Mitigating Sync Amplification for Copy-on-write Virtual Disk

Qingshu Chen, Liang Liang, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University;*
Hyunsoo Kim, *Samsung Electronics*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/chen-qingshu

This paper is included in the Proceedings of the
14th USENIX Conference on
File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Open access to the Proceedings of the
14th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX

Mitigating Sync Amplification for Copy-on-write Virtual Disk

Qingshu Chen, Liang Liang, Yubin Xia‡, Haibo Chen‡*, Hyunsoo Kim†
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
† Samsung Electronics Co., Ltd.

ABSTRACT

Copy-on-write virtual disks (e.g., *qcow2* images) provide many useful features like snapshot, de-duplication, and full-disk encryption. However, our study uncovers that they introduce additional metadata for block organization and notably more disk sync operations (e.g., more than 3X for *qcow2* and 4X for *VMDK* images). To mitigate such sync amplification, we propose three optimizations, namely *per virtual disk internal journaling*, *dual-mode journaling*, and *adaptive-preallocation*, which eliminate the extra sync operations while preserving those features in a consistent way. Our evaluation shows that the three optimizations result in up to 110% performance speedup for *varmail* and 50% for *TPCC*.

1 INTRODUCTION

One major benefit of virtualization in the cloud environment is the convenience of using image files as virtual disks for virtual machines. For example, by using the copy-on-write (CoW) feature provided by virtual disks in the *qcow2* format, a cloud administrator can provide an image file as a read-only *base file*, and then overlay small files atop the *base file* for virtual machines [16]. This can significantly ease tasks like VM deployment, backup, and snapshot, and bring features such as image size growing, data de-duplication [7, 9], and full-disk encryption. Thus, CoW virtual disks have been widely used in major cloud infrastructures like OpenStack.

However, the convenience also comes at a cost. We observe that with such features being enabled, the performance of some I/O intensive workloads may degrade notably. For example, running *varmail* on virtual disks with the *qcow2* format only gets half the throughput of running on the *raw* formats. Our analysis reveals that the major reason is a dramatic increase of sync operations (i.e., *sync amplification*) under *qcow2*, which is more than 3X compared to the *raw* format.

The extra sync operations are used to keep the consistency of the virtual disk. A CoW image (e.g., *qcow2*) contains much additional metadata for block organization, such as the mapping from virtual block numbers to physical block numbers, which should be kept consistent to prevent data loss or even disk corruption. Thus, the *qcow2* manager heavily uses the *fdatasync* system call to ensure the order of disk writes. This, however, causes notable performance slowdown as a sync operation is expensive [17, 1]. Further, since a sync operation triggers disk flushes that force all data in the write cache to be written to the disk [25], it reduces the effectiveness of the write cache in I/O scheduling and write absorption. For SSD, sync operations can result in additional writes and subsequent garbage collection. Our evaluation shows that SSD has a 76% performance speedup for random write workload after disabling write cache flushing. A workload with frequent syncs may also interfere with other concurrent tasks. Our experiment shows that sequential writes suffer from 54.5% performance slowdown if another application calls *fdatasync* every 10 milliseconds. Besides, we found that other virtual disk formats like VMDK share similar *sync amplification* issues.

One way to mitigate the *sync amplification* problem is disabling the sync operations. For example, the VirtualBox (version 4.3.10) just ignores the guest sync requests for high performance [23]. Besides, VMware Workstation (version 11) provides an option to enable write cache [24], which ignores guest sync operations as well. This, however, is at the risk of data inconsistency or even corruption upon crashes [6].

To enjoy the rich features of CoW virtual disks with low overhead, this paper describes three optimizations, *per virtual disk internal journaling*, *dual-mode journaling* and *adaptive preallocation*, to mitigate *sync amplification* while preserving metadata consistency.

Per virtual disk journaling leverages the journaling mechanism to guarantee the consistency of virtual disks. *Qcow2* requires multiple syncs to enforce ordering, which is too strong according to our observation. To address this issue, we implement an internal journal for each virtual disk, where metadata/data updates are first logged in a transaction, which needs only one sync operation to put them to disk consistently. Such a journaling mechanism, however, requires data to be written twice, which is a waste of disk bandwidth. We further introduce *dual-mode journaling* which monitors each modification to the virtual disk and only logs metadata (i.e., *reference table*, *lookup table*) when there is no data overwriting.

