# MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores

Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai
Beihang University, Beijing, China
{*kangjb, woty, hucm*}*@act.buaa.edu.cn, zblgeqian@gmail.com, huaijp@buaa.edu.cn*

## Abstract

OS-level virtualization is an efficient method for server consolidation. However, the sharing of kernel services among the co-located *virtualized environments* (VEs) incurs performance interference between each other. Especially, interference effects within the shared I/O stack would lead to severe performance degradations on many-core platforms incorporating fast storage technologies (e.g., non-volatile memories).

This paper presents MultiLanes, a virtualized storage system for OS-level virtualization on many cores. MultiLanes builds an isolated I/O stack on top of a virtualized storage device for each VE to eliminate contention on kernel data structures and locks between them, thus scaling them to many cores. Moreover, the overhead of storage device virtualization is tuned to be negligible so that MultiLanes can deliver competitive performance against Linux. Apart from scalability, MultiLanes also delivers flexibility and security to all the VEs, as the virtualized storage device allows each VE to run its own guest file system.

The evaluation of our prototype system built for Linux container (LXC) on a 16-core machine with a RAM disk demonstrates MultiLanes outperforms Linux by up to 11.32X and 11.75X in micro- and macro-benchmarks, and exhibits nearly linear scalability.

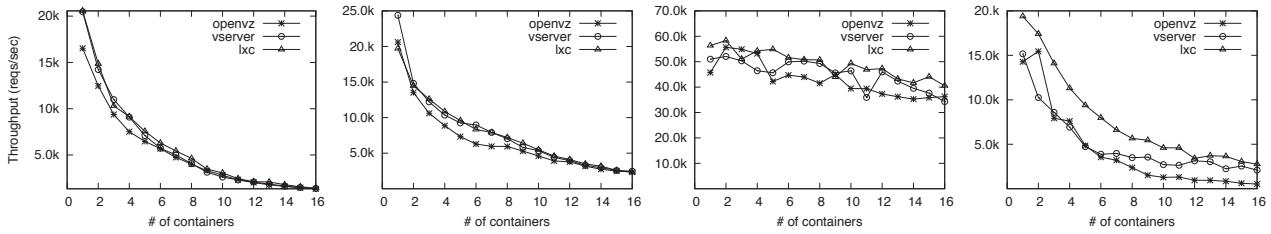
## 1 Introduction

As many-core architectures exhibit powerful computing capacity, independent workloads can be consolidated in a single node of data centers for high efficiency. Operating system level virtualization (e.g., VServer [32], OpenVZ [6], Zap [29], and LXC [4]) is an efficient method to run multiple virtualized environments (VEs) for server consolidation, as it comes with significantly lower overhead than hypervisors [32, 29]. Thus, each independent workload can be hosted in a VE for both good isolation and high efficiency [32]. Previous work on OS-level virtualization mainly focuses on how to efficiently space partition or time multiplex the hardware resources (e.g., CPU, memory and disk).

However, the advent of non-volatile memory technologies (e.g., NAND flash, phase change memories and memristors) creates challenges for system software. Specially, emerging fast storage devices built with non-volatile memories deliver low access latency and enormous data bandwidth, thus enabling high degree of application-level parallelism [16, 31]. This advance has shifted the performance bottleneck of the storage system from poor hardware performance to system software inefficiencies. Especially, the sharing of the I/O stack would incur performance interference between the co-located VEs on many cores, as the legacy storage stack scales poorly on many-core platforms [13]. A few scalability bottlenecks exist in the Virtual File System (VFS) [23] and the underlying file systems. As a consequence, the overall performance of the storage system suffers significant degradations when running multiple VEs with I/O intensive workloads. The number of concurrently running VEs may be limited by the software bottlenecks instead of the capacity of hardware resources, thus degrading the utilization of the hardware.

This paper presents MultiLanes, a storage system for operating system level virtualization on many cores. MultiLanes eliminates contention on shared kernel data structures and locks between co-located VEs by providing an isolated I/O stack for each VE. As a consequence, it effectively eliminates the interference between the VEs and scales them well to many cores. 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 VE to run a guest file system instance atop it. This approach avoids contention on shared data structures within the file system layer by providing an isolated file system stack for




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

Figure 1: **VE Scalability Evaluation.** *This figure shows the average throughput of each container performing sequential buffered writes on different file systems. We choose the latest OpenVZ, Linux-VServer and LXC that are based on Linux kernel 2.6.32, 3.7.10, and 3.8.2 respectively. The details of experimental setup is to be presented in Section 5.*

| **Ext3** | | | | **Ext4** | | | |
|---|---|---|---|---|---|---|---|
| # | lock | contention bounces | total wait time | # | lock | contention bounces | total wait time |
| 1 | zone->wait_table | 5216186 | 36574149.95 | 1 | journal->j_list_lock | 2085109 | 138146411.03 |
| 2 | journal->j_state_lock | 1581931 | 56979588.44 | 2 | zone->wait_table | 147386 | 384074.06 |
| 3 | journal->j_list_lock | 382055 | 20804351.46 | 3 | journal->j_state_lock | 46138 | 541419.08 |
| **XFS** | | | | **Btrfs** | | | |
| # | lock | contention bounces | total wait time | # | lock | contention bounces | total wait time |
| 1 | zone->wait_table | 22185 | 36190.48 | 1 | found->lock | 778055 | 44325371.60 |
| 2 | rq->lock | 6798 | 9382.04 | 2 | btrfs-log-02 | 387846 | 1124781.19 |
| 3 | key#3 | 4869 | 13463.40 | 3 | btrfs-log-01 | 230158 | 1050066.24 |

Table 1: **The Top 3 Hottest Locks.** *This table shows the contention bounces and total wait time of the top 3 hottest locks when running 16 LXC containers with buffered writes. The total wait time is in us.*

each VE. 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, and (2) how to achieve good scalability with the number of virtualized block devices on the host file system which itself scales poorly on many cores.

Hence, we propose a set of techniques to address these challenges. First, MultiLanes uses a synchronous bypass strategy to complete block I/O requests of the virtualized block device. In particular, it translates a block I/O request from the guest file system into a list of requests of the host block device using block mapping information got 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 constrains the work threads interacting with the host file system for block mapping to a small set of cores to avoid severe contention on the host, as well as adopts a prefetching mechanism to reduce the communication costs between the virtualized devices and the work threads.

