IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 16, 2020, accepted October 8, 2020, date of publication October 22, 2020, date of current version November 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033030

Designing Transactional Hypervisor for Improving the Performance of Qcow2-Based Virtual Disks

MINHO LEE[1] **AND YOUNG IK EOM**[2]
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]Department of Electrical and Computer Engineering, College of Computing, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Young Ik Eom (yieom@skku.edu)

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017R1A2B3004660).

ABSTRACT Virtualization has become one of the backbone technologies for constructing cloud servers and data centers, in that the technology supports to partition the physical computing resources of the system into multiple logical ones for concurrently running multiple VMs (virtual machines) in a single physical server. In constructing multiple VM environments, the qcow2 format is widely used as a virtual disk layout due to its attractive features such as high storage space efficiency and high reliability. Unfortunately, according to the policy of the qcow2 format, the hypervisor calls file sync operations excessively during allocating data clusters. Moreover, significant performance degradation may occur due to the CoW (Copy-on-Write) operations that are unconditionally performed by the hypervisor whenever new data clusters are allocated on the virtual disk. In this paper, we introduce a transactional hypervisor that mitigates the performance overhead caused by the data cluster allocation onto the qcow2-based virtual disk. To achieve this, we adopt transactional support to the existing hypervisor and propose a new type of file sync operation, called gsync, which flushes the modified metadata in a bundle. Moreover, we introduce an optimization technique on the CoW mechanism that performs CoW operations selectively according to the amount of data that will be updated on each data cluster. The experimental results clearly show that the proposed hypervisor outperforms the conventional hypervisor in terms of write IOPS (Input/Output Per Second) by up to 78.4%, by reducing the number of file sync calls and that of CoW operations by up to 50.6% and 52.6%, respectively.

INDEX TERMS Virtualization, qcow2 format, data cluster allocation, transactional support, sync operation.

I. INTRODUCTION

Nowadays, several types of computing systems adopt virtualization technology into their systems to fully utilize their own computing resources or to provide diverse computing platforms on a physical machine [1]. Especially, virtualization technologies have become one of the backbone technologies in constructing public cloud systems and data centers [2], [3]. It is because that virtualization technologies facilitate server consolidation of multiple logical servers into a physical one by partitioning the physical computing resources (e.g., CPU, memory, and storage) of the system into multiple logical ones for concurrently running multiple VMs (virtual machines) in a single physical server [4]–[6]. Most of the virtualized systems employ a virtual disk for each VM as its own storage, where the virtual disk takes the role of a container for storing VM data. The virtual disk is commonly created in the form of a file at the host-level, and therefore, I/O requests of each VM are transferred to the virtual disk through the hypervisor that is a software-layer for mediating between the VMs and the host [5], [7]. The hypervisor not only transforms the I/O requests issued by each VM to the file read/write operations on the virtual disk, but also reflects them into the underlying storage device by delivering them to the host. However, in spite of the tremendous growth of virtualization technologies, the virtualized systems inevitably involve significant performance overhead caused by their complicated I/O path, compared to non-virtualized systems, where the virtualized

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone.

systems have their I/O stack (e.g., page cache, filesystem, I/O scheduler, and device driver) duplicated in both the host and the VMs [8]–[12].

Meanwhile, various virtual disk formats, such as raw and qcow2, are provided by the QEMU-KVM hypervisor that is one of the representative hypervisors for Linux systems [4], [6], [13]. For the raw-based virtual disk, the hypervisor simply creates a pre-allocated fixed-size binary file, and then it stores VM data into the virtual disk without any additional operations. On the contrary, the qcow2-based virtual disk initially requires a small-size binary file that contains only the essential data for the virtual disk and basic information on the read-only *backing files*, in which the guest operating system and the essential utilities are already installed [14]. After creation of the qcow2-based virtual disk, the hypervisor newly allocates data clusters to increase the size of the virtual disk according to the on-demand policy of the qcow2 format, where the data cluster is a basic unit for handling VM data in the virtual disk. Due to such characteristics of the qcow2 format, the hypervisor can achieve high storage space efficiency in multiple VM environments. In addition, the qcow2-based virtual disk can provide high reliability via its snapshot feature that makes it more suitable for constructing multiple VM environments, contrary to the raw-based one.

Unfortunately, despite the attractive features of the qcow2 format, it has critical drawbacks with respect to I/O performance. With the qcow2-based virtual disks, the hypervisor heavily relies on the file sync calls (e.g., `fsync` and `fdatasync`) to guarantee strong consistency of the virtual disk whenever any data cluster or its corresponding metadata is updated [13]–[15]. Especially, such data updates frequently occur during the process of data cluster allocation on the qcow2-based virtual disk. Whereas file sync calls guarantee the durability of the updated data, they block the execution of the hypervisor until the modified data are entirely reflected into the underlying storage device [16]–[18]. Furthermore, whenever new data clusters are allocated into the virtual disk, the hypervisor generates massive read/write operations caused by inefficient CoW (Copy-on-Write) mechanism of the qcow2 format [14], [19]. Since the qcow2-based virtual disk works in a grow-only manner until its size reaches the configured maximum size of the virtual disk, the hypervisor continuously induces performance overhead by unconditionally incurring CoW operations between the *backing files* and the qcow2-based virtual disk.

In this paper, the proposed hypervisor is designed to answer the simple question: *how to mitigate the performance overhead caused by the data cluster allocation of the qcow2-based virtual disk?* To resolve this problem, we propose a transactional hypervisor which improves the performance of the qcow2-based virtual disk by reducing excessive file sync calls and unnecessary CoW operations in the process of data cluster allocation. We make the following specific technical contributions:

- **Transactional support:** We found that the hypervisor acquires `qemu_lock` to handle a write request transferred from a VM. Then, it splits the write request in units of data clusters and reflects them one by one into the qcow2-based virtual disk with the file sync calls. To mitigate the performance degradation caused by the chracteristics of the qcow2-based virtual disk, we adopt transactional support that is widely used for application-level and system-wide data consistency. By doing so, we improve the I/O performance of the qcow2-based virtual disk while guaranteeing the atomicity of multiple data cluster updates.
- **Group-sync (`gsync`):** Through several analyses, which will be discussed in detail in Section III, we found that the conventional hypervisor provides strong consistency for the qcow2-based virtual disk by frequently incurring file sync calls during the process of data cluster allocation. In order to reduce the frequency of file sync calls, the proposed hypervisor delays the file sync operations until all the data clusters of a transacation are allocated. It then reflects the modified data of the virtual disk at once into the underlying storage device via a new type of file sync operation, called `gsync` (group-sync), after the transaction is completely finished.
- **Selective CoW mechanism:** With some experimental results, which will be given in Section III, we found that the conventional hypervisor induces performance degradation by generating excessive read/write operations due to the inefficient CoW mechanism of the qcow2 format during the process of data cluster allocation. For this reason, we introduce an optimization technique on the CoW mechanism to selectively perform CoW operations according to the amount of data that will be updated on each data cluster.

We have implemented the proposed hypervisor based on QEMU-KVM (ver. 4.0.0) and performed several experiments to verify its effectiveness. Our experimental evaluation clearly demonstrates that the proposed hypervisor improves the write IOPS (Input/Output Per Second), by up to 78.4%, compared with the conventional hypervisor. Also, our hypervisor significantly reduces the number of file sync calls and that of CoW operations, by up to 50.6% and 52.6%, respectively, compared with the conventional hypervisor. Importantly, despite the recovery cost can slightly be high by adopting transactional support, our hypervisor efficiently increases storage space efficiency and I/O performance.

