Received: 4 May 2016 | Revised: 24 August 2016 | Accepted: 4 September 2016

DOI 10.1002/cpe.4028

WILEY

SPECIAL ISSUE PAPER

MBSA: a lightweight and flexible storage architecture for virtual machines

Xian Chen[1] | Wenzhi Chen[1] | Zhongyong Lu[1] | Yu Zhang[1] | Rui Chang[2] | Mohammad Mehedi Hassan[3] | Abdulhameed Alelaiwi[3] | Yang Xiang[4]

[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]State Key Laboratory of Mathematic Engineering and Advanced Computing, Zhengzhou, China
[3]College of Computer and Information Sciences, King Saud University, Riyadh, 11543, Saudi Arabia
[4]Centre for Cyber Security Research, Deakin University, Burwood, VIC, Australia

Correspondence
Wenzhi Chen, College of Computer Science and Technology, Zhejiang University, Hangzhou, China.
Email: chenwz@zju.edu.cn

Funding information
Deanship of Scientific Research, Grant/Award Number: RGP-318

Summary

With the advantages of extremely high access speed, low energy consumption, nonvolatility, and byte addressability, nonvolatile memory (NVM) device has already been setting off a revolution in storage field. Conventional storage architecture needs to be optimized or even redesigned from scratch to fully explore the performance potential of NVM device. However, most previous NVM-related works only explore its low access latency and low energy consumption. Few works have been done to explore the appropriate way to use NVM device for improving virtual machine's storage performance. In this paper, we comprehensively evaluate and analyze conventional virtual machine's storage architecture. We find that, even with cutting-edge optimization technologies, virtual machine can only achieve 30% of NVM device's original performance. Based on this observation, we propose a memory bus–based storage architecture, which we named MBSA. Memory bus–based storage architecture can greatly shorten the length of virtual machine's storage input/output stack and improve NVM device's use flexibility. In addition, an efficient wear-leveling algorithm is proposed to prolong NVM device's lifespan. To evaluate the new architecture, we implement it as well as the wear-leveling algorithm on real hardware and software platform. Experimental results show that MBSA can provide a big performance improvement, about 2.55X, and the wear-leveling algorithm can efficiently balance write operations on NVM device with a negligible performance overhead (no more than 3%).

KEYWORDS

nonvolatile memory, performance optimization, storage I/O stack, virtual machine

1 | INTRODUCTION

Nowadays, we are in the era of information explosion. Data should be processed within a specified time limit to provide useful value, which requires a powerful computing platform. For the strong resource management capability, virtualization technology is a good solution. But high input/output (I/O) virtualization overhead, up to 50% in some situation,[1] makes virtualization technology not suitable for data-intensive workloads. This also explains why most high-performance computing workloads and online transaction processing workloads are not deployed on virtualization platform. Things have improved when new fast storage devices emerge, such as NVM Express (NVMe) solid-state drive (SSD) and 3-dimensional X-point.[2] However, virtualization overhead still occupies a large proportion of the whole I/O latency, especially when the storage device is becoming faster and faster.

In academia and industry, a lot of work has been done to reduce I/O virtualization overhead by increasing request processing concurrency,[3–5] shortening request processing path,[6–8] or optimizing I/O request scheduler.[9,10] But all the work is based on conventional front-/back-end driver model–based storage architecture, in which virtual disks are treated as I/O devices and data are transferred through asynchronous interrupt mechanism. This applies to conventional hard disk drive (HDD) and SSD, because they are connected to host machine through SATA or PCI-E interface and the data transmission is asynchronous. But for nonvolatile memory (NVM) device, things are different. Nonvolatile memory device not only has extremely low access latency and low energy consumption but also can be accessed

at byte granularity. This means that NVM device can be connected to host machine through memory bus, which is an essential difference compared with conventional HDD and SSD. If we still treat NVM device as a faster SSD, its performance potential will not be fully explored.

Our work in this paper is to find an appropriate way to fully explore NVM device's performance potential to enhance virtual machine (VM)'s storage performance. Toward this objective, we first conduct comprehensive evaluation and analysis on conventional front-/back-end driver model–based storage architecture. In this architecture, NVM device is encapsulated into several virtual disks in virtual machine monitor (VMM) as shown in Figure 1. Then these virtual disks are assigned to VMs in the format of I/O devices. No modification is needed in guest operating system (OS), leading to pretty good compatibility. This is why NVM device is usually used just as a faster SSD. Furthermore, the cutting-edge optimization technologies such as data-plane,[11] multi-queue,[4] and vhost-blk[6] are also evaluated. We find that conventional storage architecture has greatly limited NVM device's performance potential, VM can only achieve 30% of the original performance. Based on this observation, we propose to assign NVM device directly through memory bus, bypassing the lengthy I/O stack. We call the new architecture memory bus–based storage architecture (MBSA). In MBSA, storage access is conducted as normal memory access in VM, without any involvement of VMM. Experimental results show that MBSA has great advantage over conventional storage architecture, the improvement is about 2.55 × . Furthermore, we also propose an efficient wear-leveling algorithm to enhance the NVM device's lifetime and implement it in MBSA. Experimental results show that the proposed wear-leveling algorithm has excellent effect on balancing write operations and the caused overhead is negligible, no more than 3%.

The rest of this paper is organized as follows. Section 2 gives motivation and related work. Experimental environment is given in Section 3. Conventional front-/back-end driver model–based storage architecture is evaluated and analyzed in Section 4. Section 5 describes our proposed storage architecture and the wear-leveling algorithm. Finally, Section 6 concludes the paper.

2 | MOTIVATION AND RELATED WORK

The work in this paper is motivated by prior works related to NVM and storage performance optimization technologies in virtualization environment. More details about the 2 aspects will be given in the following sections.

2.1 | Nonvolatile memory

Nonvolatile memory is a class of emerging memory technologies, such as phase change memory (PCM), ferroelectric RAM, resistive RAM, and magnetoresistive RAM. They usually have dynamic RAM (DRAM)–like data access latency and SSD-like data storage density. In addition, they can be accessed at byte granularity, making it possible to directly attach them to processor memory bus. With these features, NVM has blurred the distinction between memory device and storage device. However, it also faces some challenges. The lifetime of NVM device is usually limited, about 10^8 to 10^{15} for PCM and 10^6 to 10^{12} for resistive RAM. Magnetoresistive RAM can be used for a relatively longer time, but still less than 10^{15}, while DRAM device is usually considered cannot be worn out. Another drawback of NVM device is the asymmetric read/write (R/W) performance. For NVM device, write latency is usually several times longer than read latency.[12] The asymmetry will affect the whole system performance if read and write commands are not scheduled appropriately.