Another alternative for block device virtualization is to give VEs direct accesses to physical devices or logical volumes for native performance. However, there are several benefits in adopting plain files on the host as the back-end storage for virtualization environments [24]. First, using files allows storage space overcommitment as most modern file systems support sparse files (e.g., Ext3/4 and XFS). Second, it also eases the management of VE images as we can leverage many existing file-based storage management tools. Third, snapshotting an image using copy-on-write is simpler at the file level than the block device level.

**The partitioned VFS.** In Unix-like operating systems, VFS provides a standard file system interface for applications to access different types of concrete file systems. As it needs to maintain a consistent file system view, the inevitable use of global data structures (e.g., the inode cache and dentry cache) as well as the corresponding locks might result in scalability bottlenecks on many cores. Rather than iteratively eliminating or mitigating the scalability bottlenecks of the VFS [13], MultiLanes in turn adopts a straightforward strategy that partitions the VFS data structures to completely eliminate contention between co-located VEs, as well as to achieve improved locality of the VFS data structures on many cores. The partitioned VFS is referred to as the pVFS in the rest of the paper.

The remainder of the paper is organized as follows. Section 2 highlights the storage stack bottlenecks in existing OS-level virtualization approaches for further motivation. Then we present the design and implementation of MultiLanes in Section 3 and Section 4 respectively. Section 5 evaluates its performance and scalability with micro- and macro-benchmarks. We discuss the related works in Section 6 and conclude in Section 7. A virtualized environment is referred to as a *container* in the following sections also.

## 2 Motivation

In this section, we create a simple microbenchmark to highlight the storage stack bottlenecks of existing OS-level virtualization approaches on many-core platforms incorporating fast storage technologies. The benchmark performs 4KB sequential 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 the file systems except 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 in Table 1 presents the lock bounces and total wait time during the benchmarking, which results in the decreased performance. XFS delivers much better scalability than the other three as much less contention occurred for buffered writes. Nevertheless it would also suffer from scalability bottlenecks under other workloads, which will be described in Section 5.

The poor scalability of the storage system is mainly caused by the concurrent accesses to shared data structures and the use of synchronization primitives. The use of shared data structures modified by multiple cores would cause frequent transfers of the data structures and the protecting locks among the cores. As the access latency of remote caches is much larger than that of local caches on modern shared-memory multicore processors [12], the overhead of frequent remote accesses would significantly decrease the overall system performance, leading to severe scalability bottlenecks. Especially, the large traffic of non-scalable locks generated by cache coherence protocols on the interconnect will exacerbate system performance. Previous studies show that the time taken to acquire a lock will be proportional to the number of contending cores [13, 12].

## 3 MultiLanes Design

MultiLanes is a storage system for OS level virtualization that addresses the I/O performance interference between the co-located VEs on many cores. In this section, we present the designing goals, concepts and components of MultiLanes.

### 3.1 Design Goals

Existing OS-level virtualization approaches simply leverage *chroot* to realize file system virtualization [32, 6, 29]. The containers co-located share the same I/O stack, which not only leads to severe performance interference between them but also suppresses flexibility.

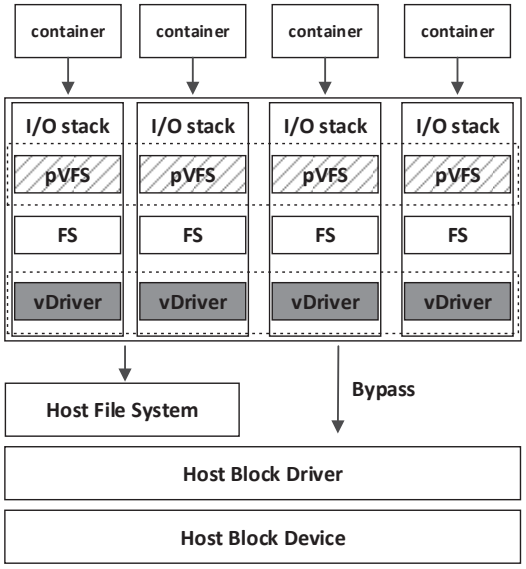



Figure 2: **MultiLanes Architecture.** *This figure depicts the architecture of MultiLanes. The virtualized storage is mapped as a plain file on the host file system and is left out in the figure.*

MultiLanes is designed to eliminate storage system interference between containers to provide good scalability on many cores. We aim to meet three design goals: (1) it should be conceptually simple, self-contained, and transparent to applications and to various file systems; (2) it should achieve good scalability with the number of containers on the host; (3) it should minimize the virtulization overhead on fast storage media so as to offer near-native performance.

### 3.2 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 abstractions of the design. We have left out other kernel components to better focus on the I/O subsystem.

At the top of the architecture we host multiple containers. A container is actually a group of processes which are completely constrained to execute inside it. Each container accesses its guest file system through the partitioned VFS that provides POSIX APIs. The partitioned VFS 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 underlying file system by providing the same standard interfaces with the VFS.

Between the guest file system and the host are the virtualized block device and the corresponding customized

block device driver. MultiLanes maps regular files in the host file system as virtualized storage devices to containers, which provides the fundamental basis for running 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 use a different file system from each other, which enables flexibility both between the host and a single guest, and between the guests. The virtualized device driver is customized for each virtualized device, which provides the standard interfaces to the Linux generic block layer. Meanwhile, MultiLanes adopts a proposed synchronous bypass mechanism to avoid most of the overhead induced by the virtualization layer.

## 3.3 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.

### 3.3.1 Virtualized Storage

Compared to full virtualization and para-virtualization that provide virtualized storage devices for virtual machines (VMs), OS-level virtualization stores the VMs' data directly on the host file system for I/O efficiency. However, the virtualized storage has inborn advantage over shared storage in performance isolation because each VM has an isolated I/O stack.

As described in Section 2, the throughput of each LXC container will fall dramatically with the increasing number of containers due to the severe contention on shared data structures and locks within the shared I/O stack. The interference is masked by the high latency of the sluggish mechanical disk in traditional disk-based storage. But it has to be reconsidered in the context of next generation storage technologies due to the shift that system software becomes the main bottleneck on fast storage devices.

