput gains for Phoenix are around 14%. Although pmfs-obj uses pVM's object library at the user level, the objects reside in files mapped and managed by the PMFS filesystem, and the PMFS page allocation component controls all persistent page allocations and accesses. In contrast, pVM benefits from faster page accesses that avoid filesystem-based metadata updates and that reduce journaling overhead cost, as discussed for the LevelDB benchmark. This results in fewer instructions and NVM accesses compared to pmfs-obj, as well as fewer TLB and cache misses, as shown in Table 19. Although the reductions are seemingly small, pVM's gains are of significant importance due to the 5x higher write latency of NVM relative to DRAM.

5.3 Summary

We make the following conclusions from the experimental results presented in this section. pVM's VM-based design with NUMA capability addresses the memory capacity scaling issue by providing applications with flexible memory placement policies, and subsequently improves performance by up to 2x compared to PMFS (the pmfs-mmap and pmfs-block approaches). Next, by extending the OS to distinguish the NVM pages between capacity and persistence use, pVM reduces TLB and cache misses significantly – by up to 80% for the applications and 29% for the Parsec benchmark compared to the VFS-based PMFS. Furthermore, pVM's object store reduces software-stack overheads, page access time, CPU instructions, and cache and TLB misses, which results in nearly 2x higher throughput and up to a 4x reduction in OS time relative to block I/O. These results also motivate the need for OS redesign to better handle object storage.

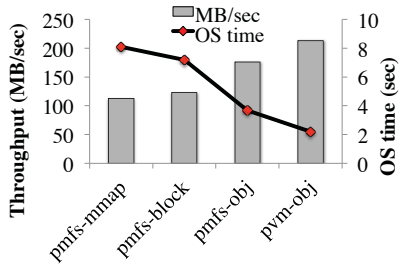

Figure 17: Snappy throughput, OS time.

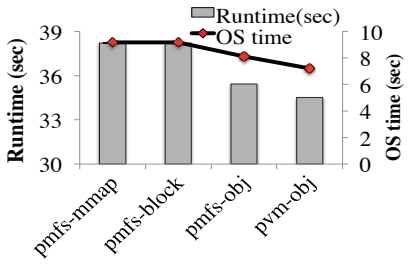

Figure 18: Phoenix runtime gains.

App	Instruct (%)	TLB Miss(%)	Cache Miss(%)
Snappy	8.20	11.7	6.32
Leveldb	9.40	8.32	6.16
Phoenix	3.17	1.1	5.6

Figure 19: pVM cache, TLB miss, CPU instruction reduction relative to pmfs-obj.

6. Related Work

We first discuss existing NVM capacity scaling research, followed by the persistence storage-related research.

Capacity scaling in hybrid memory systems. Prior research [19, 30, 34, 37] has used NVM as an alternative to DRAM with transparent page replacement strategies between DRAM and NVM. Qureshi et al. proposed a hardware-based model that treats DRAM as a cache and NVM as main memory, using the page access patterns in the OS/hardware to move data between DRAM and NVM. While this provides application and OS transparency, copying data between a fast and slow memory can add significant overhead. It limits application flexibility in memory placement, and more importantly prevents the use of persistence property. Saxena et al. [37] propose using Flash as an extended memory managed by the VFS, and use several data prefetching techniques. PMFS also enables extending memory capacity via a VFS. In contrast, pVM provides a VM-based management and, by treating NVM as a NUMA node, allows for seamless memory scaling and flexible memory placement mechanisms. To the best of our knowledge pVM is the first OS design that considers extending the VM subsystem for both capacity and persistent storage, and that provides flexible memory placement policies to applications.

Filesystem for NVM. Next, regarding persistent storage, Aerie [41], Moneta-D [15], and some early work such as FUSE [35] aim to reduce the OS overhead by moving the filesystem to the user space, and directly using the hardware for read/write operations. We discussed the decentralized design of Aerie in Section 2. Other research [14, 18, 20, 43] has proposed optimizing current block-based filesystems to adapt them for memory-based persistence, as explained earlier. pVM borrows several OS consistency and durability-related ideas from such prior work, and adapts them for object storage. Our future work will consider the co-existence of a filesystem with pVM's object based storage.

