MultiLanes: Providing Virtualized Storage for OS-Level Virtualization on Manycores

JUNBIN KANG, CHUNMING HU, TIANYU WO, YE ZHAI, BENLONG ZHANG, and JINPENG HUAI, Beihang University

OS-level virtualization is often used for server consolidation in data centers because of its high efficiency. However, the sharing of storage stack services among the colocated containers incurs contention on shared kernel data structures and locks within I/O stack, leading to severe performance degradation on manycore platforms incorporating fast storage technologies (e.g., SSDs based on nonvolatile memories).

This article presents MultiLanes, a virtualized storage system for OS-level virtualization on manycores. MultiLanes builds an isolated I/O stack on top of a virtualized storage device for each container to eliminate contention on kernel data structures and locks between them, thus scaling them to manycores. Meanwhile, we propose a set of techniques to tune the overhead induced by storage-device virtualization to be negligible, and to scale the virtualized devices to manycores on the host, which itself scales poorly. To reduce the contention within each single container, we further propose SFS, which runs multiple file-system instances through the proposed virtualized storage devices, distributes all files under each directory among the underlying file-system instances, then stacks a unified namespace on top of them.

The evaluation of our prototype system built for Linux container (LXC) on a 32-core machine with both a RAM disk and a modern flash-based SSD demonstrates that MultiLanes scales much better than Linux in micro- and macro-benchmarks, bringing significant performance improvements, and that MultiLanes with SFS can further reduce the contention within each single container.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Fast storage, OS-level virtualization, manycores, scalability, performance isolation

ACM Reference Format:
Junbin Kang, Chunming Hu, Tianyu Wo, Ye Zhai, Benlong Zhang, and Jinpeng Huai. 2016. MultiLanes: Providing virtualized storage for OS-level virtualization on manycores. ACM Trans. Storage 12, 3, Article 12 (June 2016), 31 pages.
DOI: http://dx.doi.org/10.1145/2801155

This work is supported by China 973 Program (No.2014CB340300), China 863 Program (2013AA01A213), Projects from NSFC (No.61170294, 91118008) and Beijing Higher Education Young Elite Teacher Project.
An earlier version of this work was presented in Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST 2014) [Kang et al. 2014].
Authors' addresses: J. Kang, C. Hu (corresponding author), T. Wo (corresponding author), Y. Zhai, B. Zhang, and J. Huai, P.O.Box 7-28, Beihang University, 37 Xueyuan Road, Haidian, Beijing, P.R. China; emails: kangjb@act.buaa.edu.cn, hucm@act.buaa.edu.cn, woty@act.buaa.edu.cn, zhaiye@act.buaa.edu.cn, zblgeqian@gmail.com, huaijp@buaa.edu.cn.

1. INTRODUCTION

Operating system level virtualization (e.g., VServer [Soltesz et al. 2007], OpenVZ[1], Zap [Osman et al. 2002], and Linux Container[2] (LXC)) is often used for server consolidation in data centers, as it comes with significantly lower overhead than hypervisors [Soltesz et al. 2007; Osman et al. 2002]. Thus, both good isolation and high efficiency can be achieved by using OS-level virtualization when consolidating independent workloads in a single server [Soltesz et al. 2007]. Previous work on performance isolation mainly focuses on how to enforce the underlying hardware resource allocation policies among containers to minimize the performance interference [Soltesz et al. 2007; Banga et al. 1999; Bruno et al. 1998; Verghese et al. 1998].

The advent of nonvolatile memory technologies (e.g., NAND flash, phase change memories and memristors) poses challenges to traditional system software [Bjørling et al. 2013; Caulfield et al. 2012]. Specifically, fast storage devices built with these nonvolatile memories deliver low-access latency and a high degree of parallelism to applications [Caulfield et al. 2010; Seppanen et al. 2010; Chen et al. 2011]. However, the traditional storage systems often sacrifice these hardware advances due to the scalability bottlenecks encountered on manycores [Bjørling et al. 2013]. In particular, the sharing of the I/O stack between colocated containers could cause severe contention on shared kernel data structures and locks on manycores, as the legacy storage stack has a few scalability bottlenecks in the Virtual File System (VFS) [Boyd-Wickizer et al. 2010] and the underlying file systems. Consequently, the overall storage-system performance would be adversely affected by the contention among concurrently running containers performing I/O-intensive workloads on manycore platforms.

This article presents MultiLanes, a storage system for OS-level virtualization on manycores. MultiLanes eliminates contention on shared kernel data structures and locks between colocated containers by providing an isolated I/O stack for each container. Consequently, it effectively eliminates the interference between the containers and scales them well to manycores. The isolated I/O stack design consists of two components: the virtualized block device and the partitioned VFS.

The virtualized block device. MultiLanes creates a file-based virtualized block device for each container to run a guest-file system instance on top of it. This approach avoids contention on shared data structures within the file-system layer by providing an isolated file system stack for each container. The key challenges to the design of the virtualized block device are (1) how to tune the overhead induced by the virtualized block device to be negligible; (2) how to achieve good scalability with the number of the virtualized block devices on the host-file system, which itself scales poorly on many cores; and (3) how to support storage space overcommitment between colocated containers while ensuring crash consistency across the guest and host file systems.

We propose a set of techniques to address these challenges. First, MultiLanes adopts a synchronous bypass strategy to complete block I/O requests of the virtualized block device. In particular, MultiLanes translates a block I/O request from the guest file system into a list of block I/O requests of the host block device using block-mapping information obtained from the host file system. Then, the new requests will be directly delivered to the host device driver without the involvement of the host file system. Second, MultiLanes decouples the work of block-mapping processing from the block-request handling, then constrains the working threads interacting with the host file system for block mapping to a small set of cores to reduce the contention on the host [Cui et al. 2013]. It also adopts a prefetching mechanism to reduce the communication

[1]https://openvz.org/Main_Page.
[2]https://wiki.archlinux.org/index.php/Linux_Containers.

costs between the virtualized devices and the working threads. Third, MultiLanes proposes an efficient and scalable sparse file-handling mechanism to support storage space overcommitment while ensuring crash consistency.

Although we can run multiple guest file systems on multiple physical/logical devices for native performance, adopting files as the back-end storage for virtualization environments has advantages, such as easing the management of container images and supporting storage space overcommitment [Le et al. 2012].

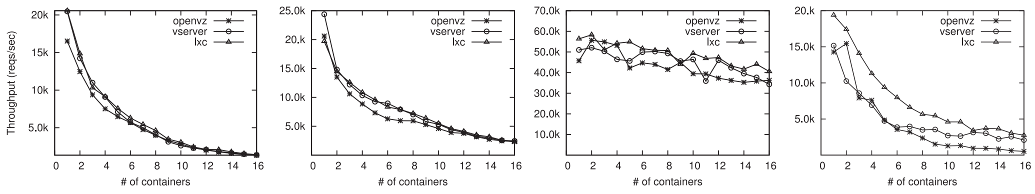
The partitioned VFS. The goal of the VFS layer in Linux is to provide and maintain a global and consistent in-memory file-system view for underlying file systems, thus speeding up application I/O access. However, the inevitable use of global data structures and the corresponding synchronization mechanisms in the VFS could cause severe scalability bottlenecks within the I/O stack on manycores. Rather than iteratively fixing or mitigating the scalability bottlenecks of the VFS [Boyd-Wickizer et al. 2010], MultiLanes, in turn, adopts a straightforward strategy that partitions the VFS data structures to completely eliminate contention between containers, as well as to achieve improved locality of the VFS data structures on manycores. The partitioned VFS is referred to as pVFS in the rest of the article.

The extension SFS. As described earlier, MultiLanes achieves scalability of OS-level virtualization on manycores by eliminating the contention between colocated containers. However, this approach does not address the scalability issues inside each single container. Hence, we further propose an extension SFS to reduce the contention within the file-system layer inside each single container. SFS is designed as a stacking file system, which runs multiple file-system instances, distributes files under each directory among the underlying file-system instances, then provides a unified namespace on top of them. We leverage the proposed virtualized block devices of MultiLanes to run multiple file-system instances and to support storage overcommitment among them. Note that SFS does not eliminate or mitigate the contention within the VFS layer inside each single container, as SFS actually lies under the pVFS as a scalable guest file system.

This article makes the following contributions: (1) We propose MultiLanes, a storage system for OS-level virtualization on manycores, which eliminates the contention between colocated containers by building an isolated stack for each container. (2) We propose a set of techniques to tune the storage-device virtualization overhead to be negligible, and to scale the virtualized devices to manycores on the host file system, which itself scales poorly. (3) We propose an extension SFS to provide better scalability for workloads inside each single container. (4) We have implemented a prototype of MultiLanes and SFS in Linux 3.8.2 kernel. (5) The evaluation of MultiLanes and SFS on a 32-core machine with both a RAM disk and a modern flash-based SSD shows that MultiLanes scales much better than Linux in micro-benchmarks and application-level benchmarks, bringing significant performance improvements, and that MultiLanes with SFS can effectively reduce the contention inside each single container.

This article significantly extends our preliminary work [Kang et al. 2014] in several aspects: (1) We proposed an extension SFS to reduce the contention inside each single container and evaluate its performance. (2) We described the details on how MultiLanes handles sparse files on the host to support storage overcommitment. (3) We constructed a set of experiments to comprehensively study the effects of the design choices in MultiLanes. (4) We evaluated the performance of MultiLanes on a present-day SSD and repeated the previous experiments on a 32-core machine.

The remainder of the article is organized as follows. Section 2 highlights the storage-stack bottlenecks in existing OS-level virtualization approaches for further motivation. We discuss related work in Section 3. Then, we present the design and implementation of MultiLanes in Sections 4 and 5, respectively. Section 6 presents the extension SFS.



(a) Ext3 (b) Ext4 (c) XFS (d) Btrfs

Fig. 1. Scalability evaluation. This figure shows the average throughput of each container performing sequential buffered writes on different file systems. We choose OpenVZ, Linux-VServer and LXC, which are based on Linux kernel 2.6.32, 3.7.10, and 3.8.2, respectively. These OS-level virtualization approaches use the shared host file system to store containers' data for high efficiency. The experiment is carried out on an Intel 16-core machine with a RAM disk.

Table I. The Top 3 Contended Locks

Ext3				**Ext4**			
#	lock	bounces	total wait time	#	lock	bounces	total wait time
1	zone->wait_table	5216k	36574	1	journal->j_list_lock	2085k	138146
2	journal->j_state_lock	1582k	56980	2	zone->wait_table	147k	384
3	journal->j_list_lock	382k	20804	3	journal->j_state_lock	46k	541
XFS				**Btrfs**			
#	lock	bounces	total wait time	#	lock	bounces	total wait time
1	zone->wait_table	22k	36	1	found->lock	778k	44325
2	rq->lock	7k	9	2	btrfs-log-02	388k	1125
3	key#3	5k	13	3	btrfs-log-01	230k	1050

Note: This table shows the contention bounces and total wait time of the top 3 contended locks when running 16 LXC containers with buffered writes. The total wait time is in microseconds.

Section 7 evaluates the performance and scalability of MultiLanes and MultiLanes with SFS with micro- and macro-benchmarks. We present our conclusions in Section 8.

2. MOTIVATION

In this section, we create a simple micro-benchmark to highlight the storage-stack bottlenecks of existing OS-level virtualization approaches on manycore platforms incorporating fast storage technologies. The benchmark performs 4KB sequential buffered writes to a 256MB file. We run the benchmark program inside each container in parallel and vary the number of containers.

Figure 1 shows the average throughput of containers running the benchmark on a variety of file systems (i.e., Ext3/4, XFS, and Btrfs). The results show that the throughput on all file systems except for XFS decreases dramatically with the increasing number of containers in the three OS-level virtualization environments (i.e., OpenVZ, VServer, and LXC).

The kernel lock usage statistics gathered by using *lock stat*[3] in Table I presents the contended locks, the lock data bounces among CPU caches, and the total lock wait time during the benchmarking, which results in decreased performance. XFS delivers much better scalability than the other three, as much less contention occurs for buffered writes. Nevertheless, it would also suffer from scalability bottlenecks under other workloads, which will be described in Section 7.

