Barrier enabled QEMU

Myeongseon Kim
Hanyang University
Seoul, Korea
+82)2-2220-4579
kms9139@hanyang.ac.kr

Joontaek Oh
Hanyang University
Seoul, Korea
+82)2-2220-4579
na94jun@hanyang.ac.kr

Youjip Won
KAIST
Daejeon, Korea
+82)2-2220-4579
youjip.won@gmail.com

ABSTRACT

Virtual machines can be run inside one physical machine. This advantage enables cloud service providers and data centers to provide the identical and independent services to multiple users with fewer hardware. QEMU, a virtual machine that is used widely, has the two problems. One is performance degradation due to the amplification of the sync operation. The other is that QEMU not support the BarrierFS which uses the barrier enabled I/O stack that can guarantee the order of writes without transfer-and-flush mechanism. In this paper, we apply the fdatabarrier() system call, which is provided by BarrierFS, to address the I/O performance degradation problem due to the amplified sync operations. We also implement the barrier handler in QEMU so that the QEMU can execute the guest with BarrierFS to reduce the I/O latency caused by transfer-and-flush. The barrier handler recognizes the barrier flag in I/O command from guest and transmits the flag to host storage. We evaluate performance of the QEMU applied BarrierFS compared with the original QEMU. We observe I/O throughput using the varmail of filebench as macro benchmark and Mobibench and GNU dd command as micro benchmark. In result of evaluation, throughput of the QEMU applied BarrierFS is improved by maximum of 13x and minimum of 3.9x compared with the original QEMU. Thus, we verified that the barrier enabled QEMU has higher I/O performance than the original QEMU.

CCS Concepts

• CSS → Software and its engineering → Software organization and properties → Contextual software domains → Operating systems → File systems management. • CSS → Software and its engineering → Software organization and properties → Contextual software domains → Software infrastructure → Virtual machines.

Keywords

Virtual machine; Operating system; Performance evaluation.

1. INTRODUCTION

Cloud services became the service that are very close and familiar to many people. The providing cloud service to multiple users means providing multiple isolated machines. However, there is numerical and economical limit on providing machines. The legacy technique that is called virtualization addresses this problem.

The virtualization is the technique that run machine that is virtualized on physical machine. Naturally, it is possible that multiple virtual machines can be run on the single physical machine. Virtual machines can continue to be created as long as performance of physical machine is enough. For that reason, the technique of virtualization became main technique for the cloud service.

Virtual machines have the advantage of being able to provide the identical and independent services to multiple users as a single physical machine. And this advantage leads to the benefits of reducing the cost of physical machines. For vendors that provide cloud service, it would be ideal to provide services to the maximum number of users with minimal physical machines.

However, virtual machines have disadvantages on performance. Hardware of virtual machines is implemented as software, so the processes that should be handled by hardware are handled by software running on host physical machine. Moreover, handling by software running on host physical machine cause handling by hardware of host physical machine. That is, one more layer is added to handle process of virtual machine.

Chen et al[2] discovered situation that the sync operation is amplified in virtual machine. When this situation occurs, one sync operation is called from guest of virtual machine can be amplificated to 8 sync operations in virtual machine. Chen et al addressed this problem by journaling. However, it accompanies overhead to calculate checksum and write metadata twice. Moreover, it should reserve separated area for journaling. They adopt an inefficient mechanism for only the ordering guarantee. However, Won et al[1] proposed an efficient mechanism for the ordering guarantee and the mechanism is not yet in use.

In this paper, we use the barrier-enabled I/O stack to guarantee ordering. We not only address the problem of ordering guarantee through the barrier but also implement the barrier handler that a virtual machine can use the barrier-enabled I/O stack. Our contributions are as follows: First, we try to address the sync amplification that is representative problem of virtual machine through the latest technology barrier. Second, we implement the first virtual machine that supports the barrier command using QEMU. Third, we perform first evaluation of the application that is applied the barrier, for verifying how much performance increased.

