Reducing Journaling Harm on Virtualized I/O Systems

Eunji Lee[1] Hyokyung Bahn[2] Minseong Jeong[1] Sunghwan Kim[1] Jesung Yeon[1]
Seunghoon Yoo[3] Sam H. Noh[4] Kang G. Shin[5]

[1]*Chungbuk National University* [2]*Ewha University* [3]*Seoul National University*
[4]*UNIST* [5]*The University of Michigan*

[1]{ejlee,msjeong,shkim,jsyeon}@oslab.cbnu.ac.kr [2]bahn@ewha.ac.kr [3]shyoo@oslab.snu.ac.kr
[4]samhnoh@unist.ac.kr [5]kgshin@umich.edu

Abstract

This paper analyzes the host cache effectiveness in full virtualization, particularly associated with journaling of guests. We observe that the journal access of guests degrades cache performance largely due to the write-once access pattern and the frequent sync operations. To remedy this problem, we design and implement a novel caching policy, called PDC (Pollution Defensive Caching), that detects the journal accesses and prevents them from entering the host cache. The proposed PDC is implemented in QEMU-KVM 2.1 on Linux 4.14 and provides 3-32% performance improvement for various file and I/O benchmarks.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management

General Terms Design, Measurement, Performance

Keywords Virtualization, Caching, Journaling, File system

1. Introduction

Virtualization is widely used in various modern computer systems ranging from personal computing devices to cloud servers [14, 16, 18–20, 22, 24, 26, 27]. Virtualization separates a software platform from hardware conditions, thereby providing flexibility, scalability and energy savings. However, these benefits are accompanied by inefficiencies associated with its additional software layers on top of existing system software. To mitigate such inefficiency, paravirtualization such as Xen is being actively explored in server systems, which runs customized guests operating systems with a light-weight hypervisor, without host operating systems [13]. However, according to recent IDC reports, more than 80% of cloud server systems rely on full-virtualization hypervisors such as Vmware, Hyper-V, and QEMU-KVM [11].

This paper analyzes and alleviates inefficiencies of the I/O mechanism in full virtualization, particularly coupled with journaling. Figure 1 shows the storage architecture of a fully virtualized system. Both the guests and the host have their own file systems and buffer caches. Although guests can configure to bypass a host cache at boot time, generally guests, by default, select to use host caching for the sake of its buffering and merging effects [12, 28]. In particular, if many virtual machines, whose lifetime is difficult to estimate, coexist, then managing a large shared buffer cache on the host side can be much more effective than allocating a large cache space for each individual guest a priori [28].

However, as storage devices increasingly become faster, whether the current I/O mechanism of full virtualization is most effective is in question. The cost for placing data into a host cache relatively increases in high-speed storage, while, in contrast, the benefit from the cache is decreased. As a straightforward approach to this issue, we can consider bypassing a host cache in fast storage.

Figure 2 shows the I/O performance when going through the host cache in comparison with the guest accessing SSD storage directly (experimental setup is described in Section

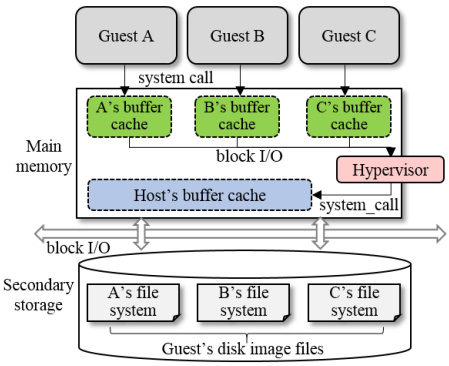


Figure 1: File system structure of a fully virtualized system

SYSTOR '16, June 6–8, 2016, Haifa, Israel.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

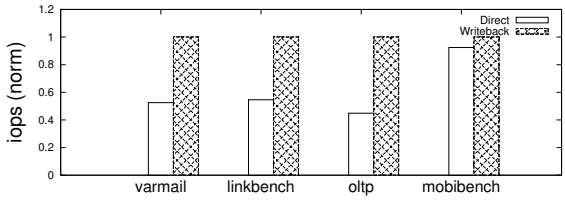


Figure 2: Performance comparison of using host cache (WB) and bypassing it (NONE) in SSD.

3). As the figure shows, using the host cache delivers 2.7x and 1.7x better performance in HDD and SSD, respectively, proving the host cache provides considerable benefits even in fast storage. However, the performance improvement decreases due to the caching overhead relative to the overhead for storage device access. Thus, more judicious use of the host cache in high-speed storage is warranted.