The remainder of the paper is organized as follows. Section II presents the background. Section III analyzes performance overhead of the qcow2-based virtual disk in various aspects. In Section IV, we describe the design and implementation details of the proposed hypervisor. We show the evaluation results and discuss the limitation of our hypervisor in Section V. Finally, we present related work in Section VI and conclude the paper in Section VII.

II. BACKGROUND

A. THE qcow2-BASED VIRTUAL DISK

In contrast with the raw-based virtual disk that is configured with a fixed-size binary file, the qcow2-based virtual disk initially requires a small-size file. Initial qcow2-based virtual disk contains only a few data clusters and the associated qcow2 metadata for accessing read-only *backing files*. By using the qcow2 metadata, the hypervisor gets the stored data in the *backing files*, where the guest operating system and essential utilities are installed [13]–[15]. When the write requests are transferred from the VMs, the hypervisor tries to reflect the write request into the virtual disk by allocating new data clusters, where the data cluster is a basic unit of space allocation on the qcow2-based virtual disk, with a default size of 64KB. To handle the VM data in the data clusters, the hypervisor exploits additional qcow2 metadata, including the qcow2 header, two-level lookup tables (e.g., L1 and L2 tables), and a reference table, as shown in Figure 1. In the case of qcow2 header, its role is approximately similar to that of the superblock of Unix/Linux filesystems, where it contains the basic layout information of the virtual disk such as the addresses of the lookup tables and the reference table. The two-level lookup tables consist of L1 and L2 tables for address translation. The L1 and L2 tables have an array of virtual disk offsets to L2 and data clusters, respectively. Lastly, the reference table is used to perceive the modification of data clusters to support the snapshot feature by recording the reference count of each data cluster.

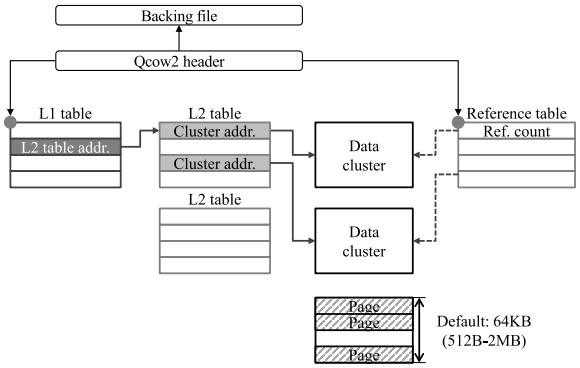


FIGURE 1. The structure of the qcow2-based virtual disk.

B. DATA CLUSTER ALLOCATION SCHEME OF THE qcow2-BASED VIRTUAL DISK

The qcow2-based virtual disk works in a grow-only manner from the initial state. In order to reflect the write requests transferred from VMs, the hypervisor should allocate new data clusters and store the associated qcow2 metadata into the virtual disk, until the size of the virtual disk reaches the configured maximum [14]. Figure 2 depicts the process of data cluster allocation on the virtual disk according to the qcow2 format. The detailed procedure is as follows. 1) When a write request is transferred from a VM, the hypervisor acquires `qemu_lock`. 2) Then, the hypervisor checks the dependency with other write requests in-flight. 3) The hypervisor confirms the offset of the two-level lookup tables and the reference table for recording the address of each data cluster into the tables. 4) The hypervisor also determines whether there are adjacent CoW regions that can be merged for the write request, using `QCowL2Meta`, which is a data structure to handle the CoW region of each data cluster in the hypervisor-level. 5) Then, the hypervisor actually performs the CoW operation onto the virtual disk with the contents of the *backing file* and the write request. 6) Finally, the hypervisor links the data clusters and the entries of the L2 table. After the reference table is also updated, the hypervisor reflects the write request transferred from the VM into the virtual disk.

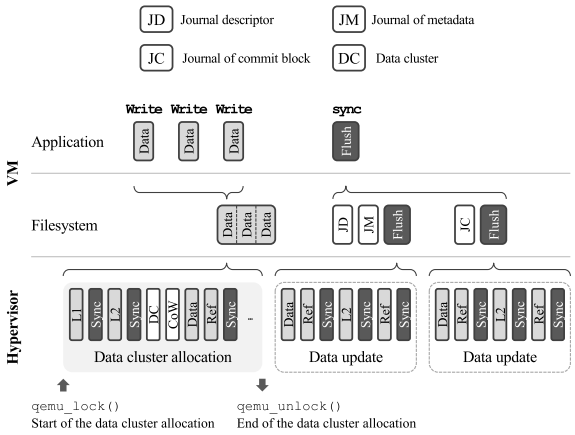


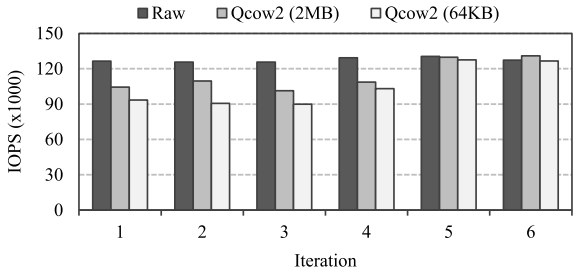
FIGURE 2. The overview of the process of data cluster allocation on the qcow2-based virtual disk.

III. MOTIVATION

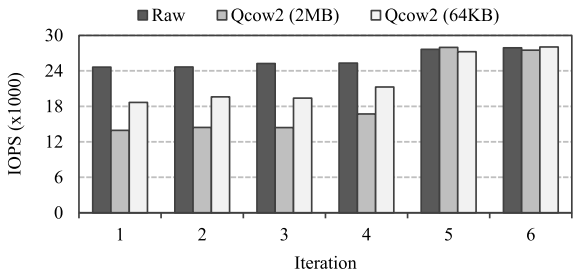
A. THE PERFORMANCE OVERHEAD OF THE qcow2-BASED VIRTUAL DISK

In this section, we study the behavior of conventional hypervisor to understand the performance characteristics of the qcow2-based virtual disk. To achieve this, we first measured the write IOPS using the FIO benchmark [20]. Also, we collected information on the number of file sync calls and the amount of data read/written caused by CoW operations during the process of data cluster allocation according to the policy of the qcow2 format. For the experiments, we launched a VM using the QEMU-KVM hypervisor with two types of virtual disks, the raw-based and qcow2-based. The data cluster size of the qcow2-based virtual disk was configured as 64KB or 2MB. Running the FIO benchmark inside the virtual machine, we created 4 files and sequentially wrote 4GB of data, in units of 4KB, onto each file. Then, we iterated this procedure six times until the capacity of the virtual disk becomes full (See Section V for the detailed description of the experimental environment). To confirm the impact of the write/sync amplification problem on the performance of virtual disk in more detail, we also varied the frequency of `fsync` calls from none to once every 256 writes in the experiments.

Our observations from the experiments can be described as follows. First, we found that the qcow2-based virtual disk has lower performance in terms of write IOPS, by up to 28.4%, compared with the raw-based disk, as shown in Figure 3. This result comes from the fact that the conventional hypervisor frequently incurs file sync calls to guarantee strong consistency on the data of the qcow2-based virtual disk whenever it allocates new data clusters. This procedure negatively affects the performance of the virtual disk because each file sync call usually suspends execution of the hypervisor for upcoming write operations until the modified data are entirely reflected onto the underlying storage device [16], [21], [22]. Furthermore, such file sync calls amplify write/sync operations by triggering journaling of the host filesystem, especially in the Ext4 filesystem, and heavily generating additional journal writes with frequent flush operations [13]. In this way, the file sync calls during the data cluster allocation process significantly degrade the performance of the virtual disk along with the write and sync amplification problem.



