Towards Virtual Machine Image Management for Persistent Memory

Jiachen Zhang, Lixiao Cui, Peng Li, Xiaoguang Liu*, Gang Wang*
Nankai-Baidu Joint Lab, College of Computer Science, KLMDASR, Nankai University, Tianjin, China
{jczhang, cuilx, lipeng, liuxg, wgzwp}@nbjl.nankai.edu.cn

***Abstract*—Persistent memory's (PM) byte-addressability and high capacity will also make it emerging for virtualized environment. Modern virtual machine monitors virtualize PM using either I/O virtualization or memory virtualization. However, I/O virtualization will sacrifice PM's byte-addressability, and memory virtualization does not get the chance of PM image management. In this paper, we enhance QEMU's memory virtualization mechanism. The enhanced system can achieve both PM's byte-addressability inside virtual machines and PM image management outside the virtual machines. We also design *pcow*, a virtual machine image format for PM, which is compatible with our enhanced memory virtualization and supports storage virtualization features including thin-provision, base image and snapshot. Address translation is performed with the help of Extended Page Table (EPT), thus much faster than image formats implemented in I/O virtualization. We also optimize *pcow* considering PM's characteristics. The evaluation demonstrates that our scheme boosts the overall performance by up to 50× compared with qcow2, an image format implemented in I/O virtualization, and brings almost no performance overhead compared with the native memory virtualization.**

***Index Terms*—persistent memory, memory virtualization, storage virtualization, virtual machine, cloud storage**

I. Introduction

Persistent memory (PM) is attached through memory buses and provides byte-addressability for persistent data. Compared with DRAM, PM's capacity is several times larger and cheaper. In order to take advantage of the best performance of PM in cloud environment, the support for using PM in virtual machines is also important.

Modern virtual machine monitors (VMMs) like QEMU and VMware vSphere can virtualize PM through current I/O virtualization or memory virtualization [1] [2]. However, although PM is already at the boundary of storage and memory, I/O virtualization and memory virtualization are still separated in a way. Since traditional storage devices are attached through I/O bus, storage virtualization features like thin-provision, base image and snapshot are also implemented along the virtual I/O stack. However, using virtual I/O stack loses the byte-addressability of PM. Although directly mapping a PM region into the virtual machine can preserve the byte-addressability, this kind of memory virtualization can not get the chance of PM image management in modern VMMs.

As storage virtualization features are based on the management of virtual machine images, we believe it is also necessary to achieve virtual PM image management when using memory virtualization for PM. However, it will be more challenging compared with I/O virtualization, since memory virtualization today are dominated by hardware-assisted address translation like AMD's Nested-Page Table (NPT) and Intel's Extended Page Table (EPT) [3]. Both NPT and EPT implement dedicate second level page tables to accelerate the address translation. When using PM under the hardware-assisted memory virtualization, once a VMM register a PM region to the host OS kernel, the control of virtualized memory is handed over to the host OS and hardware MMU. That is to say, any data access to virtual PM regions will never pass through VMMs, which reside in the user space of host servers.

To achieve PM image management, in this paper, we enhance the native hardware-assisted memory virtualization. We design an *image monitor* inside QEMU to manage PM images created on PM-aware file system. The *image monitor* acquires the control opportunity to organize PM image by handling EPT violations and write-protection violations in user space.

Data of virtual disks is usually organized in well-formatted image files. Image formats like qcow2 [4] and VMDK [5], which support abundant storage virtualization features, are widely used in cloud environments. However, as most of them are designed for I/O virtualization, some metadata are not suitable or not necessary for memory virtualization. On the other hand, although modern VMMs support creating raw PM image files for memory virtualization based on virtual PM regions, the mapping between the raw format images and virtual PM regions is linear, and no storage virtualization features can be implemented [1]. Therefore, existing image formats cannot meet our needs.

Thus, in this paper, we also design an image format for PM called *pcow* (short for PM copy-on-write). Like the modern virtual machine image formats, the *pcow* format supports thin-provision, base image and snapshot. Compared with formats designed for I/O virtualization, we eliminate unnecessary metadata like address translation tables, and consider the consistency issue caused by PM's smaller write atomicity [6].

The contributions of this paper are as follows.

1) We summarize the current schemes of PM virtualization.
2) We enhance the current memory virtualization in QEMU to support PM image management.
3) We propose the *pcow* image format, which is compatible with the enhanced memory virtualization and optimized for PM.

4) We implement several storage virtualization features based on the enhanced memory virtualization and *pcow*.

The paper is organized as follows: We describe the background in Section II and explain our motivation in Section III. In Section IV, we describe the idea of the proposed methods. Section V discusses the implementation details and the performance optimization for PM. We give experimental results in Section VI, present the related work in Section VII, and make the conclusions in Section VIII.

II. Background

A. Persistent Memory

Persistent memory (PM), also known as storage class memory (SCM), is a type of memory device based on non-volatile memory (NVM) technologies. There are plenty NVM technologies under development, such as 3D XPoint, PCM (phase-change Memory) [7], ReRAM (resistive random-access memory) [8] and STT-MRAM (spin-transfer-torque MRAM) [9]. Although most PM devices are not mass-produced, a few technologies like 3D XPoint are preparing for commercial use [10].

PM is byte-addressable and can be accessed by load/store instructions like DRAM, and non-volatile like block devices. While using PM can get the benefits of using both memory and storage, we also have to face the common issues of memory or storage:

- **As memory**, like DRAM, we have to face cache coherence and atomic visibility [6] [11] issues when programming with PM.
- **As storage**, PM also have the issues like data durability and crash consistency. As the program interface and write atomicity changed, we should switch to cache flush instructions (like *clflush*) to make the recently changed data persistence from CPU cache to PM [12]. To maintain the order of cache flush instructions or store instructions in non-TSO (total store order) architecture (like ARM) [13], a cache flush or store instruction should also have a fence instruction (*sfence* or *mfence*) followed. Facing power failure or system crash, we also need to consider the crash consistency issues [14] in the context of PM [15].

According to SNIA's NVM programming model [16], in order to take advantage of both storage and memory characteristics of PM, we should access PM in `NVM.PM.FILE` mode. According to this mode, all or part of a PM file is directly mapped as a virtual memory region to user-space applications, then the applications can directly load/store the PM region. The data durability is assured after the file system or the applications issue cache flush instructions. The feature implements `NVM.PM.FILE` is called direct access (DAX) in both Windows and Linux. File systems with DAX feature supported are called PM-aware file system [6]. Linux's XFS and ext4 already support being mounted with DAX enabled.

As virtual machine images are backed by files on host OS file systems, the image isolation is achieved based on the isolation of files. In this paper, we also achieve the isolation between PM images with the help of a PM-aware file system, ext4, which is DAX enabled.