As a means to this end, we analyze the inefficiency of the current I/O mechanism associated with *journaling* in fast storage. Journaling is a technique used to ensure data consistency in popular file systems such as Ext4 and ReiserFS [2, 7]. Journaling writes updates first to a separate storage area called the *journal area*, and then later reflects them to the permanent file location, thereby providing data consistency. However, journaling generates harmful I/O traffic for cache performance. First, the journal data is not accessed again unless the system crashes, and second, the journal area acts as a log, generating completely sequential writes in a large loop. This behavior incurs the eviction of valuable data from the cache, leading to serious cache pollution. Third, in journaling file systems, the `fsync` operation comes right after writing updates into the journal for durability. This feature nullifies the buffering effect of the host cache, particularly more so for smaller writes, which comprise the majority of *journal traffic*. Without the buffering effect, a write takes longer when passing through the host cache due to the additional memory copy and the hosts I/O stack overhead.

Based on this observation, we propose a judicious caching policy called PDC (Pollution Defensive Caching) that prevents detrimental journal data from overwhelming the host cache. Specifically, PDC, which is implemented in a hypervisor, filters out journal data from host cache, while allowing other I/O requests to go through the host cache. The strength of PDC is that it guesses the journal accesses only with *logical block addresses (LBA)*, by detecting a stream of sequential writes from I/O accesses. That is, as the LBA is given to a hypervisor in the current architecture, PDC requires no modification to guests, making it easily deployable.

The proposed PDC is implemented in QEMU-KVM 2.1 in Linux Kernel 4.14, and we measure the performance of PDC with Filebench, Linkbench, Sysbench, Mobibench, and DbBench with LevelDB. Our measurement results show that PDC improves performance by 3-32%.

Table 1: Experimental Setup

CPU	Intel Core i5-3470 3.2GHz, Quad-Core
Main memory	DDR3 Samsung 16GB
Guest Resource	Single core / 4GB Memory
Storage	Sandisk 240GB SSD x 2
OS	Ubuntu 14.04
Hypervisor	QEMU-KVM 2.1

2. Related Work

As virtualization increases the depth and the complexity of I/O paths, considerable research has been performed on the efficient management of I/O stacks in virtualized systems. Russel presented a set of virtualized device drivers, called Virtio, which allows explicit cooperation of guests device drivers and the hypervisor, thereby eliminating the overhead of emulating physical devices at the hypervisor [26]. Xens para-virtualized driver [13] and VMwares guest tools [10] also improve I/O performance in virtualized systems by providing new I/O device drivers. HarEl *et al*. present that the mode switch invoked when sending guests I/O requests to the host incurs considerable overhead to I/O performance [21]. They propose a polling mechanism that allows guests to send I/O requests to the host via shared memory without mode switch, thereby improving I/O performance. Xu *et al*. accelerate I/O processing for virtual machines by revising IRQ handling mechanisms in multi-core systems [29]. They reduce the IRQ processing latency by processing all I/O requests in a designated core with a very small time slice. Hardware-supported virtualization technologies have also been studied. The Intel VT-d technology allows guests to directly interact with the device assigned by the host, thereby eliminating the virtualization overhead [23]. Le *et al*. observe that nested interactions of guest and host file systems significantly degrade I/O performance. Based on this observation, they provide suggestions on configuring nested file systems [24]. Chen *et al*. reveal that a copy-on-write virtual disk incurs serious sync amplification due to the internal metadata updates for block organization and propose several optimization techniques to mitigate the problem [17]. Liu *et al*. analyze that the guest passing through the virtual machines device driver to access a physical device degrades I/O performance and propose to have guests directly access the device to circumvent this problem [25] Boucher *et al*. present that I/O performance is highly affected by the combination of guest and hosts I/O schedulers in duplicated storage architectures, and explore the efficient I/O scheduler combinations for virtualized systems [15].