NVM as a heap. The benefits of using NVM as a heap for persistent storage have been well documented by the earliest systems – Rio Vista [31], and RVM [36]. These solutions provide heap-based persistence and OS-level management that uses a combination of block-based storage and battery-backed RAM. In addition, Rio Vista provides persistence guarantees only at log flushes. Recent research such as [17, 40] also offers persistent data programming models, in addition to the capabilities of RVM. pVM's user-level library borrows ideas from these proposals. However, the key contribution of pVM is the novel extension of the VM subsystem for achieving both memory scaling and persistence. [17, 40] depend on VFS-backed memory, and we have already discussed their limitations. Furthermore, Volvos et al. [40], and SCMFS [43] propose an NVM persistence OS zone. pVM extends such ideas to create capacity and persistence zones, but more importantly, it provides a cleaner NUMA-based abstraction for NVMs, and provides users with flexible memory placement policies. Guerra [25] et al. discuss solutions to prevent mixing persistent and nonpersistent pointers via memory protection, which can also be adapted to pVM.

7. Conclusions & Future Work

We propose pVM, an OS-level solution for exploiting the capacity and storage benefits of byte-addressable NVMs. pVM integrates NVMs into the OS as memory (NUMA) nodes and extends the virtual memory to manage NVM nodes, instead of relying on the VFS to do so. In this manner, pVM addresses the memory capacity scaling issues of state-of-the-art designs and provides 'close to hardware performance'. pVM provides users with flexible memory placement support which is integrated with existing user-level NUMA libraries. pVM further extends the VM subsystem for fast object-based storage to overcome substantial filesystem bottlenecks. By distinguishing between capacity and persistence access at the OS level, apart from the performance benefits discussed in this paper, we believe there are several opportunities to bridge the gap between volatile memory use and persistent storage. We plan to explore more end-user centric environments, such as the Android stack for gaming, multimedia and other applications.

Acknowledgments

We would like to thank our shepherd Dr. Indrajit Roy and all of the anonymous reviewers for their valuable feedback. Dr. Greg Eisenhauer, Alexis Champsaur, Dr. Jeff Young, Alex Merritt and Dipanjan Sengupta from Georgia Tech provided valuable suggestions on earlier drafts of this paper. We also thank Intel Labs for providing access to the Intel NVM emulation platform. This work was supported by the Intel Persistent Memory ISRA Grant.

References

[1] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[2] Linux Scalability Benchmark. http://lse.sourceforge.net/.

[3] OpenCV. http://opencv.org/.

[4] S3fuse filesystem. https://github.com/s3fs-fuse/s3fs-fuse.

[5] Snappy Compession. http://tinyurl.com/ku899co.

[6] http://www.canonware.com/jemalloc.

[7] Numonyx PCM projection. http://bit.ly/1sph5Qz.

[8] Samsung 8GB chips. http://bit.ly/1oFFiSR.

[9] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[10] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9.

[11] A. Bhattacharjee and M. Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 29–40, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9.

[12] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[14] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. 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, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7.

[15] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8.

[16] Ceph. CEPH file-systems. https://ceph.com/ceph-storage/file-system/.

[17] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1.

[18] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

[19] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. DAC '09, New York, NY, USA.

[20] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6.

[21] A. C. Gallagher and T. Chen. Clothing cosegmentation for recognizing people. In *CVPR 2008*.

[22] Google. LevelDB. http://bit.ly/1wEklYD, .

[23] Google. LevelDB benchmark. http://bit.ly/1ERE4u7, .

[24] M. Gorman. Understanding the Linux Virtual Memory Manager. http://bit.ly/1n1xIhg.

[25] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conference*, pages 319–331, 2012.

[26] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6.

[27] Intel. Logging library. https://github.com/pmem/nvml.

[28] S. Kannan, A. Gavrilovska, and K. Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. HPCA 14.

[29] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0.

[31] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5.

[32] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD),*

2014 32nd IEEE International Conference on, pages 216–223, Oct 2014.

[33] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. HotOS'09, Berkeley, CA, USA, 2009.

[34] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*.

[35] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7.

[36] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, Feb. 1994. ISSN 0734-2071.

[37] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 14–14, Berkeley, CA, USA, 2010. USENIX Association.

[38] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan. A framework for emulating non-volatile memory systems with different performance characteristics. In *Proceedings of the Ninth European Conference on Computer Systems*, ICPE '15. ACM, 2015.

[39] SNIA. SNIA persistent memory specs. `http://tiny.cc/o55p4x`.

[40] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1.

[41] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6.

[42] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014. ISSN 2150-8097.

[43] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0.

[44] R. M. Yoo et al. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. IISWC '09.