Separation of Virtual Machine I/O in Cloud Systems

HYOKYUNG BAHN (Member, IEEE), AND JISUN KIM
Department of Computer Engineering, Ewha University, Seoul 120-750, South Korea
Corresponding author: Hyokyung Bahn (bahn@ewha.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIP) under Grant 2019R1A2C1009275 and in part by the ICT Research and Development program of MSIP/IITP (Developing system software technologies for emerging new memory that adaptively learn workload characteristics) under Grant 2019-0-00074.

ABSTRACT Virtualization is widely used in modern computer systems ranging from personal computers to cloud servers as it provides various heterogeneous platforms at low cost. However, due to its nested software structure in host and guest machines that are difficult to harmonize, it is challenging to manage resources efficiently in virtualized systems. In this article, we anatomize the overhead of virtualization associated with file system journaling and discover the excessively frequent commits that take place in virtualized systems. This is because the host triggers commit on every write request from all guest machines. This also generates unnecessary write traffic to storage. To remedy these problems, we propose the VM-separated commit, and implement it on QEMU-KVM and Ext4. Specifically, we devise a data structure that manages modified file blocks from each guest as a separate list and split the running transaction list into two sub-transactions based on this data structure upon a commit request from a guest. Measurement studies with Filebench and IOzone benchmarks show that the proposed policy improves the I/O throughput by 19.5% on average and up to 64.2% over existing systems. It also reduces the variability in performance.

INDEX TERMS Journaling, file system, virtualized system, virtualization, buffer cache, commit.

I. INTRODUCTION
Virtualization techniques are widely used in various modern computer systems ranging from personal computers to cloud servers [1]–[6]. Virtualization separates a software platform from hardware conditions, thereby providing independent views to heterogeneous applications with a single physical hardware set. This allows flexible resource management, thus offering a wide range of heterogeneous platforms at low hardware cost and improving system utilization. It also provides high scalability and low-power consumption in cloud systems via resource sharing.

Unfortunately, these benefits are accompanied by inefficiencies associated with its additional software layers on top of existing system software. Specifically, in a fully virtualized environment such as KVM and VirtualBox, as host and guest machines are independent systems unaware of the internal structures of other machines, it is difficult to harmonize the two systems with respect to the overall performances [7], [8]. Recently, some peculiar storage access patterns and performance degradations have been observed in virtualized systems due to nested file system structures [1], [6], [9]–[13].

The work presented here is in line with these previous studies, but our work focuses on the journaling commit feature of file systems in virtualized environments. Journaling is an essential feature of modern file systems, such as Ext4 and ReiserFS, employed to improve file system reliability [14], [15]. Journaling commits updated data by writing it first to a separate storage area called the journal area, and then later reflecting the updates to their permanent location. This mechanism protects data from being corrupted in case of a sudden system crash because it always maintains consistent data either in the journal area or in the permanent file system location.

In a virtualized system, both host and guest machines should use journaling in order for each system to be protected from corruption, and this nested journaling causes significant inefficiencies. Figure 1 shows the file system structure of a fully virtualized system, which is the target architecture of this article. Both guest and host machines have their own file systems and buffer caches, and the host manages the disk storage.
image of a guest as a single file. In this architecture, we find the following inefficiencies.

- **A chain of cascade commits**
  In virtualized systems, excessively frequent commits take place because the host performs commit on every write request from all guest machines as shown in Figure 2. In particular, when the host receives a write request from a guest, it is already committed by the guest file system. Thus, the host should commit it promptly in order to satisfy the commit semantic with respect to the guest machine. We observe that the commit frequency of a virtualized system is 3x to 11x that of non-virtualized systems.

- **Heavy write traffic to storage**
  The frequent commits also generate a significant amount of unnecessary write traffic to storage as file systems such as Ext4, Btrfs, and ZFS, manage all modified file blocks through a single transaction list. We observe that this unnecessary I/O traffic constitutes 43% on average and up to 63% of the total write traffic.