*‡Corresponding authors

Adaptive preallocation allocates extra blocks for a virtual disk image when the disk is growing. The preallocated blocks can be used directly in the following expansion of virtual disks. This saves the image manager from requesting the host file system for more free blocks, and thus avoids extra flush operations.

We have implemented the optimizations for *qcow2* in QEMU-2.1.2. Our optimizations result in up to 50% performance speedup for *varmail* and 30% speedup for *tpcc* for a mixture of different workloads. When we run *varmail* and *tpcc* with a random workload, they can achieve 110% and 50% speedup, respectively.

2 Background and Motivation

We use the *qcow2* format as an example to describe the organization of a VM image and the causes of *sync amplification*.

2.1 The *qcow2* Format

A *qcow2* virtual disk contains an image header, a two-level *lookup table*, a *reference table*, and data clusters, as shown in Figure 1. The image header resembles the superblock of a file system, which contains the basic information of the image file such as the base address of the *lookup table* and the *reference table*. The image file is organized at the granularity of *cluster*, and the size of the cluster is stored in the image header. The *lookup table* is used for address translation. A virtual block address (VBA) a in the guest VM is split into three parts, i.e., a=(a_1, a_2, a_3): a_1 is used as the L1 table's index to locate the corresponding L2 table; a_2 is used as the L2 table's index to locate the corresponding data cluster; a_3 is the offset in the data cluster. The *reference table* is used to track each cluster used by snapshots. The refcount in *reference table* is set to 1 for a newly allocated cluster, and its value grows when more snapshots use the cluster.

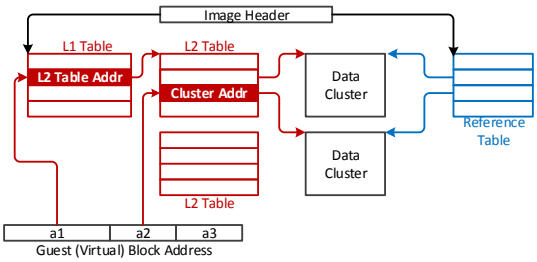

Figure 1: The orginization of *qcow2* disk format.

The process of writing some new data to a virtual disk includes following steps:

① Look up the L1 table to get the offset of the L2 table. ② If the L2 table is not allocated, then set the corresponding *reference table* entry to allocate a cluster for the L2 table, and initialize the new L2 table. ③ Update the L1 table entry to point to the new L2 table if a new L2 table is allocated. ④ Set the *reference table* to allocate a cluster for data. ⑤ Write the data to the new data cluster. ⑥ Update the L2 table entry to point to the new data cluster.

Note that, each step in the whole appending process should not be reordered; otherwise, it may cause metadata inconsistency.

2.2 Sync Amplification

The organization of *qcow2* format requires extra efforts to retain crash consistency such that the dependencies between the metadata and data are respected. For example, a data cluster should be flushed to disk before updating the lookup table; otherwise, the entry in the lookup table may point to some garbage data. The *reference table* should be updated before updating the *lookup table*; otherwise, the lookup table may point to some unallocated data cluster.

We use two simple benchmarks in QEMU-2.1.2 to compare the number of sync operations in the guest VM and the host: 1) "overwrite benchmark", which allocates blocks in advance in the disk image (i.e., the *qcow2* image size remains the same before and after the test); 2) "append benchmark", which allocates new blocks in the disk image during the test (i.e., the image size increases after the test). The test writes 64KB data and calls *fdatasync* every 50 iterations. We find that the virtual disks introduce more than 3X sync operations for *qcow2* and 4X for *VMDK* images, as shown in Figure 2.

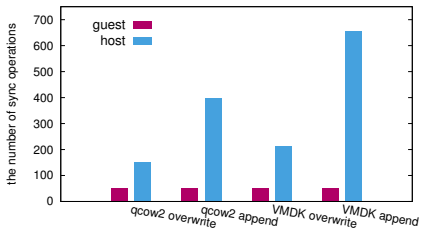

Figure 2: #sync operations observed from inside/outside of the VM.

As shown in Figure 3, a *fdatasync* of the user application can cause a transaction commit in the file system. This requires two flushes (in guest VM) to preserve its atomicity, which are then translated into two set of writes in QEMU. The first write puts the data and the journal metadata of the VM to the virtual disk, which in the worst case, causes its size to grow.

