Experience Paper: Danaus: Isolation and Efficiency of Container I/O at the Client Side of Network Storage

Giorgos Kappes
University of Ioannina
Ioannina, Greece
gkappes@cse.uoi.gr

Stergios V. Anastasiadis
University of Ioannina
Ioannina, Greece
stergios@cse.uoi.gr

ABSTRACT

Containers are a mainstream virtualization technique commonly used to run stateful workloads over persistent storage. In multi-tenant hosts with high utilization, resource contention at the system kernel often leads to inefficient handling of the container I/O. Assuming a distributed storage architecture for scalability, resource sharing is particularly problematic at the client hosts serving the applications of competing tenants. Although increasing the scalability of a system kernel can improve resource efficiency, it is highly challenging to refactor the kernel for fair access to system services. As a realistic alternative, we isolate the storage I/O paths of different tenants by serving them with distinct clients running at user level. We introduce the Danaus client architecture to let each tenant access the container root and application filesystems over a private host path. We developed a Danaus prototype that integrates a union filesystem with a Ceph distributed filesystem client and a configurable shared cache. Across different host configurations, workloads and systems, Danaus achieves improved performance stability because it handles I/O with reserved per-tenant resources and avoids intensive kernel locking. Danaus offers up to 14.4x higher throughput than a popular kernel-based client under conditions of I/O contention. In comparison to a FUSE-based user-level client, Danaus also reduces by 14.2x the time to start 256 high-performance webservers. Based on our extensive experience from building and evaluating Danaus, we share several valuable lessons that we learned about resource contention, file management, service separation and performance stability.

CCS CONCEPTS

• **Software and its engineering** → *Cooperating communicating processes*; • **Information systems** → **Storage virtualization**; **Distributed storage**; • **Computing methodologies** → **Distributed computing methodologies**; • **General and reference** → **Performance**.

KEYWORDS

cloud, virtualization, multitenancy, user-level services, client-server architectures, distributed filesystems

Middleware '21, December 6–10, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3493390

ACM Reference Format:
Giorgos Kappes and Stergios V. Anastasiadis. 2021. Experience Paper: Danaus: Isolation and Efficiency of Container I/O at the Client Side of Network Storage. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3464298.3493390

1 INTRODUCTION

Containers are popular in the cloud landscape because they offer flexible virtualization with low overhead. Although their benefits follow from the native resource management of the operating system, the improved isolation of the container execution often requires hardware-level virtualization or rule-based control [7, 20, 50, 60]. In particular, the persistent storage of container images and data through the system kernel introduces several isolation and efficiency issues. Examples include the suboptimal performance [36, 70, 79], security [24, 28, 78] and fault-containment [47, 55], or the excessive use of processing, memory and storage resources [32, 46, 60, 76].

The containers cannot isolate the I/O performance of competing workloads (e.g., noisy neighbor [45, 60]) because they share a common kernel path for the critical I/O operations of request routing [36] (e.g., VFS), page cache [51, 61], filesystem [77], and device scheduling [15]. Additionally, the security isolation and fault containment of the colocated containers are limited due to the potential vulnerabilities from the shared operating system [24, 28, 50, 78].

For backward compatibility to standard interfaces (e.g., POSIX), the existing filesystems typically run partly or fully inside the kernel. As a result, they experience inefficiencies for several reasons: first, the processor cache pollution from memory copy operations; second, the direct and indirect costs (e.g., TLB flushes) from mode and context switches [45]; third, the double caching inside and outside the kernel (e.g., FUSE [69]); and fourth, the inaccurate accounting of the shared kernel resources (e.g., page cache [82]).

Additional inefficiencies arise in cloned containers running on the same host. Indeed, their concurrent execution introduces duplication in the utilized memory and storage space, or the memory and I/O bandwidth. The multi-layer union filesystems and block-based copy-on-write snapshots reduce the waste of storage space. Yet, they do not always prevent the duplication of memory space and memory or I/O bandwidth at the host [14, 32, 76]. Therefore, the resource contention and the inflexible sharing of the system kernel reduces the container I/O isolation and increases the resource duplication. Current systems explicitly reserve resources for user-level execution, but they are blurry in the allocation and fair access of the kernel resources. Given the undesirable variability consequences, we argue that the effective container isolation requires two types of system support: *first, the explicit allocation of the hardware resources*

utilized at both the user and kernel level; and second, the fair access to the kernel operations and data structures.