In order to perform commit, a guest requests a synchronous write to the host, which then triggers its own commit to write all modified data in the buffer cache to storage. The host system manages this request as a single transaction list that includes not only the write data from this guest but also those from all the other guests. As there are no dependencies between write requests from different guests, we propose a VM-separated commit that commits writes only from the guest that makes the write request. To do so, we devise a data structure that manages the modified file blocks from each guest as a separate list by making use of the hash table with the inodes of VM image files as hash keys. Then, upon a commit request from a guest, our policy splits the running transaction into two sub-transactions based on the hash table; one works as a commit transaction consisting of the modified file blocks from the same guest and the other still works as the running transaction.

Measurement studies with popular file I/O benchmarks show that the proposed policy improves the I/O throughput by 19.5% on average and up to 64.2% on the QEMU-KVM and Ext4/JBD2 platform. It also reduces the variability in performance. Though our evaluation study is performed on KVM and Ext4, it is noted that the observations that we make and the policy that we propose are applicable to any platform that makes use of full virtualization and transaction-based file systems (e.g., journaling file systems or copy-on-write file systems).

The remainder of this article is organized as follows. Section II provides the background of this study. The journaling overhead in virtualized systems is analyzed in Section III. Sections IV describes the details of our VM-separated commit to improve the I/O performance of virtualized systems. Section V presents the performance evaluation results of the proposed policy. Section VI summarizes related work. Finally, Section VII concludes the article.

### II. COMMIT SEMANTICS AND VIRTUALIZATION

#### A. FILE SYSTEM CONSISTENCY

To buffer the wide speed gap between main memory and secondary storage, modern operating systems use a buffer cache that stores requested file blocks in a certain portion of main memory, thereby servicing subsequent requests without accessing slow storage media. As traditional buffer cache uses volatile DRAM, the file system may enter an inconsistent and/or out-of-date state when the system crashes before the changes are reflected to permanent storage. To overcome this problem, modern file systems adopt journaling or copy-on-write transaction mechanisms that prevent data corruption via out-of-place updates through periodic commits [16], [17].

**TABLE 1. Consistency Support Policies of File Systems.**

| File system | Consistency supporting policy | Commit period |
|-------------|--------------------------------|---------------|
| Ext4, ReiserFS | metadata / data journaling | 5 sec |
| JFS, XFS | metadata journaling | 5-30 sec |
| Btrfs, ZFS | Copy-on-write | 30 sec |
Table 1 summarizes consistency support policies of various file systems.

Instead of writing modified data directly to its original location in the file system, journaling writes the changes to the journal area first and then reflects them to the original location later. In this way, journaling guarantees file system consistency even when the system crashes in the middle of storage updates. Specifically, journaling groups data to be updated atomically and manages them as a single transaction list. When a transaction is successfully written to the journal area, a commit mark is placed on the journal area. This mark indicates that the transaction will be reflected to storage even in case of a system crash. The committed data are periodically transferred to their permanent location by the checkpointing operation. Unlike the journaling operation that is performed frequently (e.g., every 5 seconds) to protect data from being corrupted, the interval between checkpointing is relatively long (e.g., 5 minutes).

Journaling guarantees more reliable file system states, but it generates additional storage writes. As a compromise between reliability and performance, journaling file systems often offer different journaling modes such as metadata journaling and full data journaling. In a metadata journaling mode, only metadata is journaled and regular data blocks are directly flushed to its original location. Data corruption is possible in this mode, but the file system structure cannot be broken in any case. In full data journaling mode, the file system journals both metadata and regular data. This mode guarantees complete consistency of a file system, but incurs performance penalty due to the write-twice behavior of journaling. To balance between performance and reliability, Ext4 and ReiserFS set the default journaling mode of the file systems to the ordered mode where only the metadata is journaled but with regular data flushes preceding metadata journaling so as to reduce the chance of data corruption such as dangling pointers [18], [19].

Another approach for supporting file system consistency is the copy-on-write (COW) technique. COW creates a copy of data when the data needs to be updated and all modifications are performed on this copy. Thus, the original file data blocks are preserved during modifications. In copy-on-write file systems, such mechanisms are adopted also for the updates of indexes. That is, for a tree-based indexing structure, the COW mechanism causes a change of a child node to update its parent node in an out-of-place manner, propagating all the changes of internal nodes up to the top of the tree [35]. Unlike journaling file systems, copy-on-write file systems do not write modifications back to the original locations but the copies become part of a new file system tree. Generating a new version of a file system tree is performed periodically, and this is also called commit. Similar to commit in journaling file systems, all modifications are handled as a transaction and a commit is completed by generating a new version of a root node. Btrfs and ZFS are examples of copy-on-write file systems of which the default commit period is set to 30 seconds. In copy-on-write file systems, both regular data and metadata are protected [20].