(a) No-`fsync` mode



(b) `fsync` mode (`fsync` frequency: once for every 256 writes)

FIGURE 3. The write IOPS for each virtual disk format.

Second, we found that the qcow2-based virtual disk whose cluster size is 64KB has higher write IOPS by up to 35.7% in `fsync` mode compared with that of the data cluster size 2MB, as shown in Figure 3b. This is because the qcow2 format induces performance degradation due to its CoW mechanism. In accordance with the policy of the qcow2 format, the conventional hypervisor unconditionally pursues sharing the contents of allocated data clusters on the *backing file* and tries to execute CoW operations before it reflects the write requests transferred from VMs to new data clusters. If there are no existing data clusters associated with the offset of write requests in the *backing file*, the hypervisor simply performs zero-write on the new data clusters of the qcow2-based virtual disk. Otherwise, the hypervisor executes CoW operations on each data cluster allocated. With the CoW operations, the hypervisor reads the existing data cluster in the *backing file* using the offset of the write requests and copies the content of the corresponding data cluster to a new data cluster on the virtual disk, reflecting the write request transferred from the VM into the new cluster. In this way, as the size of the data cluster becomes larger, the cost of the CoW operation increases regardless of the I/O size issued by the application, as shown in Table 1.

TABLE 1. The overhead of the CoW mechanism for qcow2 format (`fsync` frequency: once for every 256 writes).

DC-BL	# CoW breaks	# Writes (Amount of write)	# Reads (Amount of read)
64KB-4KB	16653	178 (0.087 GB)	247 (0.008 GB)
64KB-64KB	14620	3198 (0.145 GB)	4029 (0.013 GB)
2MB-4KB	8300	8299 (16.211 GB)	8257 (8.183 GB)
2MB-64KB	7283	6861 (15.241 GB)	8433 (7.859 GB)

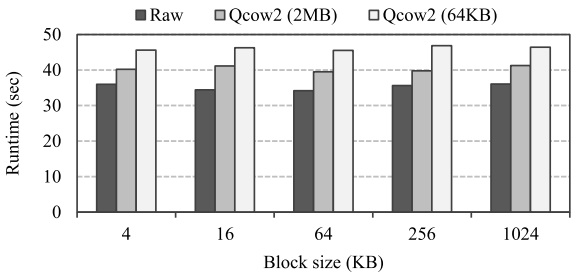
‡ DC: Data cluster size
† BL: Block size (The I/O size issued by the application)

Third, we found that the number of file sync calls increases considerably when the data cluster size becomes small. As shown in Table 2, the case of the data cluster size 64KB incurs more file sync calls, by up to 1.7 times, compared with the case of data cluster size 2MB. The reason is that the hypervisor merely iterates the procedure of data cluster allocation until the total size of the allocated data cluster on the virtual disk is equal to the size of the write requests transferred from the VM. This fact indicates that the size of data clusters has more impact on the performance of the qcow2-based virtual disk, compared with that of write requests issued from applications inside VMs. Figure 4 clearly demonstrates that the performance gap between the two data cluster sizes (64KB and 2MB) can be seen regardless of the block size issued by the application-level. Meanwhile, in `fsync` mode, the size of write request transferred from VMs becomes smaller compared with that of the no-`fsync` mode. As the size of write requests decreases, the overhead of CoW operations has more impact on the performance of the qcow2-based virtual disk, rather than that of file sync calls. The qcow2-based virtual disk whose data cluster size is 64KB has a lower runtime compared with that of data cluster size 2MB as shown in Figure 4b.

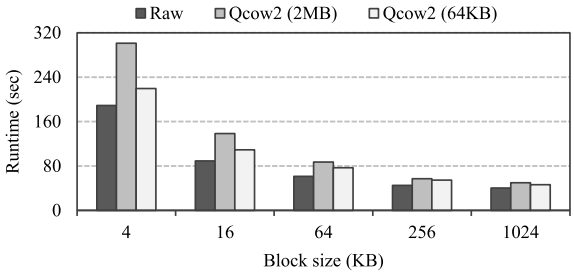
TABLE 2. The number of file sync operations (`fsync` frequency: once for every 256 writes).

DC-BL	Write IOPS	# File sync operations
64KB-4KB	19110	9678
64KB-64KB	3407	630
2MB-4KB	13930	5790
2MB-64KB	3010	405

‡ DC: Data cluster size
† BL: Block size (The I/O size issued by the application)



(a) No-`fsync` mode



(b) `fsync` mode (`fsync` frequency: once for every 256 writes)

FIGURE 4. The runtime for each virtual disk format.

B. THE PRE-ALLOCATION OF THE qcow2-BASED VIRTUAL DISK

The QEMU-KVM hypervisor provides a pre-allocation option to mitigate the performance overhead induced in the process of data cluster allocation for the qcow2 format. When the qcow2-based virtual disk is created with this option, some metadata and data clusters are pre-allocated. This option consists of four modes: *none*, *metadata*, *falloc*, and *full*. In *metadata* mode, the hypervisor pre-allocates all the metadata space needed, up to the maximum size of the qcow2-based virtual disk. The *falloc* and *full* modes also pre-allocate the whole metadata space along with the space for data clusters on the virtual disk. They are different in that the former mode does not record the information on the allocated data clusters into the L2 table.

To analyze the effectiveness of the pre-allocation option, we measured the performance of the qcow2-based virtual disk according to mode, in terms of write IOPS and creation time. On the *none* and *metadata* modes, the write IOPS of the qcow2-based virtual disk is lower, by up to 45.2%, than that of the raw-based virtual disk, as shown in Figure 5. It is because these modes need to allocate new data clusters for the write requests transferred from VMs. Meanwhile, since the *falloc* and *full* modes pre-allocate data clusters on the virtual disk, their write IOPSs are similar to those of the raw-based virtual disk. However, these modes have a limitation in that the storage space efficiency of the virtualized system may be dropped considerably. Table 3 shows the creation time of each virtual disk. Since the hypervisor simply creates the raw-based virtual disk through the fixed-size binary file, the creation time of the raw-based virtual disk is shortest, as shown in Table 3. On the contrary, the creation time of qcow2-based virtual disk increases significantly, by up to 96.1 times, according to

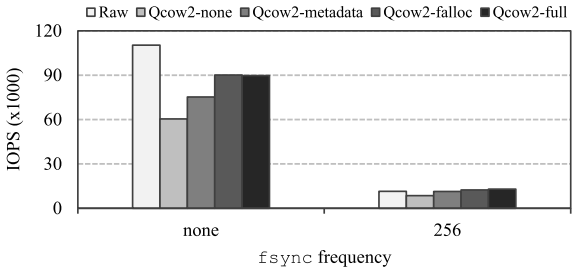


FIGURE 5. The effectiveness of the pre-allocation option.

TABLE 3. The creation time of each virtual disk.

	Raw	Qcow2			
Option	-	None	Metadata	Falloc	Full
Creation time (s)	0.29	1.76	6.25	4.38	421.22

the pre-allocation option, as shown in Table 3. It is because the pre-allocation options (i.e., *metadata*, *falloc*, and *full*) additionally involve performance overhead caused by data cluster allocation according to creation procedure.

IV. DESIGN AND IMPLEMENTATION

Our work begins with a simple question: *how to mitigate the performance overhead caused by the data cluster allocation process of the qcow2 format?* To answer the question, this section discusses the following three differentiated design principles.

