IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 3, 2020, accepted March 28, 2020, date of publication March 31, 2020, date of current version April 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984681

JCache: Journaling-Aware Flash Caching

JIANYU FU[1], YUBO LIU[2], AND GUANGMING LIU[1]

[1]School of Computer, National University of Defense Technology, Changsha 410073, China
[2]School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510275, China

Corresponding author: Jianyu Fu (fujianyu13@nudt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB1000302, in part by the National Natural Science Foundation of China under Grant 61832020, Grant U1611261, and Grant 61872392, in part by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2016ZT06D211, in part by the Pearl River S&T Nova Program of Guangzhou under Grant 201906010008, and in part by the Guangdong Natural Science Foundation under Grant 2018B030312002.

ABSTRACT Virtual machines (VMs) have become the predominant paradigm for deploying applications, and flash-based caches are widely deployed for VMs to improve their performance. However, due to the semantics of journaling in the VMs' file system are transparent to flash caches, traditional flash caching with journaling induces inefficient logging I/O traffic to the shared storage server and duplicated caching in flash, which not only underutilizes flash caches but also aggravates their wearout. This paper presents JCache, a journaling-aware flash caching solution to address these problems. First, JCache proposes the *virtual journal device* design to receive and deliver the journaling semantics in the VMs to the cache manager so as to optimize the VM's journaling. Second, JCache proposes a *cache-only logging* mechanism, which transparently uses the persistent flash caches as the journal area and does cache-only logging to safely eliminate the logging I/O traffic to the shared storage server. Third, JCache proposes the *logical caching* mechanism, which eliminates the duplicated flash caching induced by logging I/Os and in-place updates to mitigate the wearout of flash caches. Evaluations show that JCache improves the application performance by up to 11.4× and reduces the flash cache writes by up to 42% compared to traditional flash caching solutions.

INDEX TERMS Cloud computing, virtual machines, journaling file system, flash caching, caching with journaling.

I. INTRODUCTION

Cloud computing has quickly become the predominant paradigm in modern data centers, and virtual machines (VMs) are widely used as the basis for application deployment. To improve resource utilization and to reduce operation costs, more VMs are consolidated onto the same hosts to run simultaneously. As the VMs' consolidation level continues to grow, the contention for their shared storage server is more intensive, which becomes a serious bottleneck for performance [3], [21], [27]. To accelerate the VMs' storage performance, flash-based SSDs or non-volatile memory (NVM) are commonly deployed as local caches for the VMs [4], [10], [13], [17], [20]. By caching the VMs' data in flash, the VMs can experience low-latency flash cache access instead of high-latency networked storage access.

Journaling is an important technique widely used in modern file systems (e.g., EXT4 [18], NTFS [6], and XFS [29]), including the file systems running in the VMs. By logging the file system updates (e.g., dirty metadata or data) to a journal area (i.e., **logging I/Os**) before committing them to their *original location* in the main file system structure (i.e., **committing I/Os** or **in-place updates**), journaling guarantees the file system's consistency and recoverability in case of system failures [23]. However, journaling introduces duplicated writes and more frequent storage access, i.e., the aforementioned logging I/Os to the journal area beyond the in-place updates, which usually aggravates the contention of the shared storage server.

While deploying flash caching for VMs that mostly use journaling file systems, we observe a three-fold challenge to fully exert the high performance and persistence properties of flash caches. First, the cache manager that manages the shared flash caches for VMs is usually deployed outside of the VMs for simplicity and flexibility (e.g., in the hypervisor [3], [4], [8], [13], [20]), such that the semantics of journaling in the VMs' file system are transparent to flash caching, which makes it difficult to optimize the VM's journaling in the caching layer. Second, the logging I/Os are just considered as regular I/O requests by the cache manager, which then will

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

write these I/O requests to the VMs' virtual disk backend stored in the shared storage server and cache them in flash as well. This traditional procedure in fact underutilizes the persistence property of flash caches, which provides a good opportunity to optimize this I/O flow. Third, although the logging I/Os and the corresponding in-place updates contain much identical content, the cache manager will cache both of them in flash because they are performed on different blocks in the VMs' virtual disk backend (i.e., they are completely different I/O requests from the point of view of the cache manager), inducing massive duplicated caching. In summary, traditional flash caching solutions for VMs are inefficient because of the above challenges induced by the journaling mechanism of the file system in the VMs.

This paper presents JCache to address the aforementioned challenges to managing flash-based caches. First, we propose a virtual journal device design to be aware of the journaling mechanism of the VMs' file system, and to be able to distinguish I/O requests to the journal area and to the main file system structure from outside the VMs, which enables the cross-layer optimizations. Second, we propose a cache-only logging mechanism, which transparently uses the *persistent* flash caches as the objective journal area to store the logging I/Os instead of using the original one in the virtual disk backend, and thus safely eliminates writing the logging I/Os to the shared storage server. Third, we propose a logical caching mechanism, which semantically eliminates the duplicated caching for the logging I/Os and the corresponding in-place updates and thus effectively mitigates the wearout of flash caches.

To the best of our knowledge, JCache is the first to use the journaling semantics of the VM's file system to improve the flash caching efficiency. We have implemented a JCache prototype and evaluated it using various Filebench [19] workloads. The results show that for typical application workloads (i.e., Fileserver, OLTP, and Varmail), by eliminating the logging I/O traffic to the shared storage server and mitigating the duplicated flash caching, JCache improves the applications' throughput by up to 11.4× and reduces the amount of flash cache writes by up to 42%. By co-designing flash caching with the journaling semantics of the VMs' file system, JCache not only fully utilizes the flash caches but also extends the lifetime of the flash device.

The rest of the paper is organized as follows: Section II presents the background and motivations of JCache, Section III describes the design of JCache, Section IV describes the implementation, Section V presents the evaluation of JCache, Section VI examines the related work, and Section VII concludes the paper.