Nonvolatile memory–related works can be classified into 2 large categories. In the first category, NVM is used as a memory device. A lot of work has been done to combine NVM into traditional memory architecture to reduce the increasing computational energy consumption. Qureshi in one study[13] proposed a hybrid memory architecture, where a small DRAM is used as the buffer of PCM to make up the performance gap and reduce write operations to PCM. To enhance the durability of NVM device, work of Zhou et al[14] proposed to remove redundant writes at bit level and conduct row shifting and segment swapping in hardware. Work of Qureshi et al[15] explored the performance characteristics of multilevel PCM and proposed a memory system that can dynamically change PCM working mode according to memory pressure. Supports from OS are also studied in the works of Mogul

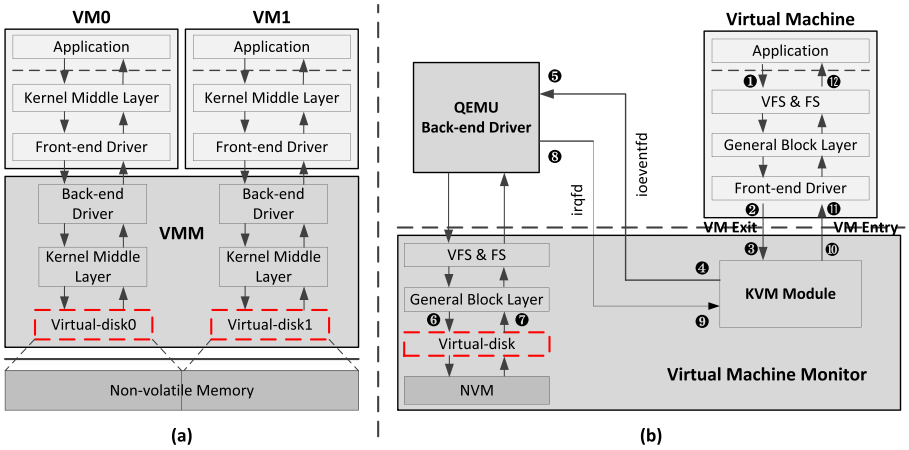

FIGURE 1 Conventional storage architecture in virtualization environment. A, Schematic diagram for multiple virtual machines; B, Detailed storage I/O stack in KVM. FS indicates file system; I/O, input/output; KVM, kernel-based virtual machine; NVM, nonvolatile memory; QEMU, Quick Emulator; VFS, virtual file system; VM, virtual machine; VMM, virtual machine monitor

et al, and Jung and Cho[16,17]; they proposed to allocate NVM and DRAM according to the access characteristics.

In the second category, NVM is used as a storage device. For compatibility and performance reasons, the nonvolatility feature can be explored in different formats. To obtain good compatibility, NVM is usually encapsulated into a block device.[12,18] Conventional file systems can be directly deployed on it, without any modification made in upper software layers. The corresponding cost is the relatively lower storage performance. This is because conventional storage system is designed for conventional slow spinning HDD. For NVM, software overhead is much higher than hardware,[19] which greatly limits NVM device's performance. To alleviate this situation, many new file systems were designed for NVM, eg, BPFS,[20] SCMFS,[21] Aerie,[22] PMFS,[23] and Shortcut-JFS.[24] Furthermore, to simplify NVM-based programming, new programming models were proposed in the works of Coburn et al and Volos et al,[25,26] providing programmers an easy way to use NVM. In addition, if NVM device is used in cloud storage, cloud storage forensic analysis[27–29] can also be accelerated.

As shown above, most prior NVM-related works are conducted at hardware level or at system software level in non-virtualization environment, while our work in this paper tries to explore NVM device's performance potential as a storage device in virtualization environment. They are complementary.

2.2 | Storage I/O stack in virtualization environment

We take the example of kernel-based virtual machine (KVM) to discuss the storage I/O stack in virtualization environment. With KVM, we can provide VM a virtual storage device through 2 methods. One is fully simulating the target device. This method does not need any modification in guest OS. But its processing path is too long and causes many processor mode switches and memory copies. Thus, the performance is not very good. Usually, this method is not used in production environment. The other method is virtio,[30] which provides a front-/back-end driver model–based I/O virtualization architecture. As shown in Figure 1, the front-end driver is deployed in guest OS, and the back-end driver is in VMM. They communicate through a ring buffer that can accumulate I/O requests from guest OS and submit them to VMM in batch. Compared with fully simulating, virtio can provide much better performance, close to the native performance for spinning HDD.[31] But for fast storage devices such as NVMe SSD, the time spent on hardware processing is very short, only accounts for a small part of the whole data access latency. Thus, the design of software layer plays a more crucial role in improving storage performance. For virtio-based storage architecture, there are 2 primary bottlenecks blocking the performance advantage of fast storage device.

The first bottleneck is the poor request handling concurrency. From the aspect of hardware interface, NVMe SSD can provide much better concurrency than conventional spinning HDD. It can provide 64K queues for OS to simultaneously submit I/O requests, and the depth of each queue can be 64K,[32] while only 1 queue for HDD and the depth is just 32. There are 2 locations in virtio-based storage I/O stack limiting the concurrency provided by fast storage device. One is the submission/completion queue. There is only 1 request queue in the general block layer in guest OS. All request-related operations, such as submission/completion, request merging, fairness scheduling, and I/O accounting, are conducted in this queue with a big queue lock. When multiple threads concurrently operate the queue, lock contention will be serious, and storage performance will be decreased. To tackle this problem, in the work of Bjørling et al,[3] Bjørling proposed to split the request queue into 2 level queues: several per-cpu software queues and 1 dispatch (hardware) queue. Input/output threads submit requests into per-cpu software queue without any lock operation. Specific device driver will process software queues and submit the requests to hardware through dispatch queue. This multi-queue mechanism removes the coarse queue lock and improves request processing concurrency. However, this mechanism needs hardware support. To make virtio-blk compatible with multi-queue mechanism, Lei[4] proposed to use multiple virtqueues as dispatch queues to improve scalability and throughput. The other limitation of concurrency is the number of QEMU I/O threads. By default, all storage I/O requests are handled by the main QEMU thread with a coarse lock. This will cause severe scalability problem. With QEMU data-plane feature, each block device can be assigned with 1 I/O thread. However, if multiple applications concurrently access the same storage device, the scalability problem still exists. For this issue, work of Kim et al[5] proposed to assign I/O threads for each VM based on the number of virtual CPUs (VCPUs). This can greatly improve storage performance, but the number of VCPUs becomes the new limiting factor. On the other hand, integrating all optimization technologies into one system platform needs many modifications in guest OS and VMM, which seems not a good method in production environment.

