IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 31, 2018, accepted December 6, 2018, date of publication December 12, 2018, date of current version January 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2886473

Improving Spatial Locality in Virtual Machine for Flash Storage

SUNGGON KIM[1], HYEONSANG EOM[1], AND YONGSEOK SON[2]

[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[2]School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) under Grant 2018R1C1B5085640, Grant 2017R1A2B4004513, and Grant 2016M3C4A7952587, in part by the Institute for the Information and Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIP) under Grant R0190-16-2012, and in part by the BK21 Plus for Pioneers in Innovative Computing (Department of Computer Science and Engineering, SNU) funded by NRF under Grant 21A20151113068.

ABSTRACT Flash-based solid-state drives (SSD) are being widely adopted in virtualized environments to improve the I/O performance due to their low latency and high throughput compared with existing hard disk drives. In the virtualized environment, the performance of SSDs fluctuates significantly according to the I/O patterns from the virtual machine. For example, random writes have a significantly negative impact on the performance of SSDs compared with sequential writes due to the characteristics of SSDs. In this paper, we propose an address reshaping technique for SSDs in the virtualization layer to improve the spatial locality and the performance of random writes. Our scheme transforms random write requests into sequential write requests in the virtualization layer and thus enables the virtualization layer to issue the transformed sequential requests to SSDs. The experimental results show that the optimized scheme improves the performance by up to 97% compared with the existing scheme under random write workloads.

INDEX TERMS Cloud computing, operating system, solid-state drive, virtualization.

I. INTRODUCTION

Cloud computing is becoming widely adapted to the industry as it significantly reduces IT costs and improves the resource efficiency. Cloud service providers such as Amazon EC2 [1] and private cloud platform such as Openstack [2] use both full virtualization such as Kernel-based Virtual Machine (KVM) [3] and Xen [4], and container-based virtualization such as Docker [5] and Linux Containers (LXC) [6] to fully exploit limited host hardware resources. These virtualization techniques allow cloud service providers to efficiently manage resources and provide dynamic scalability. From the perspective of cloud users, virtualization provides the illusion of an isolated environment even though multiple VMs are running on the same hardware [7].

Although the efficient resource management and the illusion of an isolated environment are provided by the existing virtualization techniques, a significant amount of overhead can occur due to the presence of an additional abstraction layer. For example, in the perspective of I/O operations, the I/O performance of a VM is theoretically bound to the performance of the host and is far below the host performance under real workloads [8], [9]. This is because virtualization introduces duplicate layers of the file system, block layer, and device driver to the host, as well as a new hypervisor/container layer which performs virtualization using the host resources. Additionally, to communicate between the emulated device and the host, QEMU-KVM uses virtio [10] device driver which uses an in-memory data structure for request handling. QEMU-KVM receives the request containing a *virtual block address* (VBA) and translates the VBA into an *image block address* (IBA) which is the file offset that can be issued to the host OS via system calls such as `pwrite`. However, this device virtualization can incur the management of the data structure and increases communication overhead between the VM and host [11].

To improve the I/O performance in virtualization, fast storage devices such as flash-based solid-state drives (SSDs) have attracted attention from both industry and academia since they provide low latency and high bandwidth compared with the existing hard disk drives (HDDs) [12]. The cost-per-bit of flash memory has continued to fall owing to semiconductor technology scaling and 3-D vertical NAND flash

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

technology [13]. Meanwhile, due to the characteristics of the flash memory, performance imbalances can occur according to the characteristic of I/O workloads. For example, in flash memory, erase-before-write constraint does not permit in-place updates of a page inside a flash block. When a page must be updated, a flash translation layer (FTL) inside SSD writes new data to a clean page and invalidates the old page. When SSD has an insufficient number of clean pages, garbage collection in the FTL reclaims invalid pages and creates free blocks. During the garbage collection, all valid pages in a victim block which need to be erased to create a free block have to be copied to another block. This internal copy operation generates large I/O amplification and the performance of SSD is eventually limited by the garbage collection performance of FTL [14].

Previous works [14]–[19] have reported that the performance of random writes drops by more than an order of magnitude compared with that of sequential writes. This is because sequential write workloads are more likely to generate a small number of completely invalid blocks, while random write workloads generate a large number of blocks with a small number of invalid pages. Thus, random writes increase the garbage collection overhead which induces many internal page copy operations compared with sequential writes.

To address these issues, previous studies [19], [20] optimized the existing storage stack considering the characteristics of SSDs [19], [20]. SHRD [20] transforms random write requests into sequential write requests in the host block device driver by assigning the address space of a reserved log area in the SSD. F2FS [19] builds on append-only logging to transform random writes to sequential writes. Our study is in line with these previous studies [19], [20] in terms of improving the random write performance of SSDs by remapping them into sequential writes. In contrast, our study focuses on optimizing the I/O performance in the virtualized environment instead of a host system.

In this paper, we propose an address reshaping technique for the virtualization to improve the I/O performance of SSDs. Our main scheme reshapes random write requests issued by the applications and the file system from the VM to sequential write requests to the host. To do this, we propose a metadata/data checker in the file system inside the VM and a sequentializer in QEMU-KVM. The metadata/data checker classifies the requests as metadata or data to manage the metadata and data independently for more efficiency. The sequentializer transforms random write requests from the VM into sequential write requests by remapping the IBA of the metadata and data requests. The sequentializer stores pairs of the original and transformed IBA for the write request in remapping tables. In addition, it also translates the original IBA to the transformed IBA for read requests, ensuring the correct read operation. The optimized virtualized system can transform random write requests from the VM to sequential write operations in the host.