A library operating system (libOS) reduces the isolation dependence from the kernel resource management because it shifts functionality from the kernel to the user level. Yet, the typical linking of a libOS to a single process complicates the multiprocess state sharing of typical applications [29, 33, 37, 67, 75]. In a similar sense, a filesystem provides flexible resource management when it is accessed through the kernel but runs at user level. However, it sacrifices performance and efficiency due to the extra data copying and mode or context switching (e.g., FUSE [69]).

The container isolation and resource efficiency can be addressed with improved scalability in the state management of the system kernel. Nevertheless, refactoring the kernel for scalability is a long-standing challenging problem [23]. As a pragmatic alternative, we relocate to user level both the system functionality and the interprocess communication. *Accordingly, we let each process group stick to their reserved hardware resources and run their own system services at user level.* Our approach can be beneficial for a broad range of system components, including the local and remote access to different types of data storage, or the network communication.

The network storage client is a critical component of the end-to-end path used by a scalable storage system to serve the cloud infrastructure. Consequently, our present focus is to architect a network storage client that provides I/O isolation and efficiency across the containers of different colocated tenants. We are not getting into the multitenant isolation of the network itself or the storage servers because they introduce their own challenges beyond the scope of the present work [16, 40, 47].

The key idea of Danaus is to *provision a distinct user-level filesystem client per tenant on a host.* We construct each client from a private stack of user-level functionalities based on the abstraction of libservices [38]. A client runs its own code path and data cache privately on the reserved resources of the tenant. We pin the client threads on reserved cores and run them at user level to reduce the mode and context switches. Furthermore, the cloned or collaborating containers take advantage of configurable filesystem sharing to avoid the duplication of memory and storage resources.

We built a Danaus prototype from two libservices running a union filesystem and a distributed filesystem client. We experimentally demonstrate that a mature kernel-based client and union filesystem lead to low performance due to lock contention inside the kernel and I/O handling with resources exceeding those reserved. In contrast, Danaus achieves higher performance combined with low memory and cpu utilization, especially in write-intensive or scaleout workloads. Although Danaus is slower in cached read, we identified a potential solution for performance improvement through a concurrency optimization of the user-level distributed filesystem client.

Our contributions include:

(1) Provide motivation about the critical problem of I/O isolation and efficiency; demonstrate the software and hardware contention caused in I/O operations served by the kernel.
(2) Set the design goals and principles for a unified client architecture of multitenant container storage.

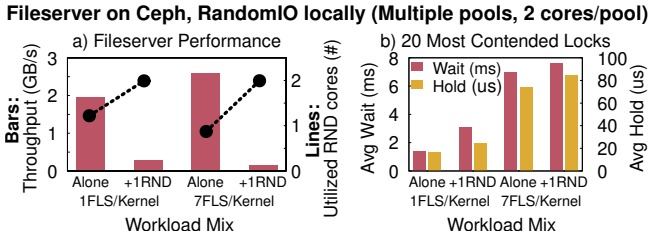

Figure 1: Dramatic drop of the Fileserver throughput due to the core (a) and lock (b) contention inside the shared host kernel.

(3) Justify the design decisions for the system interface, client cache, consistency properties and interprocess communication.
(4) Explain the Danaus prototype components that integrate the union and client libservices with a configurable cache for space and bandwidth efficiency.
(5) Analyze and justify the realization of the Danaus goals.
(6) Experimentally quantify the improved isolation, high performance and low resource consumption of Danaus in comparison with representative systems using production applications and microbenchmarks.
(7) Present insightful lessons learned from our experience with the design, implementation and evaluation of Danaus.

Next we provide experimental motivation and research background (§2), describe the design (§3) and implementation of Danaus (§4), conduct qualitative analysis (§5) and comparative performance evaluation (§6), outline and differentiate the related work (§7), discuss the limitations and future work (§9) before summarizing our conclusions (§10).

2 MOTIVATION AND BACKGROUND

In the present section, we experimentally demonstrate the application performance sensitivity to the kernel I/O contention and outline the existing container storage options.