The second bottleneck is the long request processing path. As shown in Figure 1, I/O request issued by application in VM will go through virtual file system and file system layer, general block layer, arriving at front-end driver layer. Then it will enter into VMM, and the KVM module in host kernel will notify the back-end driver in QEMU. Corresponding QEMU I/O thread will parse the request and fetch the requested data from virtual disk. If the virtual disk is managed as a file in host file system, then the I/O path in host will be gone through again. The whole I/O path from the application in VM to the virtual disk file in host is about twice the length of that in non-virtualization environment. Furthermore, processor mode switches will be triggered. Longer I/O path means higher software overhead, especially for fast storage device. To eliminate the redundant layers, work of He[6] proposed bio-based virtio-blk and vhost-blk. Bio-based virtio-blk skips the general block layer in guest OS. Vhost-blk bypasses the virtual file system and file system layer in host OS, directly submits requests from VM to the general block layer in host kernel. Vhost-blk also eliminates the kernel/user mode switches in host OS. The limitation of vhost-blk is that it can only support raw block device. Similar idea was also implemented on Xen platform in the work of Li et al.[7] Another method to shorten the storage I/O path is directly assigning block device to VM, letting VM completely manages the device. In this method, VMM can be removed from the storage I/O path with the help of software optimization[33] or hardware APIC virtualization (APIC-v) support.[34] For this method, host server needs to supply enough hardware slots for storage devices, or the storage device itself supports single root input/output virtualization (SR-IOV) interface. However, until now, SR-IOV has not been widely applied on storage device.

Based on the above analysis, we argue that if we want to fully explore the performance potential of NVM device, software layer must be thin enough. On the other hand, NVM device has some good features that normal fast SSD device does not have, such as byte addressability. Nonvolatile memory device cannot be simply treated as a common faster SSD device. Conventional interrupt-based storage architecture in virtualization environment needs to be re-evaluated for NVM device.

3 | EXPERIMENTAL ENVIRONMENT

All the experiments in this paper are conducted on the same physical machine. It has one Intel i7-2600 3.4 GHz processor with 4 cores, 16 GB DDR3 memory, and a 500 GB 7200RPM Seagate hard drive. We select QEMU/KVM as the virtualization platform. Each VM is configured with 4 VCPUs and 2 virtual disks. One virtual disk is used as system disk, and the other is used as test disk. Host and guest all run Ubuntu 12.04. Unless otherwise noted, the kernel version is 3.10. We select flexible I/O tester (FIO) as the benchmark, the version is 1.59. Because we do not have the real NVM hardware, DRAM is used to simulate NVM device. Based on the latency characteristics of NVM device presented in the work of Chen et al,[12] we set read latency the same with DRAM and write latency 3 times DRAM. The size of simulated NVM device is 8 GB.

4 | EVALUATION OF CONVENTIONAL STORAGE ARCHITECTURE

For the best compatibility, we encapsulate NVM device into several virtual disks in host as shown in Figure 1, then assign them to VMs as storage devices. With this method, no modification is needed in QEMU or guest OS. Same to common file-based virtual storage device, I/O requests from VMs also need to go through the long I/O path. The difference is that, for NVM device, I/O requests will be translated into memory operations by the virtual disk module. In Section 4.1, we will introduce the details of the virtual disk module and evaluate its raw disk performance. Virtual machine's storage performance will be analyzed in Section 4.3.

4.1 | NVM-based virtual disk in host

If NVM device is used as a permanent storage device, 2 issues must be solved. The first one is the influence introduced by volatile central processing unit (CPU) cache. With CPU cache , the performance gap between CPU and memory device can be narrowed. However, it is volatile and may cause data loss when the power is inadvertently off. This will affect the data persistency in NVM device. On the other hand, CPU cache may reshuffle write commands, which is not acceptable in transaction operations. For this problem, there are 3 solutions. First, adopt write-through strategy for CPU cache or directly disable CPU cache. This method is easy to implement, but storage performance will be greatly degraded. Second, bypass the CPU cache only for write operations. This can be realized with CPU nontemporal instructions, eg, *movntdq* and *movnti*. The sequence of memory write commands can be maintained by *sfence* instruction. This method will not cause big performance degradation as shown in the work of Chen et al.[12] We adopt this method in virtual disk module. Third, adopt write-back strategy for CPU cache, but let upper software decide when to flush the cached data. This can be implemented within synchronization operations from upper software, such as *fsync()* and *fdatasync()*. Storage performance will be good if there are not so many synchronization operations. Once the synchronization operations become frequent, storage performance will be affected greatly because all the cache lines will be flushed for every synchronization operation. In addition, this method still has the risk of losing the dirty cached data when a sudden power off occurs.

The second issue is the security risk caused by wild pointer in kernel space. Because NVM is accessed in the same way with DRAM, we have to map it into a virtual address space before any access to it. If a wild pointer carelessly points to the virtual address space mapped by NVM, the stored data may be modified unintentionally. Three methods can be used to tackle this problem. First, use R/W flag in page table entry (PTE) to control the accessibility of the mapped virtual address space. Initially, NVM is mapped read-only. When writing NVM, the corresponding address is changed to writable. After write operation, the address is changed back to read-only. This method can be implemented with Linux kernel function *set_memory_rw()*. Second, use write-protect (WP) flag in control register zero (CR0) to control whether the NVM device can be written. The CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP is set to 0, supervisor-mode writes are allowed to virtual addresses with read-only access right; if CR0.WP is set to 1, they are not. Based on this hardware feature, we set CR0.WP to 0 when writing NVM device and set it back to 0 when the write operation is finished. Third, on-demand mapping. Map NVM device only when it is accessed. This method is proposed in the work of Chen et al,[12] in which each CPU core is allocated with a virtual memory page. When a process wants to write NVM, the target physical NVM page will be mapped to corresponding virtual memory page according to the CPU ID and unmapped after write operation. In addition, interrupts are disabled to prevent context switch. Thus, 1 mapped NVM page can only be accessed on 1 core. The overhead of maintaining the consistency among all cores is avoided. On the other hand, unlike the first and second methods, on-demand mapping does not need to allocate the complete mapping table for NVM. The security is better as no access to NVM device is allowed if not mapped.