II. BACKGROUND AND MOTIVATIONS

A. JOURNALING FILE SYSTEM

Journaling file systems, such as EXT3/EXT4 [18], [30], NTFS [6], and XFS [29], are widely used as local file systems in the VMs in cloud computing environments. A typical journaling file system reserves a special journal area (e.g., a file in the file system) to persistently log the file system updates (e.g., dirty metadata or data) before writing the updates to their *original location* in the complex file system structure. By forcing the logging I/Os before the in-place updates, the journaling technique can efficiently recover the file system to a consistent state through scanning the journal and redoing any incomplete committed updates after system failures.

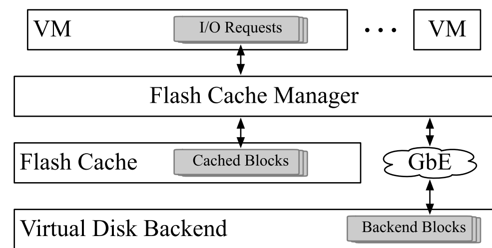


FIGURE 1. Flash caching for VMs.

There are typically three journaling modes, i.e., *writeback*, *ordered*, and *journal* mode. (1) In writeback mode, only file system metadata updates are logged to the journal, and data updates are written directly to their original location. This mode provides guarantees to metadata consistency, but does not guarantee data consistency. (2) In ordered mode, again only metadata updates are logged, but these metadata updates cannot be persisted to the journal until corresponding data updates have been written to their original location. This mode provides strong consistency guarantees for both metadata and data. (3) In journal mode, both metadata and data updates are logged, and it provides the same consistency guarantees as ordered mode.

Although the journaling mechanism has different journaling modes to provide different levels of consistency guarantees, all need to write metadata or data updates to the journal in addition to their in-place updates. In writeback and ordered modes, only metadata updates are written twice (i.e., once to the journal, and then to their original location); in journal mode, data updates are written twice as well as metadata updates.

B. FLASH CACHING WITH JOURNALING

Flash-based SSDs are being increasingly deployed in cloud computing environments as shared flash caches for concurrently running VMs to accelerate their storage performance [1], [4], [8], [10], [12], [13], [15], [17], [20], [21], [24]. Due to the easy control for the VMs and sharing the flash caches, managing flash caches outside of the VMs (e.g., in the hypervisor) is a commonplace, as discussed in [4], [8]. As illustrated in Figure 1, after deploying flash caching for VMs, the cache manager receives all the I/O requests issued from the VMs, sends them to the virtual disk backend when necessary, and caches them with various replacement policies. Take the commonly used *write-through* caching policy

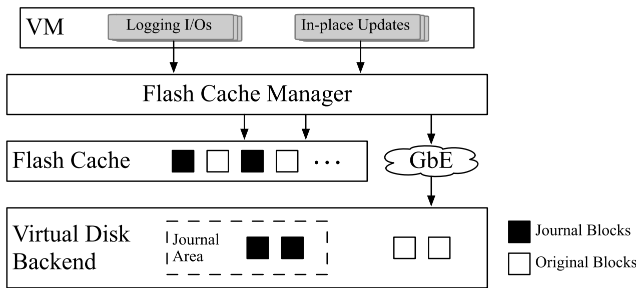


FIGURE 2. Traditional flash caching with journaling presented.

as an example. When the cache manager receives an I/O request from one VM, if it is a read, the cache manager will read the data from either flash caches (if cached) or the virtual disk backend (if not cached) and caches the data in flash in the latter case; if it is a write, the cache manager will write the data to both flash caches and the virtual disk backend before returning an acknowledgment to the VM.

Here we revisit the caching I/O flow of traditional flash caching while considering the journaling mechanism in the VMs, as illustrated in Figure 2.

Semantically speaking, the I/O requests sent by the VMs regarding the journaling mechanism consist of two types: (1) *logging I/Os*, which are written to the journal blocks in the journal area, and (2) *in-place updates*, which are written to the original blocks in the main file system structure. According to the principle of the journaling mechanism, the logging I/O and in-place update induced by the same file system modification are different I/O requests, but they have the same content. However, after these I/O requests are sent out of the VMs, such journaling semantics of the VM's file system have been lost because of the interleaving complex I/O stacks. What remains are only the I/O requests' block offset and I/O size. For example, when a block B in the VM's file system needs to be updated, the VM first logs the updates to a journal block J in the journal area and then updates the original block B in a later time. The underlying cache manager will receive two I/O requests, one to update block J and one to update block B. The cache manager does not know that block J is a journal block and that block J and B contain the same content, so that it will process and cache both blocks just as regular different block I/O requests. However, the missing of the journaling semantics of the VM's file system to the cache manager leads to a two-fold caching challenge.

First, the cache manager processes *unnecessary* logging I/Os to the virtual disk backend. When a logging I/O is issued from the VM, the cache manager receives it, caches it in flash, and writes it to the virtual disk backend in the write-through caching policy. However, this traditional caching procedure does not fully utilize the persistence property of flash caches. As illustrated earlier, the adoption of journal area is that it provides another persistent location to store the file system updates other than do in-place updates directly. The persistent flash caches can also play as such a role, which is not taken into account by traditional flash caching solutions. In the write-back caching policy, the logging I/Os can be acknowledged to the VM once they are written to flash cache, however, they can be evicted from flash cache later and thus need to be written to the virtual disk backend, inducing unnecessary logging I/Os to the virtual disk backend as well.

Second, the cache manager handles duplicated caching for file system updates that induce both logging I/Os and in-place updates. When an update needs to be written to both the journal area and its original location (e.g., metadata updates in all the journaling modes, and also data updates in journal mode), it induces logging I/O request first and then in-place update I/O request. From the point of view of the cache manager, they are different I/O requests to different blocks in the virtual disk backend, and the cache manager caches both of them to flash without knowing that they actually contain identical content, inducing duplicated caching for the same file system updates in both the write-through and write-back caching policies. Although it is possible to employ cache deduplication [7], [15], [34], it incurs unnecessary overhead for the additional management and computation.

Although flash caches are widely deployed for VMs that mostly run journaling file systems, the aforementioned challenges of flash caching with journaling have tremendous impact on the flash caching efficiency in terms of cache performance and endurance. They are important and unexplored problems, which are addressed by this paper's solution, JCache.