3. Journal Traffic Analysis

The host cache provides overall performance gains, but the guests I/O requests include a substantial number of accesses that decrease cache performance, in particular *journal traffic*. To estimate the effect of journaling on cache performance, we measure the ratio of journal accesses to the total

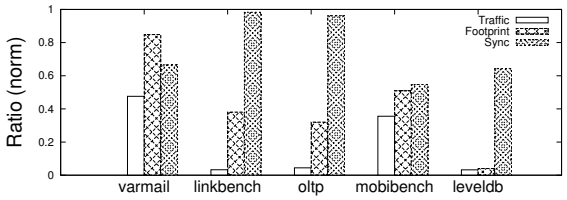
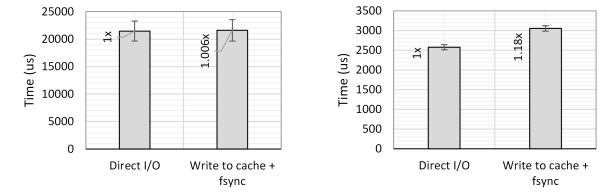


Figure 3: The ratio of journal accesses in total I/O requests



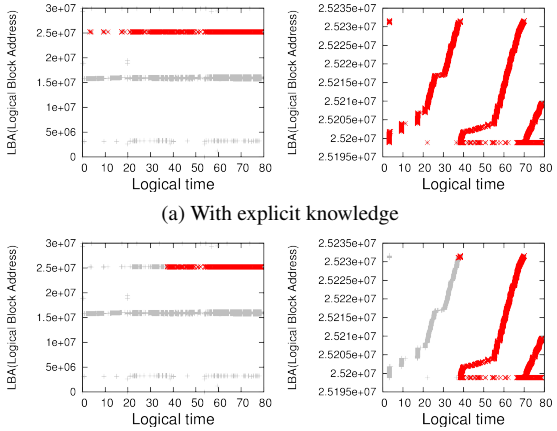
(a) HDD (b) SSD

Figure 4: Latency w/ and w.o/ the host cache

I/O traffic. Our experimental platform is described in Table 1. We evaluate performance using five benchmarks: Varmail in Filebench, Linkbench, Oltp in Sysbench, Mobibench, and DbBench with LevelDB. Varmail and OLTP generate the mail-server and the financial workloads, respectively [3, 9]. Linkbench is a database benchmark based on the Facebook Social Graph [5]. Mobibench issues a series of transactions consisting of insert, update, and delete operations using sqlite3 [6]. DbBench measures the performance of a key-value store application using LevelDB for various operations [1, 4]. The guest file system is set to Ext4 in ordered mode that performs metadata journaling, as it is currently the most common configuration in various systems.

Figure 3 shows the ratio of journal accesses in terms of traffic, footprint, and the number of `fsync` operations, relative to the entire I/O for each workload. The journal access accounts for 19% on average and up to 47% of the total I/O traffic. The effect of journaling becomes more significant in terms of the footprint. As the journal area is accessed in a completely sequential pattern, the footprint of journal accesses account for 45.2% on average and up to 84.8% of the total footprint. This result implies that the effect of forcing out important data from the cache by journaling is significant under the current caching policy.

Furthermore, the 86% of entire `fsync` operations are associated with the journal on average, as journaling issues `fsync` operations right after logging updates on every commit. In contrast to buffered I/O, the writes immediately accompanied with the `fsync` operation incurs additional overhead when they go through the host cache. This overhead becomes more pronounced as the underlying storage device becomes faster. Figure 4 estimates this overhead by measuring the I/O time issuing a 4KB write request followed by an immediate `fsync` operation. The synchronous write when using host cache takes only 1.006x longer in HDD, while it costs 1.18x longer with an SSD, compared to using Direct I/O. Thus, the cost for going through an additional host cache layer is non-trivial, calling for the need to optimize the I/O stack even further for fast storage.



(a) With explicit knowledge

(b) Prediction by heuristics

Figure 5: Accuracy of journal accesses prediction

4. Pollution Defensive Caching

Based on the above observation, we popose a new caching policy, called PDC (Pollution Defensive Caching),that protects the host cache from the harmful effect induced by guest journaling. The key challenges to the implementation of PDC are (1) how to distinguish the journal accesses from the accesses generated by the hypervisors running unmodified guest operating systems,(2) how to change the host cache usage mode dynamically to filter out journal data from the cache.

Let us consider the first challenge. In general, the journal area is created on a part of storage device at the time of its formatting, but the location is hard to figure out by LBA, as it is not fixed and varies with the storage size and other factors. One way to resolve this issue is modifying the guest operating system to explicitly mark journal accesses to the hypervisor. However, it is unfavorable in fully virtualized systems that enable guests run unmodified. To overcome this, PDC uses a heuristics that guesses journal accesses by detecting the writes with a large sequential loop, without any modification of guests. Specifically, PDC maintains a series of access flows with first and last LBAs in a hash table, and monitors if the upcoming request is in a consecutive address range. If there is a range where consecutive writes forms a large loop, PDC regards the range as journal area.

