A Case for Virtualizing Persistent Memory

Liang Liang, Rong Chen, Haibo Chen, Yubin Xia, †KwanJong Park, Binyu Zang, Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
†Samsung Electronics
Contact: haibochen@sjtu.edu.cn

Abstract

With the proliferation of software and hardware support for persistent memory (PM) like PCM and NV-DIMM, we envision that PM will soon become a standard component of commodity cloud, especially for those applications demanding high performance and low latency. Yet, current virtualization software lacks support to efficiently virtualize and manage PM to improve cost-effectiveness, performance, and endurance.

In this paper, we make the first case study on extending commodity hypervisors to virtualize PM. We explore design spaces to abstract PM, including load/store accessible guest-physical memory and a block device. We design and implement a system, namely VPM, which provides both full-virtualization as well as a para-virtualization interface that provide persistence hints to the hypervisor. By leveraging the fact that PM has similar characteristics with DRAM except for persistence, VPM supports transparent data migration by leveraging the two-dimensional paging (e.g., EPT) to adjust the mapping between guest PM to host physical memory (DRAM or PM). Finally, VPM provides efficient crash recovery by properly bookkeeping guest PM's states as well as key hypervisor-based states into PM in an epoch-based consistency approach. Experimental results with VPM implemented on KVM and Linux using simulated PCM and NVDIMM show that VPM achieves a proportional consolidation of PM with graceful degradation of performance. Our para-virtualized interface further improves the consolidation ratio with less overhead for some workloads.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Virtual memory

General Terms Design, Performance

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987551

Keywords Persistent Memory, Virtualization, Para-virtualization

1. Introduction

Computer systems have long faced a tension between performance and persistence, which poses a challenge to place data in volatile and non-volatile storage to ensure proper crash recovery under power failures. Persistent memory (PM), either in the form of battery-backed memory cards (e.g., NVDIMM) or emerging non-volatile memory like PCM [47], Memristor [65], STT-MRAM [3], is promising to relax this tension by providing high-performance recoverable systems.

Because of its intriguing promise, PM has gained significant interests from both hardware and software sides. Major memory vendors like Micron, Viking, JEDEC, Fusion-IO have provided commercial NVDIMM to the mass market. SNIA has also recently formed the NVDIMM Special Interest Group [49] to accelerate the adoption of NVDIMM. For emerging non-volatile memory, Micron has announced its 1Gb PCM available to volume production [38] in 2012. In response of this, researchers have investigated a number of ways of utilizing PM [10, 14, 20, 31, 53, 62, 64, 68]. Linux and Windows developers have started to provide support for PM-aware file systems [15, 16].

With the proliferation of hardware and software systems for PM, as well as the emerging in-memory computing instances for big data from major cloud providers [2], we envision that PM will soon be an indispensable part of cloud computing systems. Actually, recent industry talks [1, 45] indicated that cloud will likely incorporate non-volatile memory like NVDIMM to increase the services reliability, scalability, and availability. However, commodity hypervisors still lack support to virtualize and efficiently manage PM in a cloud environment.

In this paper, we make a case study on how to efficiently virtualize PM to improve cost-effectiveness and performance, which are important for several reasons. First, as new technologies, PM is still with a relatively high price. For example, currently NV-DIMM's price is at least 8-10X higher than DRAM according to the price quote from NV-DIMM providers; efficiently virtualizing and consolidating

PM may lead to significant cost-effectiveness. Second, some emerging memory technologies like PCM or Memristor still have inferior write performance than DRAM, virtualizing them may lead to performance improvement. By virtualizing PM, we provide VM with an abstraction of sufficient amount of PM, through emulating it with multiple types of media, without any relax to the persistency that PM offers.

There are several technical challenges in providing efficient virtualization for PM and VPM addresses them in a framework called VPM. First, current virtualization lacks support for an appropriate abstraction for PM, which may present as a block device or directly as a piece of memory in guest virtual machines (VMs). In this paper, we explore the design spaces of providing appropriate abstractions as well as support from hypervisors to different forms of PM. Specifically, VPM virtualizes PM as a block device and a special piece of memory and provides a few para-virtualization interfaces to bridge the semantic gap of persistency between guest VMs and the hypervisor.

Second, as PM may exhibit different performance and price characteristics from DRAM, it is critical to consolidate PM to maximize cost-efficiency, performance, and endurance. As a PM has similar characteristics (except some performance gap between DRAM and some emerging memory like PCM) with DRAM when serving read accesses from guest VM, it has no impact on persistence whether data is placed in DRAM or PM. To this end, VPM leverages DRAM to place read-mostly data in a transparent way by leveraging the two-dimensional paging (e.g., extended page table (EPT)) to dynamically migrating data between DRAM and PM.

To make an optimal decision on data placement, VPM needs to efficiently track PM accesses to predict future accesses. VPM leverages the newly introduced dirty bit in the hypervisor-managed page table (e.g., EPT) to transparently tracking writing working set of guest VMs.

Finally, crash recovery is a challenge for VPM under a hybrid memory architecture. To ensure proper and efficient crash recovery under a power failure, VPM only needs to preserve key data structures in the hypervisor to be consistent but also need to efficiently bookkeep virtualized PMs for proper crash recovery. VPM address this challenge by making the part of hypervisors leverage PM to store its key data structures for crash recovery and using a portion of PM to bookkeep guest VM's PM structures.

We have built a working prototype based on KVM, by simulating PM using DRAM and treating PM as a portion of the physical address space in the hypervisor. To measure the performance of VPM, we run a combination of workloads atop VMs. Evaluation results show that VPM can reduce the using of PM by a factor of 50% while incurring only 20% performance overhead for most tested workloads. The para-virtualization interface further improves PM consolidation rate by allowing more workloads to be executed concurrently with low performance degradation.

This paper makes the following contributions:

- A case for virtualizing PM (§3) and a set of ways to virtualize PMs (§4).
- A set of techniques to efficiently consolidate PM for better cost-efficiency and performance (§5).
- An implementation on KVM (§6) set of evaluations that confirms the effectiveness of VPM (§7).

2. Background and Motivation

2.1 Persistent Memory

Persistent memory (PM), also known as non-volatile memory (NVRAM), embraces the high performance byte addressability of DRAM and the persistency of disk and flash devices. They can be attached to a DRAM-like memory bus and thus can be accessed through *load* and *store* instructions [23]. Currently, there are several technologies providing PM potentially as a main-memory replacement: PCM (Phase-change Memory) [29], STT-MRAM (Spin-Transfer-Torque MRAM) [27], battery-backed DRAM (i.e., NVDIMM) [52], and memristors [51]. With the advances of semiconductor techniques, PM is now commercially available with sizes of 8GB [52]. In this paper, we do not consider flash-based storage (e.g., solid state disk) since they are mostly accessed as block device through I/O bus.

While it is still unclear which emerging technology would likely be a replacement of DRAM, we envision that there will be a long period where DRAM will co-exist with emerging PM, for the sake of performance, cost, and endurance. A co-existence of DRAM with PM would lead to heterogeneity in performance and cost-effectiveness. For example, NVDIMM is with 8-10X price compared to DRAM. Other emerging PM would still have a relatively higher price until massive production and wide adoption. Besides, PM tends to have two types: symmetric read/write performance or asymmetric read/write performance. Finally, some PM devices have endurance issues such that they have a limited lifetime. In this paper, we mainly consider two types of PM: 1) PM with symmetric read/write performance but higher price; 2) PM with asymmetric read/write performance. Specifically, we consider PCM and NVDIMM as their representatives.

PM in Use: PM is now being used in three categories. First, as many applications and file systems are still not PM aware, PM is used through the block device interface (e..g, PMBD [28]). While this way can make legacy applications benefit from PM, such a usage does not fully exploit PM: 1) the byte addressability is not fully exploited; 2) a large portion of PM is used for read but not write, which just treat PM as disk cache.