This change will enable the cloud service providers and data centers can provide the faster services as fewer physical machines.

ICISS 2019, March 16–19, 2019, Tokyo, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6103-3/19/03…$15.00
https://doi.org/10.1145/3322645.3322664

This paper consists as follow: First, in background, we address path of request from guest in QEMU (In this paper, the word "guest" means the guest machine running on the host machine through QEMU) and the problem of sync amplification in QEMU. And we address the BarrierFS that is file system using the barrier-enabled I/O stack. Second, in design and implementation, we address how change sync operation in QEMU and how implement the barrier handler. Third, we verify improvement of performance by evaluating the QEMU that uses the barrier-enabled I/O stack.

2. RELATED WORK

Chen et al addressed the problem of sync amplification by the journaling. Metadata and checksum of data consist of one transaction at the journal area of virtual disk. Since metadata consist of one transaction and is written atomically, its consistency can be guaranteed. Two journaling mode is provided. One is no-data journaling mode that metadata and checksum of data consist of one transaction. The other is full journaling mode that metadata and data consist of one transaction.

The consistency of metadata can be guaranteed using journaling. However, disadvantages of journaling also exist. First, virtual disk must have an area for journaling always. Second, there is overhead for calculating a checksum. Third, there is write overhead that QEMU write metadata to journal area and home place. Therefore, the journaling is not efficient manner.

3. BACKGROUND

3.1 QEMU

QEMU[3] is an emulator that enables operating system and applications to run in virtual environment. QEMU can run virtual disk images in various formats such as raw disk, QCOW2, and VMDK.

QEMU transmits the I/O commands requested from guest to the physical storage, giving the guest the same effect as using the storage directly. QEMU handles I/O commands received from guest through virtqueue. QEMU calls different handlers depending on the type of request, such as read, write, and flush. The request handler is executed as a thread and transmits the I/O command received from the guest to the physical storage. Once I/O command received has been processed, the request handler inserts the result into virtqueue via the callback function and notifies the guest.

3.1.1 Passed requests from guest

QEMU can emulate various devices using a single device driver called virtio[4], without directly emulating all physical devices using paravirtualization technique.

Virtio uses a data structure called virtqueue to enable data transmission between guest and host. The kernel in the guest inserts an I/O request into the virtqueue in the virtio device driver and then notifies the host that a new request has been registered through the function called "kick". When the host received notification from guest, the host takes the request passed through virtqueue and executes the function associated with the requested operation.

3.1.2 Amplification of sync operation

QCOW2, a virtual disk image format supported by QEMU, has advantages of copy-on-write and snapshot. Copy-on-write ensures that if the current image uses the same data as the original image and a different modification occurs than the original, the modified data is made to avoid affecting the original. Snapshot can restore the contents stored at a specific time by recording the metadata of the changed contents of the original image.

QCOW2 consists of a default 64KB cluster unit, and there are metadata for managing images. The L1 and L2 tables are lookup table for accessing the data cluster. The reference table is a table for managing how many snapshot images in each cluster are being referenced. QEMU should guarantee the order of writes to maintain consistency of metadata when executing QCOW2. QEMU writes data and calls sync operation such as fdatasync() to guarantee the order of writes. The problem comes up from this guarantee of write order. When the guest calls a single sync operation, QEMU calls the sync operation frequently to guarantee the order of metadata. Sync operation waits until data is durable. Therefore, frequent sync operation of QEMU causes I/O performance degradation.

Guest transmits two flush command to QEMU when it performs appending write. One is for journal descriptor block and the other one is for journal commit block. The flush handler in QEMU calls four fdatasync() for each flush command. So a total of eight fdatasync() are called.

In QEMU version 2.12.0, sync operations are amplified maximum 4x. Compared with version 2.1.2 used by Chen et al, amplification is reduced to half. However, the sync amplification remains.

