VM-aware Flush Mechanism for Mitigating Inter-VM I/O Interference

Taehyung Lee
Dept. of Platform Software
Sungkyunkwan University, Korea
taehyunggg@skku.edu

Minho Lee
Dept. of Electrical and Computer Engineering
Sungkyunkwan University, Korea
minhozx@skku.edu

Young Ik Eom
College of Software
Sungkyunkwan University, Korea
yieom@skku.edu

***Abstract*—Consolidating multiple servers into a physical machine is now commonplace in cloud infrastructures. The virtualized systems often arrange virtual disks of multiple virtual machines (VMs) on the same underlying storage device while striving to guarantee the performance service level objective (SLO) for each VM. Unfortunately, sync operations called by a VM make it hard to satisfy the performance SLO by disturbing I/O activities of other VMs. We reveal that the disk cache flush command is a root cause of this problem and present a novel VM-aware flush mechanism, called `vFLUSH`, which supports the VM-based persistency control of the disk cache flush command. Our evaluation shows that `vFLUSH` reduces the average latency of disk cache flush commands by up to 52.0% and improves the overall I/O performance by up to 59.6% on real workloads.**

I. Introduction

With the benefit of virtualization technology, cloud infrastructures tend to consolidate multiple servers into a physical machine [1], [2]. In such virtualized systems, virtual disks of multiple virtual machines (VMs) are commonly arranged on the same underlying storage device, while each VM wants its performance service level objective (SLO) always guaranteed [2], [3]. Unfortunately, total I/O bandwidth provided by the storage device can fluctuate depending on the type (e.g., *read*, *write*, or *sync*) and pattern (e.g., sequential or random) of I/O requests. Thus, the I/O behavior of a VM that adversely affects the storage I/O bandwidth disturbs the I/O activities of other VMs. This performance interference among VMs makes it hard to satisfy the performance SLO of the system.

Many applications use sync operations (e.g., *fsync* or *fdatasync*) to keep their data consistent, while they suffer from significant performance degradation as the sync operation is a representative factor of negatively affecting the I/O performance of the system. It involves the disk cache flush command which forces all data in the storage write buffer to be flushed to non-volatile media, thereby generating massive I/O requests internally in the storage device. In case that the disk cache flush command is frequently triggered from the host, it makes the write buffer not be fully utilized in the aspects of write request absorption. Furthermore, the I/O requests issued by the host during flushing should be delayed until the disk cache flush command is completed. Especially, in Solid State Drives (SSDs) which are widely deployed in modern computing systems, the performance impact of the disk cache flush command is more severe. Similarly to the conventional storage devices such as HDDs, SSDs employ a small sized DRAM as its write buffer. They also have several kinds of SSD metadata (e.g., map table) that supports various internal mechanisms for their performance and reliability. For fast access to such SSD metadata, they usually cache it in DRAM. Therefore, when the disk cache flush command is triggered, it forces SSD metadata as well as all data in the write buffer to be flushed to the flash memory. This results in additional writes and frequent garbage collections which are harmful to both the performance and endurance of SSDs.

Not surprisingly, there have been several prior studies to reduce the overhead of sync operations. Some work alleviates ordering constraint to improve the performance of sync operations [4]–[6] or compromises between the data durability and the performance [7]. iJournaling [8] resolves a compound transaction problem in journaling file systems and RFLUSH [9] suggests fine-grained persistency control for the disk cache flush command based on LBA range or inode. Since all these works focus on relieving the overhead of sync operations only in the host layer, they do not fit to address the performance interference problem caused by sync operations in virtualized systems. Chen et al. [10] uncovers a sync amplification problem in copy-on-write (CoW) virtual disk format and resolves it by employing the journaling mechanism. However, its optimization is limited to CoW-based virtual disk.