The second way is building a new PM-aware file system. Examples include BPFS [15], SCMFS [62], PMFS [20], and Aerie [56]. In this case, PM is accessed through the memory interface by file systems, which further provide a block interface to applications. The layout of data structures of the file systems is specially optimized to leverage PM's characteristics. Such a usage provides better usage of PM in the file system layer, but a lot of application data is still not PM optimized.

The third one is building PM-optimized applications, which can directly access PM through load/store instructions. Examples include Quill [21], Moneta-D [10], Mnemosyne [55], NV-Heaps [14], NV-Tree [64] and a lot of database logs [11, 22, 43, 54].

Such three usages PM are suitable for different use cases, according to which components can be modified or not. Thus, they may coexist for a relatively long time. Varies usages of PM lead to different access patterns: some are read-mostly; some are write-intensive; some are at byte-level while others are at block-level. The diversity will be further amplified in virtualized environments, where a number of diverse workloads are consolidated.

2.2 The Need of PM Virtualization

VPM is motivated by two important trends in the big data era. First, the intriguing features like disk-like persistency and memory-like fast byte-addressability make PM very promising in many big-data applications demanding low-latency and high-throughput, where crash recovery is a hard but an indispensable issue. This has been evidenced by a number of systems designed around PM like OLTP, file systems and concurrent data structures. Second, (virtualized) cloud has been a standard computing platform to run many big-data applications, which can be evidenced by many big-memory clouds from major cloud vendors like Amazon EC2, Windows Azure, and RackSpace. With the two trends, we envision that major cloud platforms will soon provision PM in their cloud for performance and reliability.

While a hypervisor can trivially support PM by treating it as normal DRAM or a hypervisor can directly expose PM to guest VMs, these would lose several key benefits, which are the key motivation of VPM:

Virtualization for cost-effectiveness: One sole purpose of virtualization is server consolidation, where a hypervisor can host a large number of guest VMs each with different uses of PMs. Statically and directly provisioning physical PM to guest VMs are not only inflexible but also not cost-effectiveness. The latter is especially for NVDIMM, whose price is at least 8-10X than DRAM. By adding an indirection between the VMs and the PM, the hypervisor can provide more flexible resource management to maximize the utilization of PM according to their characteristics. For example, for those read-intensive workloads running on PM, a hypervisor can use DRAM or SSD to emulate PM to release more physical PM to other VMs, while still retaining performance and persistence of PM. In this way, PM can be more efficiently used while reducing the vendor's cost.

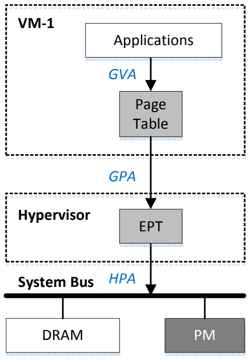


Fig. 1: Address translation in virtualization

Virtualization for performance: The additional indirection added by virtualization may also help bridge the variation of performance characteristics between DRAM and emerging PM like PCM or memristor. For example, the hypervisor may leverage DRAM to transparently serve some read-mostly workloads for PM with inferior read performance. Besides, the hypervisor may also leverage scheduling information to mitigate the NUMA-induced memory latency of PMs.

Virtualization for ease of management: From the tenants' perspective, the PM virtualization provided by the hypervisor can greatly ease the applications' management of PM. For example, a tenant can simply request a large (pseudo-) PM in the VM and use it as a block device to speed up unmodified database applications, while leaving the entire PM management to the underlying hypervisor. For other tenants that require fine-grained control over PM, they can request the hypervisor to expose PM as directly-attached memory while with little intervention from the hypervisor.

2.3 Background on Virtualization

Hardware-assisted virtualization has been a standard feature in both desktop and server platforms. On x86 processors there are two modes named "host mode" and "guest mode", the former is used to run the hypervisor and the latter is for guest VMs. Once some privileged operations, e.g., I/O instruction or system configuration, executes in the "guest mode", a trap occurs and the hypervisor takes over to handle the trap, which is also known as a "VMEXIT". The hypervisor then resumes the guest VM's execution after handling the VMEXIT. Such a process is also known as "trap-and-emulate" and is the essential part of CPU virtualization.

To virtualize memory, hardware vendors introduce a new type of page table, named "Extended Page Table" (EPT) by Intel or "Nested Page Table" (NPT) by AMD , which is uniformly denoted as *nPT* in this paper. As a page table in guest OS (mentioned as *gPT*) transfers guest virtual address (gVA) to guest physical address (gPA), the nPT transfers guest physical address (gPA) to host physical address (hPA) (Fig. 1). Each guest VM has its own nPT which is managed by the hypervisor. The second-level address mapping enables a continuous memory space of gPA built on discontinuous hPA memory, which eases the memory management of hypervisor and is hidden to the guest VM.

I/O device virtualization is done by intercepting all accesses to device registers or memory-mapped I/O through trap-and-emulate so that the hypervisor can get interposition between the guest VM and physical devices.

3. PM Virtualization

Similar as the traditional server virtualization, there are two typical ways to virtualize PM: 1) Full-virtualization, which provides functionally-identical virtualized PM to guest VM; 2) Para-virtualization, which abstracts a similar but not completely identical interface. The two ways share similar characteristics with server virtualization: the former provides transparency, but the latter may provide better performance.

Besides, as current PM like NVDIMM or PCM can be exposed to software as either a memory device or a storage device, we need to consider both cases in the design of VPM. Typically, the former one can provide better performance due to being load/store accessible and thus bypasses unnecessary interposition and formatting from systems software. The latter is mainly designed to enable existing applications to benefit from PM without notable changes.

3.1 Full-virtualization of PM

Memory interface: For PM like PCM and MRAM that attached to the memory bus, the interface is similar to DRAM that is accessed through load/store instructions. A guest VM retrieves PM information (such as caching and address space) from BIOS and then manages the PM using virtual memory hardware.

The memory interface is virtualized through *namespace* virtualization. A namespace, including address range and caching modes (e.g., write back, write through, or write combining), is a continuous address space isolated from each other. It is delivered by the virtual BIOS to the guest OS during the booting process. Specifically, Intel x86 provides a set of memory type range registers, which can be leveraged by the hypervisor to virtualize the namespace. In the hypervisor, VPM can use nested paging to map the guest PM namespaces to host PM namespaces. Thanks to the another indirection provided by virtualizing PM, the size of guest PM can be smaller or larger than the physical PM size.

Block interface: Some PMs, such as NVDIMM, also supports block interfaces to provide an evolutionary path to adopt PM. VPM can virtualize them through configuring the virtual BIOS so as to enumerate PM devices to the guest VM. This includes configuring PM-related control registers to expose the interfaces to the guest PM driver. VPM emulates the control registers of PM using a "trap-and-emulate" mechanism: when the control register is accessed, the CPU will trap to the hypervisor to emulate the access. However, as the hypervisor manage the PM through the memory interface, the upon virtual block interface is already implemented using the PM's memory interface. The hypervisor first maps the entire PM to its own memory space, and reads or writes the PM according to the commands issued by the guest VMs.

To copy data between DRAM and PM through the block interface, the guest PM driver needs to configure the PM command registers to indicate the data region information and let the hypervisor do the copy. The process is more like DMA operations. VPM completely leverages MMU instead of I/O MMU to isolate PM devices in the hypervisor.

3.2 Para-virtualization of PM

While full-virtualization can retain transparency to guest VMs, the lack of guest semantics information may lead to suboptimal performance and cost-efficiency. To this end, we also provide a para-virtualization interface, which aims to enable the guest VM to provide some hints to the hypervisor.