B. Storage Virtualization

Storage virtualization is a process to manage the mapping relationship between physical storage resources and logical views. Several features are commonly implemented based on storage virtualization. For example,

- **Storage pooling** aggregates small storage devices into a large storage pool, which facilitates storage resource management by a centralized controller.
- **Thin-provision** tends to promise users a large storage space while allocating much smaller space at the beginning. Subsequent allocations will be done as the actual usage increases.
- **Snapshot** protects data at specific time points as read-only using copy-on-write technology. It provides user the option to roll back to any snapshot point.
- **Base image** (also called **template**) provides the opportunity to build a new image based on images created before. This feature makes the deployment and backup much easier. And for cloud providers, this feature saves the storage costs [17].

Since most storage devices are connected to slow I/O buses, most of these features are implemented in VMMs' virtual I/O stacks.

C. Using PM in Virtual Machines

As displayed in Fig. 1, technically, considering storage devices of block device form or PM form, we can achieve four kinds of virtualization in the forms of "virtual device–physical device" relationship. The four forms of virtualization can be implemented through I/O virtualization or memory virtualization:

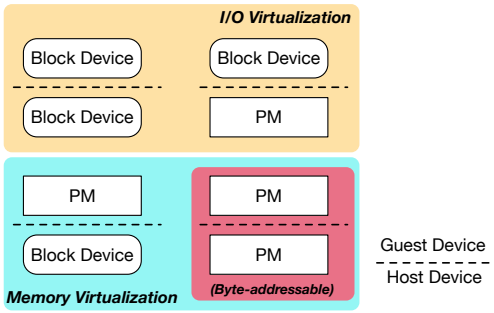


Fig. 1: Forms of storage device virtualization in modern VMMs. Only virtualizing PM under VMMs' memory virtualization ("PM–PM" form) can achieve byte-addressability in virtual machines.

I/O virtualization: Through I/O virtualization, we can achieve "block device–block device" or "block device–PM" virtualization. In this situation, an image files on physical devices is formatted by a VMM and virtualized as a block device in a virtual machine. "Block device–block device" is

the mostly used form for virtual machine storage, but it does not involve PM. The "block device–PM" form loses the byte-addressability of PM, and its complex software stack of I/O virtualization can cause serious performance loss [18].

Memory virtualization: Through memory virtualization, we can achieve the "PM–block device" and "PM–PM" forms. In this situation, a file on a block device or a PM can be mapped as a memory region of a virtual machine. With hardware-assisted memory virtualization, the two-level page tables are managed by guest OS and host OS respectively, and all the address translations are performed by the hardware MMU. As PM-aware file systems provide `mmap` interface with DAX feature, we can create image files on PM-aware file systems and map them to virtual machines. Through this kind of "PM–PM" form virtualization, both byte-addressability and data persistence of PM are maintained in the virtual machine. However, in this way, we lose the storage virtualization features implemented in I/O virtualization. Thus, this paper focus on the methods of implementing storage virtualization features in "PM–PM" form.

III. Motivation

To use PM in virtual machines, modern VMMs either virtualize PM images as block devices through I/O virtualization, or directly map them into virtual machines through memory virtualization. Take virtual machines based on QEMU-KVM as an example, Fig. 2 depicts how the PM will be accessed with the two forms of PM virtualization.

Fig. 2 (a) depicts I/O virtualization. To achieve storage virtualization features like snapshot and thin-provision, QEMU leverages an I/O layer as image *format driver*. All virtual machine image formats supported by QEMU are developed based on this layer. However, the "block device–PM" form has many drawbacks: 1) the block device interfaces in virtual machines will lose the byte-addressability of physical PM. 2) The mismatch between the virtual (512 bytes) and physical (8 bytes) write atomicity makes it difficult to guarantee the crash consistency of applications within the virtual machine. Although this problem can be solved by some host OS features like Block Translation Table (BTT) [19], the advantages of using PM will be further weakened as more middle layers will be introduced. 3) Performance will be dropped because software latency of the I/O layers have already become non-negligible for high-end NVMe SSD devices [20], let alone the faster PM. Related test results in Section VI will also verify the performance degradation of using I/O virtualization for PM ("block device–PM" form).

Fig. 2 (b) shows that in memory virtualization, memory accesses from guest applications require two address translations. A guest virtual address (GVA) will be translated to guest physical address (GPA) by a normal page table walk, and the second EPT or NPT page walk translates the GPA to host physical address (HPA) [21]. However, this two-dimensional translation only involves the two page tables managed by the guest OS and the host OS, and does not involve the VMM, which resides in the host user space. Despite the fact that

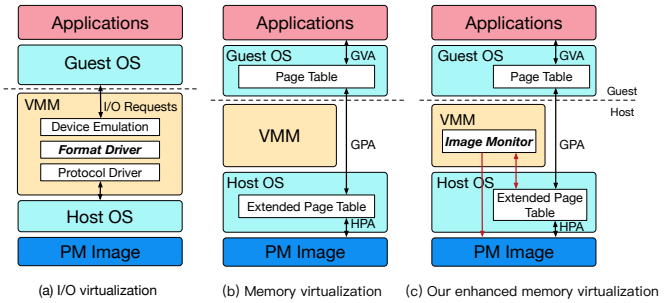


Fig. 2: I/O virtualization, memory virtualization and the proposed enhanced memory virtualization.

in this way we can take advantage of the byte-addressing feature, we cannot add an extra layer to the data path like I/O virtualization for image management.

As PM possesses data durability, storage virtualization features are also expected for virtual PM in cloud environments. Thus, we propose to enhance the current memory virtualization, and achieve the image management by dynamically manage the second level page table (NPT or EPT, we use EPT in this paper). Fig. 2 (c) depicts our general idea. Compared with the native memory virtualization, an *image monitor* is added to the VMMs, which is responsible for the management of the PM images and the related address mapping in EPT.

IV. Design

Fig. 3 depicts the overview of the enhanced memory virtualization. PM image files are created on the PM-aware file system and formatted by the proposed *pcow* format. To virtualize PM region in virtual machines, the PM images are non-linearly mapped in granularity of fixed-size clusters. In order to provide the storage virtualization features, the overall design consists of an *image monitor* inside QEMU and the *pcow* format. The *image monitor* is used to monitor and handle the demand for image expansion and copy-on-write. The *pcow* format is designed for PM images and can work with the *image monitor* to achieve storage virtualization.