To grow the virtual disk in QEMU, a data block must be allocated, and the corresponding *reference table* block should be set strictly before other operations. This necessitates the first flush. After that, the L2 data block must be updated strictly before the remaining operations. This necessitates the second flush. (In some extreme cases where the L1 data block should be updated as well, it introduces even more flushes). The third flush is used to update the base image's *reference table*. When creating

a new image based on the base image, the refcount in the *reference table* of the base image will be increased by one to indicate that another image uses the base image's data. When updating the new image, *qcow2* will copy data from the base image to a new place and do updates. The new image will use the COW data and will not access the old data in the base image, so the refcount in the base image should be decreased by one. The third flush is used to make the *reference table* of the base image durable. The fourth flush is introduced solely because of the suboptimal implementation in QEMU. The second write is the same as the first one, which needs four flushes. Consequently, we need around eight flushes for one guest *fdatasync* at most.

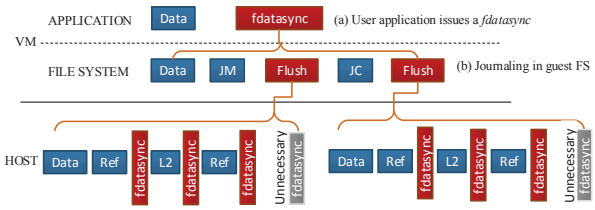

Figure 3: Illustration of sync amplification: it shows how the number of sync operations increases after the user issues *fdatasync*. The *fdatasync* with red color is necessary to impose the write ordering, while *fdatasync* with gray color are solely because of the flawed implementation of *qcow2*. J_M is for *journal of metadata*, J_C is for *journal of commit block*, *Ref* is for *reference table*, *L2* is for *L2 lookup table*.

2.3 Other Virtual Disk Formats

Other virtual disks have a similar structure to *qcow2* virtual disk. They have an image header to describe the basic information of the virtual disk, a block map table to translate virtual block address to physical block address, and many data blocks to store the guest VM's data. For example, the *grain directory* and *grain table* in the VMDK consist of a two-level table to do address translation. The VMDK also keeps two copies of the *grain directories* and *grain tables* on disk to improve the virtual disk's resilience. FVD [22] even has a bitmap for the copy-on-read feature. In summary, the organization of virtual disks will translate one update operation in the guest into several update operations in the host. Besides, the virtual disks should carefully schedule the update order to preserve crash consistency, which introduces more sync operations. Actually, our evaluation shows that VMDK has more severe sync amplification than qcow2, as shown in Figure 2.

3 Optimizations

3.1 Per Virtual Disk Internal Journaling

According to our analysis, we found that the cause of the extra sync operations is the overly-constrained semantics imposed during the modification of virtual disks. This is because the underlying file system cannot preserve the internal consistency of a virtual disk, in which certain data serves as metadata to support the rich features. As a result, the virtual disk has to impose strong ordering while being updated for the sake of crash consistency.

Based on this observation, we designed a *per virtual disk internal journaling* mechanism. As the name suggests, each virtual disk maintains an independent and internal journal, residing in a preallocated space of the virtual disk to which the journal belongs. The *per virtual disk internal journal* works in a manner similar to the normal file system journal, with the exception that it only logs the modification of the content of a single virtual disk.

On a virtual disk update, the metadata (e.g., *reference table* and *lookup table*) as well as the changed data are first logged into the preallocated virtual disk journaling area, which is treated as a transaction. At the end of this update, the journal commit block of a virtual disk is appended to the end of the transaction, indicating this transaction is committed. If any failures occur before the commit block is made durable, the whole transaction is canceled, and the virtual disk is still in a consistent state. We also leverage the checksum mechanism: by calculating a checksum of all the blocks in the transaction, we reduce the number of flushes to one per transaction.

Like other journaling mechanisms, we should install the modifications logged in the journal to their home place, i.e., checkpoint. To improve the performance of checkpoint, we delay the checkpoint process for batching and write absorbing. After a checkpoint, the area took up by this transaction can be reclaimed and reused in the future.

The *per virtual disk journaling* relaxes the original overly-constrained ordering requirement to an all-or-nothing manner, which reduces the number of flushes while retaining crash consistency.