This paper reveals that the disk cache flush command causes the performance interference among the VMs in virtualized systems. To mitigate this problem, we revisit the internal mechanisms inside the flash memory-based SSD. Then, we propose a novel VM-aware flush mechanism, called `vFLUSH`, which supports the VM-based persistency control of the disk cache flush command. `vFLUSH` identifies whether I/O requests are relevant to each VM or not, and bridges the semantic gap between the host and the SSD by transfering each I/O requests with additional semantic information. At the storage level, `vFLUSH` manages the write buffer entries and cached SSD metadata on a per-VM basis. This mechanism helps to reduce the latency for processing the disk cache flush command,

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT(NRF-2017R1A2B3004660) and was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696).

thereby mitigating the performance interference problem in the virtualized systems. Our evaluation results show that `vFLUSH` decreases the average latency of disk cache flush commands by up to 52.0% and enhances overall I/O performance by up to 59.6%.

II. Background and Motivation

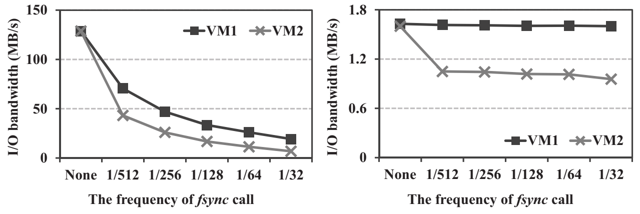
A. Brief Overview of SSDs

Flash memory-based SSD typically involves host interface logic (HIL) and flash translation layer (FTL). HIL manages the storage protocol (e.g., SATA or NVMe) used to transfer I/O requests with the host. FTL mainly handles I/O requests and supports to guarantee the reliability of the data stored on the flash memory by performing an address translation, write buffering, garbage collection, and wear-leveling. The address translation complements *out-of-place update* which is one of the flash memory features, by using a map table. This map table is stored on the flash memory and contains mapping information between logical page number (LPN) and physical page number (PPN). To provide fast accessing time to the mapping information, the map table is also cached in a portion of DRAM. Write buffering is to store the data written by the host in the write buffer space in DRAM. Since DRAM has lower read/write latency than the flash memory, write buffering reduces I/O response time on the flash storage. It also improves the endurance of the flash memory by reducing the amount of data to be written to the flash memory. If the write buffer has no free space or the host triggers the disk cache flush command, the tail or whole entries of the write buffer are reflected to the flash memory. Garbage collection tries to retain sufficient free flash blocks by recycling invalid pages, and wear-leveling is used to wear-out flash blocks evenly for increasing the lifespan of the SSD.

The I/O requests of the host are not immediately reflected to the flash memory, but only stored in the write buffer for a short period of time. To ensure the data durability and prevent unintended data reordering, applications and file systems forward the disk cache flush command to the SSD. When the SSD receives the disk cache flush command from the host, HIL transfers all I/O requests in its device queue to the FTL and prevents the host from passing further requests until the completion of the disk cache flush command. FTL flushes all entries of the write buffer to the flash memory and also reflects cached map table pages, which contain modified LPN-to-PPN (L2P) mapping information, into the flash memory for guaranteeing the persistence of the flushed data. After a while, the SSD informs the host that the disk cache flush command has been completed, and starts to receive new requests.

B. Performance Interference among VMs

In order to confirm the performance interference among the VMs in virtualized systems, we performed some experiments using a commercial SSD (Detailed experimental setup will be described in Section V). We set the cache mode of the QEMU-KVM hypervisor as *none* so that all I/O requests from VMs are directly passed to the storage device, bypassing the host page



(a) Write buffer enabled (b) Write buffer disabled

Fig. 1: The performance of VM1 (no *fsync*) and VM2 (*fsync*)

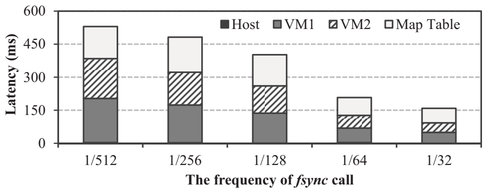


Fig. 2: The average latency of disk cache flush commands

cache. We measured the I/O performance of two VMs, each of which performed the same 4KB random write workload simultaneously, using FIO benchmark. We let only one VM invoke *fsync* calls, and varied the frequency of *fsync* calls from zero to once every 32 writes.