In order to eliminate storage system performance interference between containers on many cores, we provide lightweight virtualized storage for each container. We map a regular file as a virtualized block device for each container, and then build the guest file system on top of it. Note that as most modern file systems support sparse files for disk space efficiency, the host doesn't preallocate all blocks in accordance with the file size when the file system is built on the back-end file. The challenge is to balance performance gain achieved by performance isolation against the overhead incurred by storage virtualization. However, scalability and competitive performance can both be achieved when the virtualized storage architecture is efficiently devised.

### 3.3.2 Driver Model

Like any other virtualization approaches adopted in other fields, the most important work for virtualization is to establish the mapping between the virtualized resources and the physical ones. This is done by the virtualized block device driver in MultiLanes. As shown in Figure 2, each virtualized block device driver receives block I/O requests from the guest file system through the Linux generic block layer and maps them to requests of the host block device.

A block I/O request is composed of several segments, which are contiguous on the block device, but are not necessarily contiguous in physical memory, depicting a mapping between a block device sector region and a list of individual memory segments. On the block device side, it specifies the data transfer start sector and the block I/O size. On the buffer side, the segments are organized as a group of I/O vectors. Each I/O vector is an abstraction of a segment that is in a memory page, which specifies the physical page on which it lies, offset relative to the start of the page, and the length of the segment starting from the offset. The data residing in the block device sector region would be transmitted to/from the buffer in sequence according to the data transfer direction given in the request.

For the virtualized block device of MultiLanes, the sector region specified in the request is actually a data section of the back-end file. The virtualized driver should translate logical blocks of the back-end file to physical blocks on the host, and then map each I/O request to the requests of the host block device according to the translation. It is composed of two major components: the block translation and block handling.

**Block Translation.** Almost all modern file systems have devised a mapping routine to map a logical block of a file to the physical block on the host device, which returns the physical block information to the caller at last. If the block is not mapped, the mapping process involves the block allocation of the file system. MultiLanes achieves block translation with the help of this routine.

As shown in Figure 3, the block translation unit of each virtualized driver consists of a cache table, a job queue and a translation thread. The cache table maintains the mapping between logical blocks and physical blocks. The virtulized 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

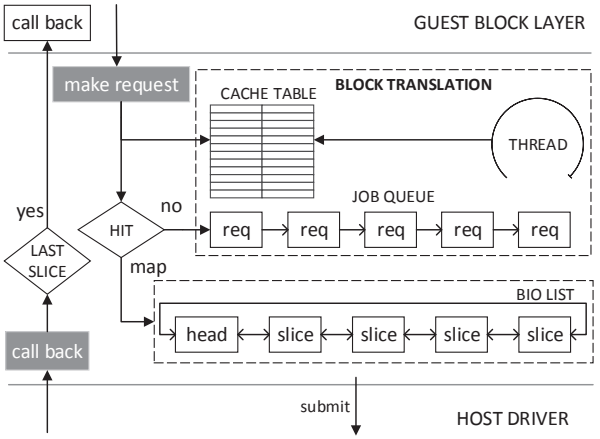



Figure 3: **Driver Structure.** *This figure presents the structure of the virtualized storage driver, which comprises the block translation unit and the request handling unit.*

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 and 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, stores 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.

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

On the other hand, as the block mapping of the host file system usually involves file system journaling, the mapping process may cause severe contention within the host on many cores when cache misses of multiple virtulized drivers occur concurrently, thus scaling poorly with the number of virtualized devices on many cores. We address this issue by constraining all translation threads to work on a small set of cores to reduce contention [18] and improve data locality on the host file system. Our current prototype binds all translation threads to a set of cores inside a processor, due to the observation that sharing data within a processor is much less expensive than that crossing processors [12].

**Request Handling.** Since the continuous data region of the back-end file may not be necessarily 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.

There are two mapping involved when handling the block I/O requests of the virtualized block device. The mapping between the memory segments and the virtualized block device sector region is specified in a scatter-gather manner. The mapping between the virtualized block device and the host block device gives the physical block number of a logical block of the back-end file. For simplicity, the block size of the virtualized block device should be the same with that of the host block device in our current prototype. For each segment of the block I/O request, the virtulized device driver first gets 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. When all the segments of a request are remapped, we have to check whether they are contiguous on the host block device. The virtualized device driver combines the segments which 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 a new block I/O request for each of the remaining segments. Thus a single block I/O request of the virtualized block device might be remapped to several requests of the host block device. Figure 4 illustrates such an example, which will be described in Section 4.1.

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. But the original request 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 finds out 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 only requires minor modification to the host file system to export the mapping routine interface.

### 3.3.3 Partitioned VFS

The virtual file system in Linux provides a generic file system interface for applications to access different types of concrete file systems in a uniform way. Although MultiLanes allows each container to run its own guest file system independently, there still exists performance interference within the VFS layer. Hence, we propose the partitioned VFS that provides a private VFS abstraction to each container, eliminating the contention for shared data structures within the VFS layer between containers.

| # | Hot VFS Locks | Hot Invoking Functions |
|---|---|---|
| 1 | inode_hash_lock | insert_inode_locked()<br>__remove_inode_hash() |
| 2 | dcache_lru_lock | dput()<br>dentry_lru_prune() |
| 3 | inode_sb_list_lock | evict()<br>inode_sb_list_add() |
| 4 | rename_lock | write_seqlock() |

Table 2: **Hot VFS Locks.** *The table shows the hot locks and the corresponding invoking functions in VFS when running the metadata intensive microbenchmark ocrd in Linux kernel 3.8.2.*

Table 2 shows the top four hottest locks in VFS when conducting the metadata-intensive microbenchmark *ocrd*, which will be described in Section 5. VFS maintains an inode hash table to speed up inode lookup and uses the *inode_hash_lock* to protect the list. Inodes that belong to different super blocks are hashed together into the hash table. Meanwhile, each super block has a list that links all the inodes that belong to it. Although this list is independently managed by each super block, the kernel uses the global *inode_sb_list_lock* to protect accesses to all lists, which would introduce unnecessary contention between multiple file system instances.

For the purpose of path resolution speedup, VFS uses a hash table to cache directory entries, which allows concurrent read accesses to it without serialization by using Read-Copy-Update (RCU) locks [27]. The *rename_lock* is a sequence lock that is indispensable for the hash table in this context because a rename operation may involve the edition of two hash buckets which might cause false lookup results. It is also inappropriate that the VFS protects the LRU dentry lists of all file system instances with the global *dcache_lru_lock*.