The poor scalability of the storage system is mainly caused by the use of global data and the corresponding protecting locks in the shared I/O stack, which could cause frequent remote CPU cache access and contention on shared cache lines on modern

[3]The lock stat tool. https://www.kernel.org/doc/Documentation/locking/lockstat.txt.

manycore processors when updating these shared data structures concurrently, thus degrading the overall performance [Boyd-Wickizer et al. 2008, 2010; David et al. 2013].

3. RELATED WORK

This section relates our work to the previous work on performance isolation, scalable operating systems, scalable I/O stacks, isolated I/O stacks, virtualized devices, and union file systems.

Performance Isolation. Most work on performance isolation mainly focuses on minimizing performance interference by enforcing the underlying hardware resource allocation policies between the colocated containers, such as VServer [Soltesz et al. 2007], Resource Containers [Banga et al. 1999], Eclipse [Bruno et al. 1998], Software Performance Units [Verghese et al. 1998], and Argon [Wachs et al. 2007]. However, these works do not focuses on the performance interference caused by the shared OS kernel data structures and locks.

In contrast, MultiLanes aims to eliminate the contention on shared kernel data structures and locks in the software to eliminate performance interference. Hence, our work is complementary to previous studies on performance isolation.

Scalable Operating Systems. Some research work proposes novel OS structures, such as clustered objects proposed by K42 [Appavoo et al. 2007] and Tornado [Gamsa et al. 1999], as well as novel OS abstractions proposed by Corey [Boyd-Wickizer et al. 2008], to eliminate the scalability bottlenecks on manycore processors. Some other research work tries to address the scalability bottlenecks of operating systems by running multiple OS instances on manycore processors through the multikernel model such as Hive [Chapin et al. 1995] and Barrelfish [Baumann et al. 2009] or through the virtualization approach such as Diso [Bugnion et al. 1997] and Cerberus [Song et al. 2011]. Similar to SFS, Cerberus [Song et al. 2011] provides a unified file system namespace on top of multiple virtual machines (VMs). However, their approach comes with the cost of hardware virtualization and inter-VM communication. Our work is influenced and inspired by the work mentioned earlier on scaling operating systems but focuses on the scalability of I/O stack on fast-storage devices.

Scalable I/O Stacks. Some work tries to address the scalability bottlenecks in other layers of the I/O stack. For example, Zheng et al. [2013] reduces the contention within the page cache layer by partitioning the global page cache into many fine-grained page sets and proposes a workaround that dedicates an I/O thread on each SSD to avoid the contention within the underlying file system. Bjørling et al. [2013] eliminates the global queue lock contention within the underlying driver layer by partitioning the global queue. However, these works do not focus on the scalability issues of the VFS layer and the on-disk file-system layer. Hence, our work aiming to scale the VFS layer and the on-disk file systems is complementary to these works. Some recent research work [Eqbal 2014; Clements et al. 2013; Gruenwald III 2014; Gruenwald III et al. 2015] mainly centers on designing in-memory file systems that can scale to many cores. However, our work focuses on addressing the scalability issues of on-disk file systems.

Boyd-Wickizer et al. [2010] and the Linux community (such as the work of Chinner [2011, 2013]) identified some contended locks in the VFS and made significant efforts to eliminate or reduce the contention within the VFS by using parallel programming techniques such as fine-grained locks [Cantrill and Bonwick 2008]. For example, the Linux community [Chinner 2011, 2013] also proposed to partition the *inode_sb_list_lock* and *dcache_lru_lock* into fine-grained, per-superblock locks to alleviate the contention within the VFS. However, our work addresses the VFS scalability bottlenecks for OS-level virtualization in a thorough way: adopting these parallel programming techniques [Cantrill and Bonwick 2008] to partition almost all the global VFS data structures and locks between containers to completely eliminate the contention within the VFS layer, as well as to improve the locality of the VFS data structures, such as dentry cache.

Cui et al. [2013] experimentally identified some hot kernel locks in the VFS layer and proposed an approach that constrains the contended threads to a small set of cores to mitigate contention. Our work scales virtualized block devices to manycores by adopting their idea: MultiLanes statically constrains the translation threads interacting with the host to a small set of cores to reduce contention on the host. However, their work cannot fully utilize the computing capacity of manycores under high contention, while our work builds isolated I/O stacks to eliminate contention and only constrains the translation threads to a small set of cores.

Our recent work on SpanFS [Kang et al. 2015] identified and analyzed the lock bottlenecks in existing file systems, and proposed a novel journaling file system to provide scalable I/O performance on fast-storage devices. SpanFS distributes all files and directories among the isolated I/O stacks called domains, builds a global file system view on top of the domains, and provides crash consistency across the domains. SFS is similar to SpanFS in that they both adopt a decentralized file system architecture to achieve scalability and both maintain crash consistency on such decentralized architecture. However, MultiLanes and its extension SFS provide scalable I/O performance in a compatible way: running unmodified guest file systems through the proposed virtualized storage device layer and a stacking file system layer. Consequently, MultiLanes and its extension provide flexibility and reliability to users: users can choose different modern file systems. Moreover, SpanFS does not address the scalability issues within the VFS layer.

Isolated I/O stacks. Vanguard [Sfakianakis et al. 2014] and its relative Jericho [Mavridis et al. 2014] build isolated I/O stacks from scratch for performance isolation. Similar to our extension SFS, Vanguard [Sfakianakis et al. 2014] and Jericho [Mavridis et al. 2014] propose a file system that distributes the top-level directories to the isolated I/O stacks in a round-robin way to eliminate the contention between independent workloads. However, SFS distributes all files under each directory across underlying file-system instances to provide scalable performance and maintains global crash consistency. IceFS [Lu et al. 2014] provides isolated I/O stacks called cubes within a file system for fault isolation and performance isolation, and dedicates a running transaction to each cube, thus allowing parallel transaction commits. However, these works (i.e., Vanguard, Jericho, and IceFS) build isolated I/O stacks from the ground up, while our work proposes the virtualized block device driver to run multiple unmodified existing file systems for performance isolation, thus bringing flexibility to users while inheriting the reliability of existing file systems. Moreover, these works do not address the scalability issues within the VFS layer and within each single container.

Virtualized Devices. MultiLanes maps a regular file as the virtualized device of a container to run guest file systems on the host. Like device emulation [Sugerman et al. 2001] and para-virtualization [Russell 2008], the use of back-end files in MultiLanes also eases the management of the storage images [Le et al. 2012]. However, our virtualized block-device approach is more efficient when compared to device emulation and para-virtualization, as it comes with little overhead by adopting a bypass strategy. Compared to direct device assignment [Gordon et al. 2012], our approach for OS-level virtualization does not need to bear the hardware virtualization cost.

The sparse file-handling process of MultiLanes is similar to that of the Linux loop device driver, which also leverages the flush request semantics to persist the back-end file. However, they differ in several aspects. First, MultiLanes offloads the synchronous request processing to the translation threads to reduce the contention on the host. Second, the Linux loop driver syncs both the dirty data and metadata of the back-end file to disk upon each flush request. In contrast, MultiLanes only needs to persist the dirty metadata of the back-end file as the data I/O requests are delivered to the host driver directly. Moreover, the adoption of the prefetching mechanism in MultiLanes as well as the use of a dirty flag to avoid redundant flushes can reduce the synchronization overhead.

To address the performance interference caused by the shared journal space among multiple containers, the OpenVZ ploop project provides a virtualized block device to run a guest file system inside a container, and adopts a direct I/O mechanism to eliminate double data caching [Patlasov 2011; Kolyshkin 2012].[4] The synchronous bypass strategy proposed in our work is similar to the direct I/O mechanism of the OpenVZ ploop.

However, as the OpenVZ ploop does not aim at addressing the scalability bottlenecks on manycores, there are many important differences between our work and the OpenVZ ploop, as follows: (1) The OpenVZ ploop does not address the VFS scalability issues, while our work proposes the pVFS abstraction to eliminate the VFS lock contention between containers. (2) The OpenVZ ploop does not focus on the poor scalability of the host file system, which can limit scaling virtualized block devices to manycores. In contrast, our work constrains all the translation threads to a small set of cores to reduce the contention on the host and adopts a prefetching mechanism to reduce the communication cost. We also have demonstrated the effectiveness of such design choices in achieving scalability in Section 7.3. (3) The OpenVZ ploop only supports direct I/O on the Ext4 file system, while our synchronous bypass strategy is almost transparent to the host file system, hence is more general.[5] (4) The OpenVZ ploop cannot reduce contention within each single container, while we further propose SFS to achieve scalability inside each container. (5) A mapping table such as Qcow2 is adopted in the OpenVZ ploop to support storage overcommitment, which requires extra disk I/Os to the mapping table for each write involving block allocation. Our work creates sparse back-end files for containers to support storage overcommitment and proposes an efficient and scalable sparse file-handling mechanism to ensure crash consistency across the guest and host file systems, achieving near-native performance.[6]

Union File Systems. Union file systems, such as Unionfs [Wright et al. 2006] and Union mount [Pendry and McKusick 1995], provide applications with a unified namespace on top of multiple directory branches. SFS is similar to Unionfs [Wright et al. 2006] in many ways, especially in the aspect of the namespace-unifying mechanism. For example, the SFS lookup operation is similar to that of Unionfs in that they both look up a directory object across all the underlying replicated parent directories to get all of the object's replicated ones and both return the lookup result immediately once the found object is identified as a file.

However, SFS differs from Unionfs [Wright et al. 2006] in the following aspects: (1) As it does not aim at addressing scalability issues, Unionfs does not care whether the underlying branches reside in different file-system instances. Although Unionfs can run multiple file-system instances by using partitions, then distributes files among them, it would introduce management costs to adjust the storage space among the file systems on demand and would even create crash inconsistencies when performing online adjusting. In contrast, SFS leverages the proposed virtualized block device driver of

[4]The OpenVZ ploop project. https://openvz.org/Ploop/Why.

[5]Only minor code modification is needed to export the host file system mapping routine interface.

[6]Similar to the approach proposed for write-back flash cache [Qin et al. 2014], MultiLanes leverages the flush request semantics to flush the dirty metadata of the sparse back-end file to ensure the upper layer ordering and durability requirements. The adoption of the prefetching mechanism as well as the use of a dirty flag to avoid redundant flushes can reduce the times of flushes, mostly eliminating the synchronization overhead. Moreover, MultiLanes offloads flush request handling to the translation threads to reduce the contention on the host.

MultiLanes to run multiple file-system instances for scalability and to support storage overcommitment. (2) Although the SFS rename operation handling process is similar to that of Unionfs, Wright et al. [2006] use different approaches to provide atomicity of the file-rename operation. Unionfs adopts an approach creating a provisional file recording the rename state to ensure the atomicity of the rename operation, which needs to recover the file system after a crash by using their high-level *fsck*. However, their *fsck* process may be time-consuming, as it needs to scan all the directories to detect inconsistencies before mounting. Moreover, Wright et al. [2006] do not explicitly describe how to recover the file system after a crash. In contrast, we propose a soft-link technique to repair crash inconsistencies online by checking the extended attribute of each accessed file. (3) Although Unionfs uses an artificial directory offset similar to the SFS directory offset for *readdir*, it can only support seeking to the start offset or the last read offset before *readdir*. In contrast, SFS uses the SFS directory offset, which can be mapped to the underlying directory offset by leveraging the VFS *readdir* callback mechanism to seek to the specified directory offset. Hence, SFS can seek to any valid directory offset before *readdir*, and is more general.

4. MULTILANES DESIGN

MultiLanes is a storage system for OS-level virtualization that addresses the I/O performance interference between colocated containers on manycores. In this section, we present the design goals, architecture, and components of MultiLanes.