A. Image Monitoring

On the startup of a virtual machine, the *image monitor* will map the data clusters of a *pcow* image file according to its logical address recorded in metadata, and mark virtual PM regions mapped from read-only clusters as write-protection. And for areas that has not been accessed, "fake" PM regions without PM image clusters backed (also called anonymous page in Linux) will be mapped to the virtual machines. After the virtual machine is started, there will be two handlers in *image monitor* in charge of access monitoring:

a) Image expansion handler: For raw PM image files already supported by modern VMMs, the mapping between virtual PM regions and image file offsets is linear, which requires the length of the image file equals to the size of the provisioned virtual PM. As thin-provision feature requires the image files grow with the actual demand, we turn to use a

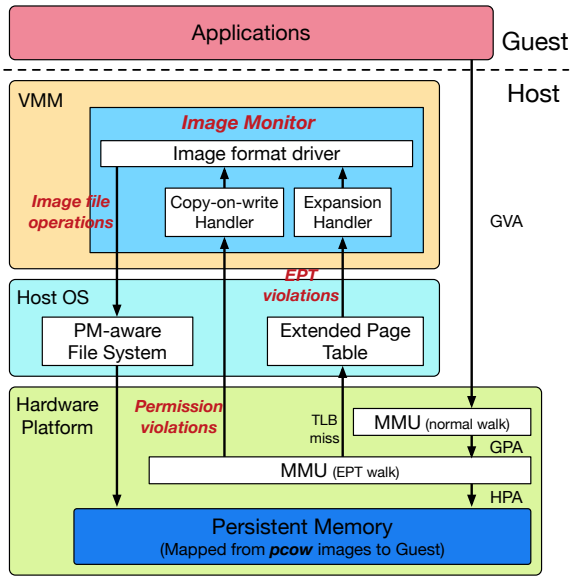


Fig. 3: Overview of the enhanced memory virtualization for PM image management.

strategy we call "fake" mapping. In a word, areas that have not been accessed are mapped as "fake" regions at the beginning, and will be replaced by "real" regions backed by PM image clusters when accessed. The "fake" mapping strategy takes advantage of modern OS virtual memory system's lazy-allocation strategy, which means memory is not allocated until the first access to the mapped address. When these "fake" PM regions are accessed by guest applications for the first time and image expansion is demanded, an EPT violation will be raised, which will subsequently call a page fault handler in OS kernel. Leveraging the `userfaultfd` system call provided by the recent version of Linux, page fault handlers can be registered in user space. In the *image monitor*, we register a user-space page fault handler as *expansion handler*. The *expansion handler* polls `userfaultfd` events caused by EPT violations, and remaps the faulting "fake" regions to newly allocated PM image clusters.

b) Copy-on-write handler: Features like snapshot or base image usually use copy-on-write to protect the read-only data. In our scheme, read-only areas are mapped with write-protection. Accessing the write-protected pages will cause permission violations, and a SIGSEGV signal will be raised. As signals can interrupt the normal procedure of the corresponding thread, the handler can get the chance of doing copy-on-write. Therefore, we register a SIGSEGV signal handler for the vCPU threads to trap the permission violations. The handler is called *copy-on-write handler* and integrated in the *image monitor*. When doing copy-on-write, write-protected data will be copied to newly allocated data clusters. The new clusters will be mapped to the violation address without write-protection and cover the original write-protected region.

B. Pcow: a Format for PM Images

The *pcow* image format is designed for PM, which supports several storage virtualization features including thin-provision, snapshot and base image. The *pcow* image files are created on the PM-aware file systems. Data and metadata are stored in fixed-size clusters, and new clusters are created in an appending manner. As *pcow* leverages EPT and hardware MMU to perform "logical–physical" address translation, dedicated translation tables commonly implemented in virtual disk image formats [18] [22] are no longer maintained by *pcow*. We switch to use a full mapping strategy. During the startup of virtual machines, the *pcow* images will be opened and all the data clusters will be mapped non-linearly to the virtual PM, and the "logical–physical" information will stay in the host OS in the form of EPT. At runtime, the EPT will be updated by the two handlers in *image monitor* when the EPT violations or permission violations happened.

Following is how *pcow* supports storage virtualization features:

a) Thin-provision: New *pcow* images are very small. As the actual virtual PM usage increase, new clusters will be appended to the end of image file, then the *image monitor* will map the new clusters to the virtual PM non-linearly.

b) Snapshot: When a snapshot is taken, a special snapshot cluster will be appended to the end of the *pcow* image. After a snapshot is taken, new data clusters will be continually appended to the image after the snapshot cluster, data before the snapshot cluster will become read-only. Any write access to the read-only area will cause a copy-on-write operation of the *image monitor*. When more than one snapshots are taken, only the data after the last snapshot cluster is appendable and writable. By truncating an image file from anyone of the snapshot clusters, we can achieve the purpose of rolling back to a specific snapshot time point.

c) Base image: The base image feature is similar to the snapshot feature. The difference is the read-only part and writable part are stored in separated image files. The base image feature allows to create a writable image based on read-only images. Images also can be chained together and only the last image can be written. To record the relationship between the writable images and their read-only base image, we add a field in the super cluster named `base_image_name` to record its base image path.

Fig. 4 is an example of *pcow* image. Each image or the area after a snapshot is started with a *super cluster* or a *snapshot cluster*. And the metadata of *data clusters* are stored in their corresponding *meta clusters*. We also define a meta cluster and its managed data clusters as a *segment*. Given a cluster size, the max segment size will be determined. When the current segment is full, new meta cluster will be created and subsequent data clusters' metadata will be stored in the new meta cluster. The image in Fig. 4 consists of three files and can be mapped as a PM region in VMs. Each file is appendable and smaller than the virtual PM size at the beginning, which indicates the thin-provision feature. These files are created on a PM-aware file system and chained together using the base

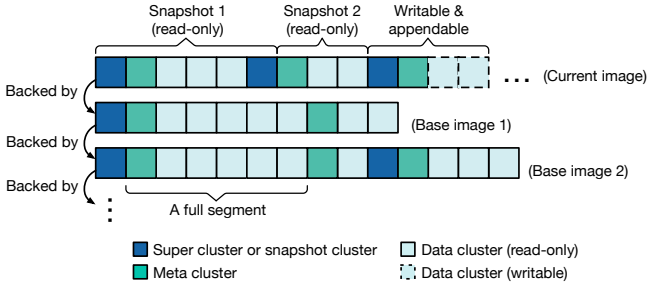


Fig. 4: An example of a *pcow* image that uses the three storage virtualization features (thin-provision, snapshot and base image).

image feature, in which the topmost one is the current image. Only the clusters after the current image's last snapshot is mapped into VMs as writable area.

V. Implementation Details

Our prototype is implemented in QEMU version 3.0 in about 2k lines of code. The host machine is an x86 server with Linux kernel version 4.16. The *image monitor* resides in QEMU and is used to virtualize *pcow* image files as PM regions in virtual machines. In this section we describe details and related optimization of the prototype.