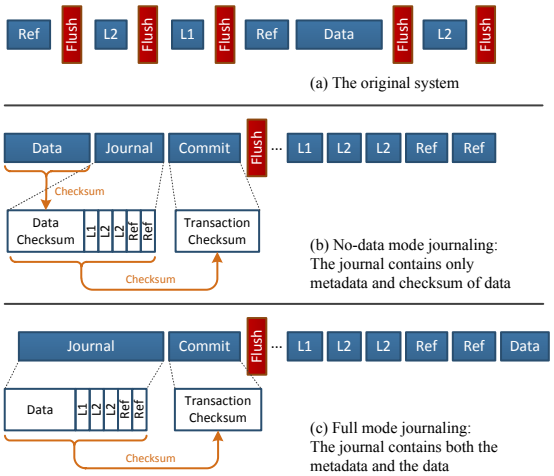

Figure 4: The dual-mode journaling optimization: it shows the process of data updating on a *qcow2* virtual disk, as mentioned in section 2.1. (*L1 means level-1 lookup table, L2 means level-2 lookup table, and Ref means reference table*)

Dual-Mode Journaling: Journaling mechanism requires logged data to be written twice. This, under certain circumstances, severely degrades performance due to wasted disk bandwidth. To this end, we apply an optimization similar to Prabhakaran et al. [18] and Shen et al. [21]. This optimization, namely *dual-mode journaling*, drives the *per virtual disk internal journaling* to dynamically adapt between two modes according to the virtual disk's access pattern. Specifically, it only logs all the data when there is an overwriting of the original data; otherwise, only metadata (*reference table* and *lookup table*) is journaled.

It should be noted that *dual-mode journaling* differs from prior work [21, 18] in that it leverages different journaling mode for a single file, and thus it does not cause any inconsistency issues discussed in [21]. This is because our journaling mechanism only deals with a single file while prior work [21, 18] needs to deal with a complete file system. The two modes used are described as follows.

No-data Journaling Mode: Instead of writing the data into the journaling, the *no-data journaling mode* simply calculates a checksum of the data and puts only the checksum into the journal. The data is written to the "home" place. The process is shown in Figure 4-b.

The Overwriting Problem and Full Journaling Mode: The *no-data journaling mode*, however, can render the recovery process inconsistent upon data overwriting. This is because the correctness of recovery relies on the checksum of the blocks in the transaction, which does not necessarily reside in the journaling area since *no-data mode journaling* does not log data blocks. The content of those data blocks can be arbitrarily affected by the following overwriting operations; the whose checksum, of course, can be different from the time when it was committed.

Consequently, if a transaction is not fully checkpointed when its following transaction aborts, and at the same time the data of the previous transaction is partially overwritten, the recovery process will erroneously consider the already committed transaction as a broken one. Therefore, if a disk write transaction needs to overwrite the data which has not been checkpointed, the system will switch to full-mode journaling and put the entire data into the journal, as shown in Figure 4-c.

Crash Recovery: During crash recovery, we scan the journal to find the first transaction that has not been checkpointed. Then, we calculate the checksum of the journal data and other related data in this transaction. If the calculated checksum is the same as the checksum recorded in the current journal transaction, we apply the logged modifications for the data and metadata to their home place and do recovery for the next transaction. It the two checksums are not equal, it means that the transaction is not completely written, and we reach the end of the journal. We abort the transaction and finish crash recovery.

Other Implementation Issues: The current suboptimal implementation of *qcow2* image format involves some unnecessary sync operations (as shown in grey boxes in Figure 3). We just remove these sync operations without affecting the consistency.

With the above operations, we can reduce the number of syncs to one for each flush request from the guest VM. In another word, if the guest OS issues a flush operation, there will be only one sync on the host.

3.2 Adaptive Preallocation

We further diagnose the behavior of the host file system when handling guest VM's sync requests, and find that the actual number of disk flushes are usually more than the number of sync requests. This is because, if a write from the guest VM increases the size of its image file, the host file system will trigger a journal commit, which flushes the disk twice, the first for the data and the second for the journal commit block.

More specifically, Linux provides three syscalls (*msync*, *fsync*, *fdatasync*) for the sync operations. The *fsync* transfers all modified in-memory data in the file referred to by the file descriptor (fd) to the disk. The *fdatasync* is similar to *fsync*, but it does not flush metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled (i.e., the file size). The *msync* is used for memory-mapped I/O. In *qcow2*, it uses *fdatasync* to make data persistent on the disk. Thus, if a guest VM's sync request does not increase the disk image's size, there will be only one flush operation for the data; but if the image size changes, the host file system will commit a transaction and cause an extra disk flush.