B. I/O STRUCTURES IN VIRTUALIZED SYSTEMS

There are two types of virtualization, full virtualization and para-virtualization, depending on the way guest systems are supported. Full virtualization runs a guest operating system (OS) as an independent application on top of the host OS. This is advantageous in that the host and guest OSs can be run without modifications. In para-virtualization, the host is equipped only with the hypervisor, which contains minimal interfaces to access hardware, and a guest OS is modified to properly run on the hypervisor. Para-virtualization can reduce the performance overhead of virtualization, but it requires modification of the guest OS. VirtualBox [8], VMware [21], and KVM [7] are examples of full virtualization systems, while Xen is a well-known para-virtualization supporting hypervisor [22]. In this article, we focus on full virtualization.

In a fully virtualized system, the guest storage device is usually managed as a single disk image file on the host system. This allows the allocation of the guest’s storage capacity upon an actual access, achieving better space utilization. Supporting other functions such as snapshot and migration of guest storage also becomes easier when managing storage as a virtual disk file rather than a raw disk.

As Figure 1 shows, both the guest and the host of a fully virtualized system have their own file systems and buffer caches. The host regards each guest as a user application and considers the buffer cache and the file system of the guest as user memory and a file, respectively. Upon request of a storage access, the guest first searches its own buffer cache and then, the hypervisor sends the request to the host via a system call. Then, the host checks its buffer cache and finally, the request is delivered to storage.

As host and guest machines manage their own buffer caches, duplicated caching may degrade space efficiency. However, with a large host cache that acts as the second-level cache, guests can expect to reap performance benefits. Specifically, if many virtual machines, whose lifetime is difficult to estimate coexist, then managing a large shared buffer cache on the host side can be much more effective than allocating a large cache space for each individual guest a priori [23]. Due to this reason, while there are options to support host cache bypassing, hypervisors, by default, generally select to use host caching [8], [23].

III. ANALYSIS OF COMMIT OVERHEAD IN VIRTUALIZED SYSTEMS

We now describe the problem of cascaded commit in virtualized systems and analyze the overhead.

A. COMMIT SEMANTICS IN VIRTUALIZED SYSTEMS

As shown in Figure 3(a), in non-virtualized systems, upon an application’s write request, the OS first writes the data to its buffer cache. The file system will later commit the data to storage according to its commit period.
Let us consider the same situation of Figure 3(a) in virtualized systems. In virtualized systems, the host OS considers each guest as a user application. Thus, upon an application’s write request, the guest OS first writes the data to its buffer cache. Then the guest file system will later commit the data to storage according to its commit period. Instead of writing to storage directly, however, the commit request of the guest OS is first sent to the host and the host writes the data to its buffer cache. If this write is treated as a regular file write, the host will write it to storage according to its commit period instead of committing it immediately. So, even though the guest has already committed the data, it is not recoverable if a crash occurs unless the host had committed the data. This contradicts the commit semantics. To resolve this problem, a guest’s commit should be delivered to the end storage immediately as shown in Figure 3(b).

This implies that the host must force a commit upon every write request from all guest machines by making use of synchronous write as shown in Figure 4. As the commit period of a journaling file system is usually set to 5 seconds, and the guest and host machines perform their own journaling commit whose periods cannot be synchronized, a system-wide commit operation occurs whenever the host or one of the guests commits. We refer to such commits as cascaded commits.

In order to perform journaling, the guest maintains a transaction list that contains blocks modified after the latest commit that are sent together to the host. To synchronously reflect the writes to end storage, a guest sends a barrier request to the hypervisor and then, the hypervisor makes the file synchronization system call to the host. As the host also manages writes with a single system-wide transaction list, all writes within the same transaction list, possibly from different guests, will be committed together. This incurs significant performance degradation as the commit of a single guest results in the commit of all modified data in the host to storage. In reality, however, write requests from different guests are unrelated as they are from different file systems. Thus, they need not be committed together.