3.2 BarrierFS

The existing file system adopts transfer-and-flush as a manner to guarantee the order of writes. However, transfer-and-flush has a limitation that should guarantee the order of write and durability at the same time. Therefore, there is a disadvantage of waiting for the data to be completely durable to the storage when using the ordering guarantee mechanism.

BarrierFS[1] is file system using the barrier-enabled I/O stack. BarrierFS provides an ordering guarantee mechanism that complements these disadvantages. By removing the durability from the existing ordering guarantee mechanisms, the next write request can be performed immediately without having to wait for the data to become durable.

BarrierFS can guarantee the order through the barrier write. The barrier write transmits a write command with a special barrier flag to the storage. When the storage controller receives the barrier flag, it ensures that the previous barrier request is written before the current barrier request or the next barrier request is written after the current barrier request. Therefore, order of write can be guaranteed.

BarrierFS provides dual-mode journaling. In the existing journaling, JBD thread guarantees the order of write by transferring and flushing the journal descriptor block and the journal commit block sequentially. However, the dual-mode journaling has divided the JBD thread into commit and flush threads. The commit thread dispatches the journal descriptor block and the journal commit block with the barrier flag attached, guaranteeing the order. The flush thread flushes the journal descriptor block and the journal commit block to be durable. When an application writes data and triggers a commit thread, the commit thread dispatches a journal descriptor block and a journal commit block and returns. And then, the application can perform the next request immediately.

4. DESIGN AND IMPLEMENTATION

4.1 QEMU with fdatabarrier()

QEMU calls a sync operation such as fdatasync() to guarantee the write order of the metadata of virtual disk. However, since fdatasync() waits until the data is completely durable, I/O performance is degraded in QEMU, which calls frequently sync operations.

We change the amplified fdatasync(), in the flush handler, to fdatabarrier() provided by BarrierFS to reduce the latency of sync operation. Since fdatabarrier() does not guarantee the durability that data is written to storage completely, we preserve the last fdatasync(). An ordering guarantee mechanism using fdatabarrier() is more efficient than the mechanism using fdatasync() because it can guarantee order without having to wait for data to become durable.

For using the fdatabarrier() in QEMU, host OS should be the BarrierFS. Because, the BarrierFS the only file system that can support fdatabarrier() system call and can transmit the barrier flag to storage of host. Thus, the QEMU using the fdatabarrier() should be executed on the BarrierFS.

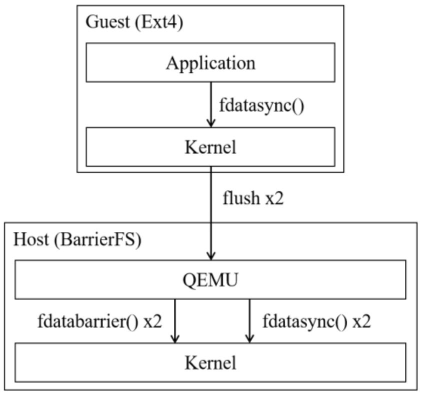

Figure 1. QEMU with fdatabarrier().

4.2 Barrier handler

Applying BarrierFS to the guest improves I/O performance. This is because the guest uses the barrier write instead of the legacy sync operation, which prevents QEMU from waiting for all requests to be processed and returned.

However, it is not enough to apply only BarrierFS to QEMU’s guest. Currently, BarrierFS does not support the virtio device driver. Therefore, even if the barrier flag exists in the request, when I/O command is generated in the virtio device driver layer, only the write flag is added without knowing the existence of the barrier flag. We modify the virtio device driver layer of BarrierFS to check whether the request contains barrier flag and to add the flag to I/O command.

Nevertheless, QEMU still cannot use proper BarrierFS. This is because QEMU receives an I/O command containing a barrier flag from the guest but cannot recognize the barrier flag. QEMU needs a handler to recognize the barrier flag and to transmit the I/O command containing the barrier flag to the storage. We modify the part of checking the request received from guest in virtqueue among QEMU. A handler is implemented to execute a routine calling fdatabarrier() if the barrier flag is present in the request. What we should note that not only the data requested by the guest but also the metadata of QCOW2 must be written to maintain the consistency. Therefore, the handler is implemented not only the recognition and transmitting of barrier flag, but also updating metadata by guaranteeing order of write through fdatabarrier().