We leverage an adaptive preallocation approach to reducing the number of journal commits in the host. When the size of the image file needs to grow, we append more blocks than those actually required. A future virtual disk write operation which originally extends the image file can now be transformed into an "overwrite" operation. In this case, *fdatasync* will not force a journal commit in the host, which can reduce the latency of sync operation.

Specifically, we compare the position of the write operation with the image size. If the position of the write operation exceeds the image size, we will do the preallocation. Currently, the size of preallocated space is 1MB.

4 EVALUATION

We implemented the optimizations in QEMU-2.1.2, which comprise 1300 LoC. This section presents our evaluation of the optimized *qcow2* from two aspects: consistency and performance.

We conducted the experiments on a machine with a 4-core Intel Xeon E3-1230 V2 CPU (3.3GHz) and 8GB memory. We use 1 TB WDC HDD and 120G Samsung 850 EVO SSD as the underlying storage devices. The host OS is Ubuntu 14.04; the guest OS is Ubuntu 12.04. Both guest and host use the ext4 file system. We use KVM [10] and configure each VM with 1 VCPU, 2GB memory, and 10GB disk. The cache mode of each VM is *writeback*, which is the default setting. It has good performance while being safe as long as the guest VM correctly flushes necessary disk caches [20].

4.1 Consistency

To validate the consistency properties, we implement a trace record and replay framework. We run a test workload in the guest and record the write operations and sync operations in the host. Then, we randomly choose a sync operation in the trace and replay all the I/O operations above this sync operation on a backup image (the image is a copy of the guest image before running the test workload). After that, we choose some write operations between this sync and the next sync operation, and apply these writes to the backup image Finally, we use the *qemu-img* tool to check the consistency of virtual disk image.

We record two traces, one for the *append* workload and the other for *append + overwrite* workload. In the *append* workload, we append 64k data and then call *fdatasync* in the VM. In the *append + overwrite* workload, we *append* 64k data, call *fdatasync*, overwrite the 64k data and then call *fdatasync*. We simulated 200 crash scenarios for each workload. We divide data in the virtual disk image into four types: guest data, metadata (i.e., Lookup table), journal data (i.e., the journal record for metadata's modification) and journal commit block. The 200 crash scenarios for each workload contain all four types of data loss. The result shows that optimized *qcow2* can recover correctly and get a consistent state for all test cases.

4.2 Performance

We first present the performance of synchronous overwrite and append workloads. For overwrite workload, we generate a base image and allocate space in advance, then overlay a small image atop the base image. For append workload, we do the experiment directly on a newly allocated image. We also run the *Filebench* [15] and *TPCC* [4] workload. *Filebench*'s *varmail* simulates a mail server workload and will frequently call sync operations. TPCC simulates a complete environment where a population of terminal operators executes against a database. We run the experiments under three configurations. For the first configuration (raw), we only boot one VM and run the tests. For the second configuration (seq), we boot two VMs, one for the experiments and the other is doing sequential I/O all the time. For the third configuration (ran), we also have two VMs, one for the experiments and the other is doing random I/O all the time.

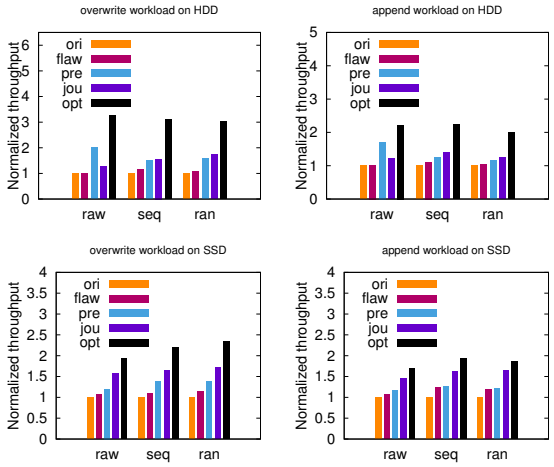

Figure 5: Micro-benchmark result on 2 storage media. *Each separate optimization as well as the overall result is showed. "ori" refers to the original system; "flaw" refers to the system which removes unnecessary sync operations caused by qcow2 flawed implementation; "pre" refers to the adaptive prealloacation; "jou" refers to the per virtual disk internal journal; "opt" refers to the overall result.*