III. DESIGN

We present JCache, a journaling-aware flash caching solution. In this section, we first discuss the architecture of JCache, which adopts a virtual journal device design to receive and deliver the journaling semantics of the VMs' file system to the cache manager. We then describe the cache-only logging mechanism, which transparently uses the persistent flash caches as the objective journal area to safely eliminate the logging I/O traffic to the shared storage server. Finally, we present the logical caching mechanism, which semantically deduplicates the redundant flash caching between logging I/Os and in-place updates to improve the lifetime of flash caches.

A. JOURNALING-AWARE CACHING ARCHITECTURE

Figure 3 shows the detailed architecture of JCache. It consists of three modules: the Virtual Journal Device module to be aware of the journaling semantics of the VMs' file system; the Cache-only Logging module to receive all (and only) the I/O requests to the journal area delivered by the Virtual Journal Device module and to do cache-only logging to flash (Section III-B); and the Logical Caching module to logically cache in-place updates, which means caching in-place updates in flash only if they have not been logged to flash by the Cache-only Logging module (Section III-C).

To solve the problems of inefficient flash caching with journaling without sacrificing the caching flexibility, the

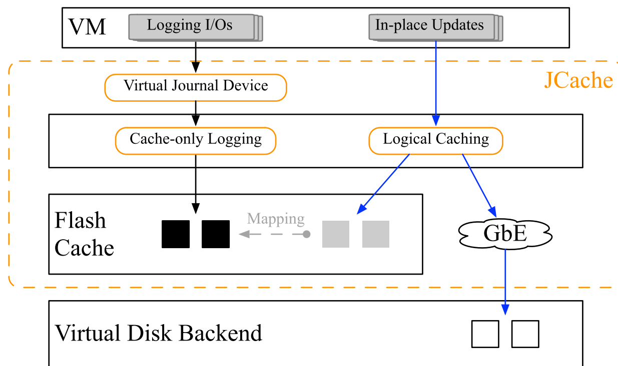


FIGURE 3. Architecture of JCache.

journaling semantics of the VMs' file system must be efficiently delivered to the cache manager. As discussed earlier, after one I/O request being issued outside of the VMs, the complex interleaving I/O stacks (e.g., the VM's kernel I/O stack) has filtered most semantic information, and only the request's block offset and I/O size remain.

We propose to create a virtual journal device for each VM, and it appears as a regular storage device in the guest VM. The guest operating system (OS) can use this device as its external journal device for the file system. Because the backend of the virtual journal device is managed by JCache, so that when the guest OS issues the logging I/O requests to this device, JCache is able to receive all the logging I/Os and knows clearly that these I/O requests are sent to the journal area by this specific VM. By checking the beginning bytes of each logging I/O block, JCache can easily identify if the block is a descriptor block or commit block. And by analyzing the descriptor block, JCache is able to identify the logged blocks' original location (i.e., block offset) in the main file system structure. Note that the logged blocks can be the file system's metadata blocks or data blocks, which depends on the journaling mode used by the file system, but JCache can do the following optimizations of cache-only logging and logical caching for both metadata and data updates. Besides, all the other I/O requests that JCache receives not through the virtual journal device will be in-place I/O requests to the VMs' file system.

B. CACHE-ONLY LOGGING

As introduced in Section III-A, JCache can distinguish all the logging I/Os from in-place updates through the virtual journal device. In traditional caching I/O flow, the logging I/Os sent to the cache manager will be cached in flash and also written to the journal area in the virtual disk backend.

For the journaling mechanism, its essence is that when the file system is updated, journaling stores the updates to a persistent location before doing in-place updates. If we revisit the existence of flash caches, we can observe that when the logging I/Os are cached in flash, the file system updates are already persistently stored because of the persistence property of the flash device, regardless of whether the logging I/Os have been written to the virtual disk backend or not.

Based on the above observation, we propose a cache-only logging mechanism. This mechanism means that, when JCache receives a logging I/O request, it only caches the logging I/O in flash, and it will not write the logging I/O to the virtual disk backend in any case. The essence of this mechanism is that JCache uses the persistent flash caches as the objective journal area to store the logging I/Os, instead of using the original journal area in the virtual disk backend. Note that the caching I/O flow changes have no impact on how the guest OS uses journaling in the VM, and how JCache manages the logging I/Os is also transparent to the guest OS. Because the journal area is originally designed to be a small circular buffer, the large flash cache space can easily handle such space requirements. Besides, if in rare cases flash caches cannot provide enough space for the required journal area, JCache can simply reissues the logging I/O requests to the virtual disk backend, which will become the traditional caching I/O flow.

By using cache-only logging to persist the logging I/Os to only flash caches, JCache provides efficient flash caching for the journaling mechanism of the VMs' file system, and it can safely eliminate all the I/O traffic to the virtual disk backend used for logging purpose. It not only improves the cache performance but also mitigates the VMs' contention for the shared storage server.

C. LOGICAL CACHING

As discussed in Section II-B, in traditional caching I/O flow, when the cache manager receives logging I/O requests and in-place update I/O requests induced by the same file system updates, it caches both of them in flash since they are different I/O requests that write different blocks in the virtual disk backend. However, if we consider the journaling semantics, the same updates are actually written for twice, and also cached for twice in flash.

Based on the above observation, we propose the logical caching mechanism. The mechanism refers to that after one modified file system block has been logged (i.e., cached) into the cache-based journal area in flash by processing the logging I/O request, JCache will do a *logical caching* operation for its latter in-place update I/O request. It means that JCache only logically maps the original block to the cache block in flash that is already mapped to the corresponding journal block without really writing the same content again to flash cache. For the in-place update I/O request, JCache does not cache the same content again to flash, but still issues it to the virtual disk backend, i.e., write the modified block to its original location in the virtual disk backend.