Fig. 1 shows the variation of I/O bandwidth for each VM under the two circumstances: one is the write buffer enabled case and the other is the write buffer disabled case. Despite the fact that VM1 does not explicitly call any sync operations, Fig. 1a shows that the I/O performance of VM1 is significantly slowed down similarly to that of VM2 which invokes *fsync* calls. This result reveals that I/O requests of VM1 are considerably delayed waiting for the completion of disk cache flush commands which are triggered by *fsync* calls of VM2, thereby deteriorating the performance of VM1. Of course, we also observe the significant performance degradation of VM2. Fig. 1b depicts the I/O performance of each VM when the write buffer is disabled. In this case, both VMs show the poor I/O performance compared with the former case since the SSD directly reflects their write requests into the flash memory. But, this synchronous handling on I/O requests results in the low overhead of the disk cache flush command. For such a reason, the performance of VM1 is constant regardless of the frequency of *fsync* calls. These notable results let us infer that reducing the latency of the disk cache flush command is one of the keys to resolve the performance interference among VMs caused by sync operations.

We analyzed various factors that affect the latency of disk cache flush commands. SSD vendors do not provide their SSD firmware structure, and so, it is hard to identify the internal mechanism of their SSDs. Thus, we conducted more experiments using OpenSSD [11] instead of the commercial SSD. Fig. 2 shows the average latency of disk cache flush commands and the ratio of the amount of each data reflected to the flash memory. Only 27.1% to 35.1% of the average latency is used for flushing the data of VM2. The rest of the time is consumed for cached map table pages, the data of VM1, and the data of the host file system. Note that the small

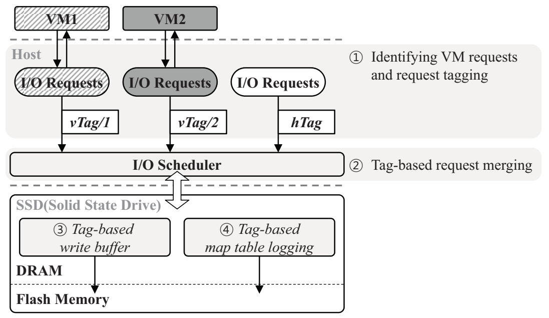


Fig. 3: The overview of `vFLUSH`

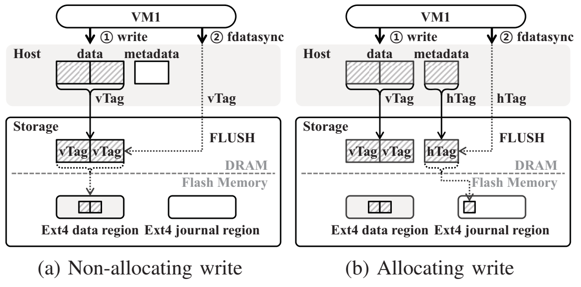


Fig. 4: The problematic case of unconditional request tagging

amount of data related to the host file system contains various file metadata such as those of image files for VMs. These results come from the semantic gap among the VMs, the host, and the SSD. Since the SSD does not have any system-side information for each data except its LBA, it unconditionally flushes all dirty pages in DRAM when the disk cache flush command is triggered. From the above experimental results, we concluded that the performance interference among the VMs severely happens in virtualized systems and the root cause of this problem is the disk cache flush command.

III. Design

Our design aims to mitigate the performance interference among VMs by reducing the latency for processing the disk cache flush command which is triggered by VMs. To achieve this goal, we propose a novel VM-aware flush mechanism, called `vFLUSH`, that supports the VM-based persistency control of the disk cache flush command. Fig. 3 shows the overview of `vFLUSH` mechanism. In our mechanism, (1) we first identify whether I/O requests are relevant to each VM at the host level, and transfer the identified I/O requests to the SSD along with corresponding tag values containing semantic information. (2) We also merge the I/O requests into a single request based on the tag values as well as their LBAs in the I/O scheduler of the host. Also, `vFLUSH` not only manages (3) the write buffer and (4) map table on the basis of tag values, but also selectively reflects the pages which actually need to be persistent. We will describe the detail of each step in the following subsections.