In the original QEMU, the write handler inserts result of the write operation to virtqueue through the callback function when the storage of host has completed the write command. However, in the modified QEMU, the write handler calls the barrier handler through callback function when the storage of host has completed the write command. And result of write operation is inserted to virtqueue by the callback of the barrier handler and is transmitted to guest. Therefore, write command from the guest is done perfectly after the barrier handler transmits the barrier flag to storage and updates metadata.

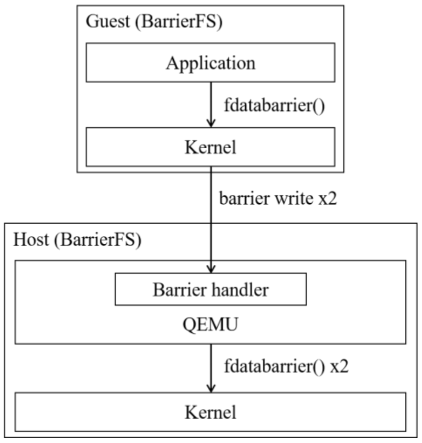

Figure 2. QEMU with the barrier handler.

5. EVALUATION

We evaluated the performance of the modified QEMU in single virtual machine and multiple virtual machines like the cloud service. The evaluation compositions are classified into three categories. In the first composition, the guest with Ext4 is executed through the original QEMU on the host with Ext4. This composition is called QEMU. In the second composition, the guest with Ext4 is executed through the QEMU changed from fdatasync() to fdatabarrier() on the host with BarrierFS. This composition is called QEMU-HostB. In the last composition, the guest with BarrierFS is executed through the QEMU with the barrier handler. This composition is called QEMU-AllB. The QEMU used in the evaluation is version 2.12.0, and the kernel of Ext4 is version 3.10.61.

Table 1. Evaluation composition.

Composition Name	Host OS	Guest OS	Version of QEMU
QEMU	Ext4	Ext4	2.12.0
QEMU-HostB	BarrierFS	Ext4	2.12.0, Changed from fdatasync() to fdatabarrier()
QEMU-AllB	BarrierFS	BarrierFS	2.12.0, with barrier handler

We used a macro benchmark and micro benchmarks to evaluate performance. The varmail workload of the Filebench is used as macro benchmark. We modified the varmail workload to I/O size to 64KB and running time to 180 seconds. The Mobibench and dd command of GNU are used as micro benchmark, and these workloads evaluate performance of the overwriting and append writing. Mobibench is set the file size to 5GB, record size to 64KB, and the sync mode to fdatasync. GNU dd command is set the I/O size to 64KB and the count to 25000 and the oflag to dsync. The reason for setting I/O size to 64KB in the above workloads is because the cluster size of QCOW2 is set to default 64KB, so that the cluster is allocated per write in evaluation so that more sync operation occurs.

The machine specs used in all evaluation are Intel(R) Core(TM) i7-4790K CPU (4.00 GHz), 32GB memory, and a 256GB Samsung 850Pro.

Evaluation in a single virtual machine is performed by executing only one virtual machine in the host.

In the varmail workload, throughput of QEMU-HostB is improved by 1.9% compared with QEMU. Throughput of QEMU-HostB is reduced by 1.2% at the Mobibench's append writing workload and 0.8% at the Mobibench's overwriting workload. And the Throughput of QEMU-HostB is improved by 76% at the GNU dd's append writing workload but it is dropped by 0.5% at the GNU dd's overwriting workload. The evaluation results of the QEMU-HostB shows similar performance to QEMU overall except the GNU dd's append writing that is improved by 76%.