Rather than iteratively fixing or mitigating the lock bottlenecks in the VFS, we in turn adopts a straightforward approach that partitions the VFS data structures and corresponding locks to eliminate contention, as well as to improve locality of the VFS data structures. 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 with the separation of the two hash tables from each other, *inode_hash_lock* and *rename_lock* are also separated. Meanwhile, each guest file system has its own *inode_sb_list_lock* and *dcache_lru_lock* also.

By partitioning the resources that would cause contention in the VFS, the VFS data structures and locks become localized within each partitioned domain. Supposing there are *n* virtualized block devices built on the host file system, the original VFS domain now is split into *n+1* independent domains: each guest file system domain and the host domain that serves the host file system along with special file systems (e.g., procfs and debugfs). We refer the partitioned VFS to as the pVFS. The pVFS is an important complementary part of the isolated I/O stack.

## 4 Implementation

We choose to implement the prototype of MultiLanes for Linux Container (LXC) out of OpenVZ and Linux-VServer due to that both OpenVZ and Linux-VServer need customized kernel adaptations while LXC is always supported by the latest Linux kernel. We implemented MultiLanes in the Linux 3.8.2 kernel, which consists of a virtualized block device driver module and adaptations to the VFS.

### 4.1 Driver Implementation

We realize the virtualized block device driver based on the Linux loop device driver that provides the basic functionality of mapping a plain file as a storage device on the host.

Different from traditional block device drivers that usually adopt a request queue based asynchronous model, the virtualized device driver of MultiLanes adopts a synchronous bypass strategy. 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. And we export the *xxx_get_block* function into the *inode_operations* structure for Ext3, Ext4, Btrfs, Reiserfs and JFS 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.

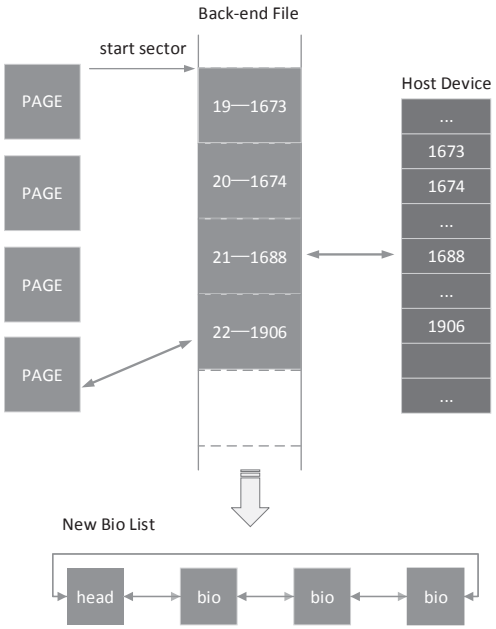



Figure 4: **Request Mapping.** *This figure shows the mapping from a single block I/O request of the virtualized block device to a request list on the host block device.*

Figure 4 shows an example of block request mapping. We assume the page size is 4096 bytes and the block size of the host block device and the virtualized storage device are both 4096 bytes. 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 152 and the block I/O size is 16KB. The *bio* contains four individual memory segments, which lie in four physical pages. After all the logical blocks of the request are mapped by the block translation unit, we can see that only the logical block 19 and 20 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, and then delivers the new *bio*s to the host driver in sequence.

## 4.2 pVFS Implementation

The partitioned VFS data structures and locks are organized in the super block of the 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* and *rename_lock*) to the super block. Meanwhile, each super block has its own LRU dentry list, and inode list along with the separated *dcache_lru_lock* and *inode_sb_list_lock*. We also 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. We first find out all the code spots where the hash tables, lists and locks are accessed. Then, a multiplexer is embedded in each code spot to do the branching. 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. This work takes much efforts to finish all the code spots. But this is non-complicated work since the idea behind all modifications is the same.

# 5 Evaluation

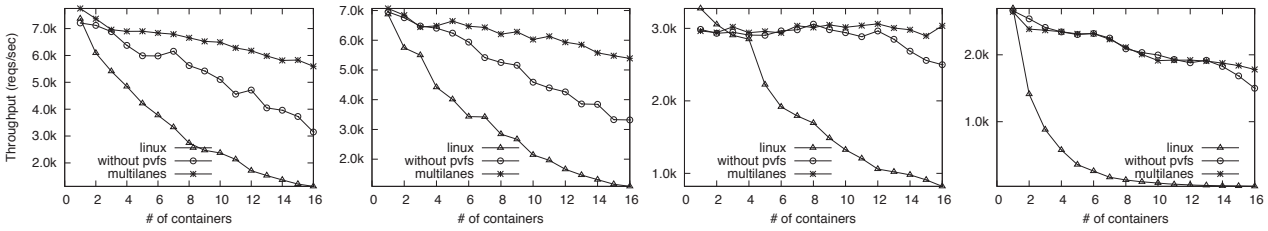
Fast storage devices mainly include prevailing NAND flash-based SSDs, and SSDs based on next-generation technologies (e.g., Phase Change Memory), which promise to further boost the performance. Unfortunately when the evaluation was conducted we did not have a high performance SSD at hand. So we used a RAM disk to emulate a PCM-based SSD since phase change memory is expected to have bandwidth and latency characteristics similar to DRAM [25]. The emulation is appropriate as Multilanes does not concern about the underlying specific storage media, as long as it is fast enough. Moreover, using a RAM disk could rule out any effect from SSDs (e.g., global locks adopted in their corresponding drivers) so as to measure the maximum scalability benefits of MultiLanes.

In this section, we experimentally answer the following questions: (1) Does MultiLanes achieve good scalability with the number of containers on many cores ? (2) Are all of MultiLanes's design components necessary to achieve such good scalability? (3) Does the overhead induced by MultiLanes contribute marginally to the performance under most workloads?