To evaluate the raw performance of NVM-based virtual disk, we directly run FIO benchmark on it, bypassing the page cache. The requested block size is 4 KB, and 4 I/O threads run concurrently. Figure 2 presents the results. The *x*-axis represents the request type; SR means sequential read, SW means sequential write, RR means random read, and RW means random write. The *y*-axis represents the bandwidth of virtual disk. In this experiment, 3 protection methods are evaluated: PMAP represents the on-demand mapping method, PTE represents PTE.R/W-based method, and CR0 represents CR0.WP-based method. Furthermore, synchronous and asynchronous request mechanisms are also evaluated, which are respectively represented by PSYNC and LIBAIO. From Figure 2, we can find that **(1)** synchronous request mechanism has a better performance than asynchronous request mechanism. For instance, With CR0.WP-based protection method, the improvement is about 34% for sequential reads, 17% for sequential writes, 31% for random reads, and 10% for random writes. This is because NVM access latency is too low that the con-

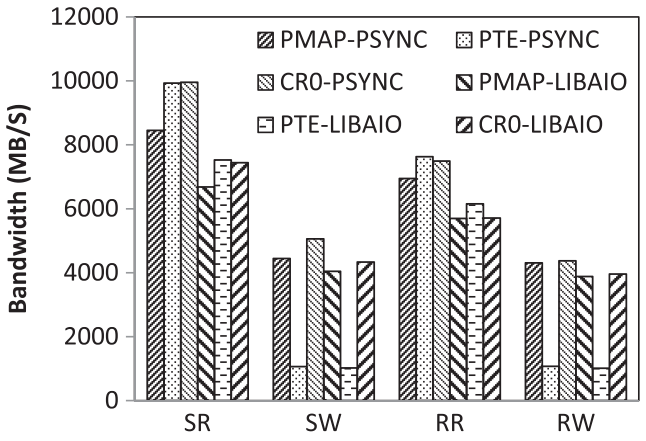

FIGURE 2 Performance of virtual disk in host with different protection mechanism. CR0 represents CR0.WP-based method; PMAP, on-demand mapping method; PTE, PTE.R/W-based method; RR, random read; RW, random write; SR, sequential read; SW, sequential write

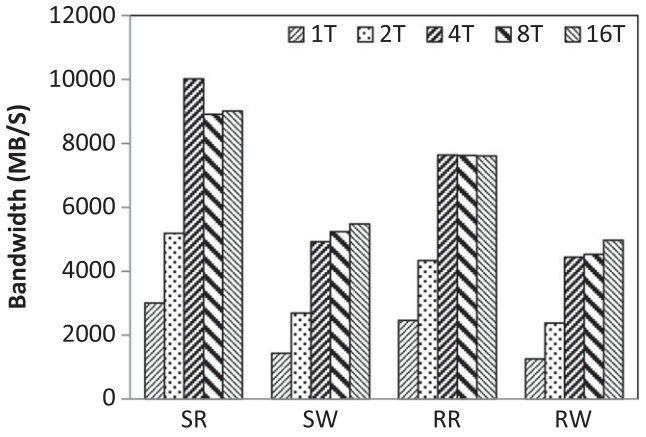

FIGURE 3 Performance of virtual disk in host with CR0-based protection mechanism. CR0 represents CR0.WP-based method; RR, random read; RW, random write; SR, sequential read; SW, sequential write

text switch overhead has a big influence on the storage performance, and asynchronous request mechanism causes more context switches than synchronous request mechanism. This phenomenon shows that, for NVM device, synchronous I/O operations may be a better choice for upper software, which is not valid for conventional HDD and SSD. **(2)** Among the 3 protection methods, CR0.WP-based method can provide the best performance in all cases. The PTE.R/W-based method has the worst write performance, because it is implemented based on kernel function *set_memory_rw()*. This function will shoot down translation lookaside buffers (TLBs) on all cores, which will cause big performance degradation. While CR0.WP-based method does not need any operation on page table, thus there is no influence on TLB. As for the on-demand mapping method, page mapping and un-mapping operations on each access will cause additional performance overhead, leading to a relatively lower read performance compared with the other 2 methods. But it only influences the accessed virtual address in TLB on the running core; no shootdown of TLBs on all cores is needed. Thus, the write performance is better than PTE.R/W-based method. **(3)** The CPU use is very high, nearly 100%. This is because in virtual disk module, I/O operations are translated into CPU intensive memory operations. This is also why when we run more than 4 I/O threads, the whole performance will not have a big improvement as shown in Figure 3. On the whole, with CR0.WP-based protection method, NVM-based virtual disk can provide excellent I/O performance, about 10 GB/s for sequential reads, 5 GB/s for sequential writes, 7.6 GB/s for random reads, and 4.5 GB/s for random writes.

4.2 | Virtual machine's storage performance

In this section, we use FIO to evaluate the performance of conventional VM storage architecture on NVM-based virtual disk device. The virtual disk is attached to VM through virtio-blk, and the page caches in guest and host are disabled to avoid their influence on storage performance. For FIO benchmark, libaio is used as the I/O engine, and the I/O depth is 32. Because the optimization technologies discussed in Section 2.2 are based on different Linux kernel and QEMU versions, we cannot deploy them all on 1 system platform. We divide the available optimization technologies into different combinations based on their supported software versions. The first combination is data-plane[11] and multi-queue,[4] the second combination is vhost-blk and bio-based virtio-blk.[6] Note that, work of Kim et al[5] is not involved in this experiment for its working source code is not available. These 2 combinations represent the available cutting-edge I/O technologies in virtualization environment, and their performance is presented in Figures 4 and 5, respectively. From the 2 figures, we can find that the combination of vhost-blk and bio-based virtio-blk can provide a much better performance than the combination of data-plane and multi-queue. This is because vhost-blk and bio-based virtio-blk focus on shortening the I/O path, while data-plane and multi-queue focus on improving the scalability of I/O request processing as depicted in Figure 4. The ideal storage I/O stack should combine all the merits of the 2 combinations, but this will need a lot of work to integrate them and this may be infeasible in some situations, eg, vhost-blk is not suitable for file-based virtual storage device.