A. Pcow Metadata Layout

Table I and Table II shows the metadata layouts of super cluster and meta cluster respectively. Although the *pcow* image format organizes data and metadata in fixed-size clusters just like other virtual machine image formats, it is designed from the ground up with PM characteristics in mind.

In super cluster described by Table I, we gather all the metadata in cacheline size (64 B), which is CPU cache friendly. The `cur_segment_num` denotes the meta cluster or segment number and will be updated frequently. However, as `cur_segment_num` occupies only 4 B, it can be updated atomically by 8 B memory write. The meta cluster described by Table II consists of a `cluster_counter` and data entries of the corresponding segments. Each data entry consists of a 4 B `cow_bitmap` and a 4 B `data_cluster_num`. The `data_cluster_num` represents the sequence number of the data cluster in the virtual PM address space. The virtual PM address thus can be represented by `data_cluster_num` × `cluster_size`. The `cow_bitmap` is used for the fine-grained copy-on-write optimization for PM, which will be described in Section V-D. All of the fields in meta cluster need to be updated on new data cluster allocation or doing copy-on-write. Fortunately, all fields can be updated under the 8 B write atomicity.

TABLE I: Super cluster layout of *pcow*.

Field Name	Size	Usage
`cluster_size`	4 B	Cluster size in KB.
`max_cluster_n`	4 B	Maximum number of clusters.
`cur_segment_num`	4 B	Allocated segment (meta cluster) number.
`magic_string`	4 B	*Pcow* magic string.
`base_image_name`	48 B	File name of the base image.
`padding`		Fill the remaining space of the cluster.

TABLE II: Meta cluster layout of *pcow*.

Field Name	Size	Usage
`cluster_counter`	4 B	Number of data clusters in this segment.
`padding`	60 B	Fill to cacheline size.
`cow_bitmap`	4 B	Fine-grained copy-on-write bitmap.
`data_cluster_num`	4 B	Logical data cluster number.
`cow_bitmap`	4 B	Fine-grained copy-on-write bitmap.
`data_cluster_num`	4 B	Logical data cluster number.
`cow_bitmap`	4 B	Fine-grained copy-on-write bitmap.
`data_cluster_num`	4 B	Logical data cluster number.
...		

B. Image Mapping at Startup

As our enhanced memory virtulization performs address translation with the help of EPT, we need to map the data clusters to the appropriate location of the virtualized address space at startup. Thus, at startup, for each *pcow* image, *image monitor* scans all the meta clusters and super clusters to map and register the virtualized address space to EPT.

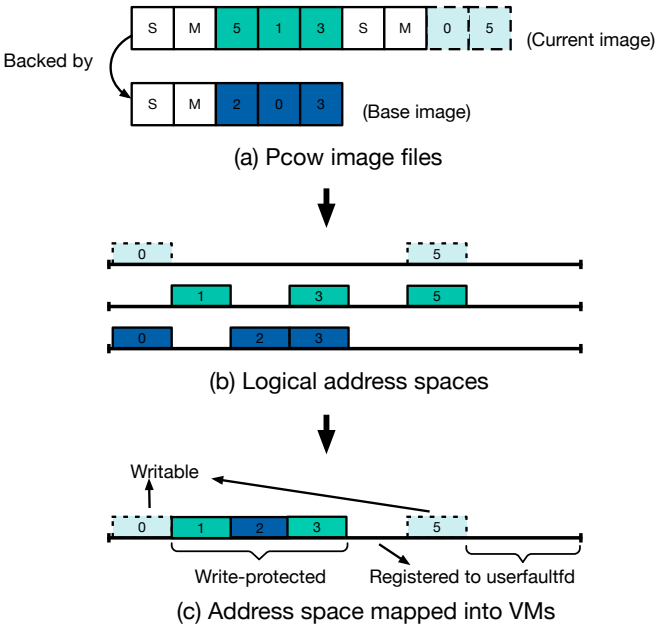


Fig. 5: Initial memory address space mapping.

For example, Fig. 5 depicts a *pcow* image consists of two image files, in which the current image is based on the base image, and the current image consists of a snapshot and the currently writable part (Fig. 5a). For readability, each data cluster is labelled with its `data_cluster_num` in the figure. As displayed, the writable part, the snapshot part and the base image part can be represented by three logical address spaces (Fig. 5b) according to the logical address that derived from `data_cluster_num` and `cluster_size`. Then the multiple logical address spaces are merged into one (Fig. 5c), which is finally mapped to a virtual machine as a virtual PM region.

It should be noted that the logical address spaces may overlap between each other. Thus when the merging step is performed (Fig. 5b to Fig. 5c), there should be priorities. Fortunately, for the base image or snapshot feature, the priorities are straightforward. Among multiple images, the priorities are based on dependencies. Base images have lower priorities than current images. Within an image file, the priorities are based on snapshot time. The priorities of more recent snapshots are higher, and the last writable part have the highest priority. Taking Fig. 5 as an example, the clusters labelled 0 and 3 of the base image are overlaid by the current image's cluster 0 and cluster 3 in the merged address space. And the cluster labelled 5 of the snapshot part is overlaid by the cluster in writable part.

As for the implementation, data clusters are mapped in the reverse order of priority, and the earlier mapped low priority address spaces will be covered by the latter mapped high priority address spaces.

Algorithm 1 Data cluster allocation procedure.

```
Input:
    virt_addr is the demanded PM virtual address aligned to cluster size.
    base_addr is the virtual address where the PM region begins.
Ensure:
    cluster_offset is the file offset of the newly allocated data cluster.
 1: function NEW_CLUSTER(virt_addr)
 2:     max_cluster_per_seg ← (cluster_size − 64)/8
 3:     if cluster_counter ≥ max_cluster_per_seg then
 4:         /* Allocate a meta cluster for the new segment. */
 5:         fallocate(pcow_fd, cluster_size)
 6:         init_new_meta()
 7:         cur_segment_num ← cur_segment_num + 1
 8:         clflush(&cur_segment_num, cacheline_size)
 9:         sfence() // Ensure the durability and order.
10:     end if
11:     /* Allocate a new data cluster. */
12:     fallocate(pcow_fd, cluster_size)
13:     cow_bitmap ← 0x00000000 // Initiate the new data entry's cow_bitmap.
14:     data_cluster_num ← (virt_addr − base_addr)/cluster_size
15:     clflush(&data_cluster_num, cacheline_size)
16:     sfence()
17:     cluster_counter ← cluster_counter + 1
18:     clflush(&cluster_counter, cacheline_size)
19:     sfence()
20:     cluster_off ← file_len + cluster_size
21:     return cluster_off
22: end function
```