As this commit overhead is too heavy, some commercial hypervisors are exploring alternatives. For example, VirtualBox, by default, ignores a guest’s flush request. However, this violates the commit semantics of data integrity and durability [8]. In another case, QEMU-KVM uses fdatasync instead of the fsync system call to lessen the heavy write traffic triggered by a guest commit request [24]. Note that both fsync and fdatasync perform cache flush for a certain file to storage; fsync writes all modified data, including regular data and metadata, of the requested file to storage, whereas fdatasync writes only regular data unless that metadata needs to be synchronized for correct handling of a subsequent data retrieval. For example, if only the modified time or the access time is changed, metadata is not flushed. In contrast, if the file size is changed, the metadata is flushed even with fdatasync. Thus, the effect of fdatasync is limited in virtualized systems. Furthermore, fdatasync has no effect at all in copy-on-write file systems such as Btrfs and ZFS as they handle all modifications on the file system tree (including regular data and metadata) as a transaction and thus do not support the fdatasync system call.

B. COMMIT OVERHEAD IN VIRTUALIZED SYSTEMS

To quantify the overhead of cascaded commits in virtualized systems, we measure the commit frequency and the write traffic to storage when multiple guests run simultaneously. We insert a profiler to JBD2, the journaling module of Ext4, and log the commit time and the amount of data committed whenever a commit occurs.

We vary the number of virtual machines from 1 to 5 and measure the commit frequency of the host. We make use of the IOzone benchmark, of which the characteristics will be described later in Section V. As shown in Figure 5, the commit frequency of the host increases dramatically as the number of virtual machines increases.

Now, let us consider the overhead of cascade commit with respect to write traffic. Figure 6 shows the write traffic of commit as the number of virtual machines is varied. For comparison purposes, we also measure the write traffic when modified blocks from only the guest that makes the request are committed. As shown in Figure 6, committing all modified blocks in the transaction list generates significantly larger...
write traffic than committing blocks only from the guest that makes the commit request. Specifically, the gap becomes wider as the number of guest machines increases. Although data in the transaction list are eventually committed in both cases, a shorter commit interval reduces the possibility of absorbing multiple write requests on the same data block, leading to increased write traffic.

### IV. THE VM-SEPARATED COMMIT

In this section, we describe the proposed VM-separated commit. We show how our VM-separated commit helps mitigate the massive storage writes caused by the single transaction management of existing journaling file systems.

Since modern reliable file systems such as Ext4, ReiserFS, Btrfs, and ZFS manage all modified data blocks in a system with a single transaction, a commit operation of one guest machine forces the commit of all modified blocks of the host and all other guest machines. To estimate the effect of this inefficiency, we modify Ext4 to manage modified blocks as a separate transaction for each virtual machine, thereby isolating the commit request between guests. We refer to this as commit isolation. In this way, the host commits only those blocks that should be written synchronously and eliminates unnecessary storage writes. We use QEMU-KVM as the hypervisor in our experiments.

#### A. OVERVIEW OF COMMIT

In journaling file systems, commit operations are triggered when a journal daemon is activated by the predefined commit period or a synchronous write is requested by an application. The periodic commit protects data from being corrupted in case of a sudden system crash. When it comes time to perform periodic commit, the journal daemon performs the commit of all modified blocks within the pending transaction list.

The commit operation is also triggered when a synchronous write is requested by an application. Similar to the periodic commit, the system commits not only the currently requested data but also all modified data within the pending transaction list. In virtualized systems, to the host, the periodic commit of a guest is regarded as an application’s synchronous write request. Thus, excessively frequent commits take place upon periodic commit requests of each guest.

To implement commit operations, Ext4/JBD2 maintains three types of transaction lists. The first is a running transaction that maintains file blocks modified after the previous commit operation. When a commit operation is issued, this running transaction is converted to a commit transaction, and a new running transaction is created to add new block writes arriving during this conversion for the next commit period. After the commit, the committed data are moved to the checkpoint transaction list. A checkpoint transaction is maintained until the committed data are flushed to their original file system locations through checkpointing.