For emerging PM like PCM, the asymmetry read/write and limited endurance are key limitations, which may be further exacerbated due to the additional virtualization layer (analogous to *sync amplification* for I/O virtualization). While both Xen and KVM are equipped with a para-virtualized memory interface called transcendent memory [36]. We found that it is not suitable for PM for several reasons:1) it is not designed with persistency awareness; 2) it uses a slow, long data path using a frontend/backend design with several interim buffers, which are not suitable for fast PM devices and may even amplify the write endurance issues.

Instead, VPM provides a slightly modified PM interface by allowing guest VMs to access virtualized PM in a manner which both preserves the semantics that of the native one and enables a relatively high performance. Table 1 lists those few APIs, which under certain circumstances, will be translated into hypercalls. For example, the *vpm_persist* is used to notify the hypervisor that a specific range of memory requires to be persisted to the PM. The *vpm_barrier* is a virtualized call of *pcommit* from Intel or *pm_barrier* [20], which informs VPM to wait until the prior persist requests to complete. This essentially enables an epoch-based persistency [15, 44] to guest VM. The *vpm_lock* and *vpm_unlock* are introduced to achieve the mutual exclusive write to PM, which prevents a certain range of memory to be concurrently accessed by both the guest VM and the hypervisor.

3.3 Architecture of VPM

Fig. 2 shows an overview of the architecture of VPM. VPM can run multiple guest VMs, each of which is with different virtualized forms of PMs, e.g., para-virtualized device, full-virtualized persistent memory. Each VM can map at most the amount of PM equaling to the size of the virtualized PM seen by the VM. Inside the hypervisor, VPM uses the PM as the memory device attached to the memory bus for better performance. A physical PM page is allocated

Table 1: Para-virtualization interfaces

vpm_persist	notifies hypervisor that a specific range of guest PM is to be persisted
vpm_lock/unlock	lock & unlock the content of a specific range of guest PM to avoid it being modified by the hypervisor
vpm_barrier	persist all prior flushed guest PM to physical PM

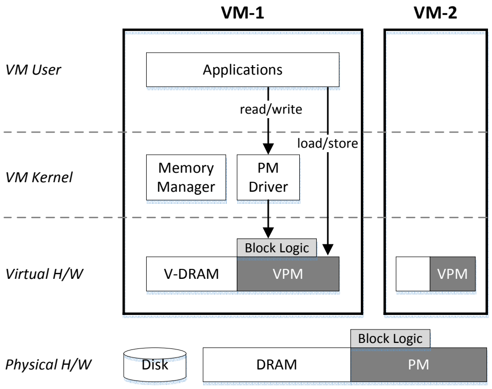


Fig. 2: Overall architecture of VPM

from a unified PM pool managed by VPM in the hypervisor. VPM intensively leverages the two-dimension address translation (e.g., nPT) to transparently mapping guest PM to host PM. According to different usages of guest PMs and the underlying PM characteristics, VPM may use different storage media to store guest PM data. The goal is to retain or even improve the performance (for PCM) while saving the usages of PM (for NVDIMM). This is done by leveraging guest provided hints and the hypervisor-based memory tracking (§ 5). However, VPM still preserves the persistency property of guest PM so that the guest PM data can be consistently recovered even in the face of arbitrary power failures (§ 4).

4. Virtualizing Persistence and Crash Recovery

One key aspect of VPM is ensuring persistency of virtualized PM, by guaranteeing that a virtualized PM enjoys the same persistence guarantee as its native counterpart. VPM achieves this by placing certain constraints over accesses of guest PM from VM layer as well as providing proper bookkeeping and crash recovery at the hypervisor layer. In the following, this section first describes necessary hardware support and assumptions and then describes how we extend the hypervisor for crash recovery.

4.1 Assumption and Hardware Support

As prior work, we assume that data stored in PM is made durable and consistent, but that stored in CPU cache is not. Besides, as there are typically some internal buffers inside a memory controller, a write operation to memory is usually buffered and in-flight reads by commit by directly reading the written value in the buffer. While this design is not an issue for DRAMs, it may lead to subtle comprise of persistence for PM due to the violation of ordering and persistence.

We address this issue by combining the newly provided instruction set by Intel and memory fencing instructions. Specifically, Intel has recently added an instruction called *PCOMMIT* (Persistent Commit). PCOMMIT serves as a persistent barrier, to enforce all memory stores flushed from CPU caches to either PM or some power-fail protected buffers. To guarantee proper memory ordering like the read-after-write reordering, VPM leverages memory fencing instructions in the hypervisor. Another feature VPM relies on is the atomic update of PM, at the granularity of 8-bytes, or 16-bytes with locking prefix or 64-bytes with transactional memory [20].

4.2 Persistency Model

To ensure persistency and consistency (i.e., memory persistency) in the hypervisor, one may either use a strict or a relaxed persistency model [44]. To provide better performance, VPM adopts a relaxed persistency model that allows reordering by using a combination of *PCOMMIT*, *clflush*[1] and *fencing* to only ensure a proper ordering and persistency when necessary. Specifically, VPM adopts an epoch-based persistency [15] by dividing execution related to PM as *epochs*. It then uses *clflush* to flush related memory stores to PM and then uses a *PCOMMIT* to force memory stores to PM to be persistent at the end of each epoch; Proper memory fences are added to preserve necessary ordering among memory accesses and force the completion of *PCOMMIT* to be visible globally.

VPM ensures persistence by preserving the invariant that, for any persistent state that can be identified by applications (either with instructions or predefined APIs), every guest PM page must have one persistent media (e.g., SSD or PM) to store its content. Fig. 3 illustrates an example of data resident types among DRAM, PM, and SSD: at any time, a guest PM page's content must be at least stored in either SSD or PM.

4.3 Persistency of Full-Virtualization PM

Full-virtualization of VPM (VPM-hv) ensures that all memory writes will be applied to native PM. To satisfy this constraint, VPM-hv heavily relies on the write protection of the two-dimension address translation mechanism. VPM-hv traps all writes to guest PM pages that are originally mapped with non-volatile storage and remaps the corresponding re-

[1] Note that, Intel provided several optimized *clfush* version like *CLFLUSHOPT* and *CLWB*. We use *clfush* to uniformly denote them.

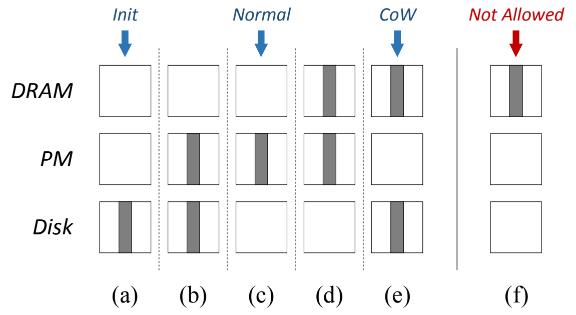


Fig. 3: Data Resident Types. (a) is the state at the very beginning. (b) shows data after being loaded from disk to the PM. (c) means the data has been modified so the only copy is now in PM. (d) stands for the temporary status when transferring from state-c to state-e. (e) is the state of PM emulation where the physical PM is revoked for other VM. (f) shows a state that will never exist since the data is only in DRAM which is vulnerable to power failure.

gion with native PM. The guest PM mapping is determined partially by the access pattern, which will be discussed in (§ 5.1).

4.4 Persistency of Para-Virtualization PM

The para-virtualization of VPM (VPM-pv), guest VM negotiates with VPM through the predefined APIs to guarantee the persistency of guest PM. As have introduced previously, *vpm_persist* is invoked to notify that a specific guest PM region is to be persisted while *vpm_barrier* acts as a sign that the previously issued *vpm_persist*s have been completed, and the corresponding data has been made non-volatile.

VPM-pv further relieves the constraints in VPM-hv and leaves more space for the VM or software in VM, that the modifications can be applied directly to non-volatile storage media; it is the applications' responsibility to maintain the persistency of data. Additionally, VPM-pv can also take the access pattern of a guest PM into considerations while making decisions on the PM mapping.