Algorithm 2 Main procedure of the *Expansion handler*.

```
Input:
    virt_addr is the demanded PM virtual address aligned to cluster size.
1: function IMAGE_EXPANSION(virt_addr)
2:     new_cluster_off ← NEW_CLUSTER(virt_addr)
3:     /* Map the new allocated data cluster to the demanded address. */
4:     mmap(virt_addr, pcow_fd, new_cluster_off)
5: end function
```

Algorithm 3 Main procedure of the *Copy-on-write handler*.

```
Input:
    virt_addr is the accessed PM address under protection (aligned to cluster size).
1: function COPY_ON_WRITE(virt_addr)
2:     new_cluster_off ← NEW_CLUSTER(virt_addr)
3:     /* Map the new allocated data cluster to a temporary address. */
4:     tmp_addr ← mmap(virt_addr, pcow_fd, new_cluster_off)
5:     memcpy(tmp_addr, virt_addr, cluster_size)
6:     clflush(&tmp_addr, cluster_size)
7:     /* Remap the temporary buffer to the accessed protected address */
8:     remap(tmp_addr, virt_addr, cluster_size)
9: end function
```

C. Detailed Procedures at Runtime

As described in Section IV-A, when guest applications attempt to access the "fake" address regions without PM image backed, the host OS kernel will raise an EPT violation. And when guest applications try to write a page with write-protection, it will cause the host OS to generate a SIGSEGV signal which is handled by the *copy-on-write handler*. We now describe the detailed implementations. The main procedures of the *expansion handler* and *copy-on-write handler* in Fig. 3 are shown in Algorithm 2 and Algorithm 3 respectively, and the two procedures are both based on Algorithm 1, which is the data cluster allocation procedure.

Algorithm 1 shows how the *image monitor* creates a new data cluster. As indicated by Line 2, a meta cluster can manage `max_cluster_per_seg` data clusters. So in Line 3 to 10, a new meta cluster should be allocated first if the current segment is full. After extending the file by `fallocate` system call (Line 12), we also need to initialize the data entry in the active meta cluster (Line 13 to 19). Noted that the metadata update order should not be changed for crash consistency. Besides, metadata on PM should be flushed by `clflush` and `sfence` after each update. Considering the PM is slower than DRAM, for efficiency, we also cache some of the metadata in DRAM, which is omitted in Algorithm 1.

Algorithm 2 is implemented in the *expansion handler*, it will be executed when an user-space page fault (`userfaultfd`) event is polled. It calls the `NEW_CLUSTER` function defined in Algorithm 1, and map the new data cluster to the faulting address.

Algorithm 3 describes the procedure of copy-on-write, which is the key procedure of snapshot and base image features. This procedure resides in the *copy-on-write handler* and will be called when write-protected area is written and a SIGSEGV signal is raised. In this procedure, a new data cluster is first allocated and mapped to a temporary address (Line 2 to 4). Then read-only data will be copied to the new cluster (Line 5 to 6). Finally, the temporary address will be remapped to the address causing the SIGSEGV signal (Line 8). Note that, as applications in virtual machines may access the same protected memory areas concurrently, thus a signal may be raised as another signal of the same address is under copy-on-writing. As a protected area should only be copied once, we should do some checks to decide whether to handle or to ignore a signal. The checking process is implemented based on a hash table in DRAM, which introduces negligible overhead and is not listed in Algorithm 3 for conciseness.

D. Optimization for PM

a) Pre-allocation: As the two key procedures (image expansion and copy-on-write) of the enhanced memory virtualization are both based on cluster allocation, Algorithm 1 is a critical part of overall performance. To optimize the performance, we use a dedicate cluster allocation thread, several clusters are allocated in the background before the expansion is required. Using pre-allocation, only Line 13 to

16 of Algorithm 1 will be executed when images need to be expanded.

b) Fine-grained Copy-on-write: As data is organized by fixed cluster size in *pcow* images, in general, copy-on-write should be done in the granularity of data clusters, which is shown in Algorithm 3 Line 5 to 6. However, as many applications take advantage of PM's byte-addressability [13] [23] [24] , the accessed area on PM is usually small. Therefore we propose to use finer-grained copy-on-write instead of copying the entire cluster at once. We only copy 4 KB (the memory page size) where the SIGSEGV signal occurs.

This optimization implies that the copy-on-write granularity and cluster size will not be identical. However, we believe larger cluster size is still necessary, because: 1) Cluster size matters more to the cluster allocation, and applications usually allocate a related large continuous PM region and then access in tiny granularity. Smaller cluster size will introduce more allocation overhead. 2) Smaller cluster size will cause more virtual memory areas maintained in OS, which increases the DRAM usage and slow down the page table construction process. 3) Smaller cluster will also slow down the startup process, as more meta clusters should be scanned and more data clusters should be mapped.

For slow devices like HDDs or low-end SSDs, a 4 KB copy usually not worth as the hardware access latency is far larger than the transfer time spend for a data cluster [25]. However, as PM has ultra-low latency, the transfer time will dominate the data copy process even for relatively small data size, which is verified in Section VI-D.

We maintained a 4 B `cow_bitmap` in each data entry as displayed in Table II. Each bit represents the 4 KB copied status of a cluster. As there are 32 bits for each cluster, *pcow* with fine-grained copy-on-write optimization supports the cluster size up to 128 KB. The Algorithm 3 is optimized as Algorithm 4. After the fine-grained 4 KB copy (Line 7 to Line 9) and before remapping (Line 15), the data cluster's `cow_bitmap` is updated (Line 11 to Line 13). The corresponding bit of the copied 4 KB unit will be set to 1 in the `cow_bitmap`. For efficiency, the metadata of clusters not fully copied are cached in DRAM, which is not listed in Algorithm 4.

Algorithm 4 *Fine-grained Copy-on-write handler .*

Input:
`virt_addr_4k` is the accessed PM address under protection (aligned to 4 KB).
`loc_in_cluster` is the location number of the 4 KB unit in its cluster.

```
1: function FINEGRAINED_COW(virt_addr_4k)
2:     cluster_off ← CLUSTER_PARTIAL_COPIED(virt_addr_4k)
3:     if cluster_off = NULL then
4:         cluster_off ← NEW_CLUSTER(virt_addr_4k)
5:     end if
6:     /* Map the new allocated data cluster to a temporary address. */
7:     tmp_addr ← mmap(virt_addr_4k, pcow_fd, cluster_off)
8:     memcpy(tmp_addr, virt_addr_4k, 4096)
9:     clflush(&tmp_addr, 4096)
10:    /* Update the copy-on-write bitmap of the data entry. */
11:    cow_bitmap |= 1 << loc_in_cluster
12:    clflush(&cow_bitmap, cacheline_size)
13:    sfence()
14:    /* Remap the temporary buffer to the accessed protected address */
15:    remap(tmp_addr, virt_addr_4k, 4096)
16: end function
```

VI. Evaluation

A. Experimental Setup

In this section, we evaluate the prototype we implemented in QEMU 3.0. All experiments are conducted on an x86 server, whose detailed configuration is listed in Table III. As PM devices are not broadly available, we reserve 48 GB DRAM of the server as an emulated PM partition [26]. The PM partition is formatted as an ext4 file system and mounted with the DAX option.