4.1. Design Goals

Most existing OS-level virtualization approaches simply leverage namespace isolation mechanisms to realize file-system virtualization [Soltesz et al. 2007; Osman et al. 2002]. The containers colocated share the same I/O stack, which not only leads to severe performance interference between them but also suppresses flexibility.

MultiLanes is designed to eliminate storage-system interference between containers to provide good scalability on manycores. We aim to meet three design goals: (1) it should support various existing file systems and be transparent to container applications; (2) it should achieve good scalability with the number of containers on the host; and (3) it should minimize the virtualization overhead on fast-storage devices to offer near-native performance.

4.2. Architectural Overview

MultiLanes is composed of two key design modules: the virtualized storage device and the pVFS. Figure 2 illustrates the architecture and the overall primary components of MultiLanes. At the top of the architecture, we host multiple containers, each of which accesses its guest file system through the pVFS. The pVFS offers a private kernel abstraction to each container to eliminate contention within the VFS layer. Under each pVFS, there lies the specific guest file system of the container. The pVFS remains transparent to the upper-layer applications and the underlying guest file system by offering the same interfaces as the VFS.

Between the guest file system and the host are the virtualized block device and the corresponding block device driver. MultiLanes maps regular files in the host file system as virtualized storage devices for containers to run multiple guest file systems. This storage-virtualization approach not only eliminates performance interference between containers in the file system layer, but also allows each container to run its own file system. In order to avoid most of the virtualization overhead and to scale the virtualized devices to manycores, MultiLanes adopts a synchronous bypass strategy to complete the I/O requests from the guest file system, and constrains the working threads

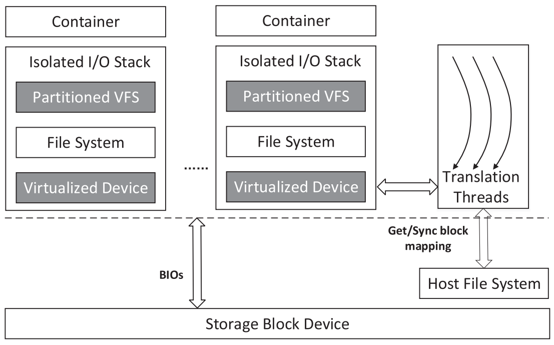


Fig. 2. MultiLanes architecture.

interacting with the host file system for block-mapping information to a small set of cores to reduce the contention on the host [Cui et al. 2013].

4.3. Design Components

MultiLanes provides an isolated I/O stack to each container to eliminate performance interference between containers, which consists of the virtualized storage device, the virtualized block device driver, and the partitioned VFS.

4.3.1. Virtualized Storage Device. For I/O efficiency, colocated containers usually use the shared host file system for storing data [Soltesz et al. 2007], which could cause significant storage-system performance interference within the shared I/O stack on manycores, as shown in Section 2.

In order to eliminate storage-system performance interference between containers on manycores, we run a separated file system instance for each container on top of a lightweight virtualized-storage device which is mapped as a back-end sparse file on the host file system. Rather than preallocating all the blocks to the back-end files, the host file system assigns physical blocks to each back-end file on-demand, thus allowing storage space overcommitment among colocated containers. Though using storage virtualization to avoid performance interference among colocated containers is intuitive, the challenge to MultiLanes is how to minimize the storage-system virtualization overhead. We propose a set of techniques to achieve both scalability and near-native performance.

4.3.2. Driver Model. Like any other device virtualization approaches, the key role of the virtualized block device driver in MultiLanes is to virtualize multiple storage block devices for containers to run guest file systems. As shown in Figure 2, each virtualized block-device driver receives block I/O requests from the guest file system and maps them to requests of the host block device.

A block I/O request (*bio*) delivered by the Linux block I/O layer specifies a data transmission requirement between the contiguous block-device sector region and a set of individual memory segments. For the virtualized block device of MultiLanes, the sector region specified in the request is actually composed of one or more contiguous logical blocks of the back-end file. The virtualized driver adopts a synchronous bypass strategy to complete the block I/O requests from the guest file system, which translates logical blocks of the back-end file to physical blocks on the host, then maps each I/O request to the requests of the host block device according to the translation. The virtualized driver is composed of three major components: the block translation, block handling, and sparse file handling.

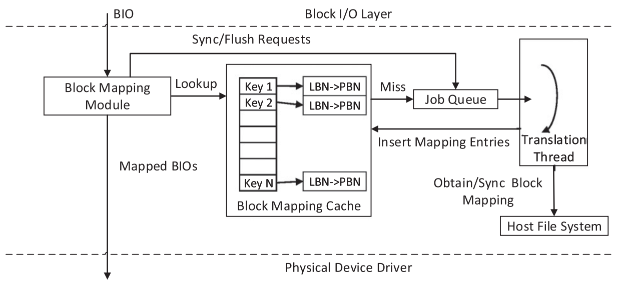


Fig. 3. Driver structure. This figure presents the structure of the virtualized storage driver, which comprises the block translation unit, the request handling unit, and the sparse file handling unit.

Block Translation. For each virtualized storage device, the host file system dynamically allocates physical blocks to the logical blocks of its back-end file, maintains the block mapping information and provides a block mapping interface, which can be invoked to obtain the block mapping information for a specified logical block. If the logical block has not been assigned physical block, the mapping process involves the host file system block allocation. The mapping interface usually supports allocating a continuous physical block region consisting of a specified number of blocks for the requested logical block region of a file at a time. When the corresponding physical blocks are allocated continuously on the host block device, the mapping interface also can support mapping a region of continuous logical blocks of a file at a time. MultiLanes achieves block translation with the help of this mapping interface. The block mapping prefetching mechanism invokes this block mapping interface to allocate and map a continuous back-end file region larger than needed at a time for efficiency.

As shown in Figure 3, the block translation unit of each virtualized block device consists of a cache table, a job queue and a translation thread. The cache table maintains the mapping between logical blocks of the back-end file and physical blocks. The virtualized driver will first look up the table with the logical block number of the back-end file for block translation when a container thread submits an I/O request to it. Note that the driver actually executes in the context of the container thread as we adopt a synchronous model of I/O request processing. If the target block is hit in the cache table, the driver directly gets the target mapping physical block number. Otherwise, it starts a cache miss event, then puts the container thread to sleep. A cache miss event delivers a translation job to the job queue and wakes up the translation thread. The translation thread then invokes the interface of the mapping routine exported by the host file system to get the target physical block number, inserts a new mapping entry in the cache table, and wakes up the container thread at last. The cache table is initialized as empty when the virtualized device is mounted.

The block mapping cache adopts a hash table to store the block mapping entries for fast lookup and uses per-hash-bucket locks to protect the hash table. Meanwhile, as we use a certain amount of memory for the block-mapping cache, the new mapping entry is also added to a FIFO list for mapping cache-entry replacement. As the block-mapping entries are inserted to the block-mapping cache by the single translation thread, there is no contention when maintaining the global FIFO list.

Block translation will be extremely inefficient if the translation thread is woken up to map only a single cache miss block every time. The driver will suffer from frequent block-mapping cache misses and thread context switches, which would cause considerable communication overhead. Hence, we adopt a prefetching approach similar to that of handling CPU cache misses. The translation thread maps a continuous logical block region starting from the missed logical block for each request in the job queue at a time.

On the other hand, as the block allocation of the host file system usually involves file system journaling, the mapping process may cause severe contention within the host on manycores when cache misses of multiple virtualized drivers occur concurrently, thus scaling poorly with the number of virtualized devices on manycores. We address this issue by statically constraining all translation threads to work on a small set of cores to reduce contention on the host file system [Cui et al. 2013]. Our current prototype statically binds all translation threads to a set of cores inside a processor, due to the observation that constraining data sharing within a processor leads to improved performance [Boyd-Wickizer et al. 2008; David et al. 2013].

As the block-mapping process is not a CPU-intensive job, constraining all the translation threads to an appropriate number of cores could suffice the block mapping requests from multiple virtualized block devices. As shown in Section 7, on our experimental machine with 32 cores, binding all the translation threads to 4 cores could achieve much better scalability than the baseline. As demonstrated in Section 7.3.2, binding all the translation threads to 16 cores (crossing two processors) also could effectively reduce the contention on the host under 16KB random direct writes. However, as the core count grows, there will be a trade-off between the computing capacity requirement for handling block-mapping requests and the contention on the host file system. Hence, users may need to experimentally handle this trade-off on the specific large manycore machine to achieve scalability.

Request Handling. Since the continuous data region of the back-end file may not be continuous on the host block device, a single-block I/O request of the virtualized block device may be remapped to several new requests according to the continuity of the requested blocks on the host block device.

The block I/O request received by the virtualized driver specifies a data transmission requirement between a set of memory segments and a contiguous logical block region of the back-end file The block size is by default the same as the page size in the host file system. Under such default configuration, each segment of a block I/O request usually contains one mapping between one memory segment and one logical block. For special cases that one segment of a bio consists of two partial logical blocks, MultiLanes splits such a segment into two segments, each corresponding to only one logical block. For other configurations that the page size is multiple times of the block size of the host file system, MultiLanes splits each segment into multiple segments, each involving one mapping between one memory segment and one logical block of the back-end file. Then, for each segment of the block I/O request, the virtualized device driver first calculates the logical block number of it, then translates the logical block number to the physical block number with the support of the block translation unit. After all the segments of a request are remapped, MultiLanes checks the contiguity of their corresponding physical blocks on the host block device. The virtualized device driver combines the segments that are contiguous on the host block device as a whole and allocates a new block I/O request of the host block device for them. Then, it creates an individual block I/O request for each of the remaining segments. Thus, a single block I/O request of the virtualized block device has been remapped to a set of block I/O requests of the host block device. Figure 4 illustrates such an example, which will be described in Section 5.1.

A new block I/O request is referred to as a slice of the original request. We organize the slices in a doubly-linked list and allocate a head to keep track of them. When the list is prepared, each slice would be submitted to the host block device driver in sequence. The host driver will handle the data transmission requirements of each slice in the same manner, with regular I/O requests.

I/O completion should be carefully handled for the virtualized device driver. As the original request is split into several slices, the host block device driver will initiate a completion procedure for each slice. The original request, however, should not be terminated until all the slices have been finished. Hence, we offer an I/O completion callback, in which we keep track of the finished slices, to the host driver to invoke when it tries to terminate each slice. The host driver will terminate the original block I/O request of the virtualized block device driver only when it determines that it has completed the last slice.

Thus, a block I/O request of MultiLanes is remapped to multiple slices of the host block device, and is completed by the host device driver. The most important feature of the virtualized driver is that it stays transparent to the guest file system and the host block device driver, and requires only minor modification to the host file system to export the mapping routine interface.

Sparse File Handling. MultiLanes creates sparse back-end files on the host file system for containers to support storage overcommitment among containers. As the data blocks of the sparse back-end file are allocated dynamically in the host file system, one challenge to MultiLanes is to maintain crash consistency across the guest and the host file system.

For example, appending data to files in the guest file system usually involves the block allocation in both the guest file system and the host file system. The guest file system allocates data blocks (logical blocks of the back-end file) by modifying allocation structures such as the block bitmap in Ext3/4, then writes data to the blocks. Then, the virtualized device driver would involve the block allocation on the host file system if the corresponding physical blocks are not allocated. For durability, the guest file system would persist the dirty data and metadata to the virtualized device upon synchronous requests such as *fsync()* system calls and periodic journaling commit actions. Inconsistencies may occur in the case of a system crash if the logical block allocation information is persisted while the corresponding physical block allocation has not yet been flushed to disk. To ensure consistency across system crashes, MultiLanes should persist the block allocation in the host file system in response to the synchronous actions in the guest file system.