Figure 5 shows how accurately our heuristics works in guessing journal data in Varmail workload. The left side plots LBA accesses for the whole file systems, and the right side shows the enlarged view of journal accesses. We mark the journal accesses in red and others in grey. As shown in the figure, our heuristics works well and provides the same result as using the explicit knowledge, except for the monitoring in the beginning. This algorithm is designed for a

Table 2: Action for `posix_fadvise flags`.

Flag	*Action*
POSIX_FADV_NORMAL	Read-ahead data on-demand
POSIX_FADV_SEQUENTIAL	Increase read-ahead window
POSIX_FADV_RANDOM	Read-ahead in a chunk size
POSIX_FADV_NOREUSE	Not implemented
POSIX_FADV_WILLNEED	Read-ahead requested pages
POSIX_FADV_DONTNEED	Invalidate data on cache

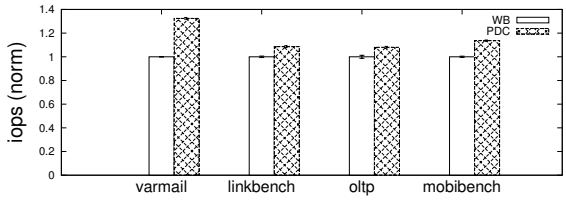
guest using *internal journaling*, which maintains a journal area in the same device with the file system, and the prediction becomes easier if a guest uses *external journaling*. The external journaling refers to maintaining the journal area in a separate device. In such a case, all we need is to guess which file refers to the journal device, as the journal area is managed in a separate file. We monitor the I/O requests on each file during a certain time and regard the write-only file as a journal device as the journal area is not read. By using both of these mechanisms, PDC manages to identify journal data irrespective of which journaling the guest uses.

Now, let us move on to the next challenge; the implementation of admission control for journal data. To meet this challenge, the hypervisor should adjust the host cache usage mode dynamically, although it is not supported in current hypervisors. We overcome this challenge by means of the `posix_fadvise` system call. It is a system call that enables user applications to provide explicit hints to the operating system about future data access patterns so that the system can take appropriate actions. Table 2 summarizes the set of flags supported in the `posix_fadvise` system call and the corresponding system actions for each flag. We use the POSIX_FADV_NOREUSE flag to implement the host cache bypassing policy. It designates that the data block is accessed only once and will not be referenced again. Since the current Linux 4.14 takes no action for this flag, we implement an action module on the host operating system such that Direct I/O is commenced when this flag set. I/O is reverted back to the buffered I/O when the POSIX_FADV_NORMAL flag is used. Thus, we make a `posix_fadvise` system call with the POSIX_FADV_NOREUSE flag before sending the journal I/O requests and make a `posix_fadvise` system call with the POSIX_FADV_NORMAL flag on completion of the I/O request.

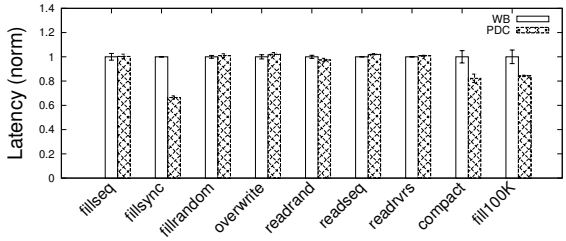
Another important issue with PDC is to maintain data consistency between host cache and storage. When the updates are directly written to storage, the obsolete data can be left in the host cache, creating a data inconsistency hazard. Thus, we invalidate obsolete data in the host cache by means of POSIX_FADV_DONTNEED flag after writing the recent data to storage.

5. Performance Evaluation

To assess the effectiveness of PDC, we measure its performance in comparison with the original policy (WB) that uses



(a) File I/O and database benchmarks

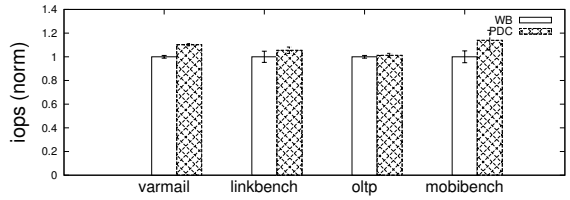


(b) Key-value store benchmark(LevelDB)