We apply our optimizations to EXT4 [21] file system in the VM and QEMU-KVM virtualization layer. We evaluate our optimized system using a widely used SATA-based SSD, Samsung 850 pro and various file systems in the host, such as EXT4 [21], XFS [22], and F2FS [19]. Our experimental result shows that our system improves the performance of random writes by up to 97% compared with the existing system, which achieves similar performance of sequential writes. To the best of our knowledge, this is the first study that proposes a reshaping technique in virtualization for SSD to improve the random write performance.

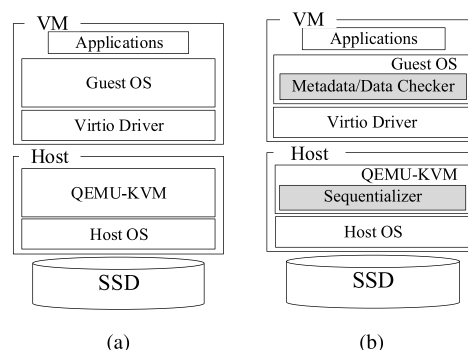

FIGURE 1. Overall architecture of existing and proposed virtualization. (a) Existing virtualization. (b) Proposed virtualization.

Our contributions are as follows:

- We analyze the I/O stack in the existing virtualization.
- We design and implement an address reshaping technique for SSD in the virtualized environment.
- We demonstrate that our optimized system can improve the performance of random writes similar to that of sequential writes.

The rest of this paper is organized as follows: Section II describes the background and motivation. Section III presents the design and implementation of the proposed scheme. Section IV shows the experimental results. Section V discusses the related work. Section VI concludes the paper.

II. BACKGROUND AND MOTIVATION

A. IO OPERATIONS IN KVM/QEMU

Kernel-based Virtual Machine (KVM) [3] is a virtualization solution for Linux included in the Linux mainline. KVM operates at the user level of the host kernel and as a user application from the perspective of the host. KVM is often used with QEMU [23] which emulates a block device in the user space and a combination of the two are commonly deployed by the name QEMU-KVM.

Figure 1 (a) shows the overall architecture of QEMU-KVM. When I/O requests are issued from applications inside the VM, the requests go through the guest OS. When the requests arrive at the VM device driver in QEMU-KVM, virtio [10] device driver is used to handle requests for the emulated device. Virtio maintains an in-memory ring buffer which resides in a shared memory region of the host and the VM and transfers the requests from the VM to the host. The requests get inserted into the ring buffer until the ring buffer is full or for a certain period after the first

request is inserted. Then, the VM ends its execution (VM exit) and yields the CPU to the host. At the host, QEMU-KVM checks the memory region used by virtio for any requests from the VM. If a request exists, QEMU-KVM pops the I/O requests from the memory to issue them. Before issuing the collected requests to the host, the requests are checked for merge opportunities. If the requests are contiguous in terms of their spatial location, QEMU-KVM merges the contiguous requests into a large request. After the merge operation, the requests are issued using the configured I/O library such as the default `pwrite` system call or `io_submit` system call from libaio [24], to the host OS like requests from a normal user process. Then, the request goes through a typical write path in the host OS such as the file system, block layer, device driver, and storage.

B. LIMITATIONS OF EXISTING IMAGE FORMATS

In QEMU-KVM, the QCOW2 [25] image format is widely used to provide functionalities for the VM such as snapshots. Since QEMU-KVM emulates the device, a block address from the VM is translated multiple times before the request is issued via system calls. When the request first arrives at QEMU-KVM, it contains a *virtual block address* (VBA). This is the logical block number for the emulated block device, which is a VM image file in the host. The VBA is translated into an *image block address* (IBA) which specifies the offset in the image file [25]–[27]. Since a VM image is a simple sequence of bytes stored in a file from the perspective of the host, the image layout of the VM is unknown to the host and can only be found inside the VM such as in the file system and block layer of the VM. Thus, modifications or optimizations regarding the image and I/O requests of VM have to consider the internals of the image format and the virtualization layer.

C. SSD CHARACTERISTIC AND PERFORMANCE IMBALANCE

Flash-based SSDs differ from existing HDDs in that in-place overwrite is not possible due to the characteristics of flash. In case of page update, the existing page gets invalidated and the data is written into a free page, resulting in an out-of-place update. This characteristic of flash storage creates a high number of invalid pages during the update operations. These invalid pages have to be erased and reclaimed by garbage collection from FTL inside SSD for future use. However, since flash can only be erased in a granularity of blocks, valid pages in a target block must be copied into free pages before erasing the block, creating many internal copy operations and amplifying the number of writes.

Many previous studies [28]–[30] reduced the garbage collection overhead and write amplification by considering locality inside a block. These works are focused on detecting sequential write requests from random write requests since random write requests are more likely to generate many blocks with a small number of invalid pages compared to sequential write requests. This characteristic of random write requests reduces the performance by more than one order of magnitude compared with sequential writes due to the increased garage collection overhead and internal fragmentation [14], [31]. Since random write requests comprise about 70% of server storage workloads [33], the I/O performance of the server is greatly impacted by random write requests.

Even though internal FTL optimizations can mitigate the overhead of random write requests, the performance of random writes, especially across in large address space, can be far less than that of sequential writes due to the limited capacity of write buffer inside SSDs [17], [18], [32]. Thus, along with optimizations in the FTL layer, optimizations in the host are also necessary to fully exploit the performance of SSDs.