As shown in Figure 4, say the guest OS needs to update block B. To ensure file system consistency, the guest OS will first log the updated content of block B to a journal block (say block J) in the journal area (in addition to writing the descriptor block and commit block), and then does in-place update for block B. So that JCache will first receive a logging

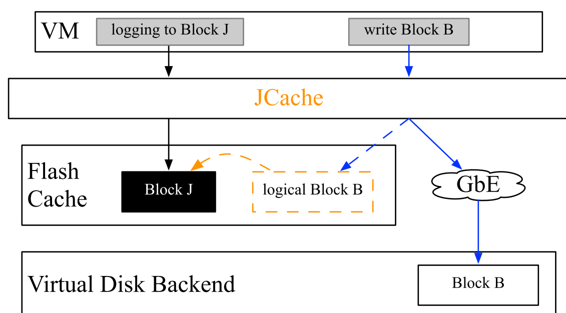


FIGURE 4. Logical caching for in-place updates.

I/O request to write block J, and then receive an in-place update I/O request to write block B. And the two I/O requests to block J and block B are writing the same content (i.e., block B's updated content) according to the principle of the journaling mechanism.

Upon receiving the logging I/O request to write block J, JCache caches block J to a cache block (say block C) in flash, and then cache block C is mapped to block J. JCache will not issue this logging I/O request to the virtual disk backend according to the cache-only logging mechanism. JCache then receives the in-place update I/O request to write block B in the virtual disk backend. With the journaling semantic information delivered to JCache, JCache knows that block J is the journal block where block B is logged to and they have the same content, and that block J has already been cached in flash. Then JCache does a logical caching operation for block B, i.e., it maps block B to block J's cache block C as its own cache block as well and does not really write block B into flash. Now cache block C is mapped to both block J and block B. Then JCache issues an in-place update I/O request to write block B to the virtual disk backend.

For the above procedure, two things need to be noted. First, JCache recognizes that block J is the journal block where block B is logged to by reading the descriptor block within the journal transaction that contains block J, because the descriptor block stores the offset of block B for block J. Second, since block J is the journal for block B, as mentioned earlier, they will always have the same content, which is the working basis of the journaling mechanism. Even so, JCache still calculates a checksum value for both block J and block B, and compares their checksum values before doing logical caching for block B to make additional verification that they indeed have the same content.

For the update or replacement regarding the cache blocks that map to more than one block (e.g., the cache block C in flash which maps to both block J and block B), they are different from traditional caching I/O flows as follows.

When the guest OS needs to update block B again, JCache cannot update its mapped cache block C directly. Before issuing the in-place update I/O request for block B, the guest OS will again issue a logging I/O request to log the new content of block B to the journal area first, which is most likely not block J but a different journal block (say block J1). Then on the new logging I/O request to write block J1, JCache caches block J1 to a newly allocated cache block (say block C1) in flash, and unmaps block B from cache block C, which then will be mapped to only block J. After JCache receives the new in-place update I/O request for block B, it again does logical caching for block B to map block B to cache block C1 without really writing the new block B in flash. If by coincidence block J and block J1 are the same journal block, then JCache can still use cache block C and directly update it instead of allocating a new cache block C1.

When the guest OS needs to journal another updated block A to the journal offset of block J, JCache will check if cache block C is still mapped to block B. If so, JCache will unmap block J from cache block C, which then will be mapped to only block B, and allocate another cache block to cache the new journal block J which contains the content of block A. If not, JCache can directly update cache block C. Then JCache will do logical caching for block A accordingly.

When block B is selected to be evicted out of cache, JCache will check if cache block C is still mapped to block J. If not, JCache can evict block B from flash cache and use cache block C to cache new blocks. If so, JCache typically does not evict journal blocks because in the cache-only logging mechanism, the journal blocks are stored only in flash caches and they do not occupy much space, so that JCache will not evict block B but look for other available blocks to evict. If the flash cache space is even too small for the circularly used journal area, then JCache just uses the journal area in the virtual disk backend as discussed earlier.

Note that the guest OS manages its journaling mechanism without any changes in the VM, and the above caching I/O flows are only operated by JCache and they are transparent to the guest OS.

D. RECOVERY

In this section, we discuss how JCache recovers the file system in the VMs to a consistent state in case of guest OS or host failures. Although there are other failure recovery issues related to flash caching as introduced in [8], [12], [24], they are out of the scope of this paper, and here we mainly concentrate on the journaling part. The general recovery procedure for the file system in the guest OS does not need any changes, and JCache can correctly process all the journal I/O requests sent by the VMs.

In case of failures of either guest OS or host, the blocks cached in flash have been persistently stored and they can be recovered. For the VMs' journal blocks, although they are not written to the virtual disk backend in the optimization of cache-only logging, they have been stored in flash cache and can be recovered and accessed as well, so that JCache does not relax the caching system's reliability. The guest OS can perform regular recovery procedure to read its journal, and JCache will handle all the journal I/O requests correctly to the cache-based journal area, so as to redo all the incomplete

committed updates and recover the VMs' file system to a consistent state. If the VM is migrated or restarted in a different host, JCache can forward all the journal I/O requests to flash caches of the host where the VM lastly run like the mechanism proposed in [1]. If the flash device encounters destructive failures that cannot be recovered, to provide better reliability, JCache can employ the peer-replication caching technique as introduced in [3] to solve such problems, which has been demonstrated to be effective.

IV. IMPLEMENTATION

We implemented a JCache prototype based on the QEMU [2] emulator with KVM [11] enabled.

We modified the VM initialization procedure of QEMU to create a virtual journal device for the VM, and the virtual journal device is not backed by any file or volume but completely managed by JCache. The virtual journal device appears as a regular block device in the guest OS. The guest OS can format the virtual journal device as an external journal device for its file system, so that JCache can receives all the journal I/O requests through the virtual journal device.

Flash cache is managed as fixed-size blocks, which is 4KB by default, and LRU is the default replacement policy. The whole flash space is split into two regions: cache header and data area. The cache header consists of the superblock and the address mapping information. The latter is used not only for caching the file system metadata and data, but also as the cache-based journal area to cache the journal blocks in the VMs' file system according to the cache-only logging mechanism.