A. Identifying VM Requests and Request Tagging

We first explain how to identify I/O requests of each VM. Since each VM employs one or more image files as its virtual disks, I/O requests of each VM are reflected in its image files. It means that we can identify I/O requests of each VM by checking whether the target file of each request is one of its own image files or not. To do this, we manually grasp the inode numbers of the image files which are used as virtual disks, and compare those numbers with the target file of incoming I/O requests. Based on this procedure, we assign the proper tag value to each request. We define two types of tag value: One is *vTag* (Tag for each VM) and the other is *hTag* (Tag for Host). *vTag* is the unique identifier for each virtual disk. Even when a VM has multiple virtual disks, the value of *vTag* is different for each virtual disk. *hTag* is used for host I/O requests such as journaling I/O requests which contains the file and filesystem metadata.

Whenever the host transfers an I/O request, corresponding tag value is transmitted together with the I/O request according to the type of each request. `vFLUSH` usually works as depicted in Fig. 4a. In the case of *non-allocating write*, which does not change the file size, `vFLUSH` conveys the write request with its *vTag* to the SSD. Afterward, if a VM triggers a *fdatasync* call, `vFLUSH` transfers the corresponding disk cache flush command with the *vTag* of the VM. In this case, the file system consistency is sufficiently guaranteed. On the other hand, in the case of *allocating write*, which incurs modification of file system metadata due to requiring new filesystem block allocations, assigning *vTag* to the write requests can lead to data inconsistency. To address this issue, our mechanism collaborates with the journaling scheme of the host file system (e.g., `jbd2` of Ext4 file system). The modified metadata blocks handled by the journaling scheme are passed with *hTag* to the SSD, as shown in Fig. 4b. If the VM triggers a *fdatasync* call, the host file system forwards disk cache flush commands with *hTag* to commit pending updates. Then, the metadata blocks with *hTag* are reflected to the flash memory. To prevent the data inconsistency caused by uncommitted data blocks with *vTag*, `vFLUSH` allocates *hTag* to both the data and metadata blocks for every *allocating write*.

B. Tag-based Request Merging

`vFLUSH` also considers the I/O scheduler of the host. Current Linux I/O schedulers (e.g., CFQ or Deadline) perform sorting and merging operations. They try to sort I/O requests in the queue based on their LBAs, and then, merge two or more adjacent requests into a single request for improving overall response time of the I/O requests. If neighbor I/O requests which have different tag values are merged into a single request, the semantic for each request could be distorted in the SSD, thereby breaking the data consistency of the system. To avoid this problem, `vFLUSH` allows requests to be merged only when they have the same tag value.

C. Tag-based Write Buffer

Since conventional write buffer in the SSDs considers only the temporal/spatial locality of the allocated data to enhance the I/O performance and utilization, we should redesign the write buffer with the tag values, *vTag* and *hTag*. In our

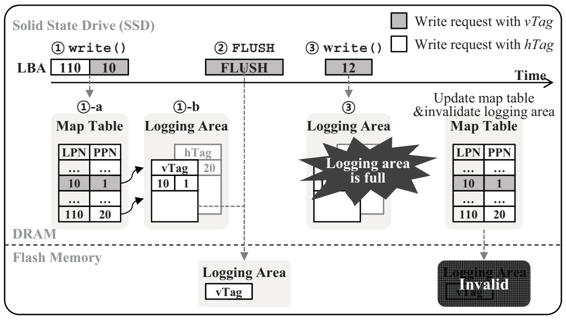


Fig. 5: Control flow of the tag-based map table logging

write buffer, which is called *tag-based write buffer*, two types of linked list exist: One is LRU list and the other is tag-hashing list. The LRU list has the role of handling entire write requests allocated in the write buffer in LRU (Least Recently Used) manner. When there is no free space, tail entries are evicted from the LRU list by reflecting them into the flash memory. Meanwhile, the tag-hashing list is per-tag linked list to efficiently manage the entries with the same tag value together. When the disk cache flush command is transferred with a certain tag value, *tag-based write buffer* first scans the tag-hashing lists to check whether the corresponding list exists. If the list exists, *tag-based write buffer* reflects whole entries of that list into the flash memory while leaving the other entries untouched. Otherwise, all entries are left in the write buffer as they are.