- **Transactional support:** One simple solution for reducing the number of file sync calls in the process of data cluster allocation is to adopt transactional support at the hypervisor-level. Traditionally, transactional support is commonly used to guarantee the atomicity of multiple page updates for application-level and system-wide consistency. The transactional support provides better I/O performance by reflecting modified data in a bundle into the underlying storage device.
- **Integrating the file sync calls:** Conventionally, the file sync call is widely used for ensuring application-level crash consistency. Note that our observations show that such file sync calls usually lead to a block in the processing of upcoming write operations until the modified data is completely reflected into the underlying storage device. For this reason, integrating the file sync calls during the process of data cluster allocation is crucially concerned to improve the performance of the virtual disk.
- **Optimizing the CoW mechanism:** In terms of storage space efficiency, the qcow2 format may be useful in that it employs the CoW mechanism with the *backing file* when the hypervisor creates the virtual disk. However, this mechanism should be re-thought because the hypervisor generates unnecessary massive read/write traffic without any consideration on the amount of data that will be updated by the write requests transferred from VM.

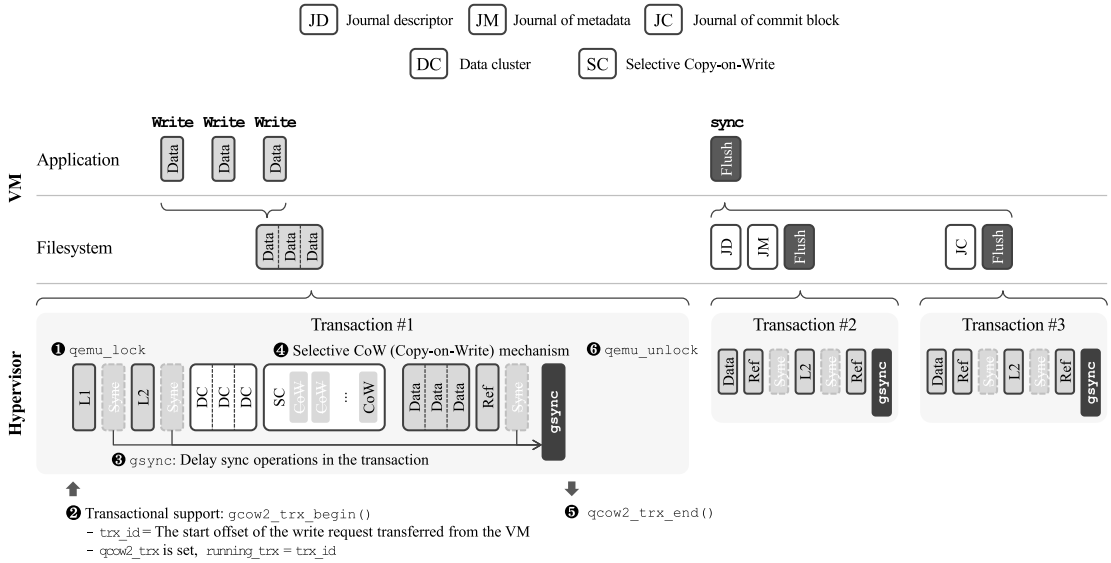


FIGURE 6. Overview of the proposed hypervisor.

A. TRANSACTIONAL SUPPORT FOR THE qcow2-BASED VIRTUAL DISK

To adopt transactional support at the hypervisor-level and efficiently optimize the data cluster allocation process of the qcow2 format, the proposed hypervisor first needs to define how to compose multiple data clusters into one transaction. In the conventional virtualized system, each VM tries not only to merge the write requests according to the re-order and merge policies of its own I/O stack but also to transfer them as a sequential pattern for storage performance gain. It contributes to an increase in the size of write requests transferred from VMs by consolidating multiple write requests issued by applications inside VMs. Moreover, since the hypervisor usually attempts to merge the write requests transferred from VMs using its own scatter-gather operation, the size of the write requests can be larger compared with that of the data cluster. From these observations, we define a single write request, which is merged at the hypervisor-level, as a transaction. Then, the proposed hypervisor allocates multiple data clusters that are needed to carry out the write request in a transactional way.

Figure 6 describes the process of the data cluster allocation in our hypervisor. Similarly to the conventional hypervisor, the proposed hypervisor first calculates how many data clusters should be newly allocated into the virtual disk to carry out the write requests transferred from VMs, using the data cluster allocation scheme of the qcow2 format. Moreover, to guarantee data consistency of the qcow2-based virtual disk, our hypervisor also designates the code region for the data cluster allocation as a critical section and attempts to acquire the `qemu_lock` before it enters the code region. At the time of acquiring the `qemu_lock`, our hypervisor assembles the data clusters needed to reflect the write request and composes them as a transaction. It also considers the transaction being started at the time of acquiring `qemu_lock`.

In order to optimize the process of data cluster allocation in a transactional way, our hypervisor additionally employs two variables (`qcow2_trx` and `running_trx`) for the data structure `BDRVQcow2State` that contains the essential information on the qcow2-based virtual disk during execution of the hypervisor. The variable `qcow2_trx` is employed to mark that each write request is handled in a transactional way, and `running_trx` exhibits which transaction is currently in progress. Moreover, to identify the transaction among the write requests in-flight, the start offset of the write request is assigned as a `trx_id` when a write request is transferred from the VM. After the proposed hypervisor acquires the `qemu_lock` (Figure 6 ❶), `qcow2_trx_begin()` sets the `qcow2_trx` and assigns `trx_id` to the `running_trx` to signify that the corresponding transaction has started (Figure 6 ❷). If the write request conflicts with other requests, the proposed hypervisor suspends the execution of the corresponding `qemu_coroutine`, which is a QEMU internal API used for efficiently managing the callback function in the event-driven program, using `qemu_co_queue_wait` function until the dependency is resolved. Otherwise, the hypervisor proceeds the data cluster allocation. The proposed hypervisor also checks the offset of the two-level lookup and reference tables. When there is no free space to store the address of each data cluster in these tables, our hypervisor extends the qcow2 metadata region by additionally allocating new data clusters for them. After that, our hypervisor stores the qcow2 metadata associated with the lookup and reference tables. Otherwise, the proposed hypervisor obtains the information on the two-level lookup and reference tables, and then actually allocates data clusters. During the data cluster

allocation, our hypervisor delays sync operations that are triggered whenever data clusters and associated qcow2 metadata are updated (Figure 6 ❸). When our hypervisor selectively performs CoW operations (Figure 6 ❹) and entirely allocates the data clusters in the qcow2-based virtual disk, it regards that the transaction is finished. Then, the proposed hypervisor triggers `qcow2_trx_end()` which unsets the `qcow2_trx` and reinitializes `running_trx` to handle the next transaction (Figure 6 ❺). If the process of data cluster allocation terminates abnormally, our hypervisor tries to revert the changes on the virtual disk by undoing the performed operations with `qcow2_abort()`. Otherwise, the hypervisor finally releases the `qemu_lock` (Figure 6 ❻) and tries to reflect the modification of virtual disk into the underlying storage device.

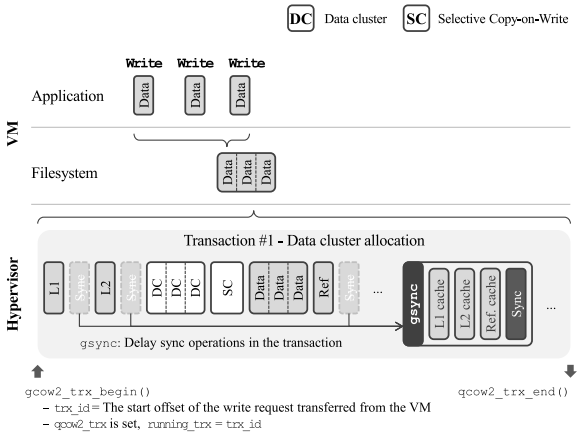


FIGURE 7. Procedure of `gsync`.