V. EVALUATION

A. EXPERIMENTAL SETUP

The evaluation environment includes two nodes, one as the VMs' backend storage server and the other as the client to run VMs. Each node has two eight-core 2.4GHz Xeon E5-2630 CPUs and 64GB of RAM. The host runs Ubuntu 14.04, and the VM runs CentOS 7. Each VM is configured with 1 vCPU, 2GB RAM, and two virtual disks: one is the guest OS and the other is a 1TB virtual disk, which is formatted as an EXT4 file system in the VM to conduct experiments. The 1TB virtual disk is stored in an NFSv4 datastore, backed by a 1TB 7.2K RPM hard disk in the storage server and connected via a 10 Gigabit Ethernet. The guest OS virtual disk is stored in the client's local hard disk to mitigate its impact on the experimental 1TB virtual disk.

In JCache, because the journal of the VMs' file system is actually backed by flash caches which usually have relatively large cache space, we set the default journal size as 1GB, although it can be configured as any size. We name traditional flash caching solutions that are not aware of the journaling semantics of the VMs' file system as *Baseline*, and it has a default journal size of 128MB as used by EXT4. To see how the journal size affects performance and to make fair comparisons between Baseline and JCache in terms of the

TABLE 1. I/O characteristics of the micro-benchmarks.

Workloads	Operations to the EXT4 file system
Makedirs	Create 500K directories
Createfiles	Create 500K files (mean size: 16KB), write the whole file
Seqwrite	Write sequential I/Os to an empty file (I/O size: 1MB)
Randomwrite	Write random I/Os to a preallocated file (I/O size: 4KB)

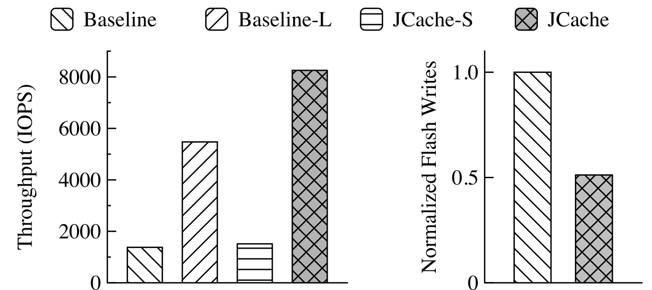


FIGURE 5. Makedirs (in ordered mode).

journal size, we added two more baselines, i.e., Baseline-Large (*Baseline-L*), which is the same as Baseline but with a larger journal size of 1GB, and JCache-Small (*JCache-S*), which is the same as JCache but with a smaller journal size of 128MB. So that Baseline and JCache-S have the same journal size of 128MB, and Baseline-L and JCache have the same journal size of 1GB. We mainly tested the ordered and journal modes, because the former logs only metadata updates and the latter logs both metadata and data updates, as illustrated in Section II-A. We will illustrate the journaling mode that is used in each evaluation.

B. MICRO-BENCHMARK EVALUATION

Filebench [19] is a commonly used file system and storage benchmark that can generate a large variety of micro-benchmark and real-world application workloads. In this section, we evaluated JCache with four basic micro-benchmarks, i.e., Makedirs, Createfiles, Seqwrite, and Randomwrite from Filebench. The micro-benchmarks are run in the VM against the aforementioned EXT4 file system in the guest OS. The I/O characteristics of the micro-benchmarks, i.e., the operations to the EXT4 file system, are shown in Table 1.

Figure 5 shows the performance comparison for Makedirs in ordered mode. Because Makedirs mainly generates metadata updates and has little data updates, there is little difference between using ordered mode or journal mode (both log metadata updates to journal), so that we only present the results under ordered mode. Because the journal size has little impact on flash writes, we only present the flash write results of Baseline and JCache for Makedirs and all the following evaluations. As the results show, compared to Baseline, JCache improves the throughput by 5× and reduces the flash writes by 48.8%. And the reason is that JCache uses flash caches as the journal area and reduces about 50% networked I/O traffic by eliminating the logging

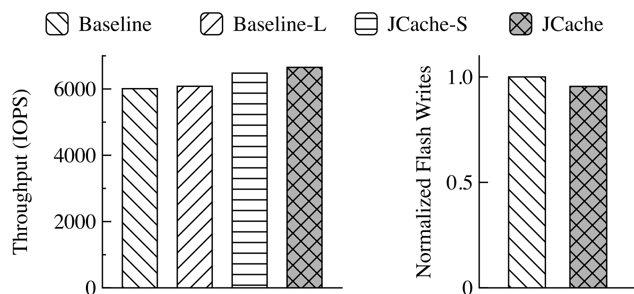


FIGURE 6. Createfiles (in ordered mode).

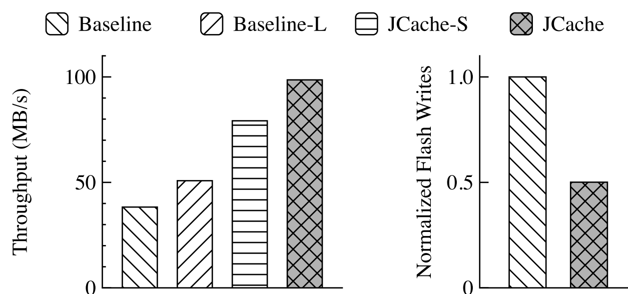


FIGURE 8. Seqwrite (in journal mode).

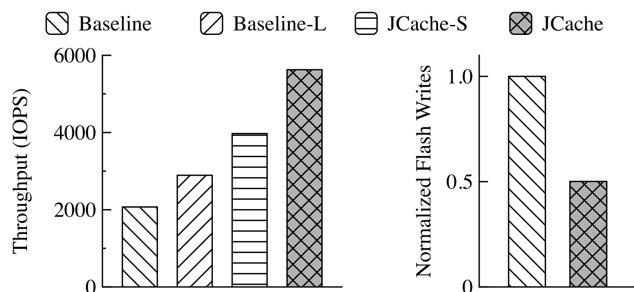


FIGURE 7. Createfiles (in journal mode).