2.1 Sensitivity to kernel I/O contention

We examine the performance variation of multiple data-intensive workloads colocated over the same host [36, 82]. We run the Fileserver (FLS) of Filebench [66] over a storage cluster accessed through the CephFS kernel-based client (setup in §6.1, §6.2, Table 2). Alternatively, we use the Stress-ng [13] RandomIO (RND) benchmark to generate random I/O over a local kernel-based filesystem at direct-attached devices. Each workload instance runs on a distinct reservation (*container pool*) of 2 cores and 8GB RAM. We activate 4 or 16 cores of the host to respectively run 1 or 7 instances of FLS (1FLS, 7FLS) alone or next to 1 RND.

At the bar chart of Fig. 1a, we observe a dramatic performance drop when FLS is colocated with RND. The FLS throughput drops 7.4x from 1FLS to 1FLS+1RND, and 16.5x from 7FLS to 7FLS+1RND. We attribute this behavior to two reasons. First, the system kernel utilizes the activated cores of the entire host to flush dirty pages. The line chart of Fig. 1a confirms that the cores of RND are utilized 122% by 1FLS and 87% by 7FLS, when either runs alone. With

the RND also running, the FLS throughput plummets because the kernel cannot "steal" any more the cores of RND to serve FLS. The second reason is the locking activity inside the kernel. In Fig. 1b, we show the average wait and hold time that the kernel spends *per lock request*. It is notable that the lock wait time of 1FLS grows 2.3x in 1FLS+1RND and 5.2x in 7FLS. Therefore, the colocation of 1FLS with either 1 RND or 6 additional FLS instances increases the locking overhead by several factors.

Our takeaway is that the colocated workloads should use their own reusable (hardware) and consumable (software) resources in order to isolate their I/O activity and performance.

2.2 Containers and existing storage options

Containers are a lightweight virtualization abstraction provided by the operating system of a host to isolate a group of processes. They are typically supported by a userspace runtime, and kernel control mechanisms for resource allocation (e.g., cpu, memory, network, and block I/O) and isolation (e.g., process IDs, user IDs, mount points, network, and IPC) [19].

An *image* is a read-only set of application binaries and system packages organized as a stacked series of file archives (*layers*) [32]. It is made available from a registry running on an independent machine. A new container is typically created on the local storage of the host. The root filesystem is prepared by copying an image and expanding it into a file tree under a private directory. An additional writable layer is added over the expanded image to enable file modifications by the container execution. A storage driver [2] of the container runtime allows sharing image layers across different containers through a union filesystem or snapshot storage.

A union filesystem offers a single logical view from multiple stacked directories (*branches*) supporting copy-on-write at the file level [56]. The upper branch can be writable, while the remaining branches are read-only and shareable. A file lookup from the top stops at the first branch that contains either the file or a whiteout entry marking the file as deleted. The branches are stored on a local filesystem, or a distributed filesystem for scalability [81]. Snapshot storage provides block-level copy-on-write. It is implemented by either a local filesystem, a local block device manager [76], or a remote block volume on network-attached storage [32].

For the storage of application data, a container can mount additional filesystems from the host (e.g., bind mount) or the network (e.g., volume plugin) [2, 5]. The host provides persistence over a local or distributed filesystem. Alternatively, a volume plugin can mount from network storage either a filesystem or a block volume formatted with a local filesystem (e.g., MS Azure [10], Amazon [6] EFS, EBS). The volume plugin is executed by the container runtime *with the necessary support from the kernel* (e.g., NFS kernel module). Finally, a container application may access cloud object storage (e.g., Amazon S3) through a RESTful API.

A distributed filesystem provides persistent storage through a client running at the host. The client memory maintains a file data cache, an inode cache, and a directory entry cache. Dirty data is flushed to network storage when it exceeds a threshold that limits the amount or age of dirty data, or the count of cached objects [4, 9, 74]. Applications access a filesystem with a system call that involves I/O directly (e.g., open) or indirectly (e.g., exec).

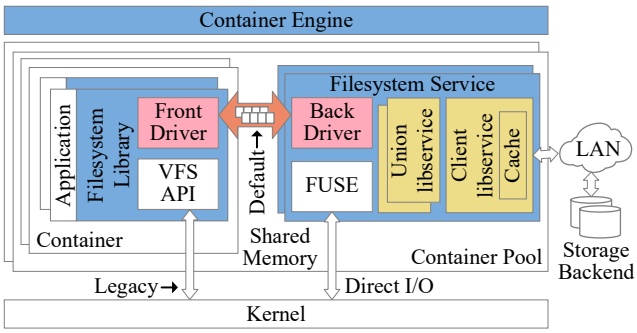

Figure 2: Architecture of the Danaus user-level system. The color shades illustrate the new components of Danaus and the white background the components of existing systems.