To reduce the number of random write requests, there have been many studies to design a copy-on-write (COW) file system since the HDD era since random writes increase the seek time and reduce batch opportunity compared to sequential writes [34], [35]. With the introduction of SSDs, COW schemes have become more popular due to the characteristics of SSDs and the impact of random writes is worse in SSDs [14]. By changing random writes into sequential writes, the number of invalid blocks, and consequently the garbage collection overhead is reduced. However, the remapping techniques for both HDDs and SSDs in previous studies [19], [20] were designed and implemented at the file system or even at the hardware level since detailed information on the logical and physical block mapping is needed to remap and manage the mapping information. Meanwhile, in the virtualized environment, a new approach which can support different system configurations regardless of the underlying systems or hardware is necessary to support various needs from multi-tenants.

III. DESIGN AND IMPLEMENTATION

In this section, we explain the design and implementation of our proposed scheme for virtualization to improve the I/O performance of SSDs.

A. OVERALL ARCHITECTURE

To improve the random write performance of SSDs, we provide an efficient address reshaping technique for virtualization. To do this, as shown in Figure 1(b), we devise two components, namely a metadata/data checker in the file system inside the VM and a sequentializer in QEMU-KVM. These components transform random write requests to sequential write requests with a minor overhead.

1) METADATA/DATA CHECKER

We devise a metadata/data checker which classifies the requests into metadata and data requests. By classifying metadata and data requests separately, the metadata and data requests are not affected by each other, increasing the efficiency of the reshaping technique [26]. For example, the proposed reshaping technique maintains data structures (e.g., remapping table) to record pairs of the original IBA (oIBA) and transformed IBA (tIBA). By isolating metadata and data

requests through the checker, we can devise separate data structure for each metadata and data. Thus, this strategy can reduce the total number of entries in the data structure to be searched or managed. Consequently, the checker enables identification of the metadata and data requests transferred from the VM at the host (e.g., QEMU-KVM), which reduces the metadata and data management overhead.

2) SEQUENTIALIZER

To improve the spatial locality of SSDs, we devise a sequentializer which transforms random requests (random IBAs) to sequential requests (sequential IBAs) in QEMU-KVM. Thus, this sequentializer enables the issuing of requests with contiguous IBA to the host OS. To do this, we adopt a remapping data structure such as a table and store pairs of an original and transformed IBA. This remapping table allows the seamless transformation of the IBAs, both original to transformed and vice-versa. For example, when a request arrives in QEMU-KVM, the sequentializer allocates a new IBA in the free space of the VM image in a sequential manner for the request. Then, the sequentializer changes the original IBA of the request to a transformed IBA and stores the pair in the remapping table. It then issues and completes the request with the transformed IBA to the host. In addition, with the help of the metadata/data checker, the sequentializer manages the IBAs of metadata and data separately using separate remapping table. Thus, the sequentializer sequentializes metadata and data to each metadata and data region in the VM image file, respectively.

B. DESIGN

1) RESHAPING OPERATIONS

Figure 2 shows the overall procedure of the proposed address reshaping technique. As shown in the figure, when applications or the guest OS in the VM issue requests, the metadata/data checker in the guest OS file system classifies and flags the metadata and data requests. After VM exit, the requests and their flags are propagated into QEMU-KVM through the file system, block layer, and device driver in the VM.

In the case of write operations, the flag is checked at QEMU-KVM to determine whether the request is for metadata or data. Then, the sequentializer finds an available IBA in the free space in the metadata or data region. If an IBA is available, the sequentializer transforms the original IBA of the request to a sequentialized IBA. Otherwise, a cleaning operation is performed, which will be discussed later in Section III-B2. Thus, the sequentializer changes the random write requests from the VM to sequential write requests. After changing the IBA to a transformed IBA, the original IBA and the transformed IBA are recorded as a single entry in the metadata or data remapping table. Then, the sequentializer checks whether the requests can be merged.

In the existing QEMU-KVM, requests with sequential IBA are merged into a large single request, reducing the request

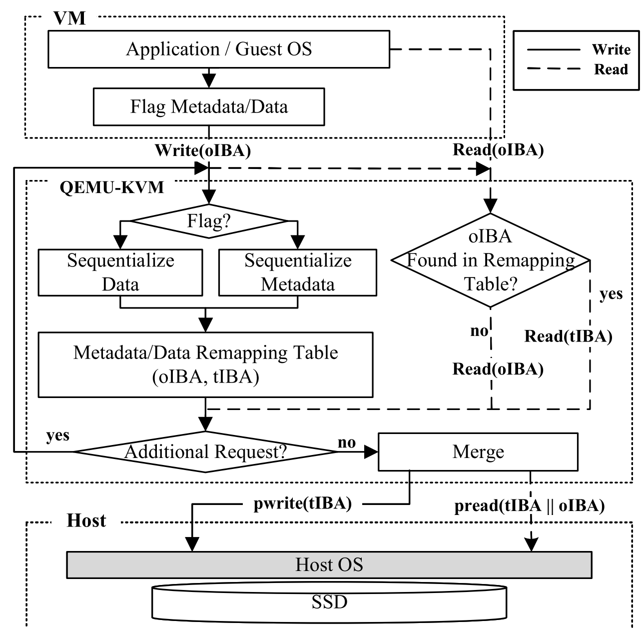

FIGURE 2. Overall procedure of proposed scheme (oIBA: original image block address, tIBA: transformed image block address).

issue overhead. With the existing merge scheme, our scheme issues at most two number of requests which are merged metadata and data request. This is because the sequentializer always creates sequential requests and separates metadata and data requests. This can also increase the performance of sequential writes as well as random writes since the sequential writes issued by multiple threads may generate a pattern of random writes to the host. Finally, all the write requests are issued to the host OS using the `pwrite` system call.