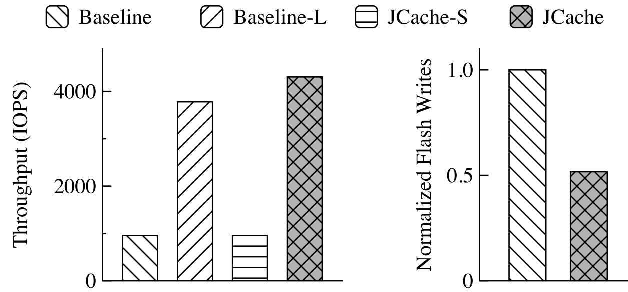


FIGURE 9. Randomwrite (in journal mode).

I/O requests to the virtual disk backend, and mitigates the duplicated caching. For Baseline-L that has the same journal size as JCache, JCache has better throughput than it by 51%. For JCache-S that has the same journal size as Baseline, JCache-S still has better throughput by 10%. JCache is better than the baselines at different journal sizes, which also demonstrates the effectiveness of JCache's optimizations.

We evaluated Createfiles in both ordered mode and journal mode, as shown in Figure 6 and 7 respectively. Because Createfiles is not metadata-intensive, the performance is similar for Baseline and JCache in ordered mode. But JCache still has the best throughput, which is 11% and 9% better than Baseline and Baseline-L respectively. JCache reduces the flash writes by 4.5% than Baseline. In journal mode, the performance of JCache is better than Baseline and Baseline-L by 172% and 95% respectively, and it reduces the flash writes by 49.9%. JCache again has significantly better performance than Baseline because it eliminates massive logging I/O traffic to the virtual disk backend and mitigates the flash cache writes.

Because both the Seqwrite and Randomwrite workloads are data-intensive and have little metadata updates, we only present their results in journal mode, as shown in Figure 8 and 9 respectively. For Seqwrite, JCache has better throughput than Baseline and Baseline-L by 157% and 94% respectively, and it reduces the flash writes by 49.9%. For Randomwrite, JCache has better throughput than Baseline and Baseline-L by 3.5× and 14% respectively, and it reduces the flash writes by 48.3%. In the Randomwrite workload, because the journaling mechanism can transfer random writes to the virtual disk backend to sequential writes by sequentially logging the writes to the journal area [23], and JCache only logs the updates to flash caches without networked logging I/O traffic, so that JCache performs much better than Baseline.

TABLE 2. I/O characteristics of the applications.

Workloads	Main operations
Fileserver	Create, write, append, read, state, and delete
OLTP	Write, dsync, and read
Varmail	Create, append, fsync, read, and delete

C. APPLICATION EVALUATION

We evaluated JCache using typical application workloads, i.e., Fileserver, OLTP, and Varmail from Filebench. Fileserver emulates a file server, OLTP emulates an online transaction processing service, and Varmail emulates a mail server. Each application's main operations are shown in Table 2. Because all the application workloads have both metadata and data updates, they are evaluated in both ordered mode and journal mode respectively.

Figure 10, 11, and 12 show the performance comparison for Fileserver, OLTP, and Varmail respectively in ordered mode, and Figure 13, 14, and 15 show the performance comparison for these applications in journal mode.

For Fileserver in ordered mode, the throughput of JCache is better than Baseline and Baseline-L by 10% and 6% respectively. Because only metadata updates are logged and Fileserver is not metadata-intensive, JCache induces less flash writes than Baseline by 3.4%. In journal mode, JCache has better throughput than Baseline and Baseline-L by 126% and 44% respectively, and it reduces the flash writes by 43%. And the improvement comes from that JCache mitigates

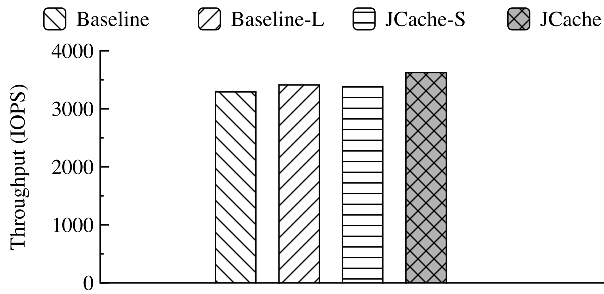


FIGURE 10. Fileserver (in ordered mode).

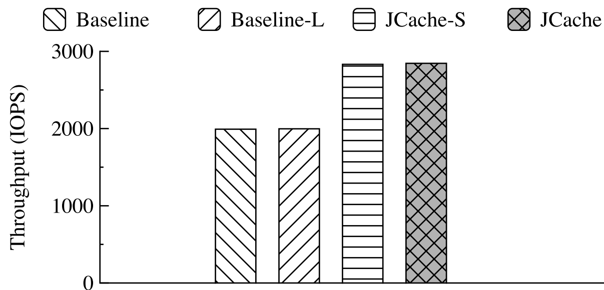


FIGURE 11. OLTP (in ordered mode).

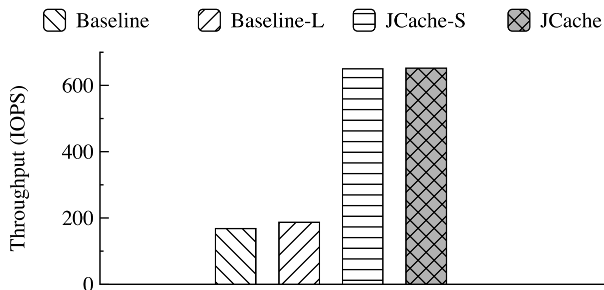


FIGURE 12. Varmail (in ordered mode).

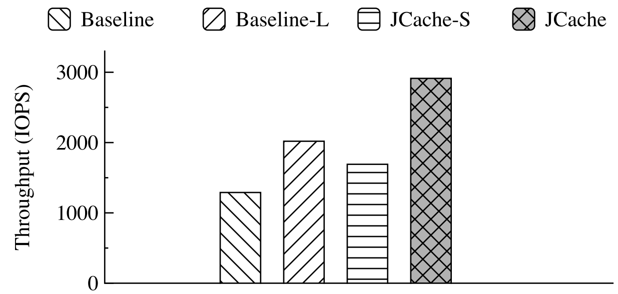


FIGURE 13. Fileserver (in journal mode).