D. Tag-based Map Table Logging

Whenever the entries in *tag-based write buffer* are reflected into the flash memory, map table pages which contain address translation information between LPNs and PPNs should be updated to guarantee the data durability. In contrast with the conventional write buffer, *tag-based write buffer* selectively flushes the data according to its tag value. For this reason, it is sufficient to flush only a part of L2P mapping entries which actually needs to be persistent. With this consideration, vFLUSH uses *tag-based map table logging* which records every modification of L2P mapping entries to the per-tag logging area in DRAM. If a disk cache flush command is triggered, only the logging area having the same tag value with that of the command is reflected into the flash memory, instead of flushing entire dirty map table pages.

For example, when an entry (*vTag*) is evicted from the write buffer as shown in Fig. 5, the corresponding map table page should be updated in that new PPN is assigned to the evicted entry (①-a). Then, its L2P mapping information is recorded in the logging area of the *vTag* in the form of (LPN, newPPN) (①-b). If the host transfers the disk cache flush command with the *vTag*, only the logging area of the *vTag* is flushed into the flash memory, leaving other logging areas and map table pages untouched (②). At this stage, the logging area of the *vTag* exists in both DRAM and the flash memory, and the size of them is equal in our mechanism. Afterwards, whenever a page is newly updated, a new logging entry is accumulated at the end of the logging area in DRAM. vFLUSH reflects the updated map table pages to the flash memory when one of the logging areas in DRAM becomes full so that no more logging entries can be kept (③). Once updated map table pages are written to the original map table in the flash memory, all logging areas in DRAM and flash memory are invalidated and start to record new mapping information.

Recovery process for system crash is as follows. On the SSD reboot, all logging areas kept in the flash memory are loaded into DRAM. If there are any logging entries, vFLUSH restores the map table by reflecting the logging entries one by one. Note that partial updates of the logging area due to sudden power failure can violate the semantic of the disk cache flush command during the recovery progress. In order to avoid this problem, each logging area must be atomically programmed to the flash memory. For this, we fix the size of the logging area as the physical page size of the flash memory.

IV. Implementation

We implemented the prototype of vFLUSH on Linux kernel 4.11 and OpenSSD Jasmine board [11]. This board is designed to allow users to modify the internal firmware. To identify I/O requests of each VM, we used the inode number of each image file as its unique identifier. In addition, we used one-byte identifier with all zero bits as *hTag* value to convey the semantic information of I/O requests that occur in the host.

Technical challenges arise when vFLUSH attempts to transfer each request with its corresponding tag value. Existing SATA interface does not support to transmit additional information for each request. To address this problem, we extended the SATA command set of the write request and disk cache flush command. If each request is delivered with a tag value to the device driver, the one byte-sized tag value is stored in the unused field of SATA command registers. However, it is difficult to transfer the inode number of the image file as its *vTag* to the SSD because the size of the inode number is too big to be recorded in the registers. We resolved this issue by making a small table in the host, which maps each inode number onto the corresponding one-byte identifier (*vTag*).

Then, OpenSSD [11] has to convey tag values transmitted with I/O requests to the FTL. To do this, we read tag values from command registers in the HIL, and pass the tag values with their corresponding I/O requests to the FTL through the event queue. In the FTL, we implemented the *tag-based write buffer* and *tag-based map table logging*, which uses the delivered tag values.