In the varmail workload, throughput of QEMU-AllB is improved by 3.9x compared with QEMU. Throughput of QEMU-AllB is improved by 12.7x at the Mobibench's append writing workload and 12x at the Mobibench's overwriting workload. And Throughput of the QEMU-AllB is improved by 11x at the GNU dd's append writing workload and 13x at the GNU dd's overwriting workload. Throughput is improved by maximum of 13x and minimum of 3.9x. The evaluation result of QEMU-AllB shows that performance is significantly improved.

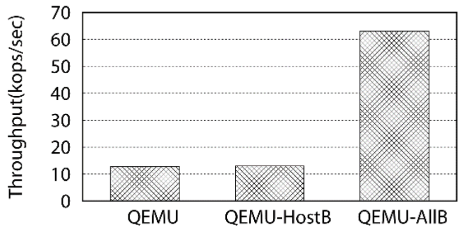

Figure 3. Varmail on single VM.

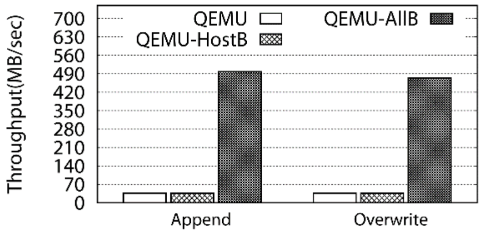

Figure 4. Mobibench on single VM.

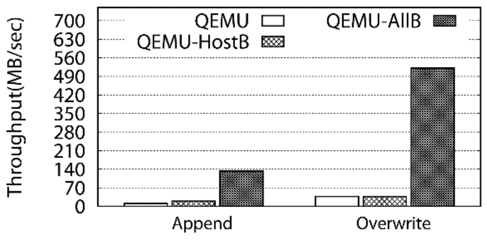

Figure 5. GNU dd command on single VM.

6. CONCLUSION

Virtual machines can provide the identical and independent services to multiple users as a single physical machine. Therefore, virtual machines are not an option but an essential key element in the cloud service providers and data centers.

In this paper, we apply BarrierFS to QEMU for increasing the efficiency of virtual machines, and we evaluate performance of the QEMU applied BarrierFS. To address the performance degradation caused by frequent sync operation of QEMU, we change the amplified fdatasync() to fdatabarrier() provided by BarrierFS. And we implement the barrier handler that QEMU can recognize the barrier flag in I/O command of guest and transmits the flag to storage of host when applying BarrierFS to guest.

The difference of performance between the original QEMU and the QEMU that changed from fdatasync() to fdatabarrier() is very small. On the other hand, when the guest applied BarrierFS is executed on the QEMU applied the barrier handler, throughput is improved by maximum of 13x and minimum of 3.9x compared with the original QEMU.

Thus, the results of this evaluation show that when applying BarrierFS to QEMU, even if more virtual machine is executed to one physical machine than current one, it can perform same or better. The barrier enabled QEMU will enable the cloud service providers and data centers to provide the faster service to more users.

7. ACKNOWLEDGMENTS

This work was supported by the BK21+ program (NRF), Basic Research Lab Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT & Future Planning (No. NRF-2017R1A4A1015498), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00549).

8. REFERENCES

[1] WON, Y. J., JUNG, J. M., CHOI, G. Y., OH, J. T., SON, S. B., HWANG, J. Y., AND CHO, S. Y. Barrier-enabled IO stack for flash storage. In Proc. of USENIX FAST 2018 (Oakland, CA, USA, Feb 2018).

[2] CHEN, Q., LIANG, L., XIA, Y., CHEN, H., AND KIM, H. S. Mitigating sync amplification for copy-on-write virtual disk. In Proc. of USENIX FAST 2016 (Santa Clara, CA, USA, Feb 2016).

[3] BELLARD, F. QEMU, a fast and portable dynamic translator. In Proc. of USENIX ATC 2005 (Anaheim, CA, USA, Apr 2005).

[4] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. In Proc. of ACM SIGOPS Oper. Syst. Rev. 2008.