## 5.1 Experimental Setup

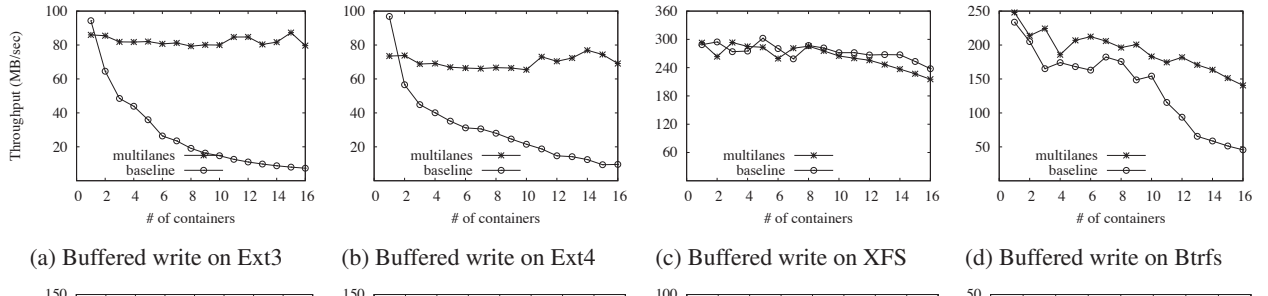
All experiments were carried out on an Intel 16-core machine with four Intel Xeon(R) E7520 processors and 64GB memory. Each processor has four physical cores clocked at 1.87GHZ. Each core has 32KB of L1 data cache, 32KB of L1 instruction cache and 256KB of L2 cache. Each processor has a shared 18MB L3 cache. The hyperthreading capability is turned off.

We turn on RAM block device support as a kernel module and set the RAM disk size to 40GB. Lock usage

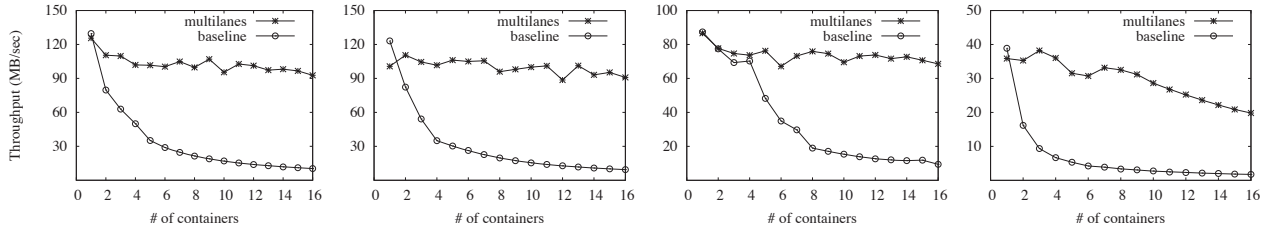



(a) Ocrd on Ext3 (b) Ocrd on Ext4 (c) Ocrd on XFS (d) Ocrd on Btrfs

Figure 5: **Scalability Evaluation with the Metadata-intensive Benchmark *Ocrd*.** *The figure shows the average throughput of the containers on different file systems when varying the number of LXC containers with ocrd. Inside each container we run a single instance of the benchmark program.*

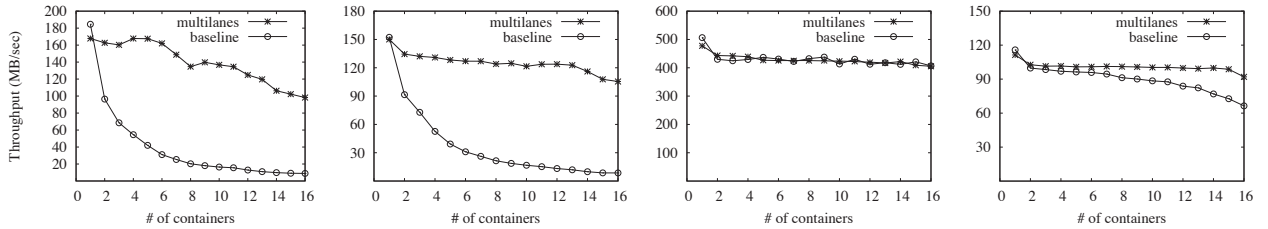



(a) Buffered write on Ext3 (b) Buffered write on Ext4 (c) Buffered write on XFS (d) Buffered write on Btrfs




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

Figure 6: **Scalability Evaluation with IOzone (Sequential Workloads).** *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 process performing sequential writes in 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

Figure 7: **Scalability Evaluation with IOzone (Random Workloads).** *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 process performing random writes in buffered mode.*

statistics is enabled to identify the heavily contended kernel locks during the evaluation. In this section, we evaluate MuliLanes against canonical Linux as the baseline. For the baseline groups, we have a RAM disk formatted with each target file system in turn and build 16 LXC containers atop it. For MultiLanes, we have the host RAM disk formatted with Ext3 and mounted in *ordered* mode, then build 16 LXC containers over 16 virtualized devices which are mapped as sixteen 2500MB regular files formatted with each target file system in turn. In all the experiments, the guest file system Ext3 and Ext4 are all mounted in *journal* mode unless otherwise specified.

## 5.2 Performance Results

The performance evaluation consists of both a collection of micro-benchmarks and a set of application-level macrobenchmarks.

### 5.2.1 Microbenckmarks

The purpose of the microbenchmarks is two-fold. First, these microbenchmarks give us the opportunity to measure an upper-bound on performance, as they effectively rule out any complex effects from application-specific behaviors. Second, microbenchmarks allow us to verify the effectiveness of each design component of MultiLanes as they stress differently. The benchmarks consist of the metadata-intensive benchmark *ocrd* developed from scratch, and IOzone [3] which is a representative storage system benchmark.

**Ocrd.** The ocrd benchmark runs 65536 transactions, and each transaction creates a new file, renames the file and at last deletes the file. It is set up for the purpose of illuminating the performance contributions of each individual design component of MultiLanes because the metadata-intensive workload could cause heavy contention on both the hot locks in the VFS, as mentioned in Table 2, and those in the underlying file systems.

Figure 5 presents the average throughput of each container running the *ocrd* 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 it is caused by severe lock contention within both the underlying file system and the VFS. Contention bounces between cores can reach as many as several million times for the hot locks. MultiLanes without pVFS achieves great performance gains and much better scalability 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, exhibits marginal degradation with the increasing number of containers, and achieves nearly linear scalability. The results have demonstrated that each design component of MultiLanes is essential for scaling containers on many cores. Table 3 presents the contention details on the hot locks of the VFS that rise during the benchmark on MultiLanes without the pVFS. These locks are all eliminated by the pVFS.