The guest file system usually issues block I/O requests marked with *REQ_FLUSH* and/or *REQ_FUA* flags to explicitly flush the storage device cache for ordering and durability [Qin et al. 2014; Chidambaram et al. 2013].[7] MultiLanes leverages these flush request semantics to persist the back-end file to ensure the upper-layer ordering and durability requirements, which is similar to the approach proposed in Qin et al. [2014] for write-back flash cache. Specifically, MultiLanes carefully handles the block I/O requests marked with these flags to maintain crash consistency across the guest and host file systems. For each block I/O request marked with *REQ_FLUSH* flag, the virtualized driver will deliver a synchronous request to the translation thread and wait for the completion before it starts to handle the block I/O request. Upon receiving a synchronous request, the translation thread then will invoke *vfs_fsync()* on the back-end file to sync its dirty metadata to the underlying durable storage. For each I/O request marked with a *REQ_FUA* flag, the virtualized driver will deliver a synchronous request to the translation thread after it has processed the block I/O request. For the empty block I/O requests marked with *REQ_FLUSH* and/or *REQ_FUA* flags, the virtualized driver also delivers a synchronous request to the translation thread to persist the back-end file, then submits a new empty block I/O request marked with these flags to the underlying device driver.

[7]Device Cache Flush Requests. https://www.kernel.org/doc/Documentation/block/writeback_cache_control.txt.

Table II. Contended VFS Locks

#	Contended VFS Locks	Description
1	inode_hash_lock	A spin lock used to protect the VFS inode hash table against concurrent inode inserts, removals and lookups to the table.
2	dcache_lru_lock	A spin lock used to serialize concurrent access to the LRU lists of unused dentries, which are used to facilitate the unused dentry reclamation under memory pressure.
3	inode_sb_list_lock	A spin lock used to serialize concurrent inode inserts or deletions to the inode lists of all file-system instances.
4	rename_lock	A sequence lock used to protect the dentry hash table when performing lookups and renames over the hash table.

Note: The table shows the contended locks in VFS when running the metadata-intensive workload CRU in Linux kernel 3.8.2.

In order to avoid redundant synchronous operations, we use a flag called *F_DIRTY* for each back-end file to record its state. The translation thread will set this flag when invoking the block-mapping routine of the host file system for write requests. The flag will be cleared after the translation thread invokes *vfs_fsync()* on the back-end file. If the *F_DIRTY* flag is set, the virtualized driver would deliver a synchronous request to the translation thread for each I/O request marked with *REQ_FLUSH* and/or *REQ_FUA* flags. If not set, the virtualized driver does not need to synchronize the back-end file. As the prefetching mechanism adopted in MultiLanes allocates and maps data blocks of the back-end file on the host in a batching way, the use of the recording flag can effectively reduce the times of synchronous operations on the back-end file, thus tuning the overhead to be negligible.

4.3.3. Partitioned VFS. Although MultiLanes allows each container to run its own guest file system independently, there still exists performance interference within the VFS layer. Table II shows the top 4 contended VFS locks collected by using the *lock stat* tool when conducting the metadata-intensive microbenchmark *CRU*, described in Section 7. Hence, we propose the pVFS that provides a private VFS abstraction to each container, eliminating the contention for shared data structures within the VFS layer between containers.

In order to speed up the file system namespace access, VFS adopts a global in-memory dentry cache and inode cache, and uses the corresponding locks shown in Table II to protect their integrity against concurrent access. Specifically, VFS organizes the dentry cache and the inode cache as a dentry hash table and an inode hash table for fast lookup. The inode hash table is accessed under the protection of the *inode_hash_lock* while the dentry hash table employs per-bucket locks cooperating with the RCU lock [McKenney et al. 2001] to maximize its access concurrency. As the rename operation may update two dentry hash buckets, it may cause conflicts with simultaneous lookup operations. A sequence rename lock is adopted to avoid such conflicts.

However, the use of the global *rename_lock* will serialize concurrent renames to the dentry hash table, introducing scalability bottlenecks. VFS also links all the inodes and unused dentries to the per-superblock inode lists and LRU dentry lists, and uses global locks (*inode_sb_list_lock* and *dcache_lru_lock*) to protect them, which will introduce severe contention under concurrent access.

Rather than iteratively fixing or mitigating the lock bottlenecks in the VFS, as in Boyd-Wickizer et al. [2010], we adopt a straightforward approach that partitions the VFS data structures and corresponding locks to eliminate contention, as well as to improve the locality of the VFS data structures such as dentry cache on manycores. In particular, MultiLanes allocates an inode hash table and a dentry hash table for each container to eliminate the performance interference within the VFS layer. Along

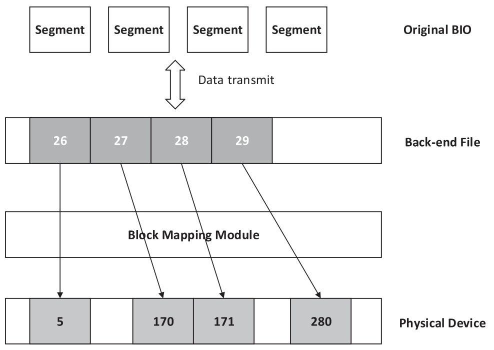


Fig. 4. Request mapping. This figure shows the mapping from a single block I/O request of the virtualized block device to a request list on the host block device.

with the separation of the two hash tables from each other, *inode_hash_lock* and *rename_lock* are also separated. MultiLanes allocates separated *inode_sb_list_lock* and *dcache_lru_lock* instances for each container guest file system.

The pVFS is an important complementary part of the isolated I/O stack.

5. MULTILANES IMPLEMENTATION

We choose to implement the prototype of MultiLanes for LXC. We implemented MultiLanes in the Linux 3.8.2 kernel, which consists of a virtualized block-device driver module and adaptations to the VFS.

5.1. Driver Implementation

We implemented the virtualized block-device driver of MultiLanes based on the Linux loop-device driver.

The virtualized device driver of MultiLanes adopts a synchronous bypass strategy to minimize the virtualization overhead. In the routine *make_request_fn*, which is the standard interface for delivering block I/O requests, our driver finishes request mapping and redirects the slices to the host driver via the standard *submit_bio* interface.

When a virtualized block device is mounted, MultiLanes creates a translation thread for it. We export the block mapping interface, such as *ext3_get_block()*, into the *inode_operations* structure so that the translation thread can invoke it for block mapping via the inode of the back-end file.

The *multilanes_bio_end* function is implemented for I/O completion notification, which will be called each time the host block-device driver completes a slice. We store global information, such as the total slice number, finished slice count, and error flags in the list head, and update the statistics every time it is called. The original request will be terminated by the host driver by calling the *bi_end_io* method of the original *bio* when the last slice is completed.

Figure 4 shows an example of block request mapping. We assume the page size and the block size of the back-end file on the host file system are both 4KB. As shown in the figure, a block I/O request delivered to the virtualized driver consists of four segments. The start sector of the request is 208 and the block I/O size is 16KB. The *bio* contains four individual memory segments, which lie in four physical pages. After all the segments of the request are mapped by the block translation unit, we can see that only the logical block 27 and 28 are contiguous on the host. MultiLanes allocates a new *bio* structure for the two contiguous blocks and two new ones for the remaining two blocks, then delivers the new *bio*s to the host driver in sequence.

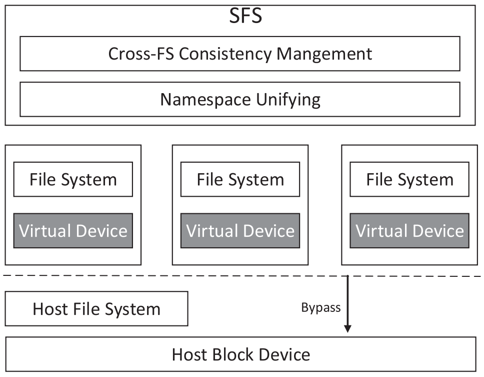


Fig. 5. SFS architecture. The underlying file system instance runs on the virtual device created by the virtualized device driver of MultiLanes.

5.2. pVFS Implementation

The pVFS data structures and locks are organized in the super block of each guest file system. We allocate SLAB caches *ihtable_cachep* and *dhtable_cachep* for inode and dentry hash table allocation when initializing the VFS at boot time. MultiLanes adds the *dentry_hashtable* pointer, the *inode_hashtable* pointer, and the corresponding locks (i.e., *inode_hash_lock*, *rename_lock*, *dcache_lru_lock* and *inode_sb_list_lock*) to the super block. We add a flag field to the *superblock* structure to distinguish guest file systems on virtualized storage devices from other host file systems. For each guest file system, MultiLanes will allocate a dentry hash table and an inode hash table from the corresponding SLAB cache when the virtualized block device is mounted, both of which are predefined to have 65536 buckets.

Then, we modify the kernel control flows that access the hash tables, lists, and corresponding locks to allow each container to access its private VFS abstraction. Accesses to each guest file system are redirected to its private VFS data structures and locks, while other accesses keep going through the original VFS.

6. THE EXTENSION SFS

SFS is designed to provide better scalability for workloads inside each single container by running multiple underlying file-system instances to reduce the contention within the file system layer. This section first gives an overview of the architecture, then describes the design and implementation details of SFS.

6.1. Architectural Overview of SFS

Figure 5 illustrates the architecture of SFS: inside each container, multiple underlying file-system instances are created through the proposed virtualized device driver of MultiLanes. SFS is designed as a stacking file system, which manages multiple underlying file-system instances and stacks a unified namespace on top of them, similar to Unionfs [Wright et al. 2006].

The key design goals of SFS are to (1) distribute all files under each directory among the underlying file-system instances, (2) provide a unified file-system namespace on top of the multiple underlying file-system instances, and (3) ensure global consistency in the face of a system crash.

6.2. Object Distribution

SFS distributes all files under each directory across the underlying file systems to reduce the contention within the file-system layer. Specifically, when creating a new object (i.e., file or directory) under a directory, SFS selects an underlying file system through a local round-robin counter for the directory, then creates the object on the underlying file system. When the parent directory does not exist in that underlying file system, SFS creates the missed ancestor directories along the file path on it, which is similar to Unionfs [Wright et al. 2006]. We also call this process *directory replication*. It should be noted that the newly replicated directories are empty directories. Consequently, for each SFS directory, the underlying directory may be replicated across the underlying file systems by directory replication when needed, similar to Unionfs [Wright et al. 2006]. However, each SFS has only one underlying file object. As the objects are distributed or replicated across the underlying file systems, when looking up a file or reading a directory, namespace unifying needs to be performed.

6.3. Unified File System Namespace

The namespace unifying mechanism of SFS is similar to that of Unionfs [Wright et al. 2006], which mainly involves two SFS operations: *lookup* and *readdir*.

Lookup. When looking up an object under an SFS directory, the SFS *lookup* operation executes lookups across the underlying replicated parent directories in sequence. For an SFS file, once the underlying file is found under one replicated directory, the SFS *lookup* operation returns the result immediately since each SFS file has only one underlying file object. For an SFS directory, SFS should get all of its replicated ones before returning.

Readdir. To support SFS directory offset seeking before *readdir*, SFS provides applications with the SFS directory offset, which can be mapped to the underlying directory offset within any underlying replicated directory. SFS builds a flat and linear SFS directory offset on top of the unified directory entries across multiple underlying replicated directories, which can be logically decomposed into the interdirectory offset and the intradirectory offset. To seek to a specified SFS directory offset, SFS first traverses the underlying directories in sequence and subtracts the underlying directory size from the SFS directory offset. SFS stops the traversal when the subtracted SFS directory offset starts to be lower than the next underlying directory size.

Then, the next underlying directory is the target directory where the directory entry that SFS directory offset points to stays.

Then the sum of the traversed underlying directories' size is the interdirectory offset and intradirectory offset is equal to the remaining SFS directory offset.

The VFS readdir callback mechanism provides a unified architecture for underlying file systems to fill directory entries into the user-space buffer, in which a VFS callback function is passed to the underlying readdir for directory entry filling and could terminate the underlying readdir on some condition.

SFS leverages the VFS readdir callback architecture to transparently locate the directory entry SFS intradirectory offset points to within the target underlying directory by implementing our specific seek callback function which counts the size of each accessed directory entry each time it is invoked by the underlying readdir for directory entry filling and will terminate the underlying readdir when the sum of the size of each accessed directory entry increases to the intradirectory offset. During this process, although the underlying directory offset will be increased in some way that is opaque to SFS by the underlying *readdir*, it would at last point to the same directory entry to which the SFS intradirectory offset points. Then, SFS can read directory entries starting from the specified SFS directory offset. SFS also records the SFS directory offset where the last directory read operation ended to avoid unnecessary seeking for subsequent directory reading.