Optimization for Para-Virtualization PM Communicating with hypervisor presents a challenge for VPM-pv. Using hypercall for every VPM-pv APIs can incur unacceptable performance overhead for VM software since it is a high-frequency event. In VPM-pv, each VM communicates with hypervisor via one piece of shared memory. This piece of shared memory holds one ring queue and its corresponding metadata. On each *vpm_persist* call, a descriptor referring to a memory region that is to be persisted will be placed on the ring queue, which will be processed by a *persist thread* that runs on a dedicated core in the hypervisor by storing it in the non-volatile media.

To avoid additional memory copy, *vpm_persist* only records the pointer to the memory region. However, this leads to another problem that the *persist thread* might access the region that is to be persisted simultaneously with the VM. VPM-pv introduces *vpm_lock* and *vpm_unlock* to achieve the mutual exclusiveness between hypervisor and VM. Both entities shall proactively protect the region with the given APIs before making changes.

4.5 Supporting Crash Recovery

VPM is designed to guarantee that the key states regarding virtualizing PM in hypervisor shall be preserved across power failures or crashes. Correspondingly, operating systems in guest VMs should have its mechanism to recover data by leveraging states accessible from guest PM.

Apart from data stored in the guest PM, there are several key states in the hypervisor that require consistency and persistency. First, the additional indirection between guest PM to host PM, which does not exist in a vallina virtualized environment, should be recoverable after failure. This includes the nPT for PM, as well as hypervisor-provided reverse mapping from host PM addresses to guest PM addresses. Recovering nPT for PM allows VM to have an identical view of data on PM across a failure. Second, the memory tracking information, including the local and global history table, should be preserved to enable continuous access prediction and improve wear-leveling.

However, those states cannot be directly stored on PM. First, they are updated rather frequently (e.g., a CPU updates nPT status bits like access and dirty automatically on each memory access), using PM to store them might reduce the endurance for PCM-like PM. Second, there are still subtle ordering issues to ensure the consistency of such data structures.

VPM uses a separate data structure in PM to bookkeep the mapping from guest PM to host PM but uses DRAM to store nPT for accesses by CPU. It uses a write-ahead style logging to ensure consistent crash recovery: before updating the nPT, VPM firstly performs the corresponding page allocation and content migration, then updates its mapping structure stored in PM and lastly applies the updates to nPT.

The history information is persisted to PM periodically. The interval can be relatively long since it does not require high accuracy and is timing-independent. Losing some accuracy only affect performance but not persistency.

Hypervisor recovery: The hypervisor does not explicitly perform data recovery. Instead, on every startup of guest VMs, the hypervisor will automatically recover data either from disks or PM, according to the mapping information, stored in PM, which the PM nPT will also be populated accordingly.

5. PM Consolidation

With multiple VMs running different PM workloads, VPM is designed to leverage the access characteristics to consolidate virtual PMs into less physical PMs. The key for VPM to do this with small overhead is efficiently tracking and predicting memory accesses. Specifically, VPM needs to track memory accesses to determine not only which page is frequently referenced, but also which page is frequently updated. VPM then leverages such information to transparently remap pages to save PM or improve performance.

5.1 PM Tracking

While there has been much work in memory access tracking and prediction, there is little work doing this at the hypervisor level for managing PM. This is unique as VPM needs to optimize for performance while retaining persistency and thus needs to track both read and write set with timing information. Commodity processors provide limited hardware support for tracking memory access patterns. Prior approaches usually use an intrusive approach by either trapping all accesses to the hypervisor [71] or only tracking read accesses [26], or both. Other approaches usually leverage dedicated hardware support currently not available in commodity processors [19, 46] and not designed for hypervisors.

To provide a readily-available approach while retaining guest transparency, VPM leverages nPT to track the working set information. While prior Intel processors before Intel Haswell only provides an *access bit* to nPT entries, recently processors (e.g., Haswell) also provide a *dirty bit* in nPT, which can be used by VPM to track the writing work set. This avoids the expensive approach by frequently trapping into the hypervisor as in prior approaches [71].

At every sampling period, VPM scans each nPT entries related to PM and gets the access bits and dirty bits of nPT entries and also clear such bits to rescan them later in next period (may need to flush TLB for nPT). VPM then records such bits for use in prediction.

Prior PM management systems usually use a variant of CLOCK algorithm [9] to predict future memory accesses [33]. While this can result in good accuracy, it requires frequent sampling of memory accesses, which may incur large overhead in a virtualized environment. As indicated by prior work [33], sampling writes instead of reads/writes may provide a better estimation of future accesses for PM. Besides, for both PCM-like and NVDIMM like PM, the write accesses are more important metrics to PM virtualization. Hence, we only sample write accesses in this paper.

5.2 Transparent Page Migration

Page displacement is done by remapping from PM pages to different storage media on the fly. VPM achieves this by dynamically changing the mapping from guest physical address to host physical address, which is stored in nPT.

Batching of flushing to disk: Whenever there is a PM page swap-out, the content of this page needs to be made persistent on the disk(SSD), which in turn requires a disk flush, after which can applications continue to run. Frequent disk flushes can dramatically waste disk bandwidth due to the low utilization of on-disk cache and little schedule space for disk-scheduler. To relieve the performance impact incurred by disk flush operations, VPM aggressively writes back PM pages to the disk in the background and only issue disk flush when the amount of dirtied data reaches a certain threshold or on applications' request (PM page swap out).

Reducing writing cost with lazy reallocation: When a page is displaced, VPM does not immediately clear its content and still retains its mapping until it is reused by other VMs. Besides, such pages are allocated to other VMs in a least-recently displaced order. These are used to reduce the writing/copying cost to this page. For example, when there is a write request to a guest PM page mapped to a DRAM while the original host PM page is still not reallocated, a copy of the page can be avoided as VPM can directly remap the host PM page back. Later, after the nPT violation due to the write request is fixed, the write can directly write to the host PM page.

5.3 Other Issues

Transparent Huge Page: Commodity hypervisors like KVM are built with a mechanism called transparent huge page (THP), which merges consecutive pages into a huge page (like 2MB or even 1GB) to reduce TLB misses and TLB miss penalty. However, this increases the paging granularity and thus reduces the flexibility of PM virtualization. Further, the cost of a page migration or huge page demotion (i.e., breaking a huge page into smaller pages) is much higher. For the sake of simplicity, VPM currently disables THP. To work with THP, VPM can register a callback to receive THP's notify whenever there's a change to nPT.

Transparent page sharing: Commodity hypervisors has a useful feature called transparent page sharing, by which the hypervisor automatically merges pages with the same content. More specifically, KVM has a feature derived from Linux called kernel same-page merging (KSM). VPM is designed to leverage this feature for PM as well. However, it leverages the fact that the merged page is write-protected such that it instead uses a DRAM page instead of host PM page to save PM pages if PM is with a higher price than DRAM in the NV-DIMM case. We currently disabled KSM for simplicity, but can support it by distinguishing the case for consolidation and KSM upon a write protection fault.

6. Implementation

We have built a prototype of VPM based on Linux 3.17.2 by modifying the KVM implementation for Intel processors. The code base of VPM comprises around 4200 lines of code.

To expose PM to guest VMs, VPM extends guest kernel with a kernel module, whose main functions are managing and allocating PM. For the current implementation, the kernel module uses a predefined reserved 1GB physical consecutive memory region as the PM pool, which is backed by VPM. This provides guest VMs with an abstraction that there is a dedicated 1GB PM provided, no matter what the amount of native PM is. VPM's guest kernel module is also responsible for managing and persisting the mapping between guest physical PM and applications, to provide a consistent view of memory across crashes.