The I/O requests are served inside the kernel or redirected from the kernel to a user-level process (e.g., FUSE). Another option is to link an application to a library and implement the I/O functionality at user level, either as part of the application or at a different process over IPC [53, 71].

3 DESIGN

Below we describe the critical parts of Danaus and justify our choices over alternatives. Our presentation includes the goals and principles, the basic structure, the interfaces, the cache, the consistency and the interprocess communication.

3.1 System structure

In the design of our system, we set the following goals:

(1) **Compatibility** Provide native container access to scalable persistent storage through a backward-compatible POSIX-like interface (e.g., open, fork, exec).
(2) **Isolation** Improve the performance isolation and the fault containment of data-intensive tenants colocated on the same client machine.
(3) **Efficiency** Serve the root and application filesystems of containers through the effective utilization of the processor, memory and storage resources.
(4) **Flexibility** Enable flexible tenant configuration of the sharing and caching policies (e.g., files, consistency).

A *container pool* (or *pool*) comprises the application and service containers of a tenant on a host (Fig. 2). We manage the colocated container pools with a *container engine* per host deployed by the cloud provider. We store the container *root* and *application* filesystems directly on the shared distributed filesystem. We focus on the file-based network storage for the following two reasons:

- First, it allows flexible data sharing among the hosts (§5), and
- Second, it avoids the multilayer virtualization overheads of block-based storage [42, 64].

The *storage backend* consists of the server nodes that store the data and metadata of the distributed filesystem. Our architecture can also support clients of block-based network storage but we leave for future work the exploration of that direction.

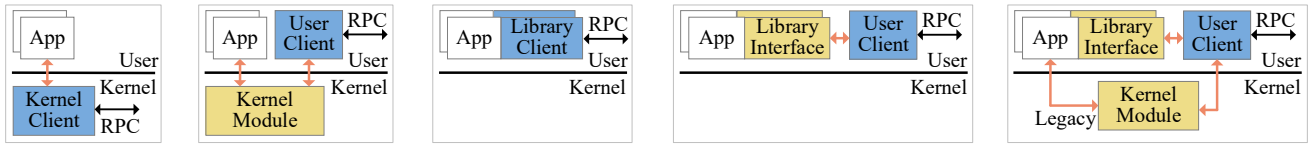

Figure 3: Client types. Three existing types of filesystem clients (a-c) and two that we introduce with Danaus (d-e).

The Danaus architecture integrates the collection of filesystem functionalities that serve the container pool of a tenant. The *backend client* is the distributed filesystem client that the containers use to access the storage backend. The *union filesystem* offers I/O deduplication across the container clones or dataset replicas. The *cache* provides caching over the root and application filesystems with configurable sharing and consistency semantics.

The Danaus architecture consists of the *filesystem library* linked to each application, the *filesystem services* that handle the storage I/O, and the interprocess communication that connects the applications with the services (Fig. 2). The *front driver* of the filesystem library passes the incoming requests at user level to the *back driver* of a filesystem service over shared memory. A *filesystem service* is a standalone user-level process that runs the *filesystem instances* mounted to the containers. A filesystem instance is implemented as a stack of libservices. A libservice is a user-level storage subsystem accessed through a POSIX-like interface. Our prototype implementation currently supports the *union libservice* of the union filesystem and the *client libservice* of the backend client. The union libservice invokes directly the client libservice to serve the branch requests without extra context switching or data copying.

In order to achieve our goals, our design is based on the following four principles:

(1) **Dual interface** Support common application I/O calls through the filesystem library, and only use the kernel for legacy software or system calls with implicit I/O.
(2) **Filesystem integration** The libservices of a filesystem instance interact directly with each other through function calls for speed and efficiency.
(3) **User-level execution** The filesystem service and the default communication path both run at user level for isolation, flexibility and reduced kernel overhead.
(4) **Path isolation** A filesystem service isolates the storage I/O of a container pool at the client side in order to reduce the interference among the colocated pools of the same or different tenants and improve their flexibility.

Below we describe the critical parts of our design and justify our choices with respect to considered alternatives.