When reading one underlying directory by calling the underlying *readdir* operation, the SFS *readdir* callback function will fill the underlying directory entry into the user-space buffer and then increment the intradirectory offset by the read directory entry size each time it is invoked by the underlying *readdir* operation for directory entry filling similar to the way most modern file systems increase their directory offset during *readdir*. When crossing the boundary of the underlying directories SFS invokes the underlying seek operation on the next underlying directory to seek to its start offset (zero), reinitializes the intradirectory offset as the start offset and then increases the interdirectory offset by the size of the underlying directory that has been read.

6.4. Consistency

Since each underlying file system can provide crash consistency through journaling or copy-on-write, the most critical challenge to SFS is how to ensure global crash consistency across multiple underlying file systems. For simplicity, each underlying file-system instance managed by SFS is also called the partition in the rest of the article.

Crash inconsistencies may occur only when performing the operations which need to span multiple partitions. Such operations in SFS include *rmdir* and *rename*. The SFS *rmdir* operation performs *rmdir* across the replicated directories, which is similar to Unionfs [Wright et al. 2006] except that SFS does not create whiteout and does not check whether all the underlying directories are empty, as the underlying file system would perform such a check upon deleting a directory. If there are objects under the underlying directory, the underlying file system would not delete the directory; instead, it will notify SFS of an error. Then, SFS will terminate the *rmdir*. Hence, if nonempty directories exist, the SFS *rmdir* operation will stop at the first nonempty replicated directory. If all the replicated directories are empty, SFS will successfully delete the underlying directories. If a system crash occurs during the SFS *rmdir*, it would not cause inconsistencies among the partitions, as the nonempty underlying directories always remain untouched during *rmdir*. Hence, we need to pay attention only to the SFS *rename* operation.

The file *rename* operation renames the source file to the destination file path that may contain an existing destination file. For the situations in which the destination file does not exist or both the source and destination file reside in the same underlying file system, SFS invokes the underlying *rename* operation to rename the source file. As the underlying operation is performed within the single partition, the underlying file system can guarantee the atomicity of the *rename* operation. Otherwise, SFS should rename the source file to the destination in the source partition, then remove the destination file in another partition. This process cannot guarantee the atomicity of the *rename* operation, as two partitions are involved.

We propose a soft-link technique to achieve the atomicity of the *rename* operation in this situation. In particular, before starting the rename process, SFS adds soft links to the destination and source file. Then, SFS can leverage the added soft links to recover the file system into a consistent state after a system crash. The *hidden* soft link added to the destination file is actually a pointer that points to the source partition, indicating that the destination file should be removed if the source file is found to have been moved to the destination in the source partition. The *stale* soft link added to the source file indicates that there may exist one stale destination file in the pointed partition that fails to be removed during the last uncompleted *rename* operation.

After having added the soft links persistently, SFS can safely process the rename operation. Upon completion of the *rename* operation, SFS flushes the modifications involved during the *rename* process to persistent storage. In particular, SFS will invoke the *fsync* operation on the source parent directory and the destination parent directory. Then, SFS removes the soft link added to the source file and persists the modifications to the storage device by invoking the *fsync* operation on the source file. Hence, it should be noted that the SFS *rename* operation comes at the cost of forcing the modifications to be persisted on the device.

After system crashes, SFS could recover the file system into a consistent state online by validating each accessed soft-linked file during the *lookup* operation. For each soft-linked file, SFS will check whether there exists a file under the same file path in the partition to which the soft link points.

For the hidden soft link, if the source file exists, the soft-linked destination file will be removed on behalf of the last uncompleted rename operation; then, SFS will remove the soft link added to the source file. Otherwise, SFS removes the soft link added to the destination file. In both cases, SFS should flush the modification to the storage device after each update operation. For the stale soft link, SFS will remove the stale destination file if it exists in the pointed partition and persist the deletion operation to the storage device. Then, the stale soft link can be removed safely and the modification is persisted to the storage device at last. Otherwise, SFS removes the soft link added to the source file and flushes the modification to the storage device.

Most modern file systems support adding extended attributes (i.e., *xattr*) to each file. SFS uses the *xattr* mechanism to support the proposed soft-link technique. In particular, SFS leverages *xattr* to add soft links in the form of <*key*, *value*> to a file in the underlying file system.

To rename a directory, SFS performs the *rename* operation across its replicated directories. If the target SFS directory exists, SFS first tries to remove the target underlying directories before performing *rename*. If a nonempty target underlying directory exists, the SFS directory rename operation will terminate. The current implementation of SFS does not provide atomicity of the directory *rename* operation.

6.5. Implementation Details

SFS is implemented in Linux 3.8.2 as a kernel module. We implemented SFS by using the stacking file-system technique [Zadok et al. 1999; Wright et al. 2006]. The implementation details of SFS are described here.

Lookup. During the *sfs_lookup()* operation, *sfs_lookup()* will check whether the found object is a soft-linked file by reading its extended attributes. If so, it will recover the file system into a consistent state using the soft links as described in Section 6.4. Hence, the most important feature of the soft-link technique is that SFS performs online checking and recovery. Meanwhile, the overhead of checking the extended attribute of each file is relatively small, as SFS only needs to perform a simple VFS operation *vfs_getxattr()* on the file. As the extended attribute values of the soft links used in SFS are a char string containing two integers, they usually would be filled into the on-disk inode by the underlying modern file systems, such as Ext4[8] and XFS[9]. Since the underlying *lookup* operation has already read the on-disk inode into the OS buffer, the subsequent *vfs_getxattr()* for online checking will not involve disk I/O when the soft links of the file are stored within its on-disk inode by the underlying file systems, thus introducing little overhead.

Readdir. The *sfs_readdir()* operation is implemented to read the directory entries of each replicated directory. During this process, the underlying *readdir* operation is invoked by SFS with the callback function *sfs_fill_dir()*, which will fill the directory entry into the user-space buffer and then increment the intradirectory offset by the size of each read entry during *readdir*. SFS also dispels duplicated directory entries across underlying file systems during *readdir*, which is similar to Unionfs [Wright et al. 2006]. To seek to a specified intradirectory offset, the SFS seek callback function

[8]https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.
[9]http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/Extended_Attributes.html.

sfs_seek_dir() is implemented to perform intradirectory offset counting, which would terminate the underlying *readdir* when the total size of the accessed directory entries increases to the intradirectory offset. Then, SFS successfully seeks to the specified intradirectory offset at the target underlying directory.

Rename. SFS uses the extended attribute mechanism to implement the soft-link technique. In particular, SFS creates an extended attribute <*"trusted.sfs.meta"*, value> to add a *hidden* or *stale* soft link. The *value* in the pair is a string containing multiple integers, which can be used as pointers. Then, the *xattr* mechanism allows SFS to add a soft link to a file through *vfs_setxattr()*, and to check a soft link through *vfs_getxattr()*. SFS removes a soft link from a file by setting the value of the extended attributes to a negative number. Specifically, when creating a soft link for a file during the rename, SFS invokes the *vfs_setxattr()* on the file to assign the partition ID to which the soft link points to the value of the extended attributes. During the *sfs_lookup()*, SFS will invoke *vfs_getxattr()* to check whether there exists any soft link for the file. During the rename operation, SFS invokes *write_inode_now()* with the sync parameter set on the file or the parent directory to persist the updates when needed.

Create and delete. The SFS *create* operation invokes the underlying *create* operation to create an object on the selected underlying file system. For file deletion–related operations, as only one underlying file exists for each SFS file, SFS simply removes the underlying file.

7. EVALUATION

We perform the evaluation of MultiLanes on both a modern PCI-Express–based SSD (785GB MLC Fusion-IO ioDrive2) and a RAM disk. Some experiments conducted are performed on a RAM disk, which allows us to evaluate the performance of MultiLanes compared to the baseline on next-generation storage devices (e.g., SSDs based on emerging nonvolatile memories). We also perform some experiments on a Fusion-IO ioDrive2 SSD to show how many performance improvements MultiLanes can bring on the present-day storage device. Note that, as the Fusion-IO SSD driver employs a global queue lock, there will be contention within the underlying driver. This can be eliminated in more advanced SSDs, however [Bjørling et al. 2013].

7.1. Experimental Setup

All experiments are carried out on an Intel 32-core machine with 512GB memory and four Intel(R) Xeon(R) E5-4650 processors, each having eight physical cores running at 2.70GHz (with the hyperthreading capability disabled). The RAM disk size is set to 320GB in our experiments, and we use 260GB of the SSD for evaluation. All experiments are carried out on the Linux 3.8.2 kernel unless otherwise noted. We also compiled a separated kernel with *lock stat* enabled to gather lock usage statistics during benchmarking.

In this section, we evaluate MultiLanes against canonical Linux as the baseline. For the baseline, we have the RAM disk or the SSD formatted with each target file system in turn and build 32 LXCs atop it. For MultiLanes, we have the host RAM disk or SSD formatted with Ext3 and mounted in *ordered* journal mode, then build 32 Linux containers over 32 virtualized devices that are mapped as thirty-two 8GB back-end files formatted with each target file system in turn. For Btrfs, the back-end file size is set to 15GB, as the back-end file size will affect the performance of random buffer writes on Btrfs. Before building containers for the baseline and MultiLanes, we preallocate all the memory pages for the RAM disk in order to avoid contention within the RAM disk driver. In all experiments, the guest file systems Ext3 and Ext4 are all mounted in ordered journal mode.

7.2. Partitioned Page-Cache

For both MultiLanes and the baseline, each LXC adopts the kernel feature cgroup[10] to provide memory resource isolation. Although the Linux cgroup provides an isolated LRU page-cache list for each container, it still uses the global zone LRU lock (i.e., *zone->lru_lock*) to protect all the isolated LRU page-cache lists.[11]

The use of the global lock will introduce unnecessary contention when multiple containers access their private LRU page-cache lists simultaneously. Hence, we partition the global LRU lock into per-container locks, each of which protects its corresponding per-container LRU page-cache list. We apply this patch to the kernel for both the baseline and MultiLanes to eliminate the contention within the page-cache layer between colocated containers. The Linux community also proposed a similar patch to partition the global LRU page-cache lock into per-container locks [Dickins 2012].

7.3. Parameter Impact Study

As described in Section 4, in order to avoid severe contention on the host file system, MultiLanes constrains the translation threads for block mapping to a small set of cores [Cui et al. 2013] and adopts a prefetching mechanism to reduce the communication overhead between the virtualized devices and the translation threads. Hence, there are two parameters in MultiLanes that can influence the performance: the number of binding CPU cores and the prefetching size value. In this section, we construct a set of experiments with the IOzone[12] benchmark to comprehensively measure the effect of the two parameters, then determine the optimal configuration for MultiLanes, which is adopted in the following experiments. Ext4 is chosen as the guest file system in this test.