It is interesting to note that the throughput of complete MultiLanes marginally outperforms that of Linux at one container on Ext3 and Ext4. This phenomenon is also observed in the below Varmail benchmark on Ext3, Ext4 and XFS. This might be because that the use of private VFS data structures provided by the pVFS speeds up the lookup in the dentry hash table as there are much less directory entries in each pVFS than in the global VFS.

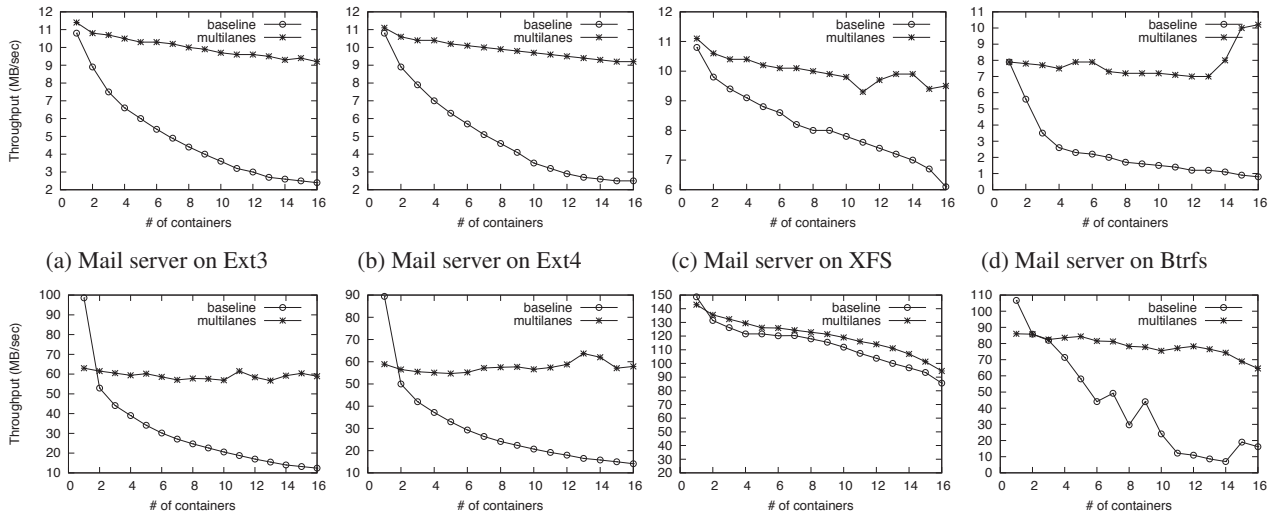
**IOzone.** We use the IOzone benchmark to evaluate the performance and scalability of MultiLanes for data-intensive workloads, including sequential and random workloads. Figure 6 shows the average throughput of each container performing sequential writes in buffered mode and direct I/O mode respectively. We run a single IOzone process inside each container in parallel and vary the number of containers. Sequential writes with 4KB I/O size are to a file that ends up with 256MB size. Note that Ext3 and Ext4 are mounted in ordered journaling mode for direct I/O writes as the data journaling mode does not support direct I/O.

| Lock | Ext3 | Ext4 | XFS | Btrfs |
|---|---|---|---|---|
| inode_hash_lock | 1092k | 960k | 114k | 228k |
| dcache_lru_lock | 1023k | 797k | 583k | 5k |
| inode_sb_list_lock | 239k | 237k | 144k | 106k |
| rename_lock | 541k | 618k | 446k | 252k |

Table 3: **Contention Bounces.** *The table shows the contention bounces using MultiLanes without pVFS.*

As shown in the figure, the average throughput of MultiLanes outperforms that of Linux in all cases except for buffered writes on XFS. MultiLanes outperforms Linux by 9.78X, 6.17X and 2.07X on Ext3, Ext4 and Btrfs for buffered writes respectively. For direct writes, the throughput improvement of MultiLanes over Linux is 7.98X, 8.67X, 6.29X and 10.32X on the four file systems respectively. XFS scales well for buffered writes owing to its own performance optimizations. Specially, XFS delays block allocation and associated metadata journaling until the dirty pages are to be flushed to disk. Delayed allocation avoids the contention induced by metadata journaling so as to scale well for buffered writes.

Figure 7 presents the results of random writes in buffered mode. Random writes with 4KB I/O size are to a 256MB file except for Btrfs. For Btrfs, we set each file size to 24MB due to the observation that when the writing data files occupy a certain proportion of the storage space Btrfs generates many work threads during the benchmark even for single-threaded random writes, which causes heavy contention and leads to sharply dropped throughput. Nevertheless, MultiLanes exhibits much better scalability and significantly outperforms the baseline at 16 containers for random writes to a 256MB file. However, in order to fairly evaluate the normal performance of both MultiLanes and Linux, we experimentally set a proper data file size for Btrfs. As shown in the figure, the throughput of MultiLanes outperforms that of Linux by 10.04X, 11.32X and 39% on Ext3, Ext4 and Btrfs respectively. As XFS scales well for buffered writes, MultiLanes exhibits competitive performance with it.




(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

Figure 8: **Scalability Evaluation with Filebench Fileserver and Varmail.** *The figure shows the average throughput of the containers on different file systems when varying the number of LXC containers, with Filebench mail server and file server workload respectively.*

### 5.2.2 Macrobenchmarks

We choose Filebench [2] and MySQL [5] to evaluation performance and scalability of MultiLanes for application-level workloads.

**Filebench.** Filebench is a file system and storage benchmark that allows to generate a variety of workloads. Of all the workloads it supports, we choose the Varmail and Fileserver benchmarks as they are write-intensive workloads that would cause severe contention within the I/O stack.