B. THE GROUP-SYNC (GSYNC) OPERATION

In the process of data cluster allocation, the qcow2 metadata must be generated for storing the information on the new data clusters before the hypervisor begins to allocate the data clusters on the qcow2-based virtual disk. In addition, to record the generated qcow2 metadata into the associated tables, the hypervisor confirms whether there is free space for recording the qcow2 metadata into the L1/L2 table and reference table or not. If there is no free space in the associated tables in the virtual disk, the hypervisor extends the qcow2 metadata regions by additionally allocating new data clusters for the tables. Otherwise, the hypervisor records the qcow2 metadata on the tables, and then calls the file sync operation for durability of the modified data by completely reflecting them into the underlying storage device.

Since the conventional hypervisor allocates new data clusters one by one by triggering the file sync call for durability of the modified data, it is inevitable that excessive file sync calls occur during the process of data cluster allocation to the virtual disk. To improve the performance of the qcow2-based virtual disk, we propose a new type of file sync operation, called `gsync` (group-sync), which flushes the modifications of the virtual disk in a bundle into the underlying storage device. With the `gsync`, our hypervisor can efficiently guarantee the atomicity of transactions and the durability of the modified data when new data clusters are allocated into the virtual disk. Figure 7 presents the detailed procedure of the `gsync` operation of our hypervisor. When a write request is transferred from a VM, our hypervisor first assigns the write request as the transaction. Then, if the associated transaction begins, the proposed hypervisor starts to allocate new data clusters. Until the transaction finishes its execution, our hypervisor delays the file sync calls instead of immediately triggering them whenever a new data cluster is allocated and the corresponding qcow2 metadata is updated. If `qcow2_trx` is set and `running_trx` has `trx_id` by the transferred write request, the proposed hypervisor leaves out the file sync call for modification of the virtual disk because the transaction is still in progress. Instead, until the associated transaction is entirely completed, the proposed hypervisor temporarily records metadata changes into the in-memory cache (e.g., the L2 table cache and the reference table cache), and then sets the `use_lazy_flush` bit on the cache entry to indicate that the cache entry should be flushed. With completion of the transaction, our hypervisor triggers the `gsync` to ensure the durability of modified data. After that, the `gsync` searches the in-memory caches to collect the entries whose `use_lazy_flush` bit is set and flushes them at once into the underlying storage device.

C. THE SELECTIVE CoW MECHANISM

The final approach to mitigate the performance degradation of the qcow2-based virtual disk is to optimize the inefficient CoW mechanism of the qcow2 format. The conventional hypervisor unconditionally incurs CoW operations onto the virtual disk with the contents of the *backing file* and the write request during the data cluster allocation process. To mitigate it, we introduce a selective CoW mechanism that efficiently eliminates unnecessary CoW operations onto the virtual disk by considering the amount of data that will be updated on the data cluster.

In the case of the conventional hypervisor, it checks the feasibility of allocating new data clusters contiguously onto the virtual disk before actually reflecting the write request. When contiguous data clusters cannot be allocated, the hypervisor gathers the data clusters necessary for reflecting the write request by scanning the virtual disk. While checking the feasibility of contiguous allocation, the hypervisor creates an object of the data structure `QCowL2Meta`, which is used for recording the information on the data clusters that will be allocated. Using the `QCowL2Meta` object, the hypervisor can simply perform the CoW operations.

In contrast with the conventional hypervisor, the proposed hypervisor proceeds two additional steps to efficiently eliminate unnecessary CoW operations during the data cluster allocation. Algorithm 1 depicts how the proposed hypervisor classifies `QCowL2Meta` objects into two groups and selectively performs the CoW operations for them. To support the selective CoW mechanism, the proposed hypervisor adds a

qcow2_cow bit to the data structure QCowL2Meta. The qcow2_cow bit takes the role of indicating whether it performs the CoW operations to the QCowL2Meta object or not. The detailed procedure of the selective CoW mechanism of the proposed hypervisor is as follows.

Algorithm 1 The Selective CoW Mechanism

```
Result: Set qcow2_cow bit of each QCowL2Meta object and
        perform the selective CoW mechanism
Input: BlockDriverState bs, QCowL2Meta l2meta;
Function perform_selective_cow(bs, l2meta):
    rq_offset ← the write request offset; ;
    rq_bytes ← the write request size; ;
    for each l2meta in the transaction do
        cr_offset ← the cow region offset of l2meta;
        cr_bytes ← the cow region size of l2meta;
        if cr_offset > rq_offset && cr_bytes < rq_bytes then
            l2meta.qcow2_cow = false;          ▷ (#1);
        else
            l2meta.qcow2_cow = true;           ▷ (#2);
        end if
    end
    for each l2meta in the transaction do
        if l2meta.qcow2_cow == true then
            perform_cow(bs, l2meta);           ▷ (#3);
        else
    end
End Function
```

The first step classifies the allocated data clusters into two groups by checking the QCowL2Meta object. One is for the data clusters whose CoW region is included in the range of the write request transferred from VM (#1). It means that they are fully updated by the contents of the write request. The other is for the data clusters that will be partially updated by the contents of the write request. In the case of data clusters that belong to the second group, the start offset of their CoW region is not included in the range of the write request, or the end offset of their CoW region is placed beyond that of the write request (#2). For these data clusters, the CoW operations should be performed with the contents of the *backing file* according to the policy of the qcow2 format. Thus, if there are data clusters that should be partially updated by the write request, the proposed hypervisor separates the QCowL2Meta objects and sets its qcow2_cow bit to perform the CoW operation. Otherwise, our hypervisor unsets qcow2_cow bit of the QCowL2Meta object to bypass the CoW operation.

The second step is selectively performing the CoW operations with the allocated data clusters and the corresponding QCowL2Meta objects. Using the classified QCowL2Meta objects, if their qcow2_cow bit is unset, the proposed hypervisor does not perform the CoW operations and just reflects the write request onto the data clusters, as shown in Figure 8. Otherwise, our hypervisor identifies the CoW region of each data cluster with the offset of the write request transferred from the VM, and then performs the CoW operations onto the virtual disk with the contents of the *backing file* and the write request (#3). Finally, the proposed hypervisor not only

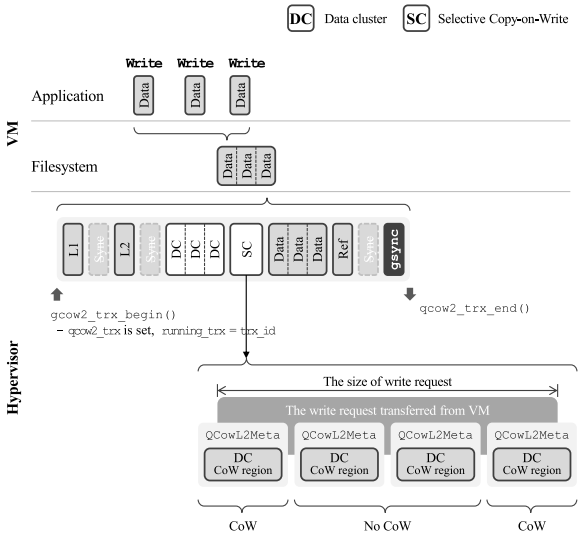


FIGURE 8. Selective CoW mechanism.

obtains the information on the allocated data clusters and the mapping table but also links each data cluster with the entries of the L2 table by recording the offset of each data cluster into the entries of the table.