#### Algorithm 1 VM-Separated Commit

```
if sync_write by a VM then
    Commit (i_numVM);
else
    Commit (); /* regular commit */
end if

procedure Commit (VM_sep)
    if VM_sep is set then
        commit_trans ← Header of Hash(VM_sep);
        for each blk in commit_trans
            remove blk from running_trans;
        end for
        else
            convert running_trans to commit_trans;
            running_trans ← NULL;
        end if
    for each blk in commit_trans
        flush blk to journal area;
    end for
    write end_of_commit_block to journal area;
    move commit_trans to checkpoint_trans;
end procedure
```
B. COMMIT ISOLATION

Commit isolation can be achieved through the implementation of our VM-separated commit. Figures 7 and 8 contrast the transaction management mechanisms of the original journaling scheme used in Ext4 and VM-separated commit. In the figure, a synchronous write request comes from VM-2. A transaction is a linked list of struct journal_head that holds the metadata of a buffer cache block associated with journaling. As Figure 7 shows, the original journaling file system commits all modified blocks within the running transaction upon the commit request of VM-2. However, with our VM-separated commit, blocks from different VMs are managed separately according to their file inode by making use of a hash table. As the file system of each VM is managed by a single file in the host, we can identify different VMs via their inodes.

If a synchronous write is requested, the host file system transfers the requested file inode number into the journal layer and wakes up the journal daemon. Then, the journal daemon splits a running transaction into two sub-transactions, one consisting of blocks from the currently requesting VM (block D in this example) and the other consisting of the remaining blocks. The former now becomes a commit transaction and the latter continues to serve as a running transaction, as shown in Figure 8(b). Note that when a commit is triggered by the host machine itself, our VM-separated commit does not split but commits all blocks within the running transaction. Algorithm 1 briefly describes the pseudocode of our VM-separated commit.

V. PERFORMANCE EVALUATION

We measure the I/O performance of our VM-separated commit in comparison with the original Ext4. Our experimental platform consists of the Intel i7-9700 eight core clocking at 3.00 GHz and 16 GB GeIL DDR4. We use a 240 GB SATA 6 Gb/s Maxtor Z1 SSD as a storage device. The host and guest machines use Ubuntu 18.04.5 LTS as their operating system [25]. We allocate 2 GB main memory and 20 GB virtual disk to each guest machine. The disk image is created on a qcow2 format and virtio is used as a device driver for secondary storage in guest machines [4]. We set the cache policy of guests to the write-back mode.

We evaluate performances for three I/O scenarios with IOzone as a micro benchmark and Filebench as a macro benchmark [26], [27]. IOzone measures file I/O performances by generating micro I/O workloads consisting of a series of operations on a single file. We set the configurations of IOzone as record rewrite to measure the performance of writing and re-writing for a particular spot and set the file size to 200 MB. Filebench provides a series of predefined I/Os that emulate real server workloads. We make use of the varmail workload with setting the number of files to 1000 and the number of threads to 100.

To assess the effectiveness of our VM-separated commit, we perform measurement experiments under the three scenarios consisting of eight virtual machines. Table 2 lists the configurations of each scenario we experimented. Scenario 1 consists of three virtual machines, which concurrently execute the same IOzone benchmark on each machine. Scenario 2 consists of three virtual machines, which concurrently execute the same Filebench benchmark. Scenario 3 consists of two virtual machines: one executes IOzone and the other executes Filebench. We experiment each scenario seven times and the mean and the standard deviation are reported.

Figures 9 to 11 show the measured performance of the original Ext4 and the VM-separated commit while performing
the three scenarios. We experiment two different journaling modes of Ext4: journal mode and ordered mode. The ordered mode only guarantees the consistency of metadata, whereas the journal mode guarantees the consistency of full data including regular data as well as metadata. The journal mode is becoming increasingly important as modern reliable file systems such as Btrfs and ZFS guarantee the consistency for both regular data and metadata.

Figure 9 shows the result for Scenario 1 consisting of three virtual machines of IOzone. As shown in the figure, VM-separated commit improves the performance by 21.9% and 15.3%, respectively, for the journal mode and the ordered mode. One surprising result is that the throughput of the ordered mode is over 3 GB/s, which is 100x to 400x better than other experiments. This is because IOzone writes an entire file while it is created, and rewrites the same file after that phase. It is a special characteristic of the IOzone benchmark, which does not incur the modification of metadata like block pointers or file size, but just updates the modified time while rewriting the file.