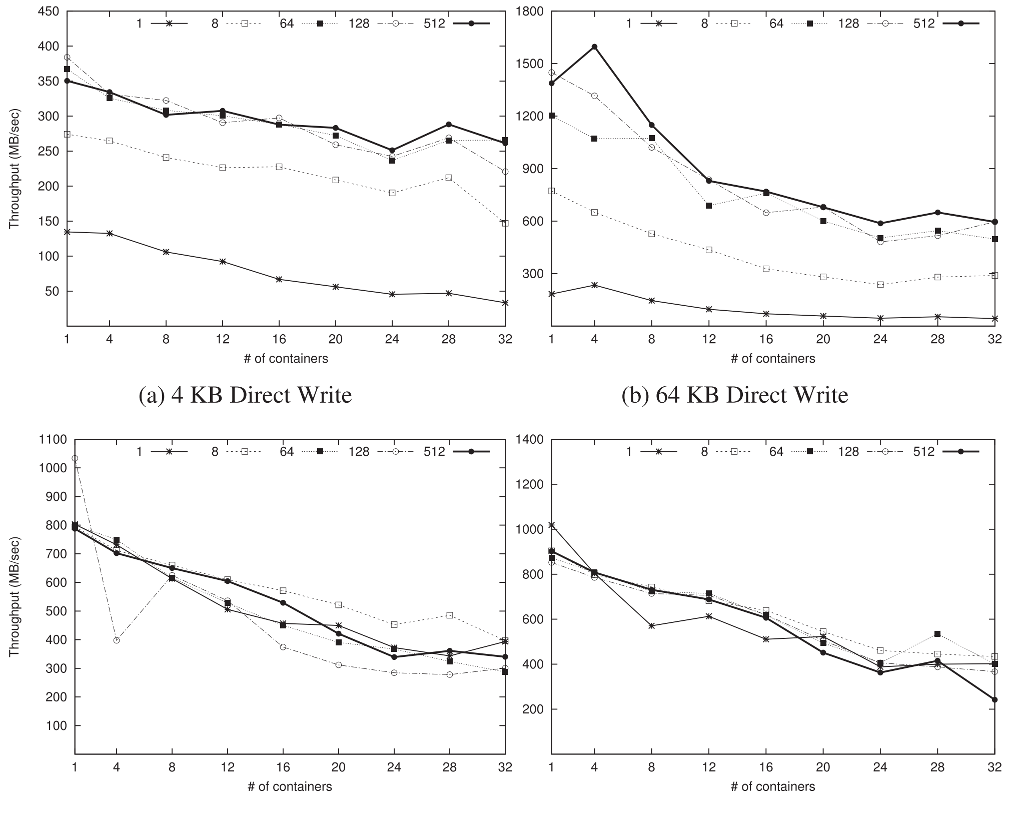
7.3.1. Prefetching Size. We evaluate the performance impact of various prefetching sizes using the IOzone benchmark. During this test, we bind the translation threads onto 4 CPU cores inside one processor. We run a single instance of the IOzone benchmark performing direct writes and buffered writes with different I/O sizes (4KB and 64KB) to a 1024MB file inside each container, and vary the number of containers.

As shown in Figure 6, we can see that, for direct writes, increasing the prefetching size improves the performance gradually due to the fact that it can effectively reduce communication between virtualized devices and translation threads. As the Ext4 file system delays the block allocation for buffered writes, there is no significant performance difference when varying the prefetching size.[13]

In particular, MultiLanes with the prefetching size of 128 outperforms that with the prefetching size of 1 by 5.6X and 13X under direct writes with 4KB and 64KB I/O size at 32 containers, respectively. However, when the prefetching size value exceeds 64, the performance improvement starts to become marginal. Meanwhile, setting a large prefetching size value will increase I/O latency as the translation thread needs to map a large continuous block region for each block cache miss. Hence, MultiLanes is configured with the prefetching size value being set to 128 by default. All the following performance experiments adopt this default value unless otherwise noted.

7.3.2. The Number of Binding CPU Cores. We evaluate the effect of varying the number of CPU cores that translation threads are bound to with IOzone and Sysbench[14]. Specifically, we use IOzone to generate 4KB sequential direct writes to a single file with a total write traffic of 1024MB inside each container. As the prefetching mechanism can

[10]Linux Cgroup. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
[11]https://www.kernel.org/doc/Documentation/cgroups/memory.txt.
[12]IOzone. http://www.iozone.org/.
[13]Delayed block allocation in Ext4. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.
[14]Sysbench. https://github.com/akopytov/sysbench.



(a) 4 KB Direct Write (b) 64 KB Direct Write

(c) 4 KB Buffered Write (d) 64 KB Buffered Write

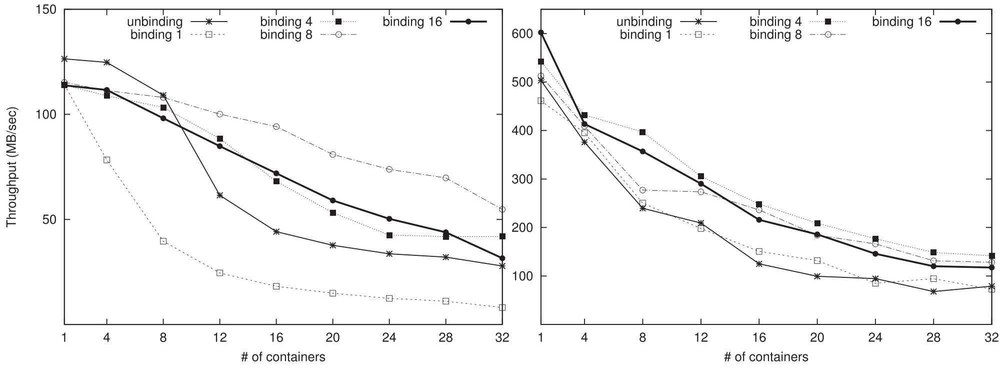
Fig. 6. Effect of various prefetching sizes on a RAM disk under the IOzone benchmark. Inside each container, we run a single instance of the benchmark program. In the figure, the prefetching size value N means that, for each block cache miss, the translation thread maps a continuous block region of N blocks.

yield maximum benefits under sequential writes, the prefetching size value is set to 1 in order to stress the translation threads. We also use Sysbench to generate 16KB random direct writes to 128 files with a total resulting size of 256MB inside each container. For 16KB random writes, the prefetching size value is set to 128 (default value).

As shown in Figure 7, MultiLanes configured with one binding CPU core performs worse than binding to a small number of cores due to the fact that the computing capacity of only one CPU core becomes the bottleneck. MultiLanes, with the translation threads being constrained to a small set of cores (i.e., 4, 8, and 16), performs much better than the "unbinding" configuration.

In order to understand the performance gains brought by the binding configuration, we collect the hot functions during this benchmarking by using the profiling tool *Perf*[15]. Table III shows the hot functions when performing 4KB direct writes on MultiLanes with the configuration "unbinding". We can see that the top hot functions are *claim_block()* and *ext3_test_allocatable()* for the unbinding configuration, which are invoked by the host ext3 file system during block allocation. The execution time of these two functions occupies 47% of the total execution time, while the execution time of the data-copying function (*brd_make_request()*) occupies only 7.27%. Table IV shows the

[15]Perf tool. https://perf.wiki.kernel.org/index.php/Main_Page.



(a) 4 KB Sequential Direct Writes (b) 16 KB Random Direct Writes

Fig. 7. Effect of various numbers of binding CPU cores on a RAM disk. The figure shows the effect of varying the number of CPU cores that translation threads are bound to with the IOzone benchmark. Inside each container, we run a single instance of the benchmark program. "Binding N" indicates that all the translation threads are bound to N CPU cores; "unbinding" means that all the translation threads are free to run on any CPU core.

Table III. Perf Statistics for Unbinding

	Proportion	Function
1	24.65%	claim_block()
2	22.72%	ext3_test_allocatable()
3	7.27%	brd_make_request()

Table IV. Perf Statistics for Binding to 4 Cores

	Proportion	Function
1	16.49%	brd_make_request()
2	11.82%	get_page()
3	10.68%	put_page()

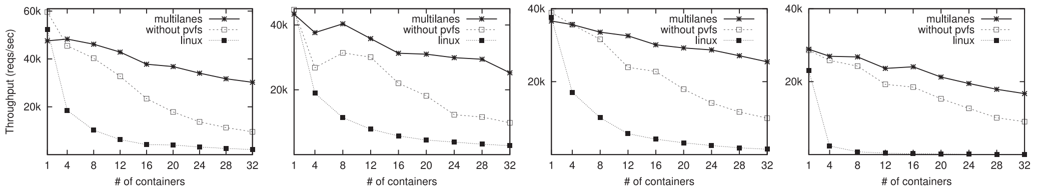
hot functions when binding all the translation threads to 4 cores. We can see that the data-copying function becomes the hottest function. The profiling results demonstrate that it can effectively reduce the contention on the host file system by binding the translation threads to a small set of cores, thus bringing significant performance improvements.

Specifically, MultiLanes with a configuration of 4 binding cores outperforms the "unbinding" configuration by 51% and 79% under 4KB sequential and 16KB random direct writes, respectively. For all the following experiments, MultiLanes adopts the configuration of 4 binding CPU cores.

7.4. MultiLanes Performance Results

7.4.1. Micro-Benchmarks. The micro-benchmarks consist of the metadata-intensive benchmark *CRU* developed from scratch, and IOzone.

CRU. The CRU benchmark creates a varying number (from 1 to 32) of processes, each of which runs inside a separated container and executes a loop of creating a new file, renaming the file and then unlinking the renamed file in parallel with other processes. The loop counter in CRU is set to a large number (64K) in order to produce I/O concurrency, thus stressing the I/O stack. For simplicity, each benchmark process runs inside a container's root directory rather than inside a complete Linux container environment. Such highly parallel metadata-intensive workload produced by the CRU



(a) CRU on Ext3 (b) CRU on Ext4 (c) CRU on XFS (d) CRU on Btrfs

Fig. 8. Metadata-intensive benchmark *CRU* on the Fusionio SSD. The figure shows the average throughput of the containers on different file systems when varying the number of containers running the CRU benchmark. The throughput is measured in loops per second.

Table V. Contention Bounces

Lock	Ext3	Ext4	XFS	Btrfs
inode_hash_lock	1605 k	2050 k	201 k	1062 k
dcache_lru_lock	130 k	113 k	166 k	163 k
inode_sb_list_lock	741 k	880 k	1978 k	515 k
rename_lock	1294 k	1041 k	1183 k	845 k

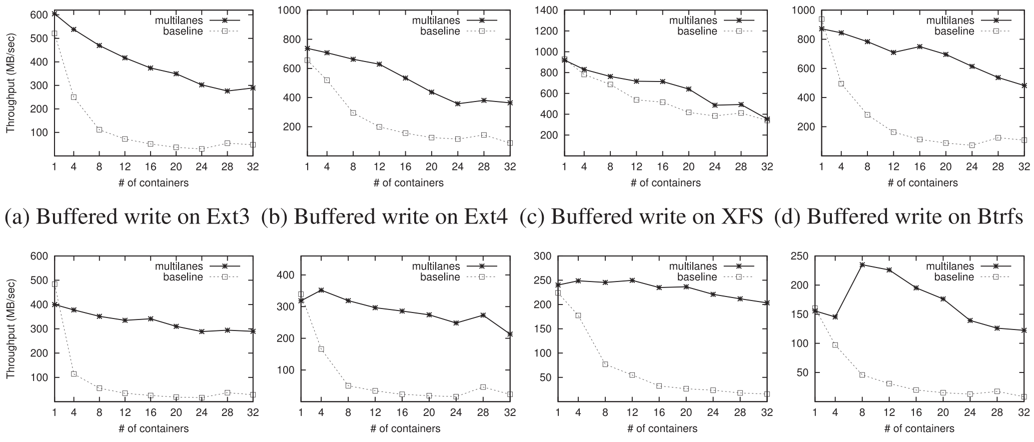
Note: The table shows the contention bounces gathered by using the *lock stat* tool when running the CRU benchmark on MultiLanes without pVFS at 32 containers.

benchmark could stress both the VFS layer and the underlying file system layer, thus allowing us to verify the effectiveness of both the pVFS and the virtualized device driver of MultiLanes.

Figure 8 presents the average throughput of the containers running the *CRU* benchmark for three situations: Linux, MultiLanes disabling pVFS, and complete MultiLanes. As shown in the figure, the average throughput suffers severe degradation with the increasing number of containers on all four file systems in Linux. Lock usage statistics show that it is caused by severe lock contention within both the underlying file system and the VFS. MultiLanes without pVFS achieves great performance gains and much better scalability than the baseline Linux, as the isolation via virtualized devices has eliminated contention in the file-system layer. The average throughput on complete MultiLanes is further improved owing to the pVFS that has eliminated the contention within the VFS layer. The results have demonstrated that each design component of MultiLanes is essential for scaling containers to manycores. Table V presents the contention details on the contended locks of the VFS that rise when the benchmark runs on MultiLanes without the pVFS.