In case of the read operations, when the read request arrives at QEMU-KVM, the original IBA is searched in the remapping table. If the original IBA is found in the table, the IBA in the request is changed into a transformed IBA to read the data correctly. Finally, the read request is issued to the host OS using the `pread` system call from libaio [24].

Figure 3 describes an example of write requests from the VM and how they are handled at QEMU-KVM. In this example, there are four write requests (request 1-4) after VM exit. First, we check whether the request is for metadata or data by checking the flag. Then, the requests are transformed into either a metadata or data request. As shown in the figure, request 1, 3, and 4 are transformed into data request (`Drequest`) 1, 2, and 3, respectively, and request 2 is transformed into metadata request (`Mrequest`) 1. After then, the sequentializer transforms the discontiguous IBAs of the requests into a contiguous IBA. As shown in the figure, the IBA of data request 1, 2, and 3 are changed into 31, 32, and 33, respectively, and the IBA of metadata request 1 is changed into 12. Then, these original and transformed IBAs are recorded into metadata and data remapping table. And then, data request 1, 2, and 3 are merged to a single request. Meanwhile metadata request 1 cannot be merged since the

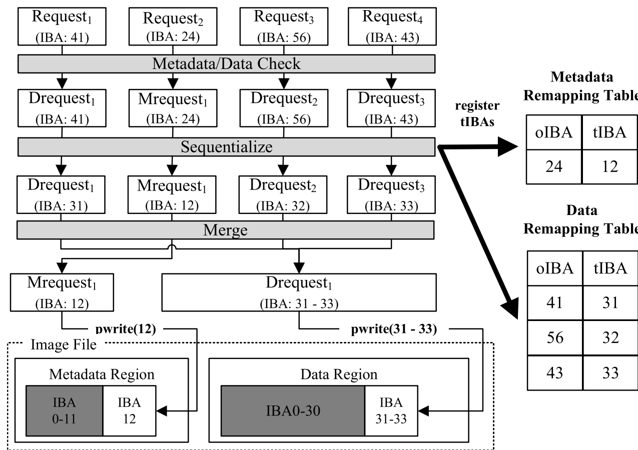

FIGURE 3. Write reshaping of proposed scheme (Mrequest: metadata request, Drequest: data request).

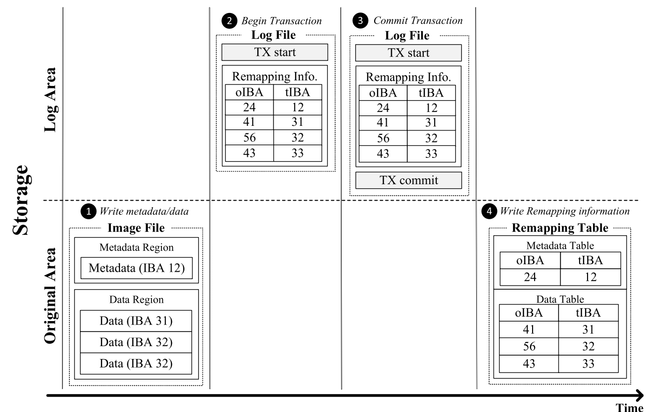

FIGURE 4. Supporting crash consistency in the proposed scheme.

metadata request is the only one in this example. Finally, metadata request 1 and data request 1 are issued to the host OS.

2) CLEANING OPERATION

When overwrite or delete operations are performed, we invalidate the previous written data by updating or removing entries in the remapping table. In case of overwrite operation, we update the previous entry associated with the overwritten data with a new tIBA. In case of delete operation, we remove the entry in the remapping table. If there is insufficient free space due to the invalidated data, a cleaning operation needs to be performed to reclaim the free space.

To reclaim the space, we use a variant of the cleaning operation used in Log-structured File System(LFS) [19], [34], [36]. When the cleaning operation starts, we first select a segment to be cleaned.[1] Then, we read the segment and determine which image blocks are still valid by checking the existence in the remapping table. And then, we start to perform a compaction. During the compaction, we compact the space by copying the data of valid image blocks to invalid or free image blocks. Then, We reflect the changed mapping information in the remapping table. Consequently, this cleaning operation reclaims the invalidated space to maintain contiguous free space.

3) CRASH CONSISTENCY AND RECOVERY

When a VM restarts (e.g., power off and VM migration), the remapping table must be reconstructed in memory to access the remapping information. To achieve this, we use a persistent red/black tree [37] for the remapping table in order to persist the remapping table. This red/black tree is an in-memory data structure and can be written in a file format to provide persistency. This allows the remapping table to be written, read, and migrated. Thus, in the event of VM restart, the remapping table can be reconstructed in memory by reading the file.

[1] Segment is a cleaning unit selected based on the total VM image size.

When sudden system failures occur while the remapping table is being written to storage, the remapping table may end up in an inconsistent state. To maintain the remapping information in storage in a consistent manner, we support crash consistency by using a variant of write-ahead logging (WAL) which is widely used in file and database systems to provide crash consistency [19], [21], [38]. To do this, after the reshaping operation, a write operation is performed in following steps:

- Both metadata and data are written from the VM to the VM image file.
- The transaction is started and this event is recorded in a log file.
- All the remapping information associated with the transaction is written to the log file.
- The transaction is committed by recording the commit mark at the end of the log file.[2]
- The remapping information is updated to the original remapping table.