As we mentioned in Section III-A, QEMU-KVM makes use of fdatasync for guest’s commit, which does not trigger commit if the transaction list contains only the modified time. Note that in the ordered mode, Ext4 does not include regular data in the transaction list. Thus, the journal daemon does not commit the rewrites of regular data in this case, but just they remain in the guest’s buffer cache. This is the reason why the throughput of the ordered mode is very high. Even in this situation, however, it can be seen that the proposed scheme improves the I/O performances.

In case of the journal mode, as regular data are also the target of commit, rewrites of the same file in IOzone are included in the transaction list. Thus, all modified data in the guest’s buffer cache are transferred to the host in every commit period of the guest, and the host, then, instantly commits all data in its buffer cache to storage. Thus, excessively frequent commits take place, which also reduces the possibility of absorbing rewrites in buffer cache. As shown in Figure 9(a), we can see a significant performance improvement of VM-separated commit in this situation.

Now, let us see the result for the Scenario 2 consisting of three VMs that execute Filebench. As shown in Figure 10, the effectiveness of VM-separated commit is relatively weaker than that in Scenario 1. Specifically, the improvement is 13.9% and 6.1% in the journal mode and the ordered mode, respectively. The improvement in this scenario is less than our expectations as VM-separated commit is effective only when rewrites on the same data occur multiple times within the commit period, and the period should not be clustered between guests. If such conditions meet, then multiple writes within a period are flushed to storage only once. However, that may not occur frequently in realistic workloads like Scenario 2. Also, executing the same workloads on all guests might be a factor that limits the effectiveness of our VM-separated commit with respect to the I/O throughput. To eliminate such possibilities, we execute heterogeneous workloads on guest machines in Scenario 3.

Figure 11 shows the result for executing Filebench on VM-1 and IOzone on VM-2. As shown in Figures 11(a)-(b), performance improvement in Filebench is larger than Scenario 2. Specifically, the I/O throughput of Filebench is improved by 64.2% in the journal mode, and 10.4% in the ordered mode. In the IOzone workload, the effect of VM-separated commit is relatively weaker than Scenario 1. The
improvement is 18.6% and 5.8%, in the journal and ordered modes, respectively. Although the improvement becomes small, the overall performance is greatly improved in case of the journal mode. Specifically, the throughput was 10.4 MB/s in Scenario 1, but is 45.9 MB/s in Scenario 3. This seems to be an effect of executing heterogeneous workloads as well as decreasing the number of virtual machines in Scenario 3. Another interesting phenomenon is that the variation of the performance is significantly reduced by adopting VM-separated commit. In a certain specific case of Figure 11(c), Ext4 may perform even better than VM-separated commit as the standard deviation of Ext4 is shown to be very large. Actually, as the isolation of I/O performance among applications is not easy in storage systems, the maximum throughput of Ext4 in Figure 11(c) may cause the performance degradation of other applications. Also, such a variability makes the stable management of given resources difficult.

In summary, the proposed VM-separated commit improves the I/O performance of virtualized systems significantly with respect to the I/O throughput and the performance variability.

VI. RELATED WORKS

As virtualization increases the depth and the complexity of I/O paths, considerable research has been performed on the efficient management of I/O stacks in virtualized systems.

Some studies focus on the optimization of device drivers in virtualized systems. Russel presents a set of virtualized device drivers, called Virtio, that allows explicit cooperation of guest’s device drivers and the hypervisor, thereby eliminating the overhead of emulating physical devices at the hypervisor [4]. Xen’s para-virtualized driver [22] and VMware’s guest tools [21] also improve I/O performance in virtualized systems by providing new I/O device drivers.

There have been several attempts to minimize delays in the I/O path of virtualized systems. Xu et al. accelerate I/O processing for virtual machines by revising IRQ handling mechanisms in multi-core systems [28]. They reduce the IRQ processing latency by processing all I/O requests in a designated core with a very small time slice. Hardware-supported virtualization technologies have also been studied. The Intel VT-d technology allows guests to directly interact with the device assigned by the host, thereby eliminating the virtualization overhead [29]. Le et al. observed that nested interactions of guest and host file systems significantly degrade I/O performance [10]. Based on this observation, they provide suggestions on configuring nested file systems.