As shown in Figure 5, with optimization technologies, VM can achieve good storage performance, about 2.4 GB/s for reads and

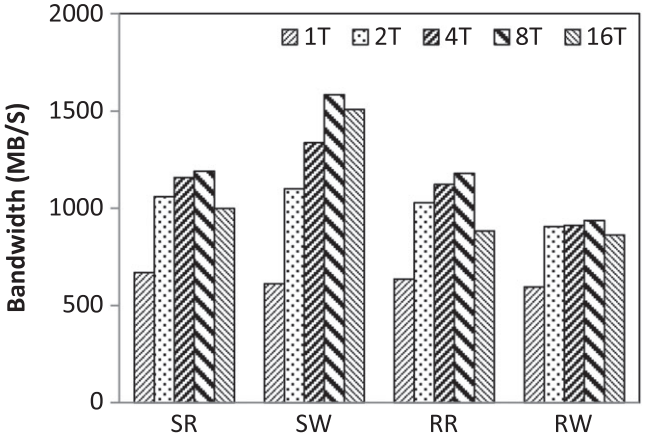

FIGURE 4 Virtual machine's storage performance with data-plane and multi-queue. RR indicates random read; RW, random write; SR, sequential read; SW, sequential write

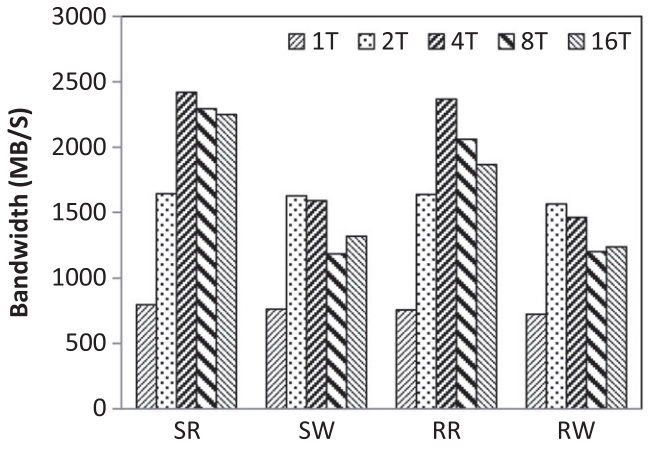

FIGURE 5 Virtual machine's storage performance with vhost-blk and bio-based virtio-blk. RR indicates random read; RW, random write; SR, sequential read; SW, sequential write

1.6 GB/s for writes. However, compared with the raw performance of NVM-based virtual disk in host as shown in Figure 3, the storage performance achieved in VM only occupies a small proportion, about 30%. To explore the details of such a large performance loss, we insert several stubs in the I/O stack as depicted by the numbers in Figure 1B. In VM, a self-made micro benchmark is used to continuously read the raw virtual disk device through system call *read()*. Page caches in guest and host are also bypassed, and the data block size is 4 KB. According to the collected statistics, we find that about 42 μs is elapsed between 1 and 12, while only about 2 μs is used between 6 and 7. The other 40 μs is spent in the middle software layers on parsing I/O requests(1 to 2 and 5 to 6), data copies, context switches (2 to 3 and 10 to 11), and asynchronous message notifications(4 to 5 and 11 to 12). This overhead may be not very important for HDD and SSD, but for NVM-based storage device, it means a lot. The performance advantage of NVM device has been greatly blocked by the inefficient, lengthy, and asynchronous VM storage I/O stack.

4.3 | Summary

Based on the above theoretical analysis and experimental results, we argue that the conventional VM storage architecture is too heavy and no longer suitable for NVM-like fast storage device. A new efficient storage architecture specified for NVM device is urgently needed.

5 | MEMORY BUS–BASED STORAGE ARCHITECTURE

Different from normal fast NVMe SSD, NVM device is byte addressable and can be directly connected to host machine by memory bus. Based on this feature, we propose a memory bus–based storage architecture for NVM device in virtualization environment. As show in Figure 6, NVM device is first assigned to VM by mapping it into VM's virtual address space as an NVM area. Then VM can deploy different storage systems on it. Compared with conventional front-/back-end driver model–based VM storage architecture, our proposed architecture has several advantages (detailed comparison is given in Table 1).

- **Shorter data access path.** This benefits from 2 aspects. The first aspect is that memory bus is closer to CPU than PCI-E or SATA interfaces. When data is stored in NVM device, we can directly access it through memory bus, while for normal HDD and SSD, we have to first move the data into memory and then process it. The second aspect is that memory requests can be directly processed in guest OS, without trapping into VMM. While for traditional VM storage architecture, VMM is involved in the I/O path as shown in Figure 1. The shorter the data access path, the lower the performance overhead.
- **Better request processing concurrency.** Nowadays, memory controller is usually integrated into processor chip and has very high bandwidth. As the number of CPU cores grows, more integrated memory controllers will be provided. Applications in VM can take the advantage of multi-core platform to fully explore NVM device's high storage performance. On the other hand, memory interface naturally supports assigning different parts of NVM device to different VMs and letting VMs manage the assigned parts individually without any involvement of VMM. Viewed from this point, NVM device is a fine-grained SR-IOV device.
- **Better use flexibility.** In conventional storage architecture, NVM device is provided to VM as an I/O block device. Any application that wants to use it must go through the lengthy I/O path. In other words, NVM device can only be used as a block device in VM. While in the memory bus–based architecture, NVM device is provided to VM with the raw memory interface. Applications can flexibly use it in different formats. As shown in Figure 6, we can encapsulate it into a virtual disk as done in Section 4.1 or deploy NVM-specified file systems on it. In addition, it can be directly used by programmers as a persistent memory library. This use flexibility makes it easier to design an efficient application-oriented software layer for NVM device. For comparison with conventional storage architecture, we encapsulate NVM device into a virtual disk in VM.

To implement the proposed storage architecture, 2 problems must be solved. The first problem is how to assign NVM device to different VMs. As depicted by the sub-figure on the lower right of Figure 6, NVM device is divided into 3 parts: global metadata block, NVM allocation table, and data area. Global metadata block contains the global information about the NVM device, such as device size, the number of allocated child devices, the block allocation bitmap, and the starting physical address of NVM allocation table. Nonvolatile memory allocation table contains the information of allocated child devices, such as child device ID, child device size, and page table pointer that points to the address mapping table. Note that the address mapping table is also allocated from data area. This allocation mechanism makes the capacity adjustment of child device easier. When a child NVM device is used by a VM, it will be mapped into the VM's virtual address space, serving as an NVM area. Furthermore, if the allocated blocks are physically adjacent, large page can be used to reduce the metadata overhead and improve TLB hit ratio.