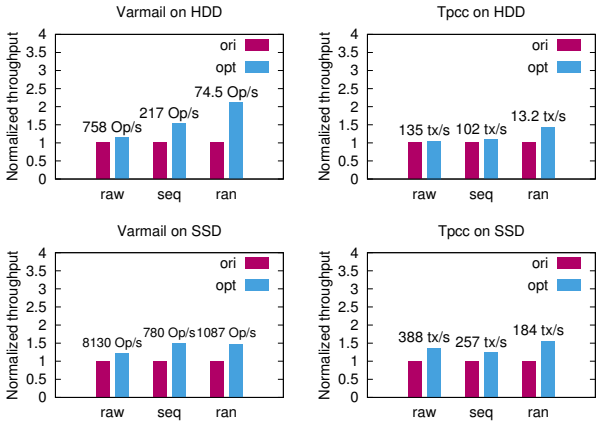

Figure 6: Macro-benchmark on 2 storage media.

Figure 5 and Figure 6 show the performance results on both HDD (hard disk driver) and SSD. For overwrite workload, our system improves the throughput by 200% for disk and 100% for SSD. For *varmail*, our system achieves 110% speedup when running *varmail* together with a random workload on HDD. The performance gain for *varmail* on SSD is about 50% when running *varmail* together with a sequential workload.

Figure 7 compares the TPCC transaction latency between our system and the original system. The results show that our system has lower latency, and the latency even decreased by 40% when TPCC is running together with a random workload.

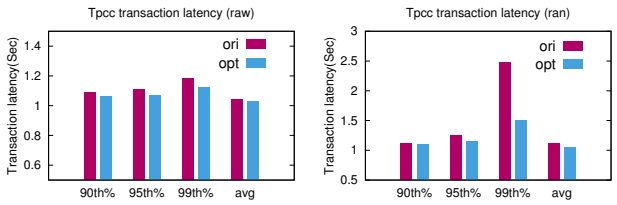

Figure 7: Latency for *TPCC*.

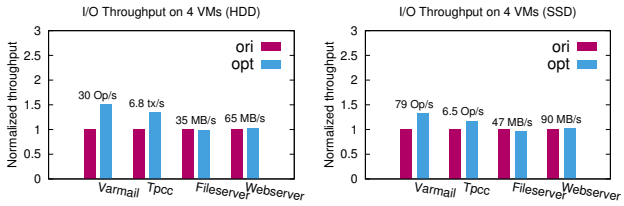

Figure 8: Evaluation for multiple VMs.

We also evaluate the multiple VM configurations. We run four VMs with *varmail*, *TPCC*, *fileserver* and *webserver* workloads respectively. *Varmail* and *TPCC* workloads issue syncs frequently, while *fileserver* and *webserver* have fewer sync operations.

Figure 8 shows the evaluation results for multiple VMs test. On HDD, the performance of *varmail* and *tpcc* improves 50% and 34%, respectively. On SSD, the performance gain is 30% and 20%, respectively. Besides, *fileserver* and *webserver* on the optimized system have similar performance to those on the original system. This is because fileserver and webserver have few sync operations and do not update *qcow2* metadata frequently.

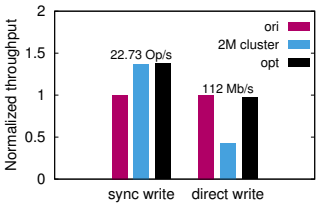

Figure 9: *"Sync write" means calling fdatasync after each write operation; "Direct write" means opening the file with O_DIRECT flag; "2M cluster" means the cluster size is 2M; The cluster size for "ori" and "opt" is 64k.*

Qcow2, VMDK and VDI support big clusters (1MB or more). Big clusters decrease the frequency of metadata operations and can mitigate the sync amplification. Figure 9 shows using big clusters may mitigate sync amplification as our approach for the *sync write* workload. However, it results in 50% performance degradation for the *direct write* workload. This is because bigger clusters increase the cost of allocating a new cluster [8]. Besides, bigger clusters reduce the compactness of sparse files. In contrast, our optimizations mitigate sync amplification without such side effects.