V. Evaluation

The experiments were performed on a machine equipped with an Intel Xeon E3-1275 v5 (3.6GHz) and 32GB DRAM. We used OpenSSD [11] which has 64GB NAND flash chips and 64MB SDRAM as its underlying storage device. OpenSSD [11] is connected to the host via the SATA interface, and its physical flash page size is 8KB. We constructed each VM by using QEMU-KVM 2.1.0 and configured it with 1 vCPU, 2GB memory, and 8GB virtual disk. Ubuntu 14.04 and 12.04 are used as the host and guest OS, respectively.

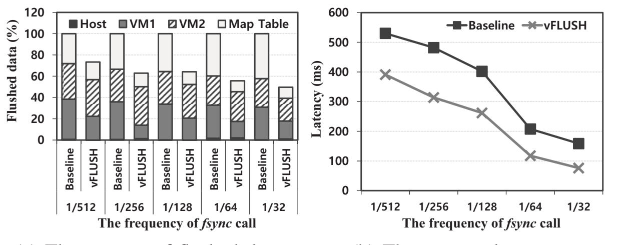

(a) The amount of flushed data (b) The average latency

Fig. 6: The amount of flushed data and the average latency of disk cache flush commands

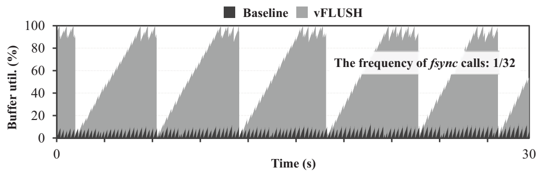


Fig. 7: The write buffer utilization

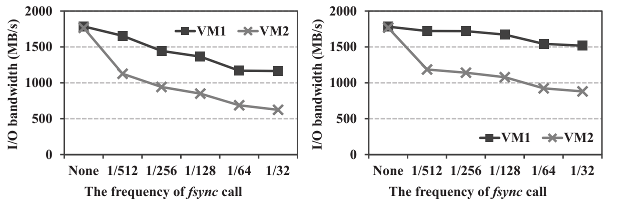
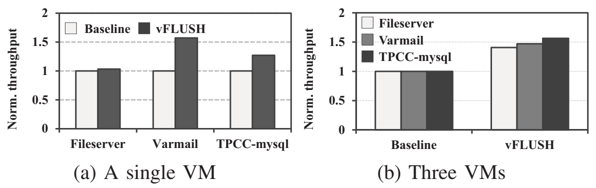

(a) `Baseline` (b) `vFLUSH`

Fig. 8: The performance of VM1 (no *fsync*) and VM2 (*fsync*)

A. Micro-Benchmark

To verify the effectiveness of `vFLUSH`, we re-performed the same experiments in Section II-B. In our experiments, `Baseline` means the conventional system where the semantic gap between the host and storage device exists. Fig. 6a shows the amount of flushed data normalized by `Baseline`. The results reveal that `vFLUSH` reduces the amount of flushed data by 26.6% to 50.2%. In particular, the amount of flushed map table pages drastically decreases by 41.1% to 75.3%. This is because `vFLUSH` fully utilizes the benefit of the *tag-based map table logging* until the logging area becomes full. The number of flushed buffer entries is also reduced by up to 32.0% by the *tag-based write buffer*. Due to these benefits, `vFLUSH` decreases the average latency of all disk cache flush commands triggered by VM1, VM2, and the host by 26.2% to 52.0% compared to `Baseline`, as shown in Fig. 6b. Furthermore, since the *tag-based write buffer* can take more chances to absorb additional writes for the VM which does not invoke any sync operations, it enhances buffer utilization as shown in Fig. 7. Buffer utilization periodically drops in `vFLUSH`, because the consistency mechanism of the guest file system in VMs commits all the updated data and metadata periodically (e.g., every 5 seconds in `jbd2` of Ext4 file system).