IOzone. We use the IOzone benchmark to evaluate the performance and scalability of MultiLanes for data-intensive workloads, including sequential and random workloads. We run a single IOzone process inside each container and vary the number of concurrently running containers. Each container performs 4KB sequential/random writes to a file. The resulting file size inside each container is set to 1024MB, except when performing buffered writes on Btrfs. For Btrfs, the written file size is set to 256MB, as the performance of random buffered writes in Btrfs will be affected by both the file system size and written file size. Figure 9 shows the average throughput of the containers concurrently performing sequential writes in buffered mode and direct I/O mode, respectively.

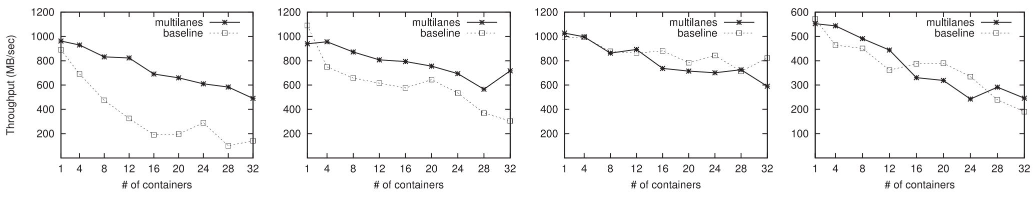
As shown in the figure, MultiLanes achieves much better scalability and outperforms the baseline Linux in all cases, except for buffered writes on XFS and random buffered writes on Btrfs. MultiLanes performs 5.09X, 3.11X, and 3.47X better than the baseline Linux on Ext3, Ext4, and Btrfs for buffered writes at 32 containers, respectively. For



(a) Buffered write on Ext3 (b) Buffered write on Ext4 (c) Buffered write on XFS (d) Buffered write on Btrfs

(e) Direct write on Ext3 (f) Direct write on Ext4 (g) Direct write on XFS (h) Direct write on Btrfs

Fig. 9. IOzone (Sequential Workloads) on a RAM Disk. The figure shows the container average throughput on different file systems when varying the number of LXC containers with IOzone. Inside each container, we run an IOzone instance performing sequential writes in the buffered mode and direct I/O mode, respectively.



(a) Buffered write on Ext3 (b) Buffered write on Ext4 (c) Buffered write on XFS (d) Buffered write on Btrfs

Fig. 10. IOzone (Random Workloads) on a RAM Disk. The figure shows the container average throughput on different file systems when varying the number of LXC containers with IOzone. Inside each container, we run an IOzone instance performing random writes in buffered mode.

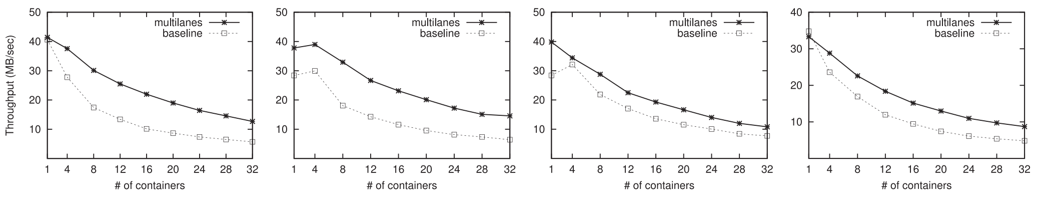
direct writes, the throughput improvement of MultiLanes against Linux is 9.34X, 8.26X, 12.09X, and 13.65X on the four file systems at 32 containers, respectively. XFS scales well for buffered writes owing to its own performance optimizations, such as delayed logging[16].

Figure 10 presents the results of random writes in buffered mode. As shown in the figure, MultiLanes outperforms Linux by 2.46X and 1.35X on Ext3 and Ext4 at 32 containers, respectively. As XFS scales well for buffered writes, MultiLanes exhibits competitive performance with it.

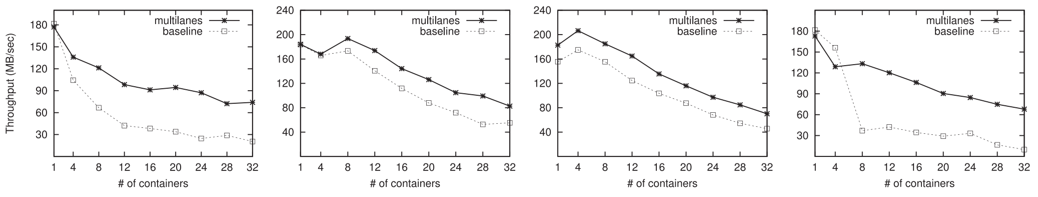
7.4.2. Macrobenchmarks. We choose Filebench, Dbench, and MySQL to evaluate performance and scalability of MultiLanes for application-level workloads.

Filebench. We use Filebench[17] to evaluate the performance and scalability of MultiLanes under application-level workloads: the Varmail and Fileserver workloads. We choose the Filebench Varmail and Fileserver workloads to evaluate MultiLanes, as they are write-intensive workloads that could cause severe contention within the I/O stack.

[16]https://www.kernel.org/doc/Documentation/filesystems/xfs-delayed-logging-design.txt.
[17]Filebench. http://filebench.sourceforge.net/wiki/index.php/Filebench.

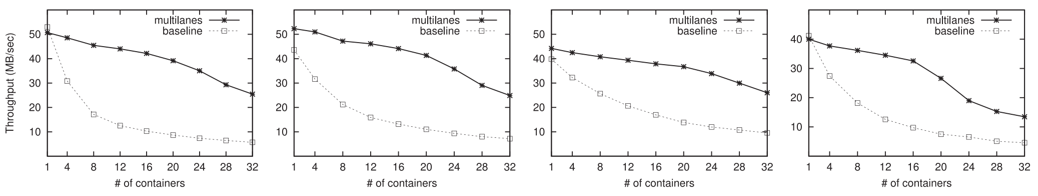


(a) Mail server on Ext3 (b) Mail server on Ext4 (c) Mail server on XFS (d) Mail server on Btrfs



(e) File server on Ext3 (f) File server on Ext4 (g) File server on XFS (h) File server on Btrfs

Fig. 11. Filebench on the Fusionio SSD. The figure shows the average throughput of the containers on different file systems when varying the number of LXC containers, with the Filebench mail server and file server workload, respectively.



(a) Mail server on Ext3 (b) Mail server on Ext4 (c) Mail server on XFS (d) Mail server on Btrfs

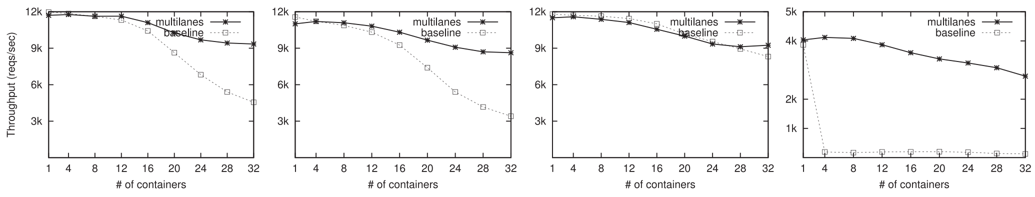
Fig. 12. Mail server on a RAM disk. This figure shows the average throughput of the containers when varying the number of LXC containers on different file systems with the Filebench mail server workload.

We run a single instance of Filebench inside each container. We adopt the Filebench default configuration for both workloads, except that the file number, the thread number, and the I/O size of the Varmail workload are set to 10000, 1, and 256KB, respectively, and the number of threads in each Fileserver instance is configured as 100.

Figure 11 shows the average throughput of multiple concurrent Filebench instances on MultiLanes compared to Linux. For the Varmail workload, MultiLanes performs 1.22X, 1.26X, 0.40X, and 0.80X better than the baseline Linux on Ext3, Ext4, XFS, and Btrfs at 32 containers, respectively. It should be noted that, as Btrfs sometimes will crash in Linux 3.8.2 when running multiple Varmail instances, we port the virtualized device driver of MultiLanes to Linux 3.13.2, which contains a more stable Btrfs for the Varmail benchmark.

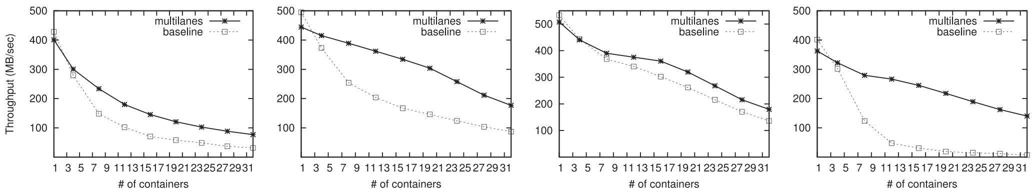
For the Fileserver workload, MultiLanes scales much better than Linux, leading to significant performance improvements. In particular, MultiLanes improves the throughput by 2.62X, 0.49X, 0.53X, and 5.93X compared to the baseline Linux on Ext3, Ext4, XFS, and Btrfs at 32 containers.

We also run the Varmail workload on the RAM disk to show the maximum performance improvement that MultiLanes can achieve. Figure 12 shows that MultiLanes scales much better than the baseline Linux, and exhibits near-linear scalability. Consequently, MultiLanes performs 3.47X, 2.45X, 1.73X, and 1.94 better than the baseline on Ext3, Ext4, XFS, and Btrfs at 32 containers, respectively. Note that, for the baseline,



(a) MySQL on Ext3 (b) MySQL on Ext4 (c) MySQL on XFS (d) MySQL on Btrfs

Fig. 13. MySQL on a RAM disk. This figure shows the average throughput of the containers when varying the number of LXC containers on different file systems with MySQL. The requests are generated with Sysbench within the same machine.



(a) Dbench on Ext3 (b) Dbench on Ext4 (c) Dbench on XFS (d) Dbench on Btrfs

Fig. 14. Dbench on the Fusionio SSD. This figure shows the average throughput of the containers on different file systems when varying the number of containers with the Dbench benchmark.

Btrfs is mounted in the SSD mode, as the Varmail workload will scale much poorly on the RAM disk without the SSD mount mode enabled.

MySQL. We evaluate the performance of running multiple MySQL[18] instances on MultiLanes against the baseline by using Sysbench. The evaluation is conducted in nontransaction mode, which performs *insert* operations. We run a MySQL server instance inside each container. For each MySQL server, one Sysbench thread is created as a client to generate 20k *insert* requests. Multiple Sysbench threads perform *insert* requests to the MySQL containers in parallel.

As shown in Figure 13, the average throughput of containers on MultiLanes exhibits nearly linear scalability with the increasing number of containers on the four file systems. As a result, MultiLanes improves the throughput by 1.05X, 1.53X, and 21.06X on Ext3, Ext4, and Btrfs at 32 containers, respectively. Once again, we have come to see that XFS scales well on manycores, and MultiLanes shows competitive performance with it.

Dbench. We use Dbench[19] to evaluate the performance of MultiLanes under application-level I/O workloads. Note that we run a Dbench instance inside each container's root directory rather than inside a complete Linux container for simplicity. Figure 14 demonstrates that MultiLanes scales much better than the baseline and improves the throughput on Dbench significantly. In particular, MultiLanes outperforms the baseline Linux by 1.45X, 1.02X, 0.32X, and 19X on Ext3, Ext4, XFS, and Btrfs at 32 containers, respectively.

7.5. Performance Results of MultiLanes-with-SFS against MultiLanes

In this section, we evaluate the performance and scalability of MultiLanes with SFS. Sixteen partitions are created by the virtualized device driver of MultiLanes for each

[18]MySQL. http://www.mysql.com/.
[19]Dbench. https://dbench.samba.org/.