3.2 Interface

The interface of the backend client is a critical part that we specified after considering the following five options:

(1) **Full Kernel** Run the client inside the kernel (e.g., Ceph, NFS) for performance and access it through the kernel (e.g., VFS) for compatibility (Fig. 3a).
(2) **User Execution** Run the client at user level for easy prototyping (e.g., FUSE) and access it through the kernel (Fig. 3b).
(3) **Linked Library** Link a library client directly to each application (e.g., Ceph [74], Gluster [4], NFS, Hare [31]) for improved efficiency at the cost of extra complexity in multiprocess cache sharing (Fig. 3c).
(4) **Full User** Run the client at user level as a standalone process and let the applications access it over shared memory through a library (Fig. 3d). This approach combines isolation and efficiency with intra-tenant cache sharing.
(5) **Dual Interface** For improved backward compatibility of legacy applications, we complement the full-user interface (described above) with a second optional interface passing through the kernel (Fig. 3e).

Danaus supports the *dual interface*. The default path connects the application with the filesystem service over user level. An application preloads the filesystem library to override the I/O functions of libc. If the source code can be recompiled, the application directly invokes the Danaus file I/O functions prefixed with `danaus_`. The shared memory between the applications and the filesystem service is isolated in the private IPC namespace of the container pool [19]. We manage the shared memory by applying the System V IPC interface instead of the mmap POSIX interface that crosses the kernel I/O path (e.g., shm_open via VFS).

The dynamic linker with preloading cannot override the statically-linked symbols. Therefore, if an application binary includes statically linked symbols or system calls with kernel-initiated I/O (e.g., exec), Danaus allows the I/O requests to automatically enter the kernel (through VFS). Subsequently, the requests arrive back at user level to the filesystem service. We use the high-level API of FUSE [69] to process these requests over dedicated FUSE threads (Fig. 2).

3.3 Client cache

We faced the dilemma of placing the client cache at either the filesystem library or the filesystem service. Managing the cache at the library side would complicate the coordination of the cache sharing across the container processes of the tenant. The complexity results from the threads of a tenant directly accessing the cache and synchronizing with each other to keep consistent the cache structures. In contrast, the filesystem service can use its own threads to run a natively shared cache inside the union filesystem or the backend client. A third possibility is to run the cache as a distinct libservice above the backend client.

We observed that the backend client already operates its own local cache with consistency to the storage backend. Thus, we

decided to operate the client cache inside the backend client at user level and run the union deduplication without cache on top of the backend client (Fig. 2). The union filesystem does not create separate file data structures (e.g., inodes) because it interacts with the backend client via function calls at the file level. The backend client recognizes a shared file from its pathname and holds in the client cache a single copy of each shared branch.

3.4 Consistency

The client cache is the first stateful part in the I/O path from the application to the storage backend. When an application write returns, it has certainly reached the client cache and probably the storage backend. Thus, it will be visible from a subsequent read to the same backend client. The filesystem consistency policy propagates the write to other backend clients at the same or different host. The synchronous requests are served in thread program order by the filesystem service. Instead, the concurrent or asynchronous requests can meet specific ordering requirements through application-level synchronization. At a client crash, the loss of cached state depends on the consistency policy of the distributed filesystem. The application will need to repeat the request if is not notified for completion, or the request is lost during the crash.

3.5 Interprocess communication

The filesystem library communicates with the filesystem service using fixed-size circular queues in shared memory. Each queue is concurrently accessed by multiple producer and consumer threads of the front and back driver, respectively. The processor hardware typically organizes the cores in groups with shared same-level cache (e.g., pair over L2). Accordingly, we maintain a separate request queue per core group to facilitate the efficient communication between the application container and the filesystem service running on the same core group. Each queue entry contains a request descriptor and a state field tracking whether the entry is currently empty, under modification, or it carries a valid request. The request descriptor contains a call identifier, an array of small arguments, and a pointer to a *request buffer*. Each application thread has a separate request buffer, which is used by the front and back driver to exchange large data items over shared memory.

Each new thread of an application first runs on a core picked by the system scheduler. Subsequently, the front driver pins the thread to the cores of the request queue that receives the first I/O request from the thread. Thus, we minimize the possible thread migrations and cache-line bouncing. The back driver initiates a default number of service threads equal to the number of request queues. Each service thread is pinned to the cores of the request queue that it serves. Extra service threads are added if the number of pending I/O requests in a request queue exceeds a threshold.