Fig. 8 shows the results in terms of I/O bandwidth for each VM. In `Baseline`, the performance of VM1 which does not trigger any *fsync* call is degraded by up to 34.8% due to disk cache flush commands of VM2. The performance of VM2 is also reduced by up to 74.7% as the frequency of *fsync* calls increases. On the other hands, in `vFLUSH`, the performance


(a) A single VM (b) Three VMs

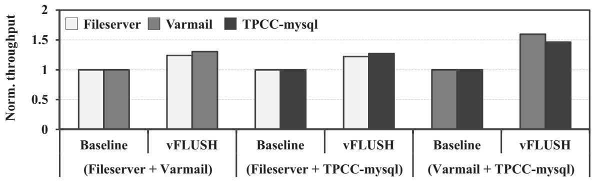

(c) Two VMs

Fig. 9: The normalized performance of real workloads

of VM1 and VM2 only decreases by up to 14.9% and 50.2%, respectively. These performance improvements derive from the low latency of disk cache flush commands as we already confirmed in Fig. 6. Moreover, `vFLUSH` provides the same performance with `Baseline` when there are no explicit *fsync* calls. This result shows that, even though `vFLUSH` needs additional costs to support the *tag-based write buffer* and *tag-based map table logging*, the benefit from the fine-grained control on flushing hides such overheads.

B. Macro-Benchmark

To evaluate `vFLUSH` on real-world workloads, we used TPCC-mysql and various workloads of Filebench including varmail and fileserver. Fig. 9 shows the performance of each workload under a single VM and multiple VM environments. All results are normalized by `Baseline` performance. On a single VM where there is the no performance interference by other VMs, `vFLUSH` outperforms `Baseline` for all workloads as shown in Fig. 9a. To be specific, for the varmail workload which invokes a lot of *fsync* calls in the process of simulating mail server environments, `vFLUSH` shows 1.57x higher throughput than `Baseline`. This performance improvement comes from the *tag-based map table logging* that flushes only the necessary logging area instead of all dirty map table pages. For the same reason, `vFLUSH` improves the performance of the TPCC-mysql workload which mimics a full database structure and frequently triggers *fsync* calls by 27.0%. Even in the fileserver workload where there is no *fsync* call, `vFLUSH` enhances the performance by 3.3%.

Fig. 9c shows the performance for two VMs where the performance interference among VMs exists. The results show that the performance of the fileserver workload increases by up to 24.0% when it is executed concurrently with sync-intensive workload. This is largely due to the fine-grained handling of the disk cache flush command for each VM, thereby mitigating the performance interference among VMs. Even in the case where two sync-intensive workloads run simultaneously, `vFLUSH` serves better performance than `Baseline` by up to 59.6% for the varmail workload and 46.3% for the TPCC-mysql workload. In three VM environments, `vFLUSH` still provides the outstanding performance than `Baseline` because the performance interference problem gets more severe as the number of VMs running concurrently increases. As shown in Fig. 9b, the performance of the fileserver workload is significantly improved by up to 40.7% in vFLUSH. The varmail and TPCC-mysql workload also show 1.47x and 1.56x higher performance, respectively, in vFLUSH.

VI. Related Works

Many studies [4]–[9] have been conducted at the host level to improve the performance of sync operations. Xsyncfs [7] prevents an application to be blocked due to synchronous requests by delaying sync operations until an external user accesses the updated data. It provides good performance, but it is hard to guarantee the data durability with the mechanism. Nofs [4] improves the performance of sync operations by removing the ordering constraint of the file system. Even though Nofs adopts the backpointer based crash consistency scheme, it cannot guarantee crash consistency for atomic operations. Optfs [5] uses two file-system primitives, osync() and dsync(), to decouple ordering and durability. It also improves I/O concurrency by using asynchronous durability notification technique. However, it does not ensure immediate data durability. CCFS [6] groups file system updates of each application in a unit of stream and separates the ordering constraint for each stream. iJournaling [8] suggests fine-grained journaling mechanism through file-level transactions. Both CCFS and iJournaling support to process *fsync* call in parallel, but the overhead of disk cache flush commands still remains in the storage device.