V. PERFORMANCE EVALUATION

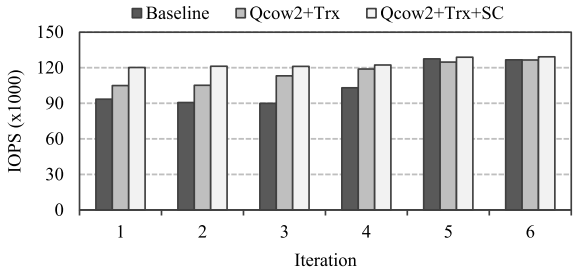
A. EXPERIMENTAL SETUP

In order to verify the performance benefits of our hypervisor, we implemented the proposed hypervisor based on QEMU-KVM (ver. 4.4.0) and performed experiments on the system equipped with Intel Core i7-3770 CPU, 32GB of DRAM, and 256GB Samsung 860 pro SSD as a secondary storage device. We launched a VM that is configured to have four vCPUs, 8GB of memory, and 64GB of virtual disk. Ubuntu 14.04 LTS with Linux kernel (ver. 5.0.1) was installed on both the host and the guest. Ext4 filesystem, which is commonly used from mobile devices to cloud servers, was installed on the physical secondary storage device and virtual disk. For the experiments, we created two types of virtual disk, including the raw-based one and the qcow2-based one. In the case of qcow2-based virtual disk, the size of its data cluster was configured as 64KB. To measure the write IOPS for each virtual disk, we used the FIO benchmark that is the representative I/O benchmark to evaluate filesystems and storage devices [20]. With the FIO benchmark, we created four files and sequentially wrote of 4GB data on each file inside the VM. Then, we iterated this procedure six times until the capacity of the virtual disk was full. In addition, to understand the performance impact of our hypervisor in more detail, we varied the frequency of fsync calls from none to once every 256 writes. We also modified the QEMU-KVM source code to collect information on the qcow2-based virtual disk such as the number of file sync calls and the amount of read/write operations, which are induced by the CoW operations.

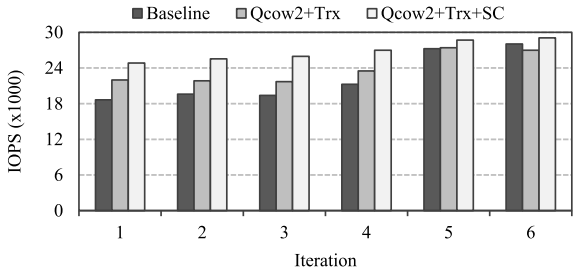
B. EXPERIMENTAL RESULTS

For evaluation, we re-performed the motivation experiments detailed in Section II with our hypervisor. In our experimental results, baseline presents that conventional hypervisor allocates new data clusters to the virtual disk according to the policy of the qcow2 format. *Trx* and *SC* indicate the adoption of the transactional support with `gsync` and applying the selective CoW mechanism to the proposed hypervisor, respectively.

Figure 9 presents the performance comparison of the baseline and our hypervisor. Assuming that there are no `fsync` operations generated while running the FIO benchmark, the write IOPS of adopting the transactional support with `gsync` (Qcow2+*Trx*) is higher, by up to 25.6%, than that of the baseline, as shown in Figure 9a. It is because the proposed hypervisor allocates multiple data clusters, as one transaction, into the qcow2-based virtual disk at once.



(a) No-`fsync` mode



(b) `fsync` mode (`fsync` frequency: once for 256 writes)

FIGURE 9. The write IOPS.

Furthermore, our hypervisor does not immediately trigger the file sync calls whenever metadata update occurs in the process of data cluster allocation. Instead, our hypervisor delays the file sync calls by temporarily recording the modified metadata into the in-memory cache such as L1/L2 and reference caches, and then it marks the `use_lazy_flush` bit of each cache entry. When the transaction finishes its execution, it guarantees consistency of the virtual disk via `gsync` that flushes the modified metadata into the virtual disk as one transaction. By doing this, our hypervisor efficiently allocates the data clusters without the performance overhead caused by the file sync calls. In contrast to our hypervisor, the conventional hypervisor induces considerable performance overhead when it reflects the write requests transferred from VMs because it allocates new data clusters one by one in the virtual disk, with each file sync call. Unfortunately, in the conventional hypervisor, such excessive file sync calls is inevitable until the write request is entirely reflected into the virtual disk.

In the case of applying the selective CoW mechanism with transactional support and `gsync`, the write IOPS of our hypervisor (Qcow2+*Trx*+*SC*) is higher, by up to 34.6%, than that of baseline, as shown in Figure 9a. The reason is that our hypervisor considerably reduces the number of read/write operations by eliminating unnecessary CoW operations for the virtual disk, as shown in Table 4. Using the selective CoW mechanism, the proposed hypervisor selectively performs CoW operations according to its criteria, in which the CoW operations are triggered only when the size of modified data is smaller than that of the data cluster. On the contrary, the conventional hypervisor unconditionally performs CoW operations without consideration on the amount of data that will be updated by the write requests transferred from VMs.

TABLE 4. The amount of read/write operations.

	# CoW breaks	The amount of read/writes (GB)
Baseline	14339	0.304 / 8.443
Qcow2+trx+SC	6895	0.073 / 2.195

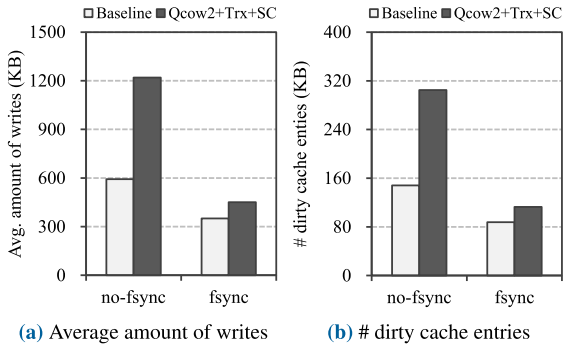
(a) No-`fsync` mode

	# CoW breaks	The amount of read/writes (GB)
Baseline	16653	0.008 / 0.087
Qcow2+trx+SC	8074	0.005 / 0.023

(b) `fsync` mode (`fsync` frequency: once for 256 writes)

When the FIO benchmark runs with `fsync` mode, journal writes can be extensively generated in the host/guest file system, incurring significant sync amplification. For this reason, the write IOPS of both our hypervisor and the baseline is considerably lowered, when compared with the case of running with no `fsync` mode. However, with the `gsync`, our hypervisor has higher write IOPS, by up to 33.8%, when compared with baseline, as shown in Figure 9b. Furthermore, our hypervisor has no memory overhead because it only marks the `use_lazy_flush` bit while the conventional hypervisor sets the `dirty` bit to each modified cache entry.