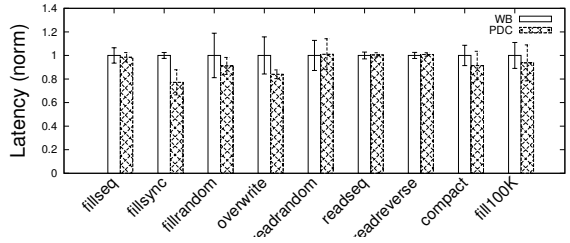
Figure 6: Performances of WB and PDC in a single guest

a host cache in write-back mode, in SSD. The proposed PDC is implemented in QEMU-KVM 2.1 and Linux 4.14. We use the same experimental setup as described in Section 3. We measure the performance for each scenario 10 times and report their average. The performance improvement by PDC comes from two factors. The first is the elimination of the host cache synchronization cost for writes immediately followed by sync, and the other is the prevention of never-reaccess data from the host cache. To investigate the effectiveness of these individual factors, we perform experiments in two different configurations, that is, single and multiple guest machines.

Figure 6 shows the performance of PDC compared to the original caching in SSD when running a single guest. As this configuration provides a large host cache space compared to the working set of a single virtual machine, it primarily presents the benefit from reducing the synchronization cost of journaling. As Figure 6(a) shows, PDC improves performance by 8-32% over the original caching in file I/O and database benchmarks. Specifically, Varmail achieves the most significant improvement, 32%, as it makes a considerable number of metadata updates and synchronization requests. This performance gain, however, seems excessively large compared to the result in Section 3. In our preliminary analysis, a pair of write and fsync operations become faster only by 18% when bypassing the host cache. Thus, the performance improvement in Varmail should be less than 18% because it generates real workloads including other advantageous requests for the host cache. This significant performance improvement turns out to be attributed to the parallelism with multi-threads. We observe that the overall I/O performance increases to some extent proportionally to the number of threads. For example, Varmail improves performance by 5% with a single thread. Similarly, Linkbench and OLTP achieves 8.5% and 8% gains with 10 and 32



(a) File I/O and database benchmarks

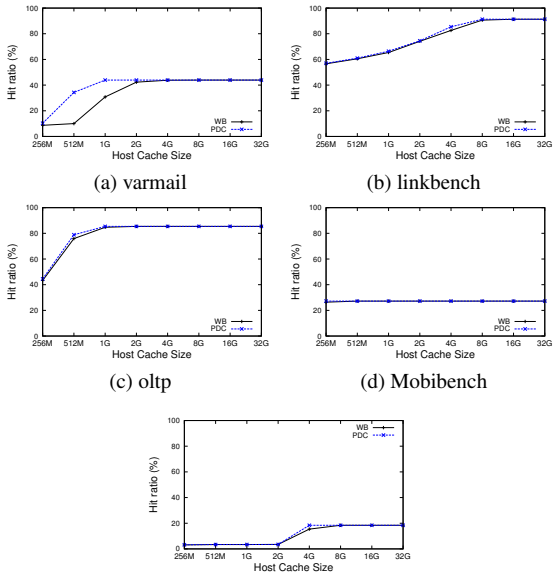


(b) Key-value store benchmark(LevelDB)

Figure 7: Performances of WB and PDC with multiple VMs

threads, respectively, while both workloads gain a marginal performance benefit with a single thread. Mobibench improves performance by 14% with PDC. Figure 6(b) compares the performances of WB and PDC for the LevelDB key-value store application. We show the LevelDB result separately from other benchmarks, as it presents the performance for each operation, not summarizing the overall performance as a single value. As the figure shows, reads and/or buffered writes achieve no remarkable performance benefit from PDC, while operations accompanied by `sync` yield considerable performance enhancement. The `fillsync` and `compact` operations improves the performance by 33% and 18%, respectively. In particular, the reduction of `compact` operation latency deserves more attention. The `compact` operation is periodically invoked and copies valid data in a higher-level file (managed in a fast storage) to the lower-level file (managed in a slow storage). During this process, as a large number of I/Os are issued and thus it takes six orders-of-magnitude longer time (nearly 1s) than other operations (under 5us), the `compact` operation is reported as a primary factor for performance degradation [8]. Accordingly, the 18% reduction of compaction latency results in 200ms reduction in time, contributing to the overall performance significantly.