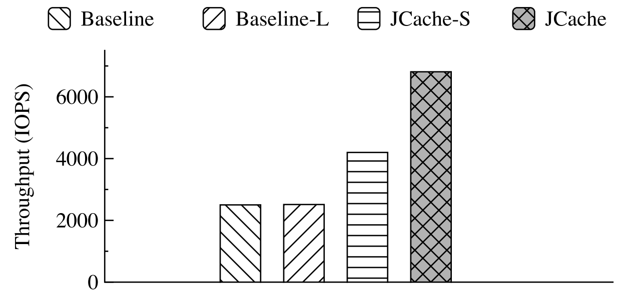


FIGURE 14. OLTP (in journal mode).

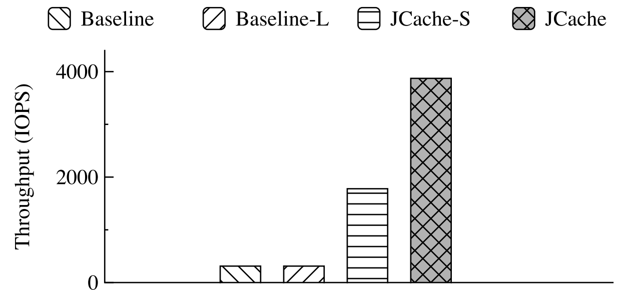


FIGURE 15. Varmail (in journal mode).

much logging I/O traffic to the virtual disk backend and massive redundant flash caching, which demonstrates the effectiveness of JCache's optimized mechanisms of cache-only logging and logical caching. Besides, JCache has better throughput than JCache-S by 72%, which also demonstrates the advantage of using large journal space enabled by using flash cache-only logging.

For OLTP in ordered mode, JCache is better than both Baseline and Baseline-L by about 42%, and it reduces the flash writes by 24.9%. For Baseline or JCache, the performance difference of changing the journal size is less than 1%, which is because that OLTP is sync-intensive and every update needs to be persisted and the journal space can store much updates before fully filled. In journal mode, JCache increases the throughput by about 1.7× than both Baseline and Baseline-L, and it reduces the flash writes by 34.4%.

For Varmail in ordered mode, JCache improves the throughput than Baseline and Baseline-L by 2.9× and 2.5× respectively, and it induces less flash writes by 28.3%. In journal mode, JCache has better throughput than both Baseline and Baseline-L by about 11.4×, and it induces less flash writes by 8.5%. The reduction of flash writes compared to that in ordered mode is because most I/O requests are processed to the cache-based journal area and the in-place update I/O requests will be handled later by checkpointing the blocks in the journal.

VI. RELATED WORK

Deploying flash-based caching for virtual machines in cloud computing environments has been extensively researched in the literatures. Mercury [4] makes a comprehensive discussion on how to deploy flash caches for the VMs and presents the effectiveness of using flash caches. In [10], the authors also discuss the various design space (e.g., regarding write-back policies, cache space) for flash caching. Flashtier [25] combines flash caching with the internal management of SSD to solve the caching limitations of using traditional SSDs. S-CAVE [17] proposes a rECS metric to guide cache

space allocation among the VMs according to their relative cache demands. vCacheShare [20] makes dynamic cache space allocation based on a cache utility model that considers the VMs' I/O access characteristics. Centaur [13] uses the VMs' miss ratio curves for cache sizing to achieve the VMs' Quality-of-Service requirements. CloudCache [1] designs a new cache demand model, Reuse Working Set, to predict a VM's cache space requirement and to make cache space allocation. CacheDedup [15] and [7] employ cache deduplication to improve the VMs' I/O performance. COWCache [8] co-designs flash caching with the management of Copy-on-Write virtual disks to improve the VMs' caching performance. In [12], [24], the authors optimize the cache writeback policies for flash caching. Flash caching is also widely used in the industries. Flashcache [28] is a general-purpose block-level caching solution, which has been deployed in the production environments in Facebook. Bcache [22] and Dm-cache [31] are block-level caching solutions supported in the Linux kernel as well. VMWare has its caching solution, VAIO [32], for its enterprise virtualization products. However, although the previous works focus on different aspects of flash caching, e.g., caching architecture, space allocation, cache replacement, or writeback policies, none of them specially considered the caching challenges brought by the journaling mechanism of the VMs' file system, which has been presented in detail and also addressed by JCache.

Journaling is a traditional consistency mechanism which is widely used in both local file systems (e.g., EXT4 [18], XFS [29]) and distributed file systems (e.g., Ceph [33], Lustre [26]). In local file systems, the journaling overhead mainly comes from the duplicated writes. JBD2 [18] in the Linux kernel supports deploying the journal on high-performance storage device such as SSD. However, it needs exclusive storage device and still does not solve the problem of duplicated writes. Some existing works co-design local file systems with non-volatile memory (NVM) to reduce the overhead of journaling. In [5] and [35], the authors propose to use NVM as a journal device and present a fine-grained approach to reduce the write amplification induced by journaling. HasFS [16] considers NVM as a journal device and page cache, and combines the techniques of journaling and Copy-on-Write to reduce the overhead of consistency guarantee. UBJ [14] reduces the duplicated writes by using an in-place checkpointing mechanism. In studies of journaling in distributed file systems, similar to local file systems, some mainstream distributed file systems, e.g., Ceph [33], reduce the journaling overhead by simply deploying it on high-performance storage device.

Different from all the previous works, to the best of our knowledge, JCache presents some new flash caching challenges, i.e., inefficient logging I/O traffic to the shared storage server and duplicated caching in flash, induced by the semantics of journaling in the VMs' file system. It observes the opportunity to use the high-performance and persistent flash caches to reduce the overhead of caching with journaling, and proposes the virtual journal device design, the mechanisms of cache-only logging and logical caching to address the challenges. The evaluation results for typical application workloads demonstrate the effectiveness of JCache's optimizations. Although the discussion in the paper focuses on flash-based caches, the general JCache approach is also applicable to new NVM technologies (e.g., 3DXpoint [9]), which will likely be used as caches for the VMs in cloud computing environments. While NVM has better performance than flash devices, it will benefit more from JCache's optimizations to improve the VMs' performance.