Furthermore, to confirm the memory overhead of our hypervisor, we measured the average amount of write requests that are transferred from the hypervisor to the host and checked the memory utilization related to the write requests. Figure 10a shows the average amount of write requests for our hypervisor is larger, by up to 2.1 times, than that of the baseline. The reason is that our hypervisor keeps the size of the write request transferred from VM by delaying sync operations triggered by the modification of qcow2 metadata until the transaction is finished. However, since the conventional hypervisor immediately updates the qcow2 metadata with sync operations, it leads to a decrease in the amount of write requests that are transferred to the host. In terms of memory utilization, both the proposed hypervisor



(a) Average amount of writes (b) # dirty cache entries

FIGURE 10. Average amount of writes and memory utilization.

and the conventional one fully use the in-memory cache during execution. In contrast with the conventional hypervisor, our hypervisor generates more dirty cache entries in that it transfers more write requests to the host. However, the portion of dirty cache entries is quite small compared with the configured cache size (e.g., L2 cache default size: 2MB). Figure 10b clearly demonstrates that the amount of qcow2 in-memory cache is enough to store the qcow2 metadata generated while the proposed hypervisor allocates multiple data clusters as a transaction until it triggers `gsync`.

In order to evaluate the effort of our approach in more detail, we evaluated the write IOPS by varying the block size from 4KB to 1MB with the FIO benchmark. Figure 11 shows the write IOPS of our hypervisor normalized to that of baseline. Since the size of transactions becomes relatively small as block size shrinks, the write IOPS of our hypervisor is slightly higher, by up to 19.3%, than that of baseline when the block size is smaller than the data cluster size (64KB). However, as block size becomes large, our hypervisor achieves significant performance improvement, by up to 42.8%, as shown in Figure 11. The reason is that the effectiveness of our hypervisor gets better as the number of data clusters that should be allocated into the qcow2-based virtual disk increases. To verify the performance impact of the proposed hypervisor from various aspects, we also recorded the number of file sync calls at the hypervisor-level while the FIO benchmark runs. There is a trade-off between the number of file sync calls and I/O performance. It is because that the file sync call suspends execution of hypervisor until the modified data are completely reflected into the underlying storage device to guarantee strong consistency for the virtual disk. Figure 12 shows the number of file sync calls of our hypervisor normalized to that of baseline. As shown in Figure 12, the number of file sync calls of our hypervisor is lower, by up to 25.8%, than that of baseline, when block size is smaller than data cluster size. But, similar to the performance trend in terms of write IOPS, the proposed hypervisor considerably reduces the number of file sync calls as block size increases, by up to 50.6%, when compared with baseline.

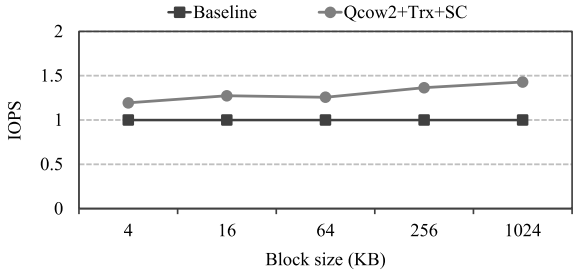


FIGURE 11. The write IOPS normalized to that of baseline.

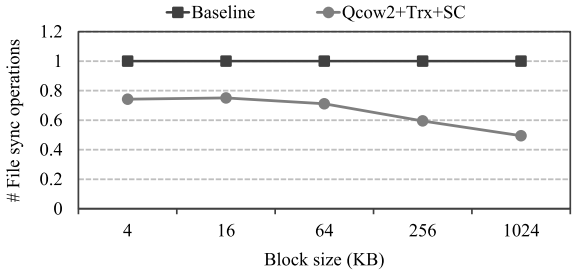
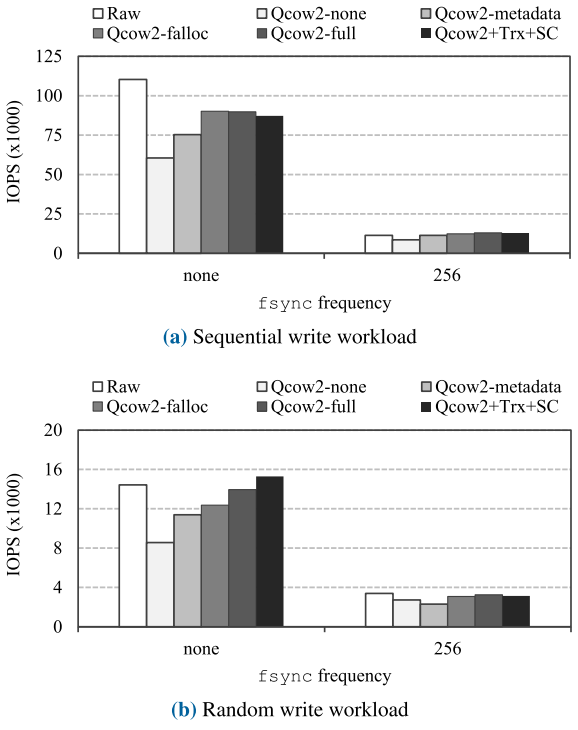


FIGURE 12. The number of file sync calls normalized to that of baseline.

For evaluation of our hypervisor in terms of write IOPS and storage space efficiency, we also performed some experiments running the FIO benchmark with sequential and random write workload. Figure 13 presents the performance comparison of our hypervisor and the conventional hypervisor, with the raw-based and qcow2-based formats. With respect to storage space efficiency, our hypervisor



(a) Sequential write workload

(b) Random write workload

FIGURE 13. Performance comparison of each virtual disk format.

outperforms the conventional hypervisor because it does not pre-allocate any qcow2 metadata and data clusters on the virtual disk. Nevertheless, with respect to I/O performance, the write IOPS of our hypervisor is higher, by up to 78.4%, than the case in which the conventional hypervisor creates the qcow2-based virtual disk configured by the *none* option. Furthermore, the write IOPS of our hypervisor is slightly higher than that of the virtual disk configured by the *falloc* and *full* options, as shown in Figure 13b.

The major difference between the conventional hypervisor and our hypervisor is on the policy for data consistency. In terms of data consistency, the conventional hypervisor guarantees strong consistency by incurring sync operations whenever data clusters and the associated qcow2 metadata are updated. For this reason, the conventional hypervisor can easily recover data inconsistency caused by sudden power failures or system crashes. Meanwhile, the proposed hypervisor guarantees the atomicity of the transaction that is composed of multiple data clusters needed for reflecting the write requests transferred from VMs, and it enforces the durability of the transaction with `gsync` operation. Since the proposed hypervisor has a relatively larger recovery unit compared with the conventional one, the recovery process may be slow relatively and it may revert the state of the virtual disk into the older version, not the latest version.

VI. RELATED WORK

Not surprisingly, many researchers have suggested various approaches to improve the I/O performance of virtualized systems by mitigating the data consistency overhead. Some studies have focused on reducing the number of file sync operations by adopting transactional support [21], [23]–[26], and others have concentrated on resolving the write/sync amplification problem in the virtualized systems [7], [12], [13], [15], [26].

A. TRANSACTIONAL SUPPORT

To provide high-performance while guaranteeing application-level and system-wide consistency, it is important to reflect multiple pages at once into the underlying storage devices using transactional atomicity. In this regard, Kang *et al.* proposed a transactional FTL that offloads the burden of guaranteeing consistency from the application-level to the storage-level [21] and reduced the number of fsync calls that occur during the process of SQLite journaling. Chidambaram *et al.* introduced an optimistic crash consistency for a journaling filesystem [27]. Their filesystem provides primitive sync operations by decoupling the durability of write requests from their ordering, inside the journal transaction. Min *et al.* proposed a filesystem, called CFS, which exploits the transactional flash storage to guarantee application-level consistency with low-overhead [21], [23]. To eliminate individual the journaling scheme of applications, they declared the critical code regions and supported them to operate atomically. Hu *et al.* implemented a transactional filesystem, called TxFS, which guarantees strong consistency, while providing high system performance [24]. Similar to the CFS [23], they also provide a simple API to designate the performance-critical code regions that can be operated atomically with low complexity. Kim *et al.* introduced a compound command, which facilitates assigning multiple key-value pairs into a single NVMe operation, by adopting the transactional support to amortize the I/O interfacing overhead [25]. Unfortunately, such transactional support does not yet consider the architectures and mechanisms of the virtualized systems, especially in the hypervisor-layer.