4 IMPLEMENTATION

The Danaus prototype consists of the filesystem library, the filesystem service, the interprocess communication, and the container engine. We developed the libservices of the Danaus prototype in C code based in part on the libcephfs client of Ceph (v10.2.7) [74] and the unionfs-fuse filesystem [54]. The total development effort required the addition of 19,484 new lines of code, modification of 2,115 and removal of 1,245.

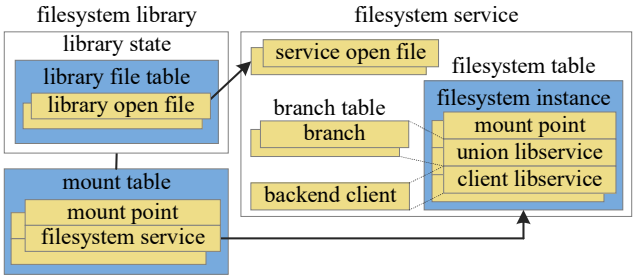

Figure 4: The data structures of the Danaus system.

4.1 Data structures

We run a distinct root filesystem per container and access the application data from private or shared filesystems. The container filesystems are served by the mount namespace of the system kernel or by the filesystem services of Danaus (Fig. 4). A mounted filesystem is identified by a unique path in the root filesystem of the host (*mount point*). The *mount table* in a container pool maps a mount point to a filesystem service. The *filesystem table* in the corresponding filesystem service maps the mount point to a filesystem instance. The filesystem instance specifies the libservices of the union filesystem and the backend client. The *branch table* in a union libservice specifies for each branch the properties of the top directory (path, access mode, file descriptor). For each open file of the filesystem service, the back driver creates a *service open file* object. The memory address of the object is returned to the front driver as *service file descriptor*.

The *library state* of the filesystem library stores the private filesystem state of a process (e.g., root directory). Danaus and the kernel assign file descriptors independently to the open files of a process. We avoid possible conflicts by specifying a distinct file descriptor per open file. For each such file descriptor, there is a unique entry, called *library open file*, in the data structure, called *library file table*. We return the entry index as private file descriptor to the process. When a file descriptor is missing from the library file table or a path is missing from the mount table, the library passes the request to the kernel as system call. We overload the library open files to also access the network sockets and directory streams using mechanisms similar to the above.

4.2 Filesystem library and service

The filesystem library implements most of the synchronous and asynchronous file I/O calls, along with several management calls for processes, threads, sockets and pipes. We implemented the union libservice as a library derived from the unionfs-fuse filesystem. We modified the I/O functions of unionfs-fuse to accept a filesystem instance as parameter and invoke a client libservice instead of a local filesystem. We derived the client libservice from the libcephfs library that supports the user-level object cache for data and metadata caching in memory. We unify the interface of the union and client libservice through function wrappers accepting a

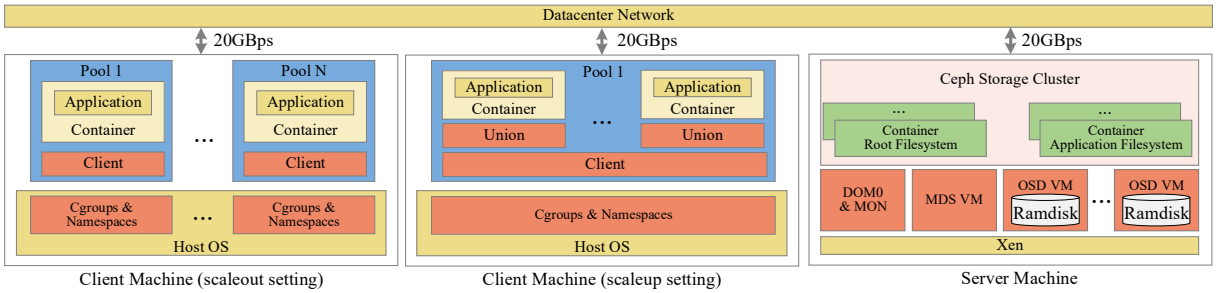

Figure 5: The testbed on which we deployed Danaus for the experimental evaluation. At the client machine we alternatively use a scaleout or a scaleup setting. The server machine is configured as a Ceph storage cluster with 7 virtual machines and the host (Domain 0) over Xen.

filesystem instance as parameter. For undefined union filesystem, the applications use the function wrappers to directly access the client libservice.