VII. CONCLUSION

As flash caching is widely deployed for the VMs in cloud computing environments to improve their performance, we uncover that the semantics of journaling in the VMs' file system bring severe challenges of inefficient logging I/O requests to the shared storage server and duplicated flash caching of logging I/Os and in-place updates. We propose JCache, a journaling-aware flash caching solution to address the challenges. JCache first proposes a virtual journal device design to bridge the semantic gap between the management of journaling in the VMs and flash caching. With the journaling knowledge in the VMs, JCache then presents the cache-only logging mechanism to fully utilize the persistence property of flash caches. It uses the persistent flash cache as the journal area to store the logging I/Os and to safely eliminate the logging I/O traffic to the shared storage server. Finally, JCache presents the logical caching mechanism to eliminate the duplicated flash caching induced by the duplicated writes in the VMs, which not only improves the cache performance but also extents the lifetime of flash caches.

REFERENCES

[1] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "CloudCache: On-demand flash cache management for cloud computing," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 355–369.

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, pp. 41–46, 2005.

[3] D. Bhagwat, M. Patil, M. Ostrowski, M. Vilayannur, W. Jung, and C. Kumar, "A practical implementation of clustered fault tolerant write acceleration in a virtualized environment," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 287–300.

[4] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Proc. 28th Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2012, pp. 1–12.

[5] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proc. 32nd Int. Conf. Mass Storage Syst. Technol. (MSST)*, 2016, pp. 1–13.

[6] H. Custer, *Inside Windows NT*. Los Angeles, CA, USA: Microcomputer Applications, 1992. [Online]. Available: https://dl.acm.org/doi/book/10.5555/138407

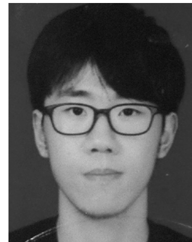
[7] J. Feng and J. Schindler, "A deduplication study for host-side caches in virtualized data center environments," in *Proc. 29th Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2013, pp. 1–6.

[8] J. Fu, Y. Lu, J. Shu, G. Liu, and M. Zhao, "COWCache: Effective flash caching for Copy-on-Write virtual disks," *Cluster Comput.*, vol. 11, pp. 1–17, Jun. 2019.

[9] J. Handy, "Understanding the Intel/Micron 3D XPoint memory," in *Proc. Storage Developer Conf.*, 2015, pp. 1–30.

[10] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer, "Flash caching on the storage client," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2013, pp. 127–138.
[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.
[12] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 45–58.
[13] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side SSD caching for storage performance control," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, 2015, pp. 51–60.
[14] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 73–80.
[15] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line deduplication for flash caching," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 301–314.
[16] Y. Liu, H. Li, Y. Lu, Z. Chen, N. Xiao, and M. Zhao, "HasFS: Optimizing file system consistency mechanism on NVM-based hybrid storage architecture," *Cluster Comput.*, vol. 15, pp. 1–15, Dec. 2019.
[17] R. Barik, J. Zhao, and V. Sarkar, "S-CAVE: Effective SSD caching to improve virtual machine storage performance," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2013, pp. 103–112.
[18] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new EXT4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, pp. 21–33.
[19] M. Richard, and J. Mauro.*Filebench*. Accessed: Mar. 1, 2020. [Online]. Available: https://github.com/filebench/filebench
[20] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "VCacheShare: Automated server flash cache space management in a virtualization environment," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 133–144.
[21] M. Oh, H. Eom, and H. Y. Yeom, "Enhancing the I/O system for virtual machines using high performance SSDs," *Proc. Int. Perform. Comput. Commun. Conf. (IPCCC)*, 2014, pp. 1–8.
[22] K. Overstreet. *Linux Bcache*. Accessed: Mar. 1, 2020. [Online]. Available: https://bcache.evilpiepirate.org
[23] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 196–215.
[24] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches," *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 451–462.
[25] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: A lightweight, consistent and durable storage cache," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 267–280.
[26] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proc. Ottawa Linux Symp.*, 2003, pp. 380–386.
[27] J. Shafer, "I/O virtualization bottlenecks in cloud computing today," in *Proc. 2nd Workshop I/O Virtualization (WIOV)*, 2010, p. 5.
[28] D. Mituzas. *Flashcache at Facebook: From 2010 to 2013 and Beyond*. Accessed: Mar. 1, 2020. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920
[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proc. USENIX Annu. Tech. Conf.*, 1996, pp. 1–5.
[30] S. Tweedie, "Ext3, journaling file system," in *Proc. Ottawa Linux Symp.*, 2000, pp. 24–29.
[31] E. V. Hensbergen and M. Zhao, "Dynamic policy disk caching for storage networking," IBM, New York, NY, USA, Tech. Rep. RC24123, 2006.
[32] VMware. *Sphere APIs for I/O Filtering (VAIO) Program*. Accessed: Mar. 1, 2020. [Online]. Available: https://code.vmware.com/programs/vsphere-apis-for-io-filtering
[33] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "CEPH: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Oper. Syst. Design Implement. (OSDI)*, 2006, pp. 307–320.
[34] Q. Yang, R. Jin, and M. Zhao, "SmartDedup: Optimizing deduplication for resource-constrained devices," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 633–646.
[35] X. Zhang, D. Feng, Y. Hua, and J. Chen, "Optimizing file systems with a write-efficient journaling scheme on non-volatile memory," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 402–413, Mar. 2019.

JIANYU FU received the B.S. degree in software engineering from Nankai University, China, in 2013, and the M.S. degree in computer science from the National University of Defense Technology, China, in 2015, where he is currently pursuing the Ph.D. degree with the School of Computer. His research interests include storage systems and cloud computing.



YUBO LIU is currently pursuing the Ph.D. degree with the School of Data and Computer Science, Sun Yat-Sen University. His research interests include local file systems, distributed and parallel file systems, non-volatile memory, and operating systems.

GUANGMING LIU received the B.S. and M.S. degrees in computer science from the National University of Defense Technology, China, in 1980 and 1986, respectively. He is currently a Professor with the School of Computer, National University of Defense Technology. His research interests include high performance computing, massive storage, and cloud computing.