Researchers have also focused on optimizing I/O scheduling in virtualized systems. Ongaro et al. explore a variety of schedulers in virtual machines and demonstrate that the performance of I/O schedulers depends heavily on the workload characteristics in virtualized systems [30]. Gulati et al. present a novel I/O scheduling algorithm called mClock that supports QoS of each virtual machine by enforcing control when I/O resource capacity is fluctuating [31]. Boucher and Chandra examine the effectiveness of traditional disk I/O scheduling in virtualized systems [32]. They show that choosing an appropriate combination of I/O schedulers in guest and host systems brings about significant performance gains. Gulati et al. present a storage resource scheduler to manage virtual disk placement and load balancing in virtualized environments, given that live migration of virtual hard disks is possible [33]. Seelam et al. study virtual I/O schedulers to provide performance isolation when various applications and OSs are running simultaneously [34].

VII. CONCLUSION

In this article, we uncovered the inefficiencies of file system commit in virtualized systems. Specifically, we observed the excessively frequent commits and unnecessary storage writes in virtualized systems as the host triggers commit on every write request from all guest machines. To remedy these inefficiencies, we proposed the VM-separated commit, and implemented it on QEMU-KVM and Ext4. Specifically, we devised a data structure that manages the modified file blocks from each guest as a separate list by making use of the hash table with the inodes of VM image files as hash keys and split the running transaction into two based on the hash table upon a guest’s commit request. Measurement studies showed that the proposed VM-separated commit policy improves the I/O throughput by 19.5% on average and up to 64.2%. It also reduces the variability in performance.

REFERENCES

[1] M. Yi, D. H. Kang, M. Lee, I. Kim, and Y. I. Eom, “Performance analyses of duplicated I/O stack in virtualization environment,” in Proc. 10th Int. Conf. Ubiquitous Inf. Manage. Commun. (IMCOM), 2016, pp. 1–6.
[2] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and B. Yassour, “Adding advanced storage controller functionality via low-overhead virtualization,” in Proc. 10th USENIX FAST Conf., 2012, pp. 1–8.
[3] D. Hildebrand, A. Povzner, R. Tewari, and V. Tar-Asov, “Revisiting the storage stack in virtualized NAS environments,” in Proc. Workshop I/O Virtualization, 2011, pp. 1–17.
[4] R. Russell, “Virtio: Towards a de-facto standard for virtual I/O devices,” ACM SIGOPS Operating Syst. Rev., vol. 42, no. 5, pp. 95–103, 2008.
[5] S. Hajnoczi, An Updated Overview of the QEMU Storage Stack. San Francisco, CA, USA: The Linux Foundation, LinuxCon-Japan, 2011.
[6] M. Lee and Y. I. Eom, “Designing transactional hypervisor for improving the performance of Qcow2-based virtual disks,” IEEE Access, vol. 8, pp. 192631–192642, 2020.
[7] KVM. [Online]. Available: http://www.linux-kvm.org
[8] VirtualBox. [Online]. Available: http://www.virtualbox.org
[9] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, “Virtual machine workloads: The case for new NAS benchmarks,” in Proc. 11th USENIX FAST Conf., 2013, pp. 307–320.
[10] D. Le, H. Huang, and H. Wang, “Understanding performance implications of nested file systems in a virtualized environment,” in Proc. 10th USENIX FAST Conf., 2012, pp. 1–13.
[11] A. Gulati, C. Kumar, and I. Ahmad, “Storage workload characterization and consolidation in virtualized environments,” in Proc. 2nd Int. Workshop Virtualization Perform., 2009, pp. 1–4.
[12] G. Casale, S. Kraft, and D. Krishnamurthy, “A model of storage I/O performance interference in virtualized systems,” in Proc. 31st Int. Conf. Distrib. Comput. Syst. Workshops, Jun. 2011, pp. 34–39.
[13] D. Li, W. Chen, Z. Wu, H. Liu, and Y. Tang, “VJournaling: Improving the write performance of a virtualized journaling file system,” in Proc. IEEE SmartWorld, Ubiquitous Intell. Comput., Adv. Trusted Comput., Scalable Comput. Commun., Cloud Big Data Comput., Internet People Smart City Innov. (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Oct. 2018, pp. 1299–1304.
[14] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Analysis and evolution of journaling file systems,” in Proc. USENIX Annual Technical Conf., 2005, pp. 196–215.