There are recent works conducted in the storage-level. Lee et al. [12] assists process-based persistency control of the disk cache flush command. RFLUSH [9] also supports a fine-grained persistency control based on LBA range or inode. However, both optimizations are limited to the write buffer; hence the overhead of flushing map table pages still exists. Furthermore, RFLUSH can incur flush amplification problems which calls frequent disk cache flush commands more than it needs when the running transactions containing multiple inodes have to be committed. OFTL [13] guarantees the write order among the journal data blocks at the storage-level to remove one disk cache flush command in the commit procedure of the Ext4 file system. Even though OFTL halves the number of disk cache flush commands, it does not fit to solve the I/O interference problem in virtualized systems.

Similar studies to relieve the overhead of sync operations have been performed at the hypervisor level in virtualized systems. Li et al. [14] proposed a new I/O model to improve the sync performance of each VM. This model buffers flushed data into the host memory and asynchronously reflects the buffered data into the storage. But, it compromises the data durability instead of the performance. Chen et al. [10] resolve sync amplification problems in copy-on-write (CoW) based disk format, by adopting the journaling mechanism. This optimization reduces the number of sync operations effectively, but it is limited to CoW-based disk format.

There are other researches [1]–[3], [15], [16] for performance isolation between VMs. However, all these works do not address the performance interference among VMs, caused by sync operations.

VII. Conclusion

This paper uncovers the performance interference problem caused by sync operations in virtualized systems. We confirmed that the disk cache flush command is a root cause of this problem, and analyzed the factors that affect its latency. Based on this analysis, we propose vFLUSH that supports VM-based persistency control of the disk cache flush command. Our evaluation results show that vFLUSH improves the average latency of disk cache flush commands by up to 52.0%. Evaluation of diverse real workloads also shows that vFLUSH enhances overall I/O performance by up to 59.6%.

References

[1] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2010, pp. 437-450.

[2] J. Kim, E. Lee, and S. H. Noh, "I/O Scheduling for Better I/O Proportionality on Flash-based SSDs," in *Proc. of IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2016, pp. 221-230.

[3] J. Kim, D. Lee, and S. H. Noh, "Towards SLO Complying SSDs Through OPS Isolation," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2015, pp. 183-189.

[4] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency Without Ordering," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2012, pp. 101-116.

[5] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proc. of the ACM Symposium on Operating Systems*, 2013, pp. 228-243.

[6] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application Crash Consistency and Performance with CCFS," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2017, pp. 181-196.

[7] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the Sync," *ACM Transactions on Computer Systems*, vol. 26, no. 3, 2008, pp. 1-26.

[8] D. Park and D. Shin, "iJournaling: Fine-grained Journaling for Improving the Latency of Fsync System Call," in *Proc. of the USENIX Annual Technical Conference*, 2017, pp. 787-798.

[9] J. Yeon, M. Jeong, S. Lee, and E. Lee, "RFLUSH: Rethink the Flush," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2018, pp. 201-209.

[10] Q. Chen, L. Liang, Y. Xia, H. Chen, and H. Kim, "Mitigating Sync Amplification for Copy-on-write Virtual Disk," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2016, pp. 241-247.

[11] Jasmine OpenSSD Platform. Internet: http://www.openssdproject.org/

[12] T. H. Lee, M. Lee, and Y. I. Eom, "An Insightful Write Buffer Scheme for Improving SSD Performance in Home Cloud Server," in *Proc. of IEEE International Conference on Consumer Electronics*, 2017, pp. 1-2.

[13] D. Park, D. H. Kang, and Y. I. Eom, "OFTL: Ordering-aware FTL for Maximizing Performance of the Journaling File System," in *Proc. of the Annual Design Automation Conference*, 2018, pp. 1-6.

[14] D. Li, X. Liao, H. Jin, B. Zhou, and Q. Zhang, "A New Disk I/O Model of Virtualized Cloud Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, 2013, pp. 1129-1138.

[15] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 349-362.

[16] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment," in *Proc. of the USENIX Annual Technical Conference*, 2014, pp. 133-144.