B. THE WRITE/SYNC AMPLIFICATION PROBLEM

Since virtualized systems have complicated I/O path due to the duplicated I/O stack on the host and guest, many studies have tried to resolve the write/sync amplification problem by mitigating the semantic gap between the host and guest. Lee *et al.* optimized the host-side page cache policy to keep high hit ratio and improve I/O performance of the virtualized systems [12]. They aimed to perceive the journal writes of the guest filesystem at the hypervisor-level, and then tried to bypass the host-side page cache in reflecting them to the storage device. Kim *et al.* proposed an address reshaping technique in the hypervisor-level to improve the performance of the qcow2-based virtual disk [15]. This technique is based on the existing technique that transforms the random write pattern into a sequential one considering the characteristics of flash storage devices [28]. Chen *et al.* re-designed the layout of the qcow2 format to resolve the sync amplification problem that induces performance degradation in virtualized systems [13]. In their techniques, they provide an individual journaling and adaptive pre-allocation method to reduce the frequency of disk sync operations. Lee *et al.* proposed a VM-aware flush mechanism to mitigate I/O performance interference among the VMs [7]. Their mechanism guarantees the service level objective (SLO) of each VM by supporting the VM-based persistency control on the disk cache flush command.

The previous work of our hypervisor proposed a data cluster allocation scheme that adopts the transactional support using `gsync` operation to improve the performance of qcow2-based virtual disk [26]. Unfortunately, the previous work has a limitation in that it cannot optimize the conventional CoW mechanism of qcow2 format, which generates unnecessary read/write operations, during the data cluster allocation. When new data clusters are to be fully updated by the write request transferred from a VM, the previous work induces performance overhead by unconditionally performing CoW operations to the new data clusters. Therefore, to mitigate performance overhead caused by inefficient CoW operations, the proposed hypervisor additionally suggests the selective CoW mechanism that identifies the CoW region of new data clusters, and then selectively performs CoW operations for them with the contents of the *backing file* during the data cluster allocation.

VII. CONCLUSION

In this paper, we propose a transactional hypervisor to mitigate the performance overhead of the qcow2-based virtual disk caused by excessive file sync calls and CoW operations incurred during the data cluster allocation process. To reduce the frequency of file sync calls, we adopt transactional support to the hypervisor and introduce a new type of file sync operation, called `gsync` (group-sync), which reflects modification of qcow2 metadata into the virtual disk in a bundle. In addition, we propose a selective CoW mechanism that efficiently eliminates unnecessary CoW operations with consideration on the amount of data that will be updated by the write requests transferred from the VMs. Our experimental results clearly show that our hypervisor not only improves write IOPS by up to 78.4% but also reduces the number of the file sync calls and that of CoW operations, by up to 50.6% and 52.4%, respectively, when compared with the conventional hypervisor.

REFERENCES

[1] M. R. Cabrer, R. P. D. Redondo, A. F. Vilas, J. J. P. Arias, and J. G. Duque, "Controlling the smart home from TV," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 421–429, May 2006.
[2] 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.
[3] N. Amit and M. Wei, "The design and implementation of hyperupcalls," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2018, pp. 97–112.
[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Ottawa Linux Symp.*, 2007, pp. 225–230.
[5] 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, Jul. 2008.
[6] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 333–348, 2014.
[7] T. Lee, M. Lee, and Y. I. Eom, "VM-aware flush mechanism for mitigating inter-VM I/O interference," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1501–1506.
[8] D. Boutcher and A. Chandra, "Does virtualization make disk scheduling passé?" *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 1, pp. 20–24, Mar. 2010.
[9] M. Kesavan, A. Gavrilovska, and K. Schwan, "On disk I/O scheduling in virtual machines," in *Proc. USENIX WIOV*, 2010, pp. 1–6.
[10] D. Hildebrand, A. Povzner, R. Tewari, and V. Tarasov, "Revisiting the storage stack in virtualized NAS environments," in *Proc. WIOV*, 2011, pp. 1–6.
[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. (ATC)*, 2013, pp. 231–242.
[12] E. Lee, H. Bahn, M. Jeong, S. Kim, J. Yeon, S. Yoo, S. H. Noh, and K. G. Shin, "Reducing journaling harm on virtualized I/O systems," in *Proc. 9th ACM Int. Syst. Storage Conf.*, 2016, pp. 1–6.
[13] Q. Chen, L. Liang, Y. Sia, H. Chen, and H. Kim, "Mitigating sync amplification for copy-on-write virtual disk," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 241–247.
[14] M. McLoughlin. *The Qcow2 Image Format*. Accessed: Mar. 18, 2020. [Online]. Available: https://people.gnome.org/~markmc/qcow-image-format.html
[15] S. Kim, H. Eom, and Y. Son, "Improving spatial locality in virtual machine for flash storage," *IEEE Access*, vol. 7, pp. 1668–1676, 2019.
[16] J. Kim, C. Min, and Y. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 217–224, May 2014.
[17] D. Park and D. Shin, "iJournaling: Fine-grained journaling for improving the latency of fsync system call," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 787–798.
[18] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, "Barrier-enabled I/O stack for flash storage," in *Proc. USENIX FAST*, 2018, pp. 211–226.
[19] Wikipedia. (2016). *Copy-on-Write*. Accessed: Jul. 30, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Copy-on-write
[20] J. Axboe. *FIO (Flexible I/O Tester)*. Accessed: Jan. 21, 2020. [Online]. Available: http://freecode.com/projects/fio
[21] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite databases," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2013, pp. 97–108.
[22] D. H. Kang, W. Kang, and Y. I. Eom, "S-WAL: Fast and efficient write-ahead logging for mobile devices," *IEEE Trans. Consum. Electron.*, vol. 64, no. 3, pp. 319–327, Aug. 2018.
[23] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight application-level crash consistency on transactional flash storage," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2015, pp. 221–234.
[24] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel, "TxFS: Leveraging file-system crash consistency to provide ACID transactions," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2018, pp. 879–891.
[25] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction support using compound commands in key-value SSDs," in *Proc. USENIX HotStorage*, 2019, pp. 1–7.
[26] M. Lee and Y. I. Eom, "Transaction-aware data cluster allocation scheme for Qcow2-based virtual disks," in *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, Feb. 2020, pp. 385–388.
[27] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proc. 24th ACM Symp. Operating Syst. Principles (SOSP)*, 2013, pp. 1–16.
[28] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device," in *Proc. USENIX FAST*, 2017, pp. 271–283.

MINHO LEE received the B.S. degree in electronic and electrical engineering and the M.S. degree in electrical and computer engineering from Sungkyunkwan University, South Korea, in 2013 and 2015, respectively, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. His research interests include virtualization, file and storage systems, and operating systems.

YOUNG IK EOM received the B.S., M.S., and Ph.D. degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. From 2000 to 2001, he was a Visiting Scholar with the Department of Information and Computer Science, University of California at Irvine. He was a President of the Korean Institute of Information Scientists and Engineers, in 2018. His research interests include virtualization, operating systems, file and storage systems, cloud systems, and UI/UX systems.