Figure 4 describes an example of a write request after the reshaping operations and how the remapping information is written in a WAL fashion. From the reshaping operation, the oIBAs of the metadata and data from the VM such as 24, 41, 56, and 43 are already remapped to 12, 31, 32, and 33, respectively. In the first phase, the metadata and data are written into their regions in the image file using the tIBAs in the original area (❶). In the second phase, a transaction is started and the mapping information between oIBAs and tIBAs is written into the log area in a WAL fashion (`TX start`)(❷). After all the information associated with the transaction is written, in the third phase, the transaction is committed and this event is recorded in the log area (`TX commit`) (❸). In the final phase, the mapping information is written into the remapping table in the original area (❹).

In the event of a crash, the transaction can be either committed or not. Our scheme performs recovery operations to recover VM in a consistent state. As the first case, if the transaction is not committed, the remapping information can

[2] This commit mark is used to determine the status of the transaction in case of crashes, guaranteeing atomicity for the transaction.

be lost or partially written. Thus, we discard the transaction by removing remapping entries associated with the transaction. By doing so, the transformed addresses in the remapping entries will be not accessible and so they will be considered free. As the second case, if the transaction is committed while the transaction is not yet applied to the original area, we apply the transaction to the original area by reading the remapping entries of the transaction from the log area and write them to the original area. Consequently, our scheme can successfully recover VM in a consistent state by discarding or applying the transaction in the event of a crash.

C. IMPLEMENTATION

To detect the metadata/data and inform the virtualization layer, we implemented the metadata/data checker inside EXT4 file system [21] and added a flag inside `bio` structure from the file system, `request` structure from the block layer, and `request` structure from virtio driver. For remapping tables used in the sequentializer, we used a red/black tree for both the metadata and data remapping table since the red/black tree has the search time of $O(\log N)$ which is low compared to other trees [20]. Modifications for EXT4 in VM and QEMU-KVM are less than about 500 lines in total which demonstrate that our scheme leads to small modifications to the VM kernel and QEMU-KVM. Note the modifications are limited to EXT4 file system inside the VM and QEMU-KVM, and so our scheme does not require the modification of file systems or kernel in the host.

IV. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETUP

For evaluation, we used Intel i7-4790 (3.6 GHz) with four physical cores and eight cores with hyper-threading, 8 GiB of memory, SATA3 interface, and Linux 4.4.0. For the storage device, we used 850 pro SATA SSD developed by Samsung. It has a capacity of 256 GiB and is widely used for data centers and consumer PCs. To verify our scheme in enterprise-class NVMe drive, we used P3700 NVMe SSD developed by Intel.S For virtualization, we used QEMU-KVM 2.5.0 for the baseline experiment and implemented our proposed scheme as well. We deployed a single VM with 8 virtual cores, 8 GiB of memory and Linux 4.4.0, utilizing the full hardware resources available in the host. We use EXT4 [21] for the guest file system, and EXT4, XFS [22], and F2FS [19] for the host file system.

We compared the performance of bare-metal which is EXT4 without virtualization, existing QEMU-KVM with modified EXT4 in the VM and various file systems in the host (E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS)), and optimized QEMU-KVM with various file systems (O-QEMU(EXT4), O-QEMU(XFS), and O-QEMU(F2FS)). We choose these file systems since EXT4 and XFS are widely used in data centers and consumer PCs, and F2FS is a log-structured file system for flash-based SSDs. For the I/O engine of QEMU-KVM, we used libaio engine [24] which is the fastest among other engines provided by QEMU-KVM. For the workloads, we used FIO benchmark [39] for a microbenchmark and fileserver workload included in filebench benchmark [40] for a macrobenchmark. All experimental results are the average value of five runs.

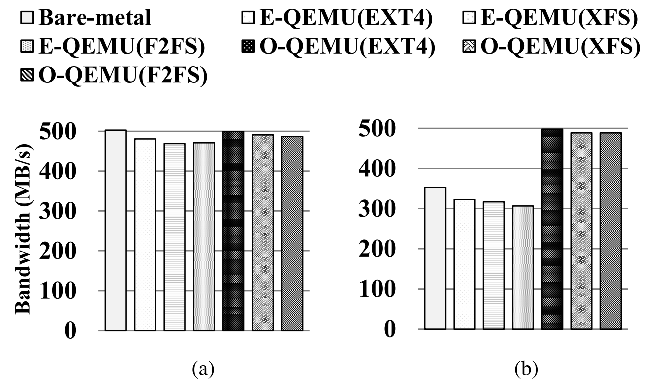

FIGURE 5. FIO bandwidth with 8 threads, 1 GiB file size, 4 KiB request size, and I/O queue depth of 64. (a) Sequential Write. (b) Random Write.

B. FIO BENCHMARK

1) BANDWIDTH

For microbenchmark, we ran FIO benchmark [39] performing sequential and random writes with identical configurations. We configured FIO benchmark to issue 8 GiB random write operations using 8 threads (1 Gib per thread), 4 KiB request size, and direct write with 64 I/O queue depth.

In case of sequential writes, as shown in Figure 5(a), the optimized QEMU improves the performance by 3.7%, 4.7%, and 2% compared with E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. Also, the performance of O-QEMU is almost similar to the performance of bare-metal. This is due to the increased merge opportunity since the proposed scheme sequentializes disjointed sequential requests from different files. However, the performance gain is relatively small compared with the gain from random writes because the maximum performance of the host storage device was already reached in the sequential writes.