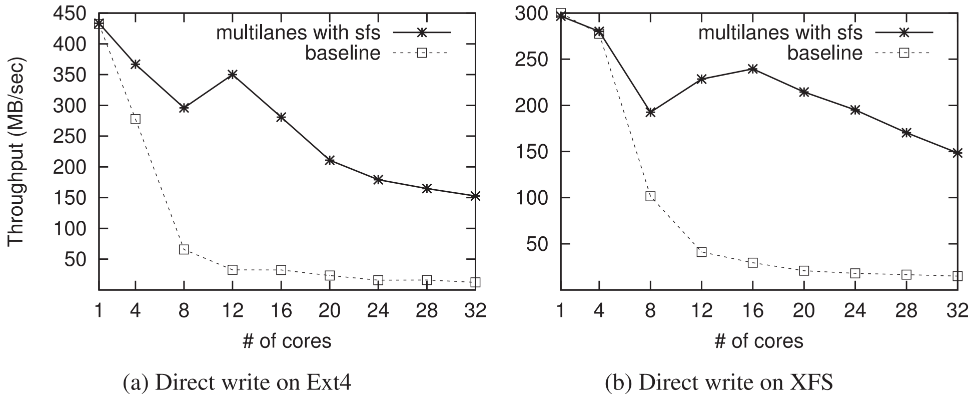


Fig. 15. IOzone on a RAM disk. This figure shows the average throughput of running multithreaded IOzone benchmark performing 4KB sequential direct writes inside one single container.

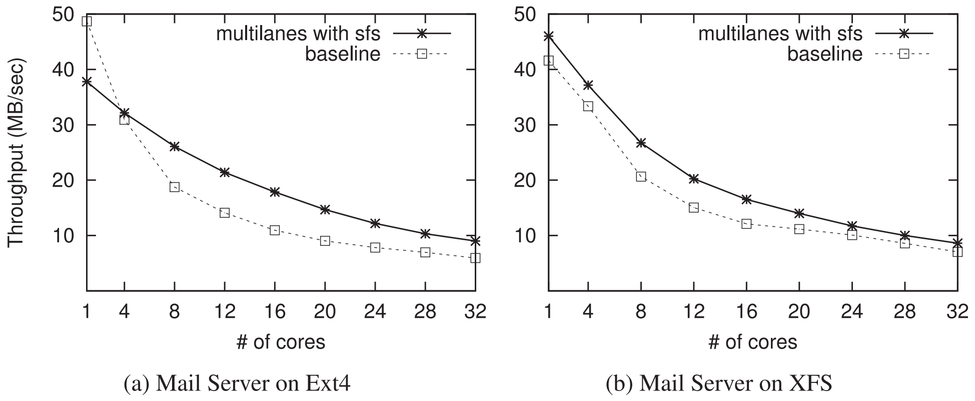


Fig. 16. Filebench Varmail on the Fusionio SSD. This figure shows the average throughput of running multiple Varmail instances inside one single container.

container. Then, we mount SFS on top of the 16 partitions inside each single container as the guest file system. In this section, we choose Ext4 and XFS as guest file systems.

IOzone. We use IOzone to evaluate the performance of MultiLanes with SFS under data-intensive workloads. Figure 15 shows the average throughput of running multithreaded IOzone benchmark inside one single container, in which each thread performs 4KB sequential direct writes to a file with a resulting file size of 512MB. For simplicity, the multithreaded IOzone benchmark runs inside the root directory of the single container rather than inside a complete Linux container. As shown in the figure, MultiLanes with SFS scales much better than the baseline Linux, leading to significant performance improvements. In particular, MultiLanes with SFS performs 11.66X and 8.86X better than the baseline Linux on Ext4 and XFS at 32 threads, respectively.

Varmail. We choose the Filebench Varmail workload to measure the performance of MultiLanes with SFS under application-level workloads. Figure 16 shows the average throughput of running multiple Varmail instances on MultiLanes with SFS against Linux inside one single container. We vary the number of threads by varying the number of running instances. For simplicity, we perform the experiment under the single container's root directory rather than inside a complete Linux container. As shown in the figure, MultiLanes with SFS scales much better than the baseline thanks to the

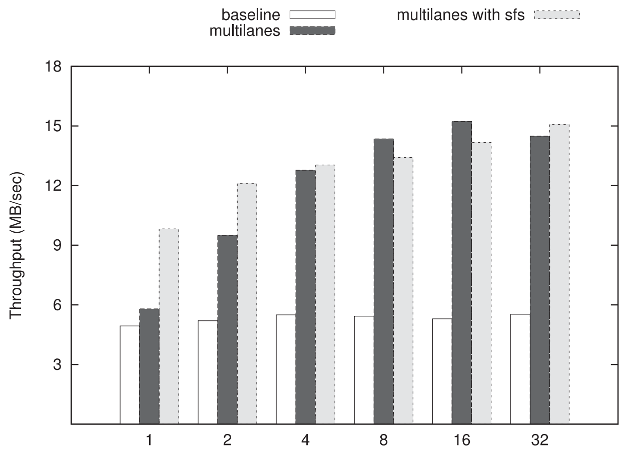


Fig. 17. Multithreading evaluation with Filebench Varmail on the Fusionio SSD. This figure shows the average throughput of 32 Varmail instances evenly distributed among multiple running containers. The horizontal axis means the number of containers.

reduced lock contention within the file system layer. MultiLanes with SFS outperforms the baseline Linux by 0.63X and 0.36X on Ext4 and XFS at 16 threads, respectively. However, the performance improvement starts to shrink beyond 16 threads due to the arising lock contention caused by other kernel components. MultiLanes with SFS performs 0.52X and 0.22X better than the baseline Linux on Ext4 and XFS at 32 threads, respectively.

The throughput degradation in MultiLanes with SFS is mainly caused by the arising lock contention in other kernel components (i.e., page cache and VFS) inside one single container. It should be noted that these lock bottlenecks have been eliminated between colocated containers in both the baseline and MultiLanes by the container namespace isolation and the patched Linux cgroup.

Multithreading Evaluation. We make a comparison of MultiLanes with SFS, MultiLanes, and the baseline on Ext4 when running 32 Varmail instances evenly distributed among multiple running containers. We construct this experiment in order to show how many performance improvements SFS can bring under multithreaded workloads inside each single container while varying the number of containers as well as the number of threads inside each container. In this test, inside each container, SFS is mounted on top of 8 partitions. The image size of each partition is set to 1GB.

From Figure 17, we can see that the average throughput of the baseline remains almost unchanged as the total number of running Varmail instances stays the same (32) regardless of the number of running containers. It is worth noting that, although the baseline has eliminated the contention within the page-cache layer between colocated containers by the patched cgroup, the throughput at 32 containers (1 Varmail instance inside each container) stays almost the same with the throughput at 1 container (32 Varmail instances inside the single container).

The average throughput on MultiLanes decreases with the decreasing number of containers due to the increasing number of Varmail instances within each single container, which causes contention inside each container. MultiLanes starts to yield significant performance improvements when we distribute the 32 Varmail instances to a number of containers.

MultiLanes with SFS can improve the performance when there are multiple running Varmail instances inside each single container thanks to the fact that SFS can mitigate the contention within the file-system layer inside each container. In particular, MultiLanes with SFS brings significant performance improvements relative to both the baseline and MultiLanes when running 32 Varmail instances inside one single container. The result demonstrates that MultiLanes with SFS is not only capable of eliminating the contention between colocated containers but also can effectively mitigate the contention within each single container.

8. CONCLUSION

Performance interference caused by the sharing of kernel resources within I/O stack among co-located containers could adversely affect the overall storage-system performance on manycore platforms. In this work, we propose MultiLanes, which consists of a virtualized storage device and a pVFS, to provide an isolated I/O stack to each container on manycores to eliminate the contention on shared kernel data structures and locks between colocated containers. The evaluation demonstrates that MultiLanes effectively addresses the I/O performance interference between the containers on manycores and exhibits significant performance improvement compared to Linux for most workloads. Moreover, we further propose an extension SFS to MultiLanes to mitigate the contention inside one single container.

REFERENCES

Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. 2007. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems* 25, 3.

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*.

Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

Matias Bjørling, Jens Axboe, David W. Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *6th Annual International Systems and Storage Conference (SYSTOR'13)*.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

John L. Bruno, Eran Gabber, Banu Özden, and Avi Silberschatz. 1998. The eclipse operating system: Providing quality of service via reservation domains. In *1998 USENIX Annual Technical Conference*.

Edouard Bugnion, Scott Devine, and Mendel Rosenblum. 1997. DISCO: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*.

Bryan Cantrill and Jeff Bonwick. 2008. Real-world concurrency. *ACM Queue* 6, 5, 16–25.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*.

Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995*.

Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *17th International Conference on High-Performance Computer Architecture (HPCA'11)*.

Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP'13)*.

Dave Chinner. 2011. dentry: move to per-sb LRU locks. Retrieved April 4, 2016 from https://lkml.org/lkml/2011/8/8/34.

Dave Chinner. 2013. Sync and VFS scalability improvements. Retrieved April 4, 2016 from http://lwn.net/Articles/561569/.

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2013. The scalable commutativity rule: Designing scalable software for multicore processors. In *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP'13)*.

Yan Cui, Yingxin Wang, Yu Chen, and Yuanchun Shi. 2013. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *ACM Transactions on Architecture and Code Optimization* 9, 4, 44:1–44:25.

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP'13)*.

Hugh Dickins. 2012. mm/memcg: per-memcg per-zone lru locking. Retrieved April 4, 2016 from https://lwn.net/Articles/482726/.

Rasha Eqbal. 2014. *ScaleFS: A Multicore-Scalable File System*. Master's thesis. Massachusetts Institute of Technology, Cambridge, MA.

Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*.

Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

Charles Gruenwald III. 2014. *Providing a Shared File System in the Hare POSIX Multikernel*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Hare: A file system for non-cache-coherent multicores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*.

Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. 2014. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*.

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*.

Kir Kolyshkin. 2012. Introducing container in a file aka ploop. Retrieved April 4, 2016 from http://openvz.livejournal.com/40830.html.

Duy Le, Hai Huang, and Haining Wang. 2012. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Stelios Mavridis, Yannis Sfakianakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2014. Jericho: Achieving scalability through optimal data placement on multicore systems. In *IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*.

Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-copy update. In *Ottawa Linux Symposium*.

Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The design and implementation of zap: A system for migrating computing environments. In *5th Symposium on Operating System Design and Implementation (OSDI'02)*.

Maxim Patlasov. 2011. Containers in a File. Retrieved April 4, 2016 from https://openvz.org/images/f/f3/Ct_in_a_file.pdf. (2011).

Jan-Simon Pendry and Marshall K. McKusick. 1995. Union mounts in 4.4BSD-lite. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*.

Dai Qin, Angela Demke Brown, and Ashvin Goel. 2014. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*.

Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *Operating Systems Review* 42, 5, 95–103.

Eric Seppanen, Matthew T. O'Keefe, and David J. Lilja. 2010. High performance solid state storage under Linux. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*.

Yannis Sfakianakis, Stelios Mavridis, Anastasios Papagiannis, Spyridon Papageorgiou, Markos Fountoulakis, Manolis Marazakis, and Angelos Bilas. 2014. Vanguard: Increasing server efficiency via workload isolation in the storage I/O path. In *Proceedings of the ACM Symposium on Cloud Computing*.

Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy C. Bavier, and Larry L. Peterson. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2007 EuroSys Conference*.

Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. 2011. A case for scaling applications to many-core with OS clustering. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*.

Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. 2001. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*.

Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1998. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. 2007. Argon: Performance insulation for shared storage servers. In *5th USENIX Conference on File and Storage Technologies (FAST'07*).

Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage* 2, 1, 74–105.

Erez Zadok, Ion Badulescu, and Alex Shender. 1999. Extending file systems using stackable templates. In *Proceedings of the 1999 USENIX Annual Technical Conference*.

Da Zheng, Randal Burns, and Alexander S. Szalay. 2013. Toward millions of file system IOPS on low-cost, commodity hardware. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*.

Received November 2014; revised June 2015; accepted July 2015