TABLE III: Server Configurations.

CPU	Intel Xeon E5-2609 1.70 GHz ×2
CPU cores	8 ×2
Processor cache	32 KB L1i, 32 KB L1d, 256 KB L2, 20 MB L3
DRAM	80 GB
PM	48 GB (emulated by DRAM)
OS	CentOS 7.0, kernel version 4.16.0 (same in VMs)
VMM	QEMU version 3.0.0
File system	ext4 (mounted with DAX option)

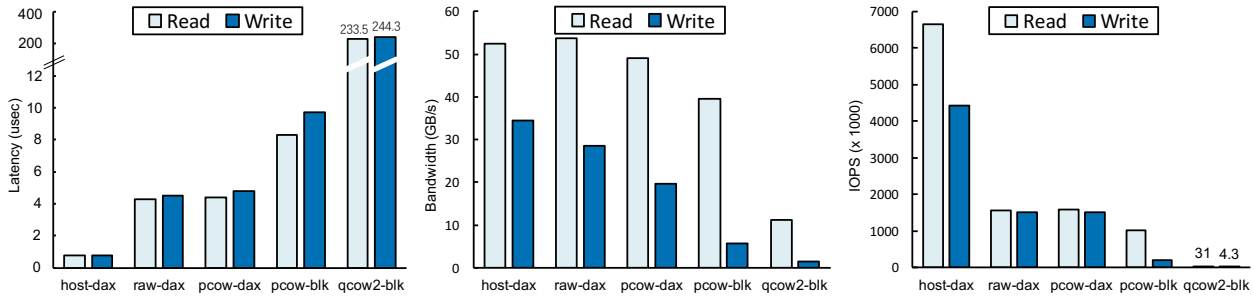
We compare our scheme ("PM–PM" form, denoted as *pcow*) with several baseline schemes, including a block device backed by a qcow2 image created on the PM partition ("block device–PM" form, denoted as *qcow2*) and a native virtual PM region without any storage virtualization feature ("PM–PM" form, denoted as *raw*). Some of the experiments are also performed directly on the host server's PM partition (denoted as *host*).

On the host, images of *pcow*, *qcow2* and *raw* are all created on the PM partition. The cluster size of *pcow* and *qcow2* are set to 64 KB. In guest, PM regions virtualized by *pcow* and *raw* are also formatted by ext4 with DAX. As the qcow2 images serve as block devices in guest, they are mounted as ext4 without the DAX option.

B. Micro-benchmarks

We make some micro-benchmarks using `fio` [27] to understand the basic performance metrics of the proposed scheme. `Fio`'s mmap engine uses `mmap/memcpy` interfaces, which can take advantage of the DAX feature to bypass OS's page cache and achieve the best performance of PM. Thus, we use the mmap engine for *pcow*, *raw* and *host*. As *qcow2* does not support DAX, we use pvsync engine and set the `direct` option to eliminate the impacts of page cache. To understand the difference introduced by pvsync engine, we also test *pcow* using pvsync engine. In Fig. 6, we use the *-dax* and *-blk* suffixes to denote mmap engine and pvsync engine respectively. All test data are generated and accessed randomly by `fio`, each data set is 20 GB, and the results are averaged over 30s executions.

a) Latency: We run `fio` in a single-threaded mode and small 4 KB random requests are submitted one at a time. The test results are displayed in Fig. 6 (a). The latency of the native memory virtualization (*raw-dax*) increases about 4 μs compared to the host (*host-dax*) . Our scheme (*pcow-dax*) brings almost no overhead compared with *raw-dax*. As expected, *qcow2-blk*, as is implemented in I/O virtualization, is 50 times slower than memory virtualization. Even the latency of *pcow-blk* is 20 times better than *qcow2-blk*.



(a) Latency (lower is better). (b) Bandwidth (higher is better). (c) IOPS (higher is better).

Fig. 6: Micro-benchmarks. The suffixes *dax* and *blk* indicate respectively the mmap engine and pvsync engine of `fio`.

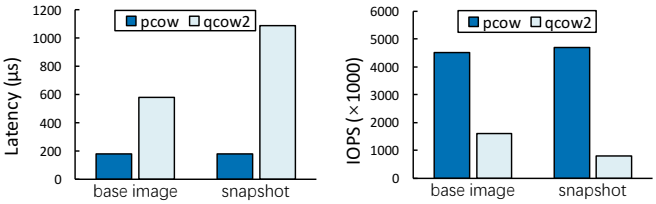
b) Bandwidth: The bandwidth is measured using 16 `fio` threads and the request size is 1 MB. As shown in Fig. 6 (b), *pcow-dax* achieves 91% and 69% of the *host-dax*'s read and write bandwidth respectively, while *qcow2-blk* only achieves 21% and 4.4% respectively. Even *pcow-blk* is 4 times better than *qcow2-blk*.

c) IOPS: The IOPS (short for I/O operations per second) is measured using 16 `fio` threads and 4 KB request units. Although IOPS of *host-dax* is over 6000k and 4000k in read and write test respectively, which are about 5 times better than *raw-dax*, our scheme does not bring additional overhead compared with *raw-dax*. And the read and write IOPS of *pcow-dax* are still 50 times and 350 times better than those of *qcow2-blk* respectively. Write performance drops sharply when *pcow-blk* is used, which indicates DAX is important for PM.

Through these measurements, we can see that although memory virtualization (*raw* and *pcow*) results in some degradation penalty compared with *host*, I/O virtualization (*qcow2*) severely drops the performance by hundreds of times. In addition, while PM virtualized *qcow2* only supports block device interfaces, our scheme can take full advantages of PM's byte-addressability. In summary, by enhancing the native memory virtualization, our scheme provides the storage virtualization features with almost no performance loss.