PM Tracking for HV: VPM-hv leverages the Access/Dirty Bit on EPT entries to track the accesses and updates of every PM page. The tracking process piggybacks on one VM-Exit event for every 500 nanoseconds. For each round of tracking, VPM-hv clears the Access/Dirty bit (if set) for the scanned pages and record this information for the page remapping afterward. However, checking all page entries of guest PM for each round of scanning can hurt the performance, which is a dilemma between accuracy and overhead of statistics. To relieve this problem, VPM-hv employs a two-level scanning, PM page entries in the first level refer to the hot pages, and they will be checked for each round of PM scanning. PM page entries in the second level refer to the relatively cold pages, and they will be scanned in a clock-like manner. The number of page entries in the first level and pages to scan in the second level are both limited. Page entries will be moved between the two levels according to the frequency that it is modified.

PM Tracking for PV: Access tracking for VPM-pv is relatively easy since the tracking information can be directly obtained when processing *vpm_persist* APIs.

Managing PM for HV: Managing PM for VPM-hv includes handling PM access EPT-violation as well as guest PM Page remapping. During the handling of EPT-violation, VPM-hv needs to allocate native PM since it requires that all updates are applied to physical PM, which under the worst situation, needs to reclaim an already allocated PM Page by unmapping it from the page table. VPM-hv uses the CLOCK algorithm [9] to pick the PM Page for unmapping. The PM page remapping remaps the guest PM page with a DRAM page if the updating frequency is not high enough. Currently, PM Page Remapping for VPM happens at the same frequency as the PM access tracking.

Managing PM for PV: Handling EPT-violation for VPM-pv does not require the allocated page to be a native PM page. This is because the persistency of VPM-pv is guaranteed by the VPM-pv APIs, which is processed by a *persist thread*. In VPM-pv, *persist thread* shares with each guest VM a 128MB non-volatile memory metadata region. This region includes 2 sections, the lock section, and ring queue section. The former one holds locks for each guest PM pages, while the latter one holds *vpm_persist* requests. On processing *vpm_persist* requests, *persist thread* locks the corresponding memory region, copies the content to non-volatile media (if necessary), and then unlocks the region. On processing *vpm_barrier* requests, *persist thread* flushes the whole ring queue for this VM. VPM-pv uses Peterson algorithm to implement *vpm_lock* and *vpm_unlock*.

Para-virtualizating PMBD: To demonstrate the effectiveness of providing hints to the hypervisor using para-virtualization, we modified PMBD, a persistent memory block device to use the interface provides by VPM. We found that the process is relatively easy due to the module design of PMBD: we only need to modify few lines of code: we wrap the block writing with *vpm_lock* and *vpm_unlock* and persist this writing with *vpm_persist*. Another change we need is to change calls to *clflush* to use the batching interface *vpm_barrier* of VPM.

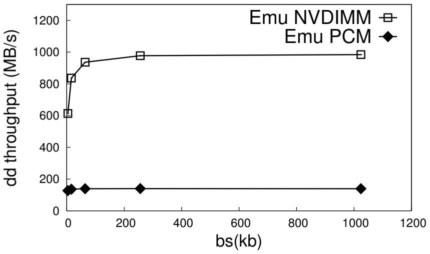


Fig. 4: PMBD throughput slowdown

7. Evaluation

7.1 Experimental Setup

We evaluate VPM using a host machine with a Quad-core Intel Core processor, 16 GB DDR3 memory and 256 GB Samsung SSD on SATA-3 bus. The memory is divided into two parts: one is the main memory and the other is emulated PM. Two types of PM are emulated, including NV-DIMM and PCM. NV-DIMM is relatively straightforward to emulate since it uses a DRAM component for read and write operations, which shares the same performance characteristics with DRAM. PCM is emulated by configuring PMBD to deliberately inject a slowdown on write operations. We use *dd* to test the effect of PMBD slowdown, by writing to PMBD at different granularity with *DIRECT_IO* mode. As shown in Fig. 4, the write throughput of emulated NVDIMM is roughly 8X of emulated PCM when the size of write unit is over 16KB. For a smaller write unit, the slowdown is around 5X.

It might be common to use a Linux RAMDISK-like block device as a native PM device for the baseline. However, it is not fair enough since a RAMDISK device relies on the OS virtual machine management policies and cannot be partitioned between PM and DRAM. Thus, it could be interfered by OS paging. A better choice could be using Persistent Memory Block Device (PMBD) which supports partitioning between PM and DRAM [12]. For block writes, PMBD uses *pm_wbarrier* operations to ensure that the selected data is flushed from CPU cache to memory. PMBD is also able to emulate the throughput of PM.

We evaluate VPM by considering the three dimensions: 1). *PM types*, including NV-DIMM, PCM, etc. NV-DIMM has similar performance with DRAM and has no endurance problem, while PCM has a limitation on write and is slow on write. 2). *Virtualization types*, including full-virtualization and para-virtualization, the former requires no modification

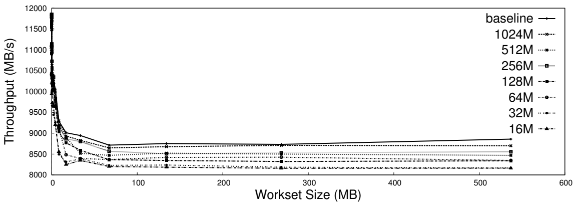


Fig. 5: fwr bandwidth

on guest VM while the latter does; 3). *Guest workloads*, including different file systems and applications, where the file systems can be PM-aware or not.

7.2 Microbenchmark

We use *lmbench* to analyze VPM's influence on bandwidth and latency of memory access. We select *fwr* to measure write bandwidth of VPM with different amount of PM given. To enable *fwr* to access PM, we adopt *libvmalloc* library provided by *NVML*[2], which replaces the original memory allocation functions (e.g., *malloc* and *free*) with PM-aware ones. We also make small modifications (less than 10 LoC) on *libvmalloc* so that it can communicate with VPM's guest kernel module. As shown in Fig. 5, the x-axis indicates the size of touched memory for the *fwr_bw* benchmark; the y-axis stands for the throughput, and lines in the figure represent memory bandwidth measured for different sizes of the touched memory area when the amount of backing PM is different. As can be observed from this figure, the size of provided native PM does not have a big effect on the final result, which contributes to less than 10% drop in bandwidth. This is because *fwr* access memory in a relatively sequential pattern, which is friendly by VPM's prediction and prefetching mechanism.

7.3 PM Requirement

One motivation of VPM is that PM is usually a precious hardware resource on servers due to its high price. VPM is designed to provide most of the performance benefit from PM with much less physical PM hardware. We run Filebench with ext4 over PMBD on a server with NV-DIMM. The baseline test runs with enough NV-DIMM, and the workload runs on VPM using full-virtualization PM. The size of virtual PM is the same as the size of PM in the baseline test but is backed with different sizes of physical PM. In both cases, the physical PM is emulated with DRAM by partitioning a region of the main memory as NV-DIMM. The results are shown in Fig. 6-(a),(b) and (c). The figures above illustrate the performance (throughput) of VPM when running workloads with different native PM size given. The "base" line in each of the figure denotes the performance of the workload when running on unmodified VM/hypervisor with enough native PM. For the unmodified VM/hypervisor, none of the benchmarks above will be able to execute if memory provided is not enough.

Fig. 6 (a) shows the performance of fileserver on emulated NVDIMM when running on the PV and HV versions of VPM with insufficient PM. Fileserver is a workload which issues disk flush operations, which in our case is translated into *clflush* for VPM-hv or *vpm_barrier* for VPM-pv. This figure shows that both VPM-hv and VPM-pv can achieve over 80% of the performance with only 20% of the PM. For this workload, VPM-pv outperforms VPM-hv because the former does not require write operations to be put on PM, which saves time for VM-Exits. Since fileserver flushes disk in a relatively conservative manner, the data persistence cost required by PV is hidden.