The second problem is how to encapsulate the NVM in VM. For this issue, we adopt the same method discussed in Section 4.1. Nonvolatile memory assigned to VM is also divided into 3 parts: super block, virtual disk mapping table, and data area. Super block contains the global

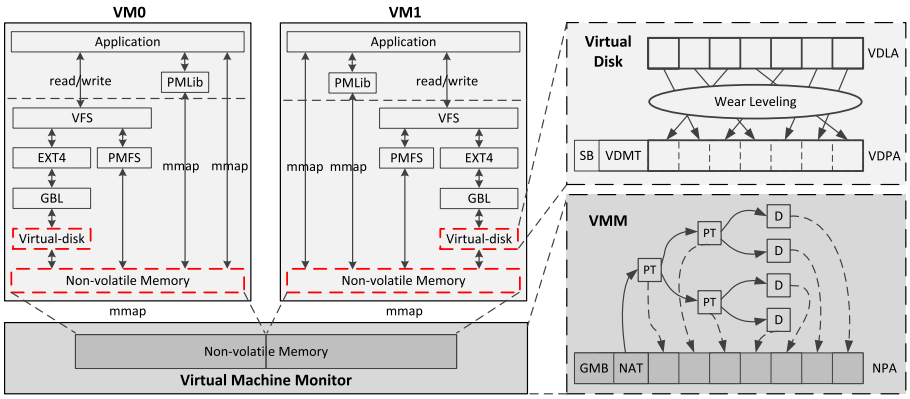

FIGURE 6 Memory bus–based storage architecture in virtualization environment. GMB indicates global metadata block; NAT, NVM allocation table; SB, super block; VDLA, virtual disk logical address; VDPA, virtual disk physical address; VFS, virtual file system; VM, virtual machine; VMM, virtual machine monitor

TABLE 1 The comparison of conventional storage architecture and MBSA

	Conventional Storage Architecture	MBSA
Compatibility	Good	Poor
Data access path	Long	Short
Request processing concurrency	Poor	Good
Usage flexibility	Poor	Good

Abbreviation: MBSA, memory bus–based storage architecture.

information about the virtual disk, such as disk size, supported block size, and the starting address of VMDT and data area. Virtual disk mapping table is used to map the virtual disk logical address to virtual disk physical address. In addition, a wear-leveling algorithm is proposed to prolong NVM device's lifespan.

5.1 | NVM-based virtual disk in guest

To evaluate the storage performance of virtual disk in VM, we directly deploy the source code used in Section 4.1 without any modification. FIO is used as the benchmark. The accessed block size is 4 KB, and synchronous I/O engine is used. Also, page cache in VM is bypassed. Virtual machine is configured with 4 VCPUs, and the NVM is 8 GB. Figure 7 presents the performance results. From this figure, we can find that on-demand mapping–based protection method and PTE.R/W-based protection method have similar performance characteristics to that in host. But for CR0.WP-based protection method, the situation is very different. In host, CR0.WP-based method can provide the best read and write performance, while in VM write performance is much worse than on-demand mapping–based method. With further investigation, we find that this is caused by 2 factors. First, CR0 can only be accessed at privilege level 0, and in VM, any access (read or write) to CR0 will cause VM-Exit. The CR0.WP-based protection method needs to modify the WP bit in CR0; thus, more VM-Exits will be caused than the other 2 methods. But this can be optimized by setting the *guset/host masks* in *VM-execution control fields*. The *guest/host masks* controls whether 1 bit in CR0 can be directly modified by guest OS. If 1 bit in *guest/host masks* is set to 0, then the guest OS can directly modify the corresponding bit in

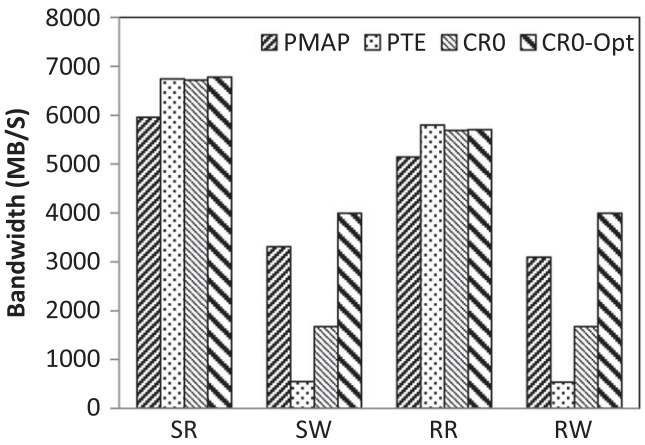

FIGURE 7 Performance of virtual disk in guest with different protection mechanism. CR0 represents CR0.WP-based method; PMAP, on-demand mapping method; PTE, PTE.R/W-based method; RR, random read; RW, random write; SR, sequential read; SW, sequential write

CR0 without any VM-Exits. Second, in the original version of virtual disk module, *movntdq* is used to update NVM, but this instruction uses XMM registers. To avoid interfering user mode process, XMM registers must be saved before this instruction by kernel function *kernel_fpu_begin()*. This will cause modification of task-switched bit in CR0, and the save operation may be very slow if the user mode process is using FPU or XMM registers. To tackle this issue, we change to use *movnti* instruction to write data to NVM. The difference is *movnti* only operate general purpose registers, no need to store the FPU or XMM registers. The drawback is that *movnti* moves less bytes than *movntdq* at a time.

We apply the optimizations discussed earlier to CR0.WP-based protection method, and its performance is depicted by CR0-Opt in Figure 7. As shown in Figure 7, among the 4 methods, CR0-Opt can provide the best performance, about 6.8 GB/s for sequential reads, 4 GB/s for sequential writes, 5.7 GB/s for random reads, and 4 GB/s for random writes. The performance is about 2.55X that of the conventional storage architecture as shown in Figures 4 and 5. However, there is still a gap when compared with the performance in host, about 20% on average. We will investigate it as our future work.