C. Copy-on-write Performance

The mirco-benchmarks mentioned above are conducted without copy-on-write. As copy-on-write is the key procedure of many storage virtualization features (such as *pcow*'s snapshot feature and base image feature), we further compare the performance with copy-on-write involved between *pcow* and *qcow2*. To produce copy-on-write requests, we first generate 20 GB test data in one image, and then create another image backed by it (denoted as *base image*) or take a snapshot on it (denoted as *snapshot*). We use `fio` to overwrite on the 20 GB read-only data. Both copy-on-write latency and copy-on-write IOPS are tested. The test results are shown in Fig. 7.



(a) Copy-on-write latency. (b) Copy-on-write IOPS.

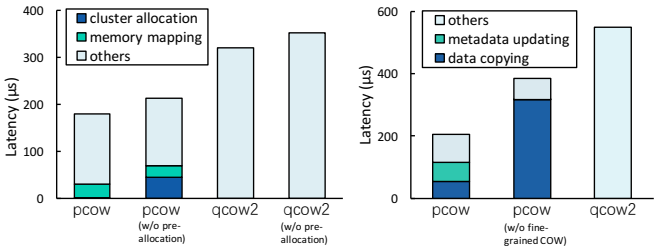
Fig. 7: Copy-on-write performance of *pcow* and qcow2.

a) Latency: The copy-on-write latency is measured using a single `fio` thread and 4 KB write requests. In Fig. 7 (a), *pcow*'s latency is about 200 μs in both *base image* and *snapshot*, while the results of *qcow2* is about 600 μs and 1100 μs. Our scheme is 3 to 5 times better than the qcow2 format.

b) IOPS: In Fig. 7 (b), IOPS with copy-on-write is also tested using 16 random access `fio` threads, and *pcow* is also 3 to 5 times better than *qcow2*.

D. Optimization Decomposition

As described in Section V-D, we optimize the image expansion and copy-on-write procedure by pre-allocation and finer-grained copy-on-write. To understand the performance improvement, we test the latency breakdowns with the optimization enabled or disabled.



(a) Pre-allocation (b) Fine-grained Copy-on-write

Fig. 8: Optimization Decomposition.

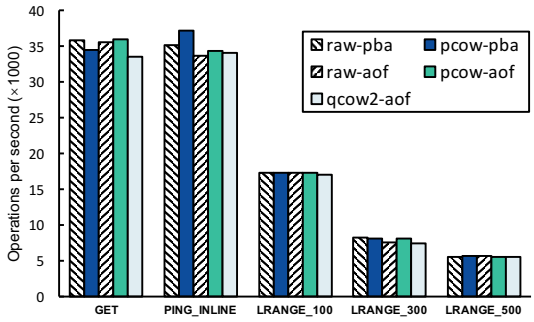
a) Pre-allocation: To measure the image expansion latency, we append a file on virtual PM regions using `fio` and measured the latency breakdown of *pcow*. As the qcow2 format also support this optimization, we also test the *qcow2* with or without pre-allocation. As shown in Fig. 8 (a), with pre-allocation, the *pcow*'s cluster allocation latency is almost eliminated (from 45 μs to 1 μs), and the total latency is reduced by 34 μs. The *qcow2*'s expansion latency is reduced

by 31 μs with pre-allocation. The overall expansion latency of *pcow* is 140 μs lower than *qcow2*.

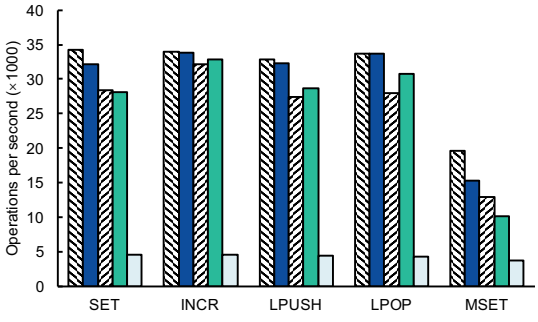
b) Fine-grained Copy-on-write: As shown in Fig. 8 (b), *pcow*'s copy-on-write latency without fine-grained optimization is 385 μs. With fine-grained copy-on-write, although additional metadata updating latency (61 μs) is introduced, the overall copy-on-write time is still decreased by 179 μs. With fine-grained optimization, *pcow*'s latency is 3 times lower than *qcow2*.

E. Real-world Workloads

We further measure the performance of real-world key-value workloads in *raw*, *pcow* and *qcow2* schemes. We select Redis [28], a widely used key-value store, to evaluate the performance. The Redis version we use is optimized for PM (denoted as Redis-PMDK) [29] based on the Persistent Memory Development Kit (PMDK) [30].



(a) Redis query performance.



(b) Redis update performance.

Fig. 9: Redis performance. The suffixes *pba* and *aof* indicate respectively the pointer-based AOF (PBA) mode optimized for PM and the native AOF mode.

Data persistence of native Redis is implemented as Append Only File (AOF), which flushes the changed data to an AOF file. Redis-PMDK implements pointer-based AOF (PBA), which improves the performance by utilizing PM's byte-addressability. We perform evaluation in AOF mode and PBA mode.

We use `Redis-benckmark` [31] to compare the performance between *pcow*, *raw* and *qcow2*. For *raw* and *pcow*, we record the operation per second (OPS) in both PBA and AOF mode. For *qcow2*, as PBA is not supported, we only record its OPS performance in AOF mode. The suffixes *-pba* and *-aof* indicate PBA mode and AOF mode respectively. We test two types of workloads, one only involves query (GET, PING_INLINE, LRANGE_100, LRANGE_300 and LRANGE_500), and another involves update (SET, INCR, LPUSH, LPOP and MSET). For each workload, 100000 requests are submitted by 10 clients concurrently.

Fig. 9 (a) shows the OPSs of query workloads. The OPSs are almost the same, because Redis is an in-memory key-value storage system, and all query workloads are practically handled in DRAM [28]. The test results of update workloads are displayed in Fig. 9 (b). The OPS performance between *pcow* and *raw* are equally matched. The OPS of *pcow-pba* is better than that of *pcow-aof*, and achieves 94%, 99%, 98%, 101% and 78% of *raw-pba* in SET, INCR, LPUSH, LPOP and MSET workloads respectively. The OPS of *pcow* is much higher than *qcow2*, as the OPS of *qcow2-aof* only achieves 16%, 14%, 16%, 14% and 36% of *pcow-aof* in SET, INCR, LPUSH, LPOP and MSET workloads respectively.

The test results indicate that, with abundant storage virtualization features implemented, our scheme is still compatible with the real-world application's optimization for PM in virtual machines, and the write performance is much better than the qcow2 format.

VII. Related Work

PM virtualization: Modern VMMs are adding support for persistent memory virtualization. QEMU starts to support the use of virtual PM since version 2.6.0 by simply leveraging file-backed memory mapping and current memory virtualization mechanisms [1]. Recently, VMware released vSphere v6.7, which starts to support PM device in both persistent memory mode and block device mode [2]. However, both QEMU and vSphere cannot maintain storage virtualization features and PM's byte-addressability at the same time.