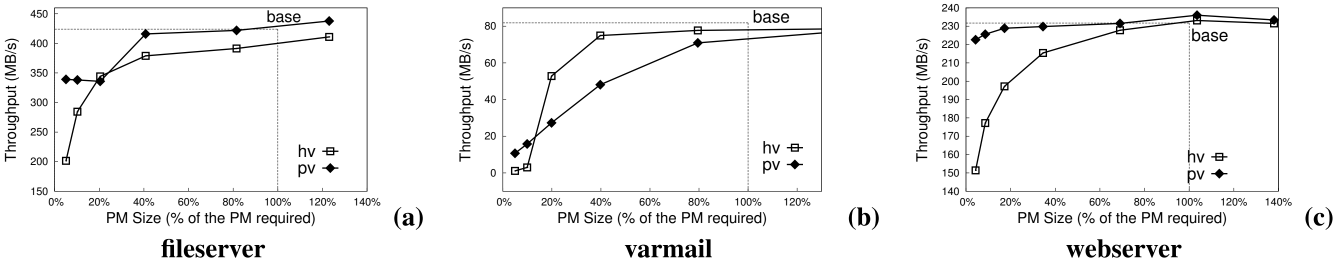
Fig. 6 (b) shows the performance of varmail on emulated NVDIMM when with insufficient PM. Varmail is a workload which frequently issues disk flush operations. From the figure, we can see that both VPM-hv achieves 60% of the performance with 20% of PM and 95% of the performance with 60% of PM. The para-virtualization version of VPM achieves 40% of the performance with 40% of PM and requires around 80% of PM to help performance grow over 95%. This is because *vpm_barrier* caused by disk flush operations introduces many VM-Exits, which impairs the chance to hide the flush latency in the background.

The Fig. 6 (c) shows the performance of webserver on emulated NVDIMM with insufficient memory given. Both VPM-pv and VPM-hv can achieve over 80% of the performance with only 20% PM provided since webserver is a benchmark which issues write operations in a low frequency and seldom flushes disk. Webserver requires around 750 to 800MB memory to have a complete running.

We also evaluate the performance on the emulated PCM. The workload in the guest VM is filebench on Ext4 file system which using PMBD as the block device. Under VPM-pv, the PMBD is modified to use hypercalls to replace instructions like *pcommit*. The results are shown in Fig. 7-(a),(b),(c). As shown in the figure, VPM-hv achieves 95% of the baseline performance using 70% of PM, while VPM-pv achieves 95% performance only using 50% of PM. The performance of VPM-pv is better than VPM-hv since the DRAM is faster than PCM on writing, as well as that there are more write absorptions in VPM-pv.

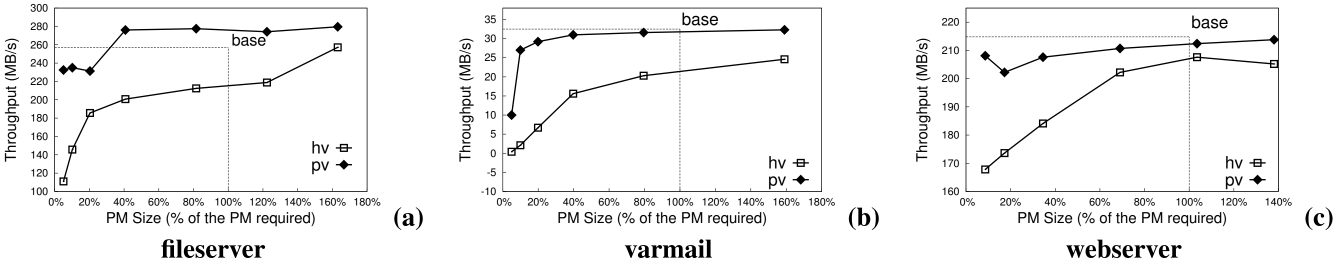
Fig. 7(a) shows the performance of fileserver on emulated PCM with other configurations unchanged. It can be noticed that the overall performance degrades due to PCM write performance slowdown. However, VPM-pv can perform better than the original system under this configuration. This is because VPM-pv allows applications to directly write on DRAM, which absorbs multiple modifications over PM pages and avoids writing to PCM on the critical path. Additionally, VPM-hv' s performance is furthermore impaired due to handling EPT violation, which requires migrating a DRAM page to a PCM page.

[2] https://github.com/pmem/nvml



fileserver **varmail** **webserver**

Fig. 6: Relationship between Native NVDIMM PM Size and Performance (Throughput)



fileserver **varmail** **webserver**

Fig. 7: Relationship between Native PCM PM Size and Performance (Throughput)

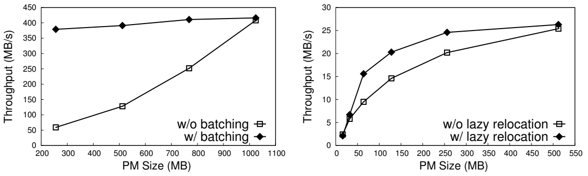


Fig. 8: (a) The effect of batching of flushing, and (b) the effect of lazy relocation.

Fig. 7(b) presents the performance of varmail on emulated PCM. One major difference between the performance on emulated NVDIMM is that VPM-pv can produce higher throughput than VPM-hv. In this case, the slowdown of PCM affects VPM-hv more than VPM-pv. This is because that the relatively random write pattern of varmail makes VPM-pv put content directly on DRAM.

Fig. 7(c) shows the performance of webserver on emulated PCM, which is hardly affected by underlying PM type due to the webserver's asymmetric read/write pattern.

7.4 Optimization Decomposition

We further analyze the effect of each optimization on VPM's performance. The optimizations include batching of disk flushing and lazy relocation (§ 5.2). Both optimizations can be applied to VPM-hv and VPM-pv. Here, we use VPM-hv to demonstrate its effect.

We run fileserver over emulated NVDIMM to show the performance improvement contributed by batching of disk flushing. As shown in Fig. 8(a), batching of flushing increases the utilization of disk bandwidth, which in return speeds up the handling of page remapping.

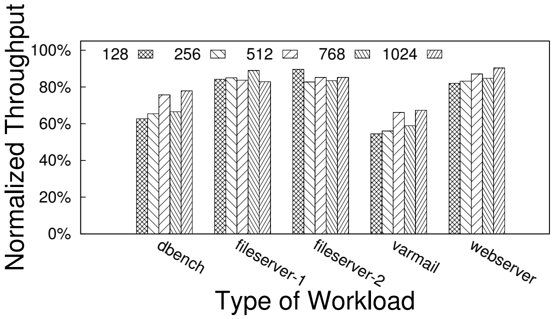


Fig. 9: Performance for workloads under consolidation

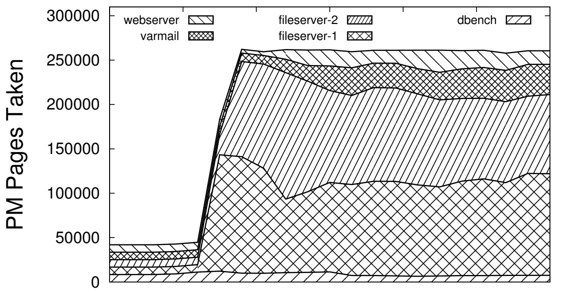


Fig. 10: PM usage for workloads under consolidation

The lazy relocation reduces the memory copy due to page remapping which requires copying the content of a non-volatile page to a PM page. Hence, we run varmail over emulated PCM to show the lazy relocation's effect. As shown in Fig. 8(b), lazy relocation increases performance by over 30% when the amount of native PM is more than 25%

of that is required on the unmodified system. For the case when the available PM pages are few, the handling of EPT-violation needs to aggressively allocate PM pages, which renders lazy relocation useless.

7.5 VM Consolidation

VM Consolidation part is evaluated using a host machine with a Quad-core (configured as 8 cores using hyper-threading) Intel Xeon processor, 32 GB DDR3 memory with 2 channels and 256 GB Samsung SSD on SATA-3 bus.