The Varmail workload emulates a mail server, performing a sequence of create-append-sync, read-append-sync, reads and deletes. The Fileserver workload performs a sequence of creates, deletes, appends, reads, and writes. The specific parameters of the two workloads are listed in Table 4. We run a single instance of Filebench inside each container. The thread number of each instance is configured as 1 to avoid CPU overload when increasing the number of containers from 1 to 16. Each workload was run for 60 seconds.

| Workload | # of Files | File Size | I/O Size | Append Size |
|---|---|---|---|---|
| Varmail | 1000 | 16KB | 1MB | 16KB |
| Fileserver | 2000 | 128KB | 1MB | 16KB |

Table 4: **Workload Specification.** *This table specifies the parameters configured for Filebench Varmail and Fileserver workloads.*

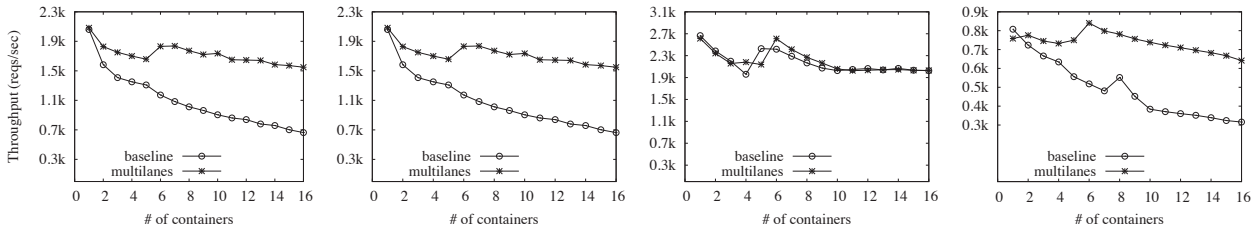
Figure 8 shows the average throughput of multiple concurrent Filebench instances on MultiLanes compared to Linux. For the Varmail workload, the average throughput degrades significantly with the increasing number of containers on the four file systems in Linux. MultiLanes exhibits little overhead when there is only one container, and marginal performance loss when the number of containers increases. The throughput of MultiLanes outperforms that of Linux by 2.83X, 2.68X, 56% and 11.75X on Ext3, Ext4, XFS and Btrfs respectively.

For the Fileserver workload, although the throughput of MultiLanes is worse than that of Linux at one single container, especially for Ext3 and Ext4, it scales well to 16 containers and outperforms that of Linux when the number of containers exceeds 2. In particular, MultiLanes achieves a speedup of 4.75X, 4.11X, 1.10X and 3.99X over the baseline Linux on the four file systems at 16 containers respectively. It is impressive that the throughput of MultiLanes at 16 containers even exceeds that at one single container on Btrfs. The phenomenon might relate to the design of Btrfs which is under actively development and does not become mature.

**MySQL.** MySQL is an open source relational database management system that runs as a server providing multi-user accesses to databases. It is widely used for data storage and management in web applications.

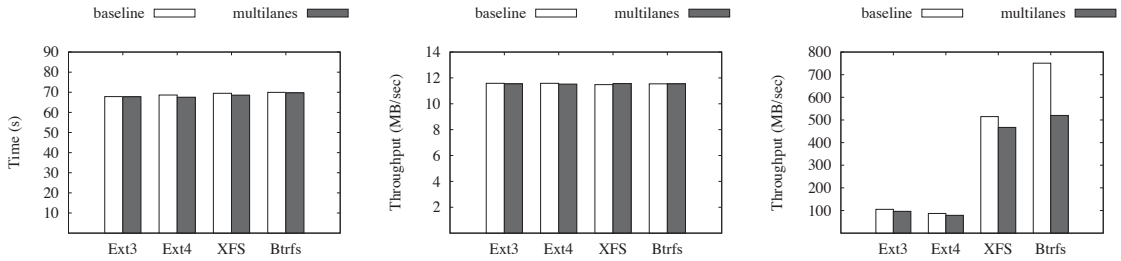
We install mysql-server-5.1 for each container and start the service for each of them. The virtualized MySQL servers are configured to allow remote accesses and we generate requests with Sysbench [7] on another identical machine that resides in the same LAN with the experimental server. The evaluation is conducted in non-transaction mode that specializes update_key operations as the transaction mode provided by Sysbench is dominated by read operations. Each table is initialized with 10k records at the prepare stage. We use 1 thread to generate 20k requests for each MySQL server.

As Figure 9 shows, MultiLanes improves the throughput by 87%, 1.34X and 1.03X on Ext3, Ext4, and Btrfs




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

Figure 9: **Scalability Evaluation with MySQL.** *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 on another identical machine in the same LAN.*




(a) Apache Build (b) Webserver (c) Streamwrite

Figure 10: **Overhead Evaluation.** *This figure shows the overhead of MultiLanes relative to Linux, running Apache build, Filebench Webserver and Filebench single-stream write inside a single container respectively.*

respectively. And once again we have come to see that XFS scales well on many cores, and MultiLanes shows competitive performance with it. The throughput of MultiLanes exhibits nearly linear scalability with the increasing number of containers on the four file systems.

## 5.3 Overhead Analysis

We also measure the potential overhead of MultiLanes's approach to eliminating contention in OS-level virtualization by using an extensive set of benchmarks: Apache Build, Webserver and Streamwrite, which is file I/O less intensive, read intensive and write intensive respectively.

**Apache Build.** The Apache Build benchmark, which overlaps computation with file I/O, unzips the Apache source tree, does a complete build in parallel with 16 threads, and then removes all files. Figure 10a shows the execution time of the benchmark on MultiLanes over Linux. We can see that MultiLanes exhibits almost equivalent performance against Linux. The result demonstrates that the overhead of MultiLanes would not affect the performance of workloads which are not dominated by file I/O.

**Webserver.** We choose the Filebench Webserver workload to evaluate the overhead of MultiLanes under read-intensive workloads. The parameters of the benchmark is configured as default. Figure 10b presents the throughput of MultiLanes against Linux. The result shows that the virtualization layer of MultiLanes contributes marginally to the performance under the Webserver workload.

**Streamwrite.** The single-stream write benchmark performs 1MB sequential writes to a file that ends up with about 1GB size. Figure 10c shows the throughput of benchmark on MultiLanes over Linux. As the sequential stream writes cause frequent block allocation of the back-end file, MultiLanes incurs some overheads of block mapping cache misses. The overhead of MultiLanes compared to Linux is 9.0%, 10.5%, 10.2% and 44.7% for Ext3, Ext4, XFS and Btrfs respectively.

# 6 Related Work

This section relates MultiLanes to other work done in performance isolation, kernel scalability and device virtualization.