In case of random writes, as shown in Figure 5(b), O-QEMU improves performance by up to 59% compared with E-QEMU. As shown in the figure, O-QEMU(EXT4), O-QEMU(XFS), and O-QEMU(F2FS) perform 59%, 54%, and 59% better than E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. The performance gains of the random writes are higher than those of sequential writes since the optimized QEMU transforms random write requests into large sequential write requests. Compared with bare-metal, O-QEMU(EXT4) improves the performance by up to 41%. This result demonstrates that our reshaping technique can achieve higher performance compared with bare-metal even though bare-metal does not include the virtualization overhead.

2) LATENCY

To evaluate the latency of proposed scheme, we ran FIO benchmark under identical settings and measured the average

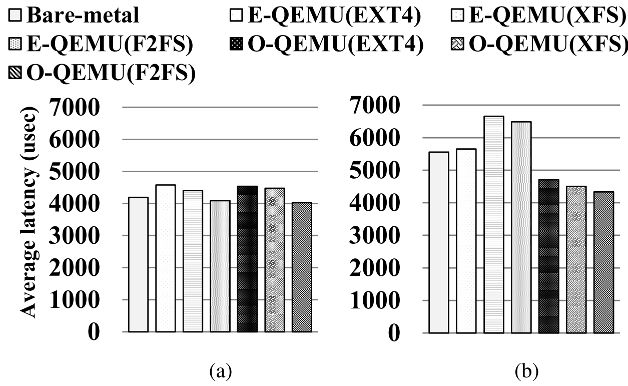

FIGURE 6. FIO average latency with 8 threads, 1 GiB file size, 4 KB request size, and I/O queue depth of 64. (a) Sequential Write. (b) Random Write.

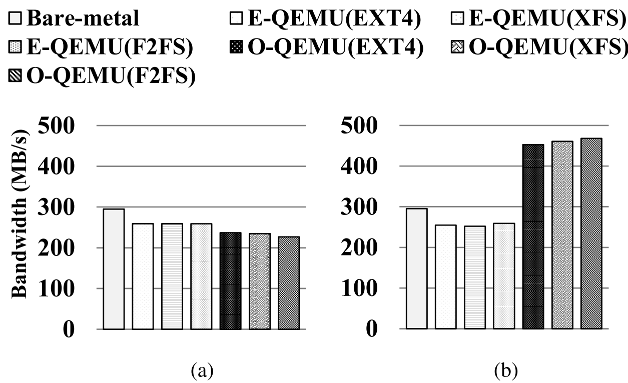

FIGURE 7. Filebench result with 100,000 files, 8 threads, and 4KiB request size. (a) Read. (b) Write.

latency in both sequential and random write workload. In case of sequential writes, as shown in Figure 6(a), the latencies of O-QEMUs are similar to those of E-QEMUs. This result shows that the managing remapping table does not incur significant overhead.

In case of random writes, as shown in Figure 6(b), O-QEMUs have 16.7%, 32.4%, and 33.1% lower latency than E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. Similar to the results of bandwidth, the average latency of the random writes is similar to that of sequential writes since our scheme transforms random writes into sequential writes.

C. FILEBENCH BENCHMARK

For a macrobenchmark, we ran fileserver workload included in filebench benchmark [40]. Fileserver workload simulates a file server by executing create, write, append, and read operations. We configured fileserver with 100,000 files, 8 threads, and 4KiB request size.

In case of read operation, as shown in Figure 7(a), O-QEMUs performs 8.5%, 6%, and 9.6% lower than E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. The read bandwidth is decreased due to the remapping overhead. This is because the transformed IBAs of the requests have to be found from the remapping table and the IBAs of the requests have to be changed to transformed IBAs before issuing the request to the host.

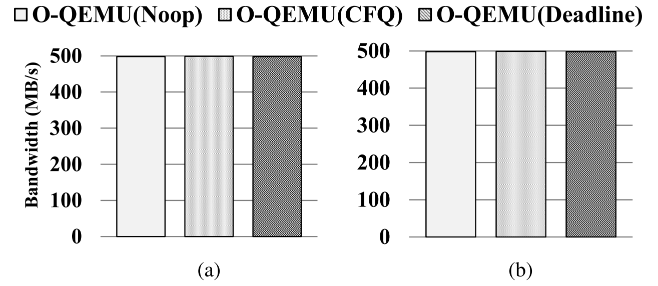

FIGURE 8. FIO bandwidth with 8 threads, 1 GiB file size, 4 KiB request size, and I/O queue depth of 64. (a) Sequential Write. (b) Random Write.

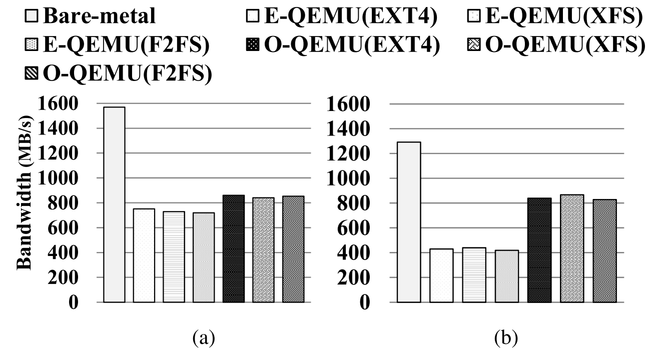

FIGURE 9. FIO bandwidth of NVMe SSD with 8 threads, 1 GiB file size, 4 KiB request size, and I/O queue depth of 64. (a) Sequential Write. (b) Random Write.