4.3 Container engine

The Danaus container engine is a user-level daemon that manages the container pools of a host. A cgroup restricts the resource usage and a namespace isolates the processes of a container pool or an individual container. The cpuset (cgroup v1) specifies the processor cores and memory nodes on which a process group runs, while the memory subsystem (cgroup v2) limits the respective memory usage. The container engine supports the combination of a union filesystem and a backend client running at the user or kernel level. Apart from the Danaus user-level filesystem, we optionally support:

- As backend client, the kernel-based CephFS client and the FUSE-based ceph-fuse,
- As union filesystem, the kernel-based AUFS and the FUSE-based unionfs-fuse (fairly comparable as both are derived from Unionfs)[1].

Each kernel-based mount (e.g., through VFS) corresponds to a different kernel filesystem instance along with a different user-level process in case of FUSE [68]. The kernel-based filesystems use the system page cache by default with optional disabling through direct I/O (e.g., avoid double caching in FUSE).

5 DISCUSSION

We qualitatively analyze how the design and implementation of Danaus realize our goals following our principles (§3).

Compatibility The containers mount the root and application filesystems of Danaus from a shared distributed filesystem and access them directly through a POSIX-like interface.

Isolation A container pool obtains a set of namespaces potentially distinct from those of the host [24, 25, 43, 63]. Correspondingly, a filesystem service runs on the hardware resources of the container pool rather than those of the shared kernel. Through the user-level execution, Danaus improves the multitenant I/O isolation and reduces the contention at processor cores, kernel locks and memory caches. The host kernel is a single point of failure with attack surface inherited to all the colocated containers. Yet, the execution of the filesystem and cache at user level involves thinner kernel interface and results into smaller trusted computing base [18]. For fault containment, Danaus decomposes the filesystem into separate user-level services with possibly distinct implementation per pool rather than fixed by the host kernel. A failed filesystem service affects the processes of a single pool but not the system kernel or the host root filesystem.

Efficiency At the server side, the *shared* network filesystem naturally prevents the storage space duplication of shared application and system files. At the client side, the client cache serves the shared files without duplication of memory space and network bandwidth. The integration of the union filesystem with the backend client prevents resource duplication and page-cache crossing for their interaction. The union filesystem deduplicates the memory space and bandwidth when serving common I/O requests across cloned containers. The user-level interaction of the application with the filesystem service avoids costly memory copies and mode switches through the kernel. In kernel-based filesystems, a typical call enters the kernel once or multiple times. On the contrary, Danaus only involves the kernel in network processing and thread scheduling, in the common case.

Flexibility A tenant uses a Danaus backend client on a host to let the containers share files or branches in read-only or writable mode. A tenant can use *multiple* filesystem services with distinct settings in resource naming, memory reservation, page replacement (e.g., swappiness) or data consistency (e.g., writeback) [51]. Multiple tenants can collaborate through a shared pool. Casual administration tasks (e.g., software updates, malware scanning, container migration) can conveniently run centrally through the backend storage rather than separately inside the individual containers.

6 EXPERIMENTAL EVALUATION

We quantify the ability of different filesystem clients to

(1) Prevent I/O contention between homogeneous or heterogeneous resource-demanding workloads (§6.2),
(2) Achieve high performance at low resource consumption across real-world applications and microbenchmarks in scaleup and scaleout settings with read or write and sequential or random I/O requests (§6.3).

[1]We do not support *OverlayFS* because it does not run over remote filesystems (e.g., CephFS) unless caching is completely disabled (*N=0* [1]).

Summary of Configurations

Symbol	Union Filesystem		Backend Client	
	System	*Cache*	*System*	*Cache*
D	Danaus (opt.)		Danaus	UlcC
K			CephFS	PagC
F			ceph-fuse	UlcC
FP			ceph-fuse	UlcC+PagC
K/K	AUFS	PagC	CephFS	PagC
F/K	unionfs-fuse		CephFS	PagC
F/F	unionfs-fuse		ceph-fuse	UlcC
FP/FP	unionfs-fuse	PagC	ceph-fuse	UlcC+PagC

Table 1: Client system components. UlcC: user-level client cache; PagC: system kernel page cache.