We evaluate the degree of VM consolidation of VPM with 5 VMs running concurrently with 4 types of workloads: Two VMs run *fileserver*, the other 3 VMs run *varmail*, *webserver*, and *dbench* respectively. The 5 VMs make up one set of test VMs, which is also as known as a *tile*. As previously mentioned, all benchmarks will not be able to have a complete run if not sufficient memory is provided in an unmodified VM and hypervisor. Table 2 introduces the performance and PM size that is required for the workloads in the tile. Therefore, it requires around 2.1GB to 2.2GB in total to run a tile to completion on an unmodified VM and hypervisor.

Table 2: Performance and required PM size

Benchmark	Memory (MB)	Throughput (MB/s)
dbench	50 to 60	1,168
fileserver	600 to 650	430
varmail	150 to 160	90
webserver	700 to 750	251.3

For each round of the test, 128MB, 256MB, 512MB, 1024MB of native PM is provided respectively. We run the tile on VPM-pv, whose results are shown in Fig. 9. Dbench can achieve 60% of the throughput with around 15% of the amount of PM provided, around 80% of the throughput with 50% of the amount of PM provided. Varmail can achieve 30% of the throughput with around 15% of the amount of PM provided and 60% of the throughput with 50% of the amount of PM provided. The performance of fileserver is hardly affected by the size of given PM, this is because the frequent flush operations issued by varmail and dbench reduce the possibility of background persisting. Since webserver contains mostly read operations, a different PM size has little impact on its overall performance. Fig. 10 shows how native PM is multiplexed across different virtual machines with 1024MB PM provisioned. When all PM are used by each VM, VMS running dbench, fileserver, varmail and webserver consume 4%, 37%, 14% and 8% of the total PM respectively.

7.6 Crash Recovery

We run dbench on VPM-hv to demonstrate VPM's ability to recover from a crash. In this case, 32MB of native PCM is provided to VM. Dbench is configured to run for 140 seconds, of which the warm-up phase runs for the first 20 seconds; after that dbench starts the real execution phase. During the whole execution phase, 2 crashes are injected. The first crash happens at around 6th second since the execution phase starts, while the 2nd one occurs at around 86 seconds.

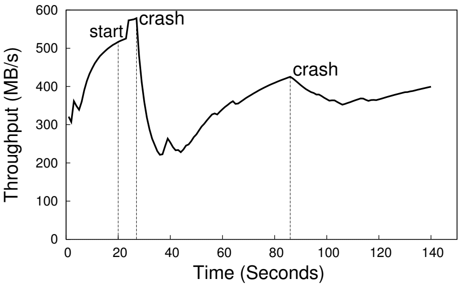


Fig. 11: Recovery performance

Since guest PM as well as the mapping information either resides on PM or other non-volatile media, we simulated a crash by unmapping all the non-volatile pages from the guest PM. Thus, the following accesses will trigger EPT-violations which requires the VPM to find the corresponding data by reading from the non-volatile media other than native PM.

From Fig. 11, we can see that the 2 crashes that are injected decrease the throughput due to the additional EPT-violation handling. After each crash injection, dbench continues its execution and the throughput gradually recovers.

8. Limitation and Future Work

While VPM makes the first step towards virtualizing persistent memory (PM), the work is still preliminary and left a number of research questions unanswered. We leave these as future work.

NUMA. As PM can approach the performance of DRAM and is byte-addressable through the memory bus, there will be a similar NUMA issue for PM, where accesses to different PM placed in different sockets may have different performance. Hence, a PM virtualization platform should also take this into account to efficiently provisioning PM for guest VM according to its access characteristics. Specifically, we plan to extend auto-NUMA in Linux/KVM to provide automatic management of NUMA effects for virtualized PM.

Wear leveling. The current implementation of VPM does not consider wear leveling issue. This is important for some PM devices that have limited life cycles. As VPM now has the global knowledge of the access pattern of guest PM, it is possible for VPM to perform global wear leveling to improve the life cycles of PM.

Different characteristics. The performance characteristics for emerging PM is still speculative, which may be varied from currently predicted ones. In this paper, we only evaluated two settings for NVDIMM and PCM. In future, we plan to investigate how to different performance characteristics of future PM may affect the decision of VPM.

9. Related Work

Systems support for PM: With the emergence of persistent memory, researchers have started to think proper systems support for persistent memory [15, 33, 39]. In a position paper, Mogul et al. [39] discussed proper OS support for hybrid DRAM with PCM or NOR Flash and proposed Time-Between-Writes-to-Page (TBWP) to ensure write endurance. There are a number of storage systems designed for PM, including BPFS [15], SCMFS [62], Shortcut-JFS [30], PMFS [20] and NOVA [63]. Maze et al. [37] propose using a single-level store to unify PM DRAM and storage, which eliminates the need for address translation between PM and storage but requires a reconstruction of systems software atop. A close work to VPM is NV-Hypervisor [48], which briefly describes the use of PM to provide transparent persistence to whole guest VMs. Yet, it does not aim at virtualizing and efficiently managing PM to be used by PM as in VPM, nor does it consider a para-virtualized interface for PM.

The emergence of PM has also stimulated a number of applications of PM, including libraries [14, 58], database transactions [4, 13, 60] and data structures [53]. VPM also leverages some PM-aware concurrent and durable data structures to support proper crash recovery.

Software/hardware interfaces for PM: There have been much work in providing proper hardware/software interface for PM to bridge the persistence of memory and the volatility of CPU structures [14, 15, 25, 34, 35, 55, 59, 69, 70]. Mnemosyne [55] provide an application interface that leverages a lightweight transaction mechanism to preserve data consistency upon failures. Kiln [69] uses a non-volatile last-level cache (LLC) and NV-aware cache replacement to ensure atomicity and ordering of memory updates. FIRM [70] further provides a memory scheduling algorithm specifically for PM-based applications. WRaP [25] uses a victim persistence cache to coalesce updates to NVM and a redo log in memory. Pelley et al. [44] observe the similarity between memory consistency and memory persistency and describes strict and relaxed persistent models. VPM also aims at studying the interfaces for PMs but at the level of virtual machines and its hypervisor.

Memory virtualization: Memory virtualization has been a key component of system virtualization, leading to various schemes like writable page table [40], shadow paging [57] and nested paging [17, 41] and a set of optimizations [5, 6, 8]. Recently, there are interests in investigating elimination of two-dimensional paging [24] using direct segments [7]. Ye et al. [66] leverage the memory tracing mechanism of VMware to prototype a hybrid memory system with the heterogeneous performance by slowing down accesses to a portion of memory. Lee et al. [32] extend Xen to manage hybrid fast 3D die-stacked DRAM and off-chip DRAM. A recent effort [42] also considers managing hybrid on-chip DRAM and off-chip DRAM/NVRAM in a cloud environment. However, it mainly considers how to design OS kernel support inside a VM by designing a lightweight OS kernel, instead of efficiently virtualizing and managing PM in the hypervisor layer. Compared to existing memory virtualization work, VPM focuses on virtualizing and managing hybrid PM/DRAM that has completely different performance and persistency characteristics from the hypervisor layer. It will be our future work on how to leverage existing memory virtualization schemes to further optimize VPM.

Tracking memory accesses: There is also much work aiming at tracking memory accesses for memory management either in virtualized environment [26, 71], or native environment [72], yet does not consider hybrid PM/DRAM environment. Recently, there are some efforts in tracking memory accesses in hybrid DRAM/PM system for page placement [18, 33, 46], but require special hardware support and does not work for virtualized environment. VPM extends prior in tracking memory accesses by leveraging A/D bits in extended page tables for low-overhead, transparent memory access tracking and leverages writing working set to predict future memory access behavior.

Virtualizing Flash: Recent work has started to explore virtualization support for flash-based storage [50, 61, 67]. However, they mainly focus on providing I/O virtualization instead of memory virtualization in VPM.