Liang et al. [32] proposed a PM virtualization solution named VPM. VPM provides full-virtualization and para-virtualization interfaces for guest system by extending the VMM. However, VPM mainly focuses on the performance improvement and cost-efficiency of using PM in virtual machines but neglects PM image management.

Compared with existing work, this paper focuses on providing an effective management for virtual PM while maintaining PM performance and byte-addressability.

Image format optimization: Modern virtual machine image formats provide convenience for storage resource management, but degrade the performance due to their complex metadata. There is a lot of work aiming to optimize the image formats. Chen et al. [18] mitigated the sync amplification caused by qcow2 by journaling and proper prealloction. Tang [22] designed a new high-profermance virtual image format, namely FVD, which introduces new features like copy-on-read and prefetching. However, all these solutions are committed to optimizing traditional I/O virtualization. The proposed *pcow* format in this paper is designed for PM images and is optimized considering PM's characteristics.

Others: Linux is also supporting storage virtualization features for PM in its device-mapper layer [33]. However, only linear target and striped target support the DAX feature at present. Our scheme is implemented in VMMs, which is orthogonal to device-mapper and provides more features.

There are also many existing works provide virtualization for flash storage [34] [35]. Our work dedicates to enhance the native memory virtualization and bring storage virtualization features for virtual PM rather than flash-based block devices.

VIII. Conclusions

As persistent memory is blurring the boundary between memory and storage, we in this paper bring several storage virtualization features into virtual machine's memory virtualization. The implemented prototype provide snapshot, base image and thin-provision features for PM images with their byte-addressability and ultra-low latency maintained.

The source code of this work is released on GitHub [1]. In the future, we aim to develop more storage virtualization features for PM, such as data deduplication and encryption.

IX. Acknowledgments

This work is partially supported by National Science Foundation of China (61872201, 61702521, 61602266, U1833114); Science and Technology Development Plan of Tianjin (17JCYBJC15300, 16JCYBJC41900, 18ZXZNGX00140, 18ZXZNGX00200); and the Fundamental Research Funds for the Central Universities and SAFEA: Overseas Young Talents in Cultural and Educational Sector, TKLNDST.

References

[1] "The memory API, QEMU Documentation." https://github.com/qemu/qemu/blob/master/docs/devel/memory.rst. [Online; accessed 20-Apr-2019].

[2] "PMem (Persistant Memory) NVDIMM support in vSphere." https://storagehub.vmware.com/t/vsphere-storage/vsphere-6-7-core-storage-1/pmem-persistant-memory-nvdimm-support-in-vsphere/. [Online; accessed 20-Apr-2019].

[3] R. McDougall and J. Anderson, "Virtualization performance: perspectives and challenges ahead," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 40–56, 2010.

[4] "Qcow2 Image Format, QEMU Documentation." https://github.com/qemu/qemu/blob/master/docs/interop/qcow2.txt. [Online; accessed 20-Apr-2019].

[5] "Virtual Disk Format 5.0." https://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf. [Online; accessed 20-Apr-2019].

[6] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[7] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[8] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, IEEE, 2015.

[9] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, *et al.*, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, p. 13, 2013.

[10] "Intel Optane DC Persistent Memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. [Online; accessed 20-Apr-2019].

[11] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: Durable hardware transactional memory," in *Proceedings of the International Symposium on Computer Architecture*, 2018.

[12] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, 2015.

[13] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in *16th USENIX Conference on File and Storage Technologies (FAST)*, p. 187, 2018.

[14] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash consistency," *Queue*, vol. 13, no. 7, p. 20, 2015.

[15] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *32nd International Conference on Computer Design (ICCD)*, pp. 216–223, IEEE, 2014.

[16] SNIA NVM Programming Technical Working Group, "NVM Programming Model (Version 1.2)," 2017.

[17] J. Xu, W. Zhang, Z. Zhang, T. Wang, and T. Huang, "Clustering-based acceleration for virtual machine image deduplication in the cloud environment," *Journal of Systems and Software*, vol. 121, pp. 144–156, 2016.

[18] Q. Chen, L. Liang, Y. Xia, H. Chen, and H. Kim, "Mitigating sync amplification for copy-on-write virtual disk.," in *14th USENIX Conference on File and Storage Technologies (FAST)*, pp. 241–247, 2016.

[19] "BTT - Block Translation Table, Linux Kernel Documentation." https://www.kernel.org/doc/Documentation/nvdimm/btt.txt. [Online; accessed 20-Apr-2019].

[20] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: a NVMe storage virtualization solution with mediated pass-through," in *USENIX Annual Technical Conference (ATC)*, pp. 665–676, 2018.

[21] T. Merrifield and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 25–35, 2016.

[22] C. Tang, "FVD: a high-performance virtual machine image format for cloud," in *USENIX Annual Technical Conference (ATC)*, pp. 16–24, 2011.

[23] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *USENIX Annual Technical Conference (ATC)*, pp. 993–1005, 2018.

[24] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *33rd Symposium on Mass Storage Systems and Technologies (MSST)*, 2017.

[25] J. Zhang, P. Li, B. Liu, T. G. Marbach, X. Liu, and G. Wang, "Performance analysis of 3D XPoint SSDs in virtualized and non-virtualized environments," in *24th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 51–60, IEEE, 2018.

[26] "How to emulate Persistent Memory." https://pmem.io/2016/02/22/pm-emulation.html. [Online; accessed 20-Apr-2019].

[27] "Flexible I/O Tester." https://github.com/axboe/fio. [Online; accessed 20-Apr-2019].

[28] "redis." https://redis.io/. [Online; accessed 20-Apr-2019].

[29] "pmem-redis." https://github.com/pmem/pmem-redis. [Online; accessed 20-Apr-2019].

[30] "Persistent Memory Development Kit." https://pmem.io/pmdk/. [Online; accessed 20-Apr-2019].

[31] "redis-benchmark." https://redis.io/topics/benchmarks. [Online; accessed 20-Apr-2019].

[32] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *SOCC*, pp. 126–140, 2016.

[33] "Using Persistent Memory Devices with the Linux Device Mapper." https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html. [Online; accessed 20-Apr-2019].

[34] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ANViL: Advanced virtualization for modern non-volatile memory devices.," in *13rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 111–118, 2015.

[35] X. Song, J. Yang, and H. Chen, "Architecting flash-based solid-state drive for high-performance I/O virtualization," *Computer Architecture Letters*, vol. 13, no. 2, pp. 61–64, 2014.

[1] https://github.com/zhangjaycee/qemu-pcow