Figure 7 compares the performances of PDC and original write-back caching policies running four guests concurrently. This result would also reflect the effects of preventing cache pollution by PDC as the host cache space becomes relatively smaller. For an accurate analysis, we also measure the host cache hit ratio under original and PDC policies, using trace-driven simulation. We capture the I/O requests from guests in a hypervisor, and simulate the host cache under the LRU replacement policy. The result shown in Figure 8 demonstrates, however, that there is no significant difference in the hit ratio between two policies, despite the high ratio of journal data in footprint (shown in Section 3). That



(a) varmail (b) linkbench

(c) oltp (d) Mobibench

(e) LevelDB

Figure 8: Cache hit ratio of PDC and WB

is because journaling does not generate a huge number of sequential writes at once, but it writes small updates periodically that are consecutive to previous accesses. Thus, it has little effect of evicting likely-to-be accessed data from the host cache faster than original caching policy. As a result, we observe no significant further enhancement when increasing the number of guests as shown in Figure 7. In this case, Varmail, Linkbench, and OLTP workloads show slightly less performance enhancements, while LevelDB and Mobibench offer better or almost identical performance enhancement as a single guest execution. The slight decrease in performance enhancement of PDC might come from that the performance bottlenecks are caused by various components, such as resource contention and scheduling latency, when the system gets busier. LevelDB makes performance improvements in most write operations by 2-21% with multiple guests, while it makes performance improvements only in `fillsync` and `compact` operations in a single guest. When multiple guests are running together, as the operation executions are interleaved, the reduction of time-consuming operation latency, such as `compact`, might lead to overall write performance improvement eventually.

6. Conclusion

This paper analyzed the effect of guests journaling on the host cache performance in fully virtualized systems. We uncovered the host cache deteriorates I/O performance when coupled with write-once and synchronous journal data, and proposed a pollution-defensive caching to remedy this problem. Our measurement study showed that the proposed caching policy improves the I/O performance of the virtualized system by 3-32%.

References

[1] DBbench. URL https://github.com/memsql/dbbench.

[2] Ext4 wiki. URL https://ext4.wiki.kernel.org/index.php/Main_Page.

[3] Filebench. URL http://www.solarisinternals.com/wiki/index.php/FileBench.

[4] Leveldb benchmarks. URL https://leveldb.googlecode.com/svn/trunk/doc/benchmark.html.

[5] Linkbench. URL https://github.com/facebook/linkbench.

[6] Mobibench. URL https://github.com/ESOS-Lab/Mobibench.

[7] Reiserfs. URL https://en.wikipedia.org/wiki/ReiserFS.

[8] Rocksdb. URL https://github.com/facebook/rocksdb/wiki/RocksDB-Basics.

[9] Sysbench. URL https://launchpad.net/sysbench.

[10] Vmware tools for linux guests. URL http://www.vmware.com/sup-port/ws5/doc/ws_newguest_tools_linux.html.

[11] Vmworld 2013 : Is vmware the mumford and sons of the cloud? URL http://www.techweekeurope.co.uk/work-space/vmworld-vmware-cloud-mumford-and-sons-12950.

[12] Virtualbox. URL http://www.virtualbox.org.

[13] Xen source - progressive paravirtulization. URL http://xen.org/files/summit_3/xen-pv-drivers.pdf.

[14] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and A. B. Yassour. Adding advanced stor-age controller functionality via low-overhead vir-tualization. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[15] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passe? *ACM SIGOPS Op-erating Systems Review*, 44(1):20–24, 2010.

[16] G. Casale, S. Kraft, and D. Krishnamurthy. A model of storage i/o performance interference in virtualized systems. In *Proceedings of the Interna-tional Workshop on Data Center Performance (DCPerf)*, 2011.

[17] Q. Chen, L. Liang, Y. Xia, and H. Chen. Mitigating sync amplification for copy-on-write virtual disk. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[18] J. N. Christoffer Dall. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[19] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Proceedings of 2nd Interna-tional Workshop on Virtualization Performance*, 2009.

[20] S. Hajnoczi. An updated overview of the qemu storage stack. LinuxCon Japan, 2011.

[21] N. Har'El, A. Gordon, A. Landau, M. B. Yehuda, A. Traeger, and R. Ladelsky. Efficient and scalable paravirtual i/o system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[22] D. Hildebrand, A. Povzner, R. Tewari, and V. Tarasov. Revisiting the storage stack in virtualized nas environments. In *Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2011.

[23] R. Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[24] D. Le, H. Huang, and H. Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technol-ogies (FAST)*, 2012.

[25] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

[26] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[27] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Za-dok. Virtual machine workloads: The case for new nas benchmarks. In *Proceedings of the 11th USENIX Conference on File and Storage Technol-ogies (FAST)*, 2013.

[28] K. Wolf. A block layer overview. KVM Forum, 2012. URL http://www.linux-kvm.org/page/KVM_Forum_2012.

[29] C. Xu, S. Gamage, H. Lu, R. R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.