[15] E. Lee, S. Hoon Yoo, and H. Bahn, “Design and implementation of a journaling file system for phase-change memory,” IEEE Trans. Comput., vol. 64, no. 5, pp. 1349–1360, May 2015.

[16] E. Lee, H. Kang, H. Bahn, and K. G. Shin, “Eliminating periodic flush overhead of file I/O with non-volatile buffer cache,” IEEE Trans. Comput., vol. 65, no. 4, pp. 1145–1157, Apr. 2016.

[17] E. Lee, H. Bahn, and S. Noh, “Unioning of the buffer cache and journaling layers with non-volatile memory,” in Proc. USENIX Conf. File Storage Technol. (FAST), 2013, pp. 73–80.

[18] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new ext4 filesystem: Current status and future plans,” in Proc. Ottawa Linux Symp., 2007, pp. 1–16.

[19] H. Reiser, ReiserFS. [Online]. Available: http://www.namesys.com/

[20] Accessed: Nov. 18, 2020. [Online]. Available: https://oss.oracle.com/-mason/btrfs/btrfs-design.html

[21] VMware Tools for Linux Guests. [Online]. Available: http://www.vmware.com/support/ws5/doc/ws_newguest_tools_linux.html

[22] Xen Source–Progressive Paravirtualization, pp. 1–8. [Online]. Available: http://xen.org/files/summit_3/xen-pv-drivers.pdf

[23] K. Wolf. (2012). A Block Layer Overview. KVM Forum. [Online]. Available: http://www.linux-kvm.org/page/KVM_Forum_2012.

[24] QEMU. [Online]. Available: http://www.qemu.org

[25] Ubuntu. [Online]. Available: http://ubuntu.com

[26] IOzone. [Online]. Available: http://www.iozone.org/

[27] Filebench. [Online]. Available: https://github.com/filebench/filebench/wiki

[28] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, “vTurbo: Accelerating virtual machine i/o processing using designated turbo-sliced core,” in Proc. USENIX Annual Tech. Conf., 2013, pp. 243–254.

[29] R. Hiremane, “Intel virtualization technology for directed I/O (Intel VT-d),” Technol. @ Intel Mag., vol. 4, no. 10, 2007.

[30] D. Ongaro, A. L. Cox, and S. Rixner, “Scheduling I/O in virtual machine monitors,” in Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE), 2008, pp. 1–10.

[31] A. Gulati, A. Merchant, and P. J. Varman, “mcLock: Handling throughput variability for hypervisor I/O scheduling,” in Proc. 9th USENIX OSDI Conf., 2010, pp. 1–14.

[32] D. Bouchter and A. Chandra, “Does virtualization make disk scheduling passé?” ACM SIGOPS Operating Syst. Rev., vol. 44, no. 1, pp. 20–24, 2010.

[33] A. Gulati, C. Kumar, and I. Ahmad, “Modeling workloads and devices for I/O load balancing in virtualized environments,” ACM SIGMETRICS Perform. Eval. Rev., vol. 37, no. 3, pp. 61–66, Jan. 2010.

[34] S. R. Seelam and P. J. Teller, “Virtual I/O scheduler: A scheduler of schedulers for performance virtualization,” in Proc. Int. Conf. Virtual Execution Environ., 2007, pp. 105–115.

[35] J. Choi, J. Hong, Y. Kwon, and H. Han, “Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface,” in Proc. USENIX Annual Technical Conf., 2020, pp. 1–16.

**HYOKYUNG BAHN** (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a Full Professor of computer science and engineering with Ewha University, Seoul, South Korea. He has published more than 100 articles in leading conferences and journals in these fields, including USENIX FAST, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and ACM TRANSACTIONS ON STORAGE. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. He was a recipient of the Best Paper Awards at the USENIX Conference on File and Storage Technologies, in 2013.

**JISUN KIM** received the B.S. degree in computer science and engineering from Hanshin University, in 2011. She is currently pursuing the Ph.D. degree in computer science and engineering with Ewha University, Seoul, South Korea. Her research interests include operating systems, storage systems, caching algorithms, system optimizations, mobile systems, software platform technologies, block chain technologies, and embedded systems.