10. Conclusion

This paper presented a study on the interfaces as well as the underlying hypervisor support to virtualize persistent memory (PM). A prototype, namely VPM, has been implemented and evaluated atop emulated PCM and NVDIMM. Performance evaluation shows that VPM can efficiently manage and multiplex PM while leading to small performance degradation even with under provisioned PM. The evaluation also shows that the para-virtualized interfaces can lead to even more saving of PM while achieving a similar level of performance.

Acknowledgments

We thank our group member Yang Hong to help implement part of the para-virtualization of VPM and the anonymous reviewers for their helpful comments. This work is supported in part by National Key Research Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314), the Top-notch Youth Talents Program of China, Shanghai Science and Technology Development Fund (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), a grant from Samsung and Singapore NRF (CREATE E2S2).

References

[1] N. Alvares. Satisfying cloud data center requirements with new memory storage hierarchy. `http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150812_S202A_Alvares.pdf`, 2015.

[2] Amazon EC2. Ec2 for in-memory computing the high memory cluster eight extra large instance.

https://aws.amazon.com/blogs/aws/ ec2-for-in-memory-computing- the-high-memory -cluster-eight-extra-large/.

[3] 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)*, 9(2):13, 2013.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722. ACM, 2015.

[5] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). In *ISCA*, pages 48–59. ACM, 2010.

[6] T. W. Barr, A. L. Cox, and S. Rixner. Spectlb: a mechanism for speculative address translation. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 307–317. IEEE, 2011.

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, pages 237–248. ACM, 2013.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGOPS Operating Systems Review*, 42(2):26–35, 2008.

[9] R. W. Carr and J. L. Hennessy. Wsclocka simple and effective algorithm for virtual memory management. *ACM SIGOPS Operating Systems Review*, 15(5):87–95, 1981.

[10] 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. *ACM SIGARCH Computer Architecture News*, 40(1):387–400, 2012.

[11] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile datastructures. *VLDB*, 8(5):497–508, 2015.

[12] F. Chen, M. P. Mesnier, and S. Hahn. A protected block device for persistent memory. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–12. IEEE, 2014.

[13] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *EuroSys*, 2016.

[14] 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 *ASPLOS*, pages 105–118. ACM, 2011.

[15] 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 *SOSP*, pages 133–146. ACM, 2009.

[16] J. Corbet. Supporting filesystems in persistent memory. `https://lwn.net/Articles/610174/`, 2014.

[17] A. M. Devices. Amd, secure virtual machine architecture reference manual, 2005.

[18] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 664–669. IEEE, 2009.

[19] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[20] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, page 15. ACM, 2014.

[21] L. A. Eisner, T. Mollov, and S. J. Swanson. *Quill: Exploiting fast non-volatile memory by transparently bypassing the file system*. Citeseer, 2013.

[22] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1221–1231. IEEE, 2011.

[23] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, 2008.

[24] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–189. IEEE Computer Society, 2014.

[25] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using wrap. In *Conf. on Computing Frontiers*, 2013.

[26] V. Gupta, M. Lee, and K. Schwan. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *VEE*, pages 79–92. ACM, 2015.

[27] Y. Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[28] Intel. Intel persistent memory block driver. `https://github.com/linux-pmbd/pmbd`.

[29] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[30] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn. Shortcut-jfs: A write efficient journaling file system for phase change memory. In *Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2012.

[31] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *FAST*, pages 73–80, 2013.

[32] M. Lee, V. Gupta, and K. Schwan. Software-controlled transparent management of heterogeneous memory resources in virtualized systems. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, page 5. ACM, 2013.

[33] S. Lee, H. Bahn, and S. H. Noh. Characterizing memory write references for efficient management of hybrid pcm and dram memory. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 168–175. IEEE, 2011.

[34] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, pages 216–223. IEEE, 2014.

[35] Y. Lu, J. Shu, and L. Sun. Blurred persistence in transactional persistent memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–13. IEEE, 2015.

[36] D. Magenheimer. Transcendent memory in a nutshell. `https://lwn.net/Articles/454795/`, 2011.

[37] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A case for efficient hardware-software cooperative management of storage and memory. In *Proceedings of the 5th Workshop on Energy-Efficient Design (WEED)*, pages 1–7, 2013.

[38] Micron. Micron announces availability of phase change memory for mobile devices. `http://investors.micron.com/releasedetail.cfm?releaseid=692563`, 2012.

[39] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi. Operating system support for nvm+ dram hybrid main memory. In *HotOS*, 2009.

[40] J. Nakajima, A. Mallick, I. Pratt, and K. Fraser. X86-64 xenlinux: architecture, implementation, and optimizations. In *Linux Symposium*, page 173, 2006.

[41] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[42] K. H. Park, W. Hwang, H. Seok, C. Kim, D.-j. Shin, D. J. Kim, M. K. Maeng, and S. M. Kim. Mn-mate: Elastic resource management of manycores and a hybrid memory hierarchy for a cloud node. *J. Emerg. Technol. Comput. Syst.*, 12(1): 5:1–5:25, 2015.

[43] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *VLDB*, 7(2):121–132, 2013.

[44] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 265–276. IEEE Press, 2014.

[45] J. Pinkerton. Bring the public cloud to your data center. `http://www.snia.org/sites/default/files/Pinkerton_NVMSummit-2015.pdf`, 2015.

[46] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[47] 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*, 52 (4.5):465–479, 2008.

[48] V. Sartakov, R. Kapitza, et al. Nv-hypervisor: Hypervisor-based persistence for virtual machines. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 654–659. IEEE, 2014.

[49] SNIA. Nvdimm special interest group. `http://www.snia.org/forums/sssi/NVDIMM`, 2015.

[50] X. Song, J. Yang, and H. Chen. Architecting flash-based solid-state drive for high-performance i/o virtualization. *Computer Architecture Letters*, 13(2):61–64, 2014.

[51] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

[52] V. Technology. Arxcis-nv (tm): Non-volatile dimm. `http://www.vikingtechnology.com/arxcis-nv`, 2014.

[53] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.

[54] S. D. Viglas. Data management in non-volatile memory. In *SIGMOD*, pages 1707–1711. ACM, 2015.

[55] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Notices*, 46 (3):91–104, 2011.

[56] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, page 14. ACM, 2014.

[57] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI*, pages 181–194. Usenix, 2002.

[58] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 957–968. IEEE, 2012.

[59] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent transactional memory. *Computer Architecture Letters*, 14: 5861, 2015.

[60] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *SOSP*, 2015.

[61] 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 *FAST*, pages 111–118. USENIX, 2015.

[62] X. Wu, S. Qiu, and A. Narasimha Reddy. Scmfs: a file system for storage class memory and its extensions. *ACM Transactions on Storage (TOS)*, 9(3):7, 2013.

[63] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX FAST*, pages 323–338, 2016.

[64] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *FAST*, pages 167–181. USENIX, 2015.

[65] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *J. Emerg. Technol. Comput. Syst.*, 9(2):11:1–11:20, 2013.

[66] D. Ye, A. Pavuluri, C. Waldspurger, B. Tsang, B. Rychlik, S. Woo, et al. Prototyping a hybrid main memory using a virtual machine monitor. In *ICCD*, pages 272–279. IEEE, 2008.

[67] Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Removing the costs and retaining the benefits of flash-based ssd virtualization with fsdv. In *MSST*, 2015.

[68] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2015.

[69] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432. ACM, 2013.

[70] J. Zhao, O. Mutlu, and Y. Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *MICRO*, pages 153–165. IEEE, 2014.

[71] W. Zhao, Z. Wang, and Y. Luo. Dynamic memory balancing for virtual machines. In *VEE*, pages 37–47. ACM, 2009.

[72] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically tracking miss-ratio curve for memory management. In *ASPLOS*, 2004.