**Performance Isolation.** Most work on performance isolation mainly focuses on minimizing performance interference by space partitioning or time multiplexing hardware resources (e.g., CPU, memory, disk and network bandwidth) between the co-located containers. VServer [32] enforces resource isolation by carefully allocating and scheduling physical resources. Resource containers [10] provides explicit and fine-grained control over resource consumption in all levels in the system. Eclipse [14] introduces a new operating system ab-

straction to enable explicit control over the provisioning of the system resources among applications. Software Performance Units [35] enforces performance isolation by restricting the resource consumption of each group of processes. Cgroup [1], which is used in LXC to provide resource isolation between co-located containers, is a Linux kernel feature to limit, account and isolate the resource usage of process groups. Argon [36] mainly focuses on the I/O schedule algorithms and the file system cache partition mechanisms to provide storage performance isolation.

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

**Kernel Scalability.** Improving the scalability of operating systems has been a longstanding goal of system researchers. Some work investigates new OS structures to scale operating systems by partitioning the hardware and distributing replicated kernels among the partitioned hardware. Hive [17] structures the operating system as an internal distributed system of independent kernels to provide reliability and scalability. Barrelfish [11] tries to scale applications on multicore systems using a multi-kernel model, which maintains the operating system consistency by message-passing instead of shared-memory. Corey [12] is an exokernel based operating system that allows applications to control the sharing of kernel resources. K42 [9] (and its relative Tornado [20]) are designed to reduce contention and improve locality on NUMA systems. Other work partitions hardware resources by running a virtualization layer to allow the concurrent execution of multiple commodity operating systems. For instance, Diso [15] (and its relative Cellular Diso [22]) runs multiple virtual machines to create a virtual cluster on large-scale shared-memory multiprocessors to provide reliability and scalability. Cerberus [33] scales shared-memory applications with POSIX-APIs on many cores by running multiple clustered operating systems atop VMM on a single machine. MultiLanes is influenced by the philosophy and wisdoms of these work but strongly foucses on the scalability of I/O stack on fast storage devices.

Other studies aim to address the scalability problem by iteratively eliminating the bottlenecks. MCS lock [28], RCU [27] and local runqueues [8] are strategies proposed to reduce contention on shared data structures.

**Device Virtualization.** Traditionally, hardware abstraction virtualization adopts three approaches to virtualize devices. First, device emulation [34] is used to emulate familiar devices such as common network cards and SCSI devices. Second, para-virtualization [30] customizes the virtualized device driver to enable the guest OS to explicitly cooperate with the hypervisor for performance improvements. Such examples include KVM's VirtIO driver, Xen's para-virtualized driver, and VMware's guest tools. Third, direct device assignment [21, 19, 26] gives the guest direct accesses to physical devices to achieve near-native hardware performance.

MultiLanes maps a regular file as the virtualized device of a VE rather than giving it direct accesses to a physical device or a logical volume. The use of backend files eases the management of the storage images [24]. 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.

## 7 Conclusions

The advent of fast storage technologies has shifted the I/O bottlenecks from the storage devices to system software. The co-located containers in OS-level virtualization will suffer from severe storage performance interference on many cores due to the fact that they share the same I/O stack. In this work, we propose MultiLanes, which consists of the virtualized storage device, and the partitioned VFS, to provide an isolate I/O stack to each container on many cores. The evaluation demonstrates that MultiLanes effectively addresses the I/O performance interference between the VEs on many cores and exhibits significant performance improvement compared to Linux for most workloads.

As we try to eliminate contention on shared data structures and locks within the file system layer with the virtualized storage device, the effectiveness of our approach is based on the premise that multiple file system instances work independently and share almost nothing. For those file systems in which the instances share the same worker thread pool (e.g., JFS), there might still exist performance interference between containers.

## 8 Acknowledgements

We would like to thank our shepherd Anand Sivasubramaniam and the anonymous reviewers for their excellent feedback and suggestions. This work was funded by China 973 Program (No.2011CB302602), China 863 Program (No.2011AA01A202, 2013AA01A213), HGJ Program (2010ZX01045-001-002-4) and Projects from NSFC (No.61170294, 91118008). Tianyu Wo and Chunming Hu are the corresponding authors of this paper.

## References

[1] Cgroup. https://www.kernel.org/doc/Documentation/cgroups.

[2] Filebench. http://sourceforge.net/projects/filebench/.

[3] IOzone. http://www.iozone.org/.

[4] LXC. http://en.wikipedia.org/wiki/LXC.

[5] MySQL. http://www.mysql.com/.

[6] OpenVZ. http://en.wikipedia.org/wiki/OpenVZ.

[7] Sysbench. http://sysbench.sourceforge.net/.

[8] AAS, J. Understanding the Linux 2.6.8.1 CPU scheduler. http://josh.trancesoftware.com/linux/.

[9] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M. A., OSTROWSKI, M., ROSENBURG, B. S., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst. 25*, 3 (2007).

[10] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource Containers: A new facility for resource management in server systems. In *OSDI* (1999).

[11] BAUMANN, A., BARHAM, P., DAGAND, P.-É., HARRIS, T. L., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (2009).

[12] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., HUA DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *OSDI* (2008).

[13] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *OSDI* (2010).

[14] BRUNO, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. The Eclipse operating system: Providing quality of service via reservation domains. In *USENIX Annual Technical Conference* (1998).

[15] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[16] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO* (2010).

[17] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *SOSP* (1995).

[18] CUI, Y., WANG, Y., CHEN, Y., AND SHI, Y. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *TACO 9*, 4 (2013), 44.

[19] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[20] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI* (1999).

[21] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ASPLOS* (2012).

[22] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP* (1999).

[23] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer* (1986).

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

[25] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *ISCA* (2009).

[26] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Euro-Par Workshops* (2007).

[27] MCKENNEY, P. E., SARMA, D., ARCANGELI, A., KLEEN, A., KRIEGER, O., AND RUSSELL, R. Read-copy update. In *Linux Symposium* (2002).

[28] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (1991), 21–65.

[29] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *OSDI* (2002).

[30] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review 42*, 5 (2008), 95–103.

[31] SEPPANEN, E., O'KEEFE, M. T., AND LILJA, D. J. High performance solid state storage under Linux. In *MSST* (2010).

[32] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A. C., AND PETERSON, L. L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys* (2007).

[33] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with OS clustering. In *EuroSys* (2011).

[34] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001).

[35] VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *ASPLOS* (1998).

[36] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *FAST* (2007).