Summary of Workload Symbols

Symbol	Description
FLS	Fileserver (Filebench) on Ceph
RND	Random I/O with readahead (Stress-ng) on ext4/RAID0
SSB	CPU benchmark (Sysbench)
WBS	Webserver (Filebench) on ext4/RAID0
1FLS/D	1x Fileserver on user-level Danaus/Ceph cluster
7FLS/D	7x Fileserver on user-level Danaus/Ceph cluster
1FLS/K	1x Fileserver on kernel CephFS/Ceph cluster
7FLS/K	7x Fileserver on kernel CephFS/Ceph cluster
X+Y	X next to Y, X=(1|7)FLS/(D|K), Y=(RND|SSB|WBS)

Table 2: Contention Workloads (§2.1,§6.2).

6.1 Experimentation Environment

Machines The client and server are two identical x86-64 machines of 4 sockets clocked at 2.4GHz. Each machine has four 16-core processors (AMD Opteron 6378), 256GB RAM, and a 20Gbps bonded connection to a 24-port 10GbE switch. A core pair shares 64KB L1 data cache and 2MB L2 cache. Each machine has 6 local disks (125-204MB/s) with the root system installed on 2 of them (in RAID1). Both machines run Debian 9 Linux (kernel v4.9). We repeated several of our experiments on a more recent Linux kernel (v5.4.0) but obtained similar results.

The client machine runs containers configured with the Linux cgroup (v1,v2) and namespaces. The server machine is organized into 8 nodes, each of 8 cores and 32GB RAM. We use one node to run the host system. The rest of the nodes are configured as 7 VMs (Xen v4.8) running a Ceph (v10.2.7) cluster of 6 OSDs (object storage devices) and 1 MDS (metadata server). On each OSD we use 8GB RAM for main memory and the remaining 24GB as ramdisk with XFS for the fast storage of the OSD data and journal.

All the Ceph clients access the stored images and data from the ramdisks of the storage servers. We only use the hard disks of the client and server machines to store their operating system, or generate contention to the Ceph clients from local I/O (§6.2). In Fig. 5 we show our testbed environment with two alternative configurations for the client machine at the left (scaleout) and middle (scaleup), and the configuration of the server machine at the right.

Filesystems We compare Danaus (D) to several combinations of union filesystems and Ceph clients:

(1) As representative mature system, we examine the kernel-based Ceph client (CephFS). We run it either standalone (K) or below the kernel-based AUFS (K/K,§4.3), with the page cache used by both the union and client.
(2) As user-level baseline, we use the FUSE-based Ceph client (ceph-fuse) using the page cache (FP) or bypassing it (F). Over the FUSE-based client, we optionally run the FUSE-based Unionfs (unionfs-fuse), with the page cache used by both the union and client (FP/FP) or none (F/F). F/F occupies the least memory space because the FUSE-based Ceph client only uses the user-level object cache, and neither the union nor the client uses the kernel page cache.
(3) Finally, we examine the FUSE-based Unionfs without cache running over CephFS with page cache (F/K).

In FUSE, we enable several known optimizations from prior research [69]. In order to bypass the page cache (in F, F/K or F/F), we use the direct I/O mount option. Table 1 summarizes these configurations.

Workloads We conduct extensive experimental evaluation using the following workloads:

(1) the Filebench [66] Fileserver (FLS) to emulate simple fileserver read/write I/O,
(2) the Filebench Webserver (WBS) for local read-intensive I/O,
(3) the Filebench Singlestreamwrite and Singlestreamread (briefly Seqwrite and Seqread) micro-workloads to generate sequential write and read,
(4) the RandomIO of Stress-ng (RND) to generate local random read/write I/O [13],
(5) the CPU benchmark of Sysbench (SSB) for cpu-intensive processing [12],
(6) the RocksDB persistent key-value store for writes and out-of-core reads [11],
(7) the Lighttpd [8] standards-compliant high-performance web server for read-intensive I/O at container startup,
(8) and our own file append (Fileappend) and read (Fileread) for sequential write and read of a single file with low metadata activity.

We run each workload instance on a separate container with basic Debian 9 Linux.

Methodology The comparison of different systems on the same platform provides crucial intuition about the relative benefits and potentials of Danaus. The kernel-based Ceph, the user-level libcephfs and the FUSE-based ceph-fuse are functionally similar but considerably different implementations of the Ceph client. AUFS is a rewrite of the kernel-based Unionfs for improved reliability and performance