5.2 | Wear-leveling algorithm

Algorithm 1: Wear-leveling algorithm

```
Input: block p to be updated, threshhold T
if p in E1 then
    update_block_in_E1(p);
if p in E2 then
    if E1 is empty then
        q = L1.tail;
        swap(p, q);
        update_block_in_L1(p);
        move q to the head of L2;
    else
        q = E1.tail;
        swap(p, q);
        update_block_in_E1(p);
        move q to the head of E2;
if p in L1 then
    update_block_in_L1(p);
if p in L2 then
    if p.count < T then
        move p to the head of L2;
    else
        if E1 is empty then
            q = L1.tail;
            swap(p, q);
            update_block_in_L1(p);
            move q to the head of L2;
        else
            q = E1.tail;
            swap(p, q);
            update_block_in_E1(p);
            move q to the head of E2;
update p;
p.count ++;
check_if_upgrade();
Subroutine update_block_in_L1(p)
if p.count < T then
    move p to the head of L1;
else
    p.count -= T;
    move p to the tail of L2;
Subroutine update_block_in_E1(p)
if p.count = 0 then
    move p to the tail of L1;
else
    p.count -= T;
    move p to the tail of L2;
Subroutine check_if_upgrade()
if L1 is empty AND E1 is empty then
    exchange L1 and L2, E1 and E2;
```

Our proposed wear-leveling algorithm is depicted by Algorithm 1. In this algorithm, we use 4 lists: L_1, L_2, E_1, and E_2. Valid blocks are linked into L_1 and L_2, and unused blocks are linked into E_1 and E_2. Blocks in L_1 and E_1 are younger (written less) than blocks in L_2 and E_2. A predefined threshold T is used to determine whether a block can be continuously updated. Initially, all the blocks are unused and linked into E_1. Then they will be moved into L_1, L_2, or E_2 according to Algorithm ??. When all the data blocks in L_1 and E_1 are moved to L_2 and E_2, we exchange L_1 with L_2 and E_1 with E_2. Then L_2 and E_2 will be empty and used to store following older blocks. This rotating mechanism will make the usable continuous updates not less than T. In addition, we use Least Recently Used–like mechanism to manage the lists; frequently updated blocks are inserted in the head position. Then the rarely updated blocks will be moved to the tail gradually. When we need to do wear-leveling, the blocks in E_1 will be the first priority (lines 9-13 and lines 25-29). If E_1 is empty, then the tail block in L_1 will be selected (lines 4-8 and lines 20-24). This algorithm only contains simple list operations and will not generate big performance overhead.

To evaluate the efficiency of this algorithm, we deploy it together with CR0-Opt–based protection method in virtual disk module in VM. FIO is configured to perform random writes to different parts of the virtual disk for different durations. The address ranges and corresponding durations are 30 minutes for 0 to 1 GB, 10 minutes for 1 to 2 GB,

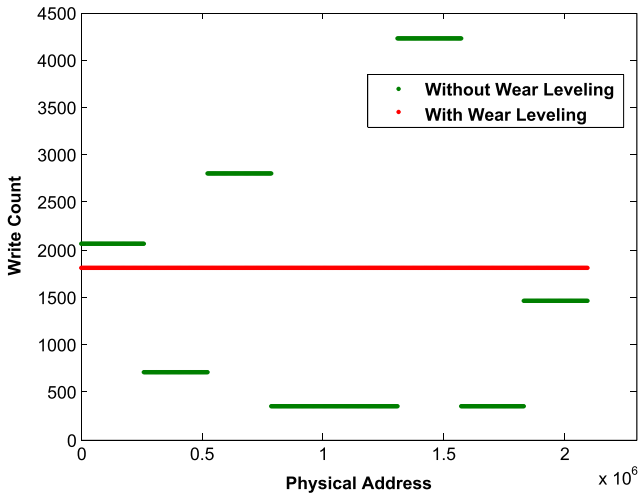

FIGURE 8 Distribution of write counts with/without wear-leveling

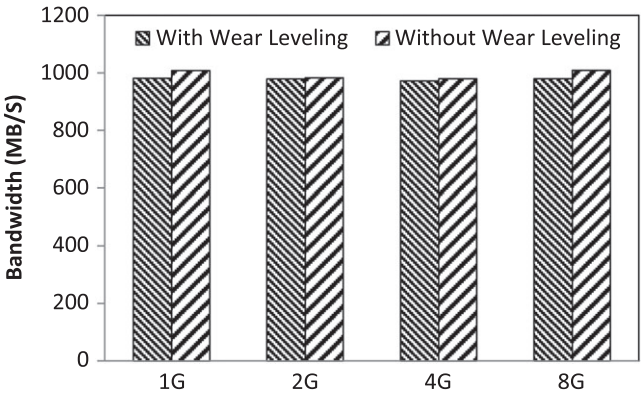

FIGURE 9 Performance of virtual disk with/without wear-leveling

40 minutes for 2 to 3 GB, 10 minutes for 3 to 5 GB, 60 minutes for 5 to 6 GB, 5 minutes for 6 to 7 GB, and 20 minutes for 7 to 8 GB. Figure 8 shows the distribution of write counts with and without wear-leveling. From this figure, we can find that if wear-leveling is not applied in virtual disk module, the write count distribution is very uneven, some blocks may be written heavily, while some blocks are only written several times. With the proposed wear-leveling algorithm, the write operations will be distributed very evenly on the virtual disk as shown by the red line in Figure 8.

To evaluate the performance overhead of this algorithm, we configure FIO to perform random writes to the virtual disk with different data set size, and the duration is 30 minutes. As depicted by Figure 9, the caused performance overhead is very low, only about 2.6% for 1 GB data set, 0.4% for 2 GB data set, 0.8% for 4 GB data set, and 2.8% for 8 GB data set. Note that in all experiments in this section, the threshold in Algorithm 1 is set to 2. If we increase the threshold, then less swap operations (eg, lines 6 and 22) will be needed for wear-leveling, and the performance overhead will be decreased accordingly. But the evenness of the write distribution will be worse. Anyhow, as the result shows, even when the threshold is set to 2, the performance with wear-leveling is still very good. This high-performance benefits from the simple but efficient wear-leveling algorithm.

6 | CONCLUSION AND FUTURE WORK

Compared with HDD and SSD, NVM device can provide much better storage performance. It has great potential in big data processing. But conventional storage architectures are designed for spinning HDD, no longer suitable for emerging fast storage devices such as NVMe SSD and NVM device. The situation is even worse in virtualization environment where the storage I/O stack is usually longer than that in non-virtualization environment. Thus, providing a high-performance storage architecture for these fast storage devices is an urgent task. Toward this objective, we reconsider conventional VM's storage I/O stack and find that NVM device's high performance has been greatly limited. Virtual machine can only explore 30% of the original performance. Inspired by the byte addressability characteristic of NVM device, we propose to discard the asynchronous interrupt mechanism-based storage architecture, directly assign NVM device to VMs through memory bus. We call this new architecture MBSA. Memory bus–based storage architecture can greatly release NVM device's performance and also increase its use flexibility. Furthermore, an efficient wear-leveling algorithm is designed to prolong NVM device's lifespan. The experimental results show that it has an excellent effect on balancing the write operations to NVM device and the caused overhead is very low.