5 Related Work

Reducing sync operations has been intensively studied [1, 2, 21, 18, 12, 17]. For example, adaptive journaling [21, 18] is a common approach to reducing journaling incurred sync operations. OptFS [1] advocates separation of durability and ordering and split the sync operation into *dsync* and *osync* for file systems accordingly. However, it requires the underlying device to provide asynchronous durability notification interface. Our work focuses on the virtual disk image format and is transparent to the guest VM. NoFS [2] eliminates all the ordering points and uses the *backpointer* to maintain crash consistency; but it is hard to implement atomic operations, such as *rename*. Xsyncfs [17] also aimed to improve sync performance. It delays sync operations until an external observer reads corresponding data. By delaying the sync operations, there is more space for I/O scheduler to batch and absorb write operations.

Improving file system performance for the virtual machine is also a hot research topic [11, 14, 22, 19]. Le [11] analyzed the performance of nested file systems in virtual machines. Li [14] proposed to accelerate guest VM's sync operation by saving the syncing data in host memory and returning directly without writing to disk. FVD [22] is a high-performance virtual disk format. It supports many features, such as copy-on-read and prefetching. QED [19] is designed to avoid some of the pitfalls of qcow2 and is expected to be more performant than qcow2. All these work did not address the *sync amplification* problem.

Our work leverages several prior techniques such as checksum [3, 5] and pre-allocation [13], but applies them to solve a new problem.

6 Conclusion

This paper uncovered the *sync amplification* problem of copy-on-write virtual disks. It then described three optimizations to minimize sync operations while preserving crash consistency. The evaluation showed that the optimizations notably boost some I/O intensive workloads.

7 Acknowledgment

We thank our shepherd Nisha Talagala and the anonymous reviewers for their constructive comments. This work is supported in part by China National Natural Science Foundation (61572314, 61303011), a research grant from Samsung, Inc., National Youth Top-notch Talent Support Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and Singapore NRF (CREATE E2S2).

REFERENCES

[1] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.

[2] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *FAST* (2012), p. 9.

[3] COHEN, F. A cryptographic checksum for integrity protection. *Computers & Security 6*, 6 (1987), 505–510.

[4] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment 7*, 4 (2013), 277–288.

[5] EXT4 WIKI. Ext4 metadata checksums. `https://ext4.wiki.kernel.org/index.php/Ext4_Metadata_Checksums`.

[6] FORUMS, V. B. Problem for virtualbox doesn't flush/commit. `https://forums.virtualbox.org/viewtopic.php?t=13661`.

[7] HAJNOCZI, S. cluster size parameter on qcow2 image format. `http://www.linux-kvm.org/images/d/d6/Kvm-forum-2013-toward_qcow2_deduplication.pdf`.

[8] HAJNOCZI, S. cluster size parameter on qcow2 image format. `http://lists.gnu.org/archive/html/qemu-devel/2012-02/msg03164.html`.

[9] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 7.

[10] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[11] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *FAST* (2012), p. 8.

[12] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. Waldio: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 235–247.

[13] LEUNG, M. Y., LUI, J., AND GOLUBCHIK, L. Buffer and i/o resource pre-allocation for implementing batching and buffering techniques for video-on-demand systems. In *Data Engineering, 1997. Proceedings. 13th International Conference on* (1997), IEEE, pp. 344–353.

[14] LI, D., LIAO, X., JIN, H., ZHOU, B., AND ZHANG, Q. A new disk i/o model of virtualized cloud environment. *Parallel and Distributed Systems, IEEE Transactions on 24*, 6 (2013), 1129–1138.

[15] MCDOUGALL, R., AND MAURO, J. Filebench. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench`, 2005.

[16] MCLOUGHLIN, M. The qcow2 image format. `https://people.gnome.org/~markmc/qcow-image-format.html`, 2011.

[17] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6.

[18] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), pp. 105–120.

[19] QEMU WIKI. Qed specification. `http://wiki.qemu.org/Features/QED/Specification`.

[20] QEMU WIKI. Qemu emulator user documentation. `http://qemu.weilnetz.de/qemu-doc.html`.

[21] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free. In *FAST* (2014), pp. 287–293.

[22] TANG, C. Fvd: A high-performance virtual machine image format for cloud. In *USENIX Annual Technical Conference* (2011).

[23] VIRTUAL BOX MANUAL. Responding to guest ide/sata flush requests. `https://www.virtualbox.org/manual/ch12.html`.

[24] VMWARE SUPPORT. Performance best practices for vmware workstation. `www.vmware.com/pdf/ws7_performance.pdf`.

[25] WIKIPEDIA. diskbuffer. `https://en.wikipedia.org/wiki/Disk_buffer`.