In case of write operation, O-QEMUs perform 81.6%, 82.9% and 85% better than E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. Compared with the bare-metal performance, O-QEMU(EXT4) achieves higher performance by up to 54%. Similar to the FIO benchmark, this is because our reshaping technique transforms the random write requests into the sequential write requests. Consequently, this result shows that O-QEMU can successfully sequentialize the random write requests even if fileserver workload includes a mixture of read and write requests unlike the FIO benchmark.

D. IMPACT OF I/O SCHEDULER

To evaluate the impact of I/O schedulers such as noop, CFQ, and deadline on proposed scheme, we ran FIO benchmark and measured the performance under identical configuration with EXT4 file system. In case of both sequential and random writes, as shown in Figure 8, the performance of optimized QEMU with all the I/O schedulers is similar. Since the performance of our proposed scheme already reaches the maximum performance of the host storage device, the effect of I/O schedulers are almost not visible. This result demonstrates that our scheme works well with underlying optimizations such as I/O schedulers without any conflict.

E. FIO BENCHMARK USING NVME SSD

Figure 9 shows the performance of NVMe SSD when we ran FIO benchmark under identical configurations. In case of sequential writes, as shown in Figure 9(a), the performance of O-QEMU(EXT4) is 45.2% lower

compared with bare-metal. The overhead from duplicated layers between VM and the host is more visible under ultra-low latency provided by NVMe SSD compared with SATA SSD. Meanwhile, the optimized QEMU improves the performance by 14.4%, 15.2%, and 15.6% compared with E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. Similar with the performance of SATA SSD, this is due to the increased merge opportunity. The performance improvement from our scheme is larger compared with the case of SATA SSD since E-QEMU(EXT4) cannot reach full performance in NVMe SSD.

In case of random writes, as shown in Figure 9(b), the performance of O-QEMU(EXT4) is 34.9% lower than that of bare-metal due to the overhead from duplicated layer. Compared with the existing scheme, O-QEMU(EXT4), O-QEMU(XFS), and O-QEMU(F2FS) perform 95.1%, 96.9%, and 97.8% better than E-QEMU(EXT4), E-QEMU(XFS), and E-QEMU(F2FS), respectively. Similar with the performance of SATA SSD, random writes with O-QEMU reach similar performance of sequential writes due to the transformation of random writes to sequential writes and the increased merge opportunity. This result shows that our scheme can improve the performance of both sequential and random writes in enterprise-class NVMe SSDs.

V. RELATED WORK

A. VIRTUALIZATION

There have been several studies on reducing overhead from virtualization. ELVIS [11] proposes a host functionality daemon pinned on a single core. By creating a single daemon thread for handling requests at the host, ELVIS contiguously handles the requests from the host without the need for VM exits. Vpipe [41] also reduces the number of VM exits by creating a pipe between the VM application and the underlying storage devices. Our study is in line with these approaches in terms of investigating virtualized devices and request handling in QEMU-KVM. In contrast, we focus on fast storage devices such as SSDs and considering the characteristics of SSD in the virtualized environment. Fvd [42] proposes a new file layout for virtualization that supports fast data allocation, near-instant creation, and migration of the VM image by designing a new file format and data lookup table for the VM image.

Our study is in line with these studies [11], [41], [42] in terms of optimizing the file layout of the VM image and improving the write performance. In contrast, we focus on sequentializing write requests for SSD by considering the VM image.

B. FLASH AWARE MODIFICATIONS

There have been several studies on designing append-only file systems for storage devices to reduce random writes. The concept of sequentializing the random write request was first introduced in Log-structured File System (LFS) [34], [35]. By managing the storage device as a circular log, write requests are always written in append-only fashion, increasing spatial locality. Although these schemes are designed for HDDs, the concept of append-only file system have been adapted for SSDs. SFS [14] shows that the performance of random writes in SSDs is far worse than that of sequential writes and designs a file system which transforms the random write requests into sequential write requests. F2FS [19] is a Linux file system for SSDs that introduces a flash friendly disk layout, hot/cold separation in the log, and optimized `fsync` call for accelerating small synchronous writes. Z-MAP [43] separates random accessed data from sequential accessed data and store them in separate region in NAND flash memory. Z-MAP uses a streaming buffer zone to log data sequentially to reduce the number of random writes and classify the workloads.

SHRD [20] transforms random write requests into sequential write requests in the device driver of the host layer. SHRD restores sequentially written data to the original location by utilizing the address mapping scheme of the FTL.

Our study is in line with these studies [14], [19], [20], [34], [35], [43] in terms of sequentializing the random write requests. In contrast, our study focuses on the virtualized environment by utilizing VM features to support sequentializing operations in the VM layer. This allows our scheme to support different system configurations (e.g., different underlying file systems and hardware) in the virtualized environment.

VI. CONCLUSIONS

In this paper, we propose an address reshaping technique for the virtualization to improve the I/O performance of flash-based SSDs. To do this, we design and implement an address reshaping technique which transforms random write requests issued from VM to sequential write requests to the host. In experiment results, we demonstrate that our optimized system improves the performance by up to 97% compared with the existing system, which achieves similar performance of sequential writes. In addition to the performance, our scheme can improve the endurance of SSDs and reduce the garbage collection overhead in a virtualization environment.

REFERENCES

[1] *Amazon Elastic Compute Cloud*. Accessed: Oct. 1, 2018. [Online]. Available: https://aws.amazon.com/ec2/

[2] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, 2012.

[3] A. K. Qumranet, Y. K. Qumranet, D. L. Qumranet, U. L. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, vol. 1, 2007, pp. 225–230.