Even though the memory bus–based storage architecture has many merits, it also faces some challenges. In this architecture, conventional I/O intensive storage workloads are changed to memory/CPU intensive workloads. Thus, storage performance is very sensitive to processor load, and providing the quality-of-service guarantee of storage performance will be more difficult. We will explore this problem in our future work.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their hard work. We also appreciate the constructive comments and suggestions from the members of ARC Lab in Zhejiang University. The authors extend their appreciation to the Deanship of Scientific Research at King Saud University, Riyadh, Saudi Arabia, for funding this work through the research group project no RGP-318.

REFERENCES

1. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*. IEEE, Philadelphia, PA; 2015:171–172.
2. Breakthrough nonvolatile memory technology 2015. Available from: https://www.micron.com/about/emerging-technologies/3d-xpoint-technology, Acessed March 18, 2016.
3. Bjørling M, Axboe J, Nellans D, Bonnet P. Linux block io: introducing multi-queue ssd access on multi-core systems. *Proceedings of the 6th International Systems and Storage Conference*. ACM, Haifa, Israel; 2013:22.
4. Lei M. Virtio-blk multi-queue conversion and QEMU optimization. KVM forum; 2014.
5. Kim TY, Kang DH, Lee D. Improving performance by bridging the semantic gap between multi-queue ssd and I/O virtualization framework. *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Santa Clara California; 2015:1–11.
6. He A. Virtio-blk performance improvement. KVM forum; 2012.
7. Li D, Jin H, Liao X, Zhang Y, Zhou B. Improving disk I/O performance in a virtualized system. *J Comput Syst Sci*. 2013;79(2): 187–200.
8. Song X, Yang J, Chen H. Architecting flash-based solid-state drive for high-performance I/O virtualization. *Comput Archit Lett*. 2014;13(2): 61–64.
9. Tan H, Li C, He Z, Li K, Hwang K. VMCD: a virtual multi-channel disk I/O scheduling method for virtual machines; 2015.
10. Ota T, Okamoto S. Using I/O schedulers to reduce I/O load in virtualization environments. *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA)*. IEEE, Biopolis, Singapore; 2011:59–62.
11. Hajnoczi S. Towards multi-threaded device emulation in QEMU. KVM forum; 2014.
12. Chen F, Mesnier MP, Hahn S. A protected block device for persistent memory. *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Santa Clara, California; 2014:1–12.
13. Qureshi MK, Srinivasan V, Rivers JA. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Comput Archit News*. 2009;37(3):24–33.
14. Zhou P, Zhao B, Yang J, Zhang Y. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH computer architecture news*, vol. 37. ACM, Austin, Texas; 2009:14–23.
15. Qureshi MK, Franceschini MM, Lastras-Montaño LA, Karidis JP. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. *ACM SIGARCH Computer Architecture News*, vol. 38. ACM, Saint-Malo; 2010:153–162.
16. Mogul JC, Argollo E, Shah MA, Faraboschi P. Operating system support for NVM+ DRAM hybrid main memory. *HotOS*. 2009:14–14.
17. Jung JY, Cho S. Memorage: emerging persistent ram based malleable main memory and storage architecture. *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, Eugene, Oregon; 2013:115–126.
18. Jung J, Won Y. NVRAMdisk: a transactional block device driver for ile RAM.
19. Swanson S, Caulfield AM. Refactor, reduce, recycle: restructuring the I/O stack for the future of storage. *Comput*. 2013;8: 52–59.

20. Condit J, Nightingale EB, Frost C, Ipek E, Lee B, Burger D, Coetzee D. Better I/O through byte-addressable, persistent memory. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, Montana, USA; 2009:133–146.
21. Wu X, Qiu S, Narasimha Reddy A. SCMFS: a file system for storage class memory and its extensions. *ACM Trans Storage (TOS)*. 2013;9(3):1–11.
22. Volos H, Nalli S, Panneerselvam S, Varadarajan V, Saxena P, Swift MM. Aerie: flexible file-system interfaces to storage-class memory. *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Amsterdam; 2014:1–14.
23. Dulloor SR, Kumar S, Keshavamurthy A, Lantz P, Reddy D, Sankaran R, Jackson J. System software for persistent memory. *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Amsterdam; 2014:1–15.
24. Lee E, Yoo SH, Bahn H. Design and implementation of a journaling file system for phase-change memory. *IEEE Trans Comput*. 2015;64(5):1349–1360.
25. Coburn J, Caulfield AM, Akel A, Grupp LM, Gupta RK, Jhala R, Swanson S. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, vol. 39. ACM, California; 2011:105–118.
26. Volos H, Tack AJ, Swift MM. Mnemosyne: lightweight persistent memory. *ACM SIGPLAN Notices*. 2011;46(3):91–104.
27. Quick D, Martini B, Choo KKR. *Cloud Storage Forensics*. Massachusetts, United States: Syngress; 2013.
28. Quick D, Choo KKR. Digital droplets: microsoft skydrive forensic data remnants. *Future Gener Comput Syst*. 2013;29(6):1378–1394.
29. Martini B, Choo KKR. Distributed filesystem forensics: Xtreemfs as a case study. *Digital Invest*. 2014;11(4):295–313.
30. Russell R. Virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Syst Rev*. 2008;42(5):95–103.
31. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *ACM SIGOPS Operating Syst Rev*. 2003;37(5):164–177.
32. Nvm express 2014. Available from: http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf. Accessed March 20, 2016.
33. Gordon A, Amit N, Har'El N, Ben-Yehuda M, Landau A, Schuster A, Tsafrir D. Eli: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices*. 2012;47(4):411–422.
34. Apic virtualization performance testing and iozone 2013. Available from: https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone. Accessed March 20, 2016.

How to cite this article: Chen X, Chen W, Lu Z, Zhang Y, Chang R, Hassan MM, Alelaiwi A, Xiang Y. MBSA: A lightweight and flexible storage architecture for virtual machines. *Concurrency Computat: Pract Exper*. 2017;e4028. https://doi.org/10.1002/cpe.4028