[4] P. Barham *et al.*, "Xen and the art of virtualization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Dec. 2003.

[5] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[6] M. Helsley, "LXC Linux container tools: Tour and set up the new container tools called Linux containers," *IBM DeveloperWorks*, pp. 1–10, 2009.

[7] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Proc. 2nd Int. Conf. Comput. Netw. Technol. (ICCNT)*, 2010, pp. 222–226.

[8] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proc. IEEE*, vol. 102, no. 1, pp. 11–31, Jan. 2014.

[9] N. Huber, M. von Quast, M. Hauck, and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments," in *Proc. CLOSER*, 2011, pp. 563–573.

[10] R. Russell, "virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[11] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual I/O system," in *Proc. USENIX Annu. Tech. Conf.*, vol. 26, 2013, pp. 231–242.

[12] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 52–54, Jul./Aug. 2010.

[13] H.-T. Lue *et al.*, "A highly scalable 8-layer 3D vertical-gate (VG) TFT NAND Flash using junction-free buried channel BE-SONOS device," in *Proc. Symp. VLSI Technol. (VLSIT)*, Jun. 2010, pp. 131–132.

[14] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. FAST*, 2012, p. 12.

[15] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and repairing write performance on flash devices," in *Proc. 5th Int. Workshop Data Manage. New Hardw.*, 2009, pp. 9–14.

[16] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High performance solid state storage under linux," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–12.

[17] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, 2009.

[18] L. Bouganim, B. Jónsson, and P. Bonnet. (2009). "uFLIP: Understanding flash IO patterns." [Online]. Available: https://arxiv.org/abs/0909.1780

[19] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. FAST*, 2015, pp. 273–286.

[20] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizng in device," in *Proc. FAST*, 2017, pp. 271–284.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, vol. 2, 2007, pp. 21–33.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. USENIX Annu. Tech. Conf.*, vol. 15, 1996, p. 1.

[23] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, vol. 41, 2005, p. 41.

[24] H. C. Yoo, "Comparative analysis of asynchronous I/O in multithreaded UNIX," *Softw., Pract. Exper.*, vol. 26, no. 9, pp. 987–997, 1996.

[25] M. McLoughlin. (2008). The Qcow2 Image Format. Gnome. [Online]. Available: http://people.gnome.org/~markmc/qcow-image-format.html

[26] C. Tang, "FVD: A high-performance virtual machine image format for cloud," in *Proc. USENIX Annu. Tech. Conf.*, 2011, p. 2.

[27] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li, "Liquid: A scalable deduplication file system for virtual machine images," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1257–1266, May 2014.

[28] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. SYSTOR, Israeli Exp. Syst. Conf.*, 2009, p. 10.

[29] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, p. 18, Jul. 2007

[30] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operat. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, Oct. 2008.

[31] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, May 2016.

[32] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level FTL to optimize address translation in flash memory," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, p. 12.

[33] Q. Li *et al.*, "Access characteristic guided read and write cost regulation for performance improvement on flash memory," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, 2016, pp. 125–132. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/li-qiao

[34] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[35] Z. N. J. Peterson, "Data placement for copy-on-write using virtual contiguity," Ph.D. dissertation, Univ. California, Santa Cruz, Santa Cruz, CA, USA, 2002.

[36] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proc. USENIX Winter*, 1993, pp. 307–326.

[37] N. Watkins. (2018). *Persistent Red/Black Tree*. [Online]. Available: https://github.com/noahdesu/persistent-rbtree

[38] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 183–191.

[39] J. Axboe. (Apr. 1998). *Fiobenchmark*. [Online]. Available: http://freecode.com/projects/fio

[40] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *Login, USENIX Mag.*, vol. 41, no. 1, pp. 6–12, 2016.

[41] S. Gamage, C. Xu, R. R. Kompella, and D. Xu, "vPipe: Piped I/O offloading for efficient data movement in virtualized clouds," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[42] Q. Chen, L. Liang, Y. Xia, H. Chen, and H. Kim, "Mitigating sync amplification for copy-on-write virtual disk," in *Proc. FAST*, 2016, pp. 241–247.

[43] Q. Wei, C. Chen, M. Xue, and J. Yang, "Z-MAP: A zone-based flash translation layer with workload classification for solid-state drive," *ACM Trans. Storage*, vol. 11, no. 1, p. 4, 2015.

SUNGGON KIM received the B.S. degree in computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2015. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. He was an Intern with the Scientific Data Management Research Group, Lawrence Berkeley National Laboratory, CA, USA, in 2018. His research interests are file systems, virtualization, cloud computing, distributed systems, multi-core systems, database systems, and operating systems.

HYEONSANG EOM received the B.S. degree in computer science and statistics from Seoul National University (SNU), Seoul, South Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of Maryland at College Park, College Park, MD, USA, in 1996 and 2003, respectively. He is currently a Professor with the Department of Computer Science and Engineering, SNU, where he has been a Faculty Member, since 2005. He was an Intern with the Data Engineering Group, Sun Microsystems, CA, USA, in 1997, and a Senior Engineer with the Telecommunication R&D Center, Samsung Electronics, South Korea, from 2003 to 2004. His research interests include high performance storage systems, operating systems, distributed systems, cloud computing, energy efficient systems, fault-tolerant systems, security, and information dynamics.

YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science, Seoul National University, in 2012 and 2018, respectively. He was a Post-Doctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana-Champaign. He is currently an Assistant Professor with the School of Computer Science and Engineering, Chung-Ang University. His research interests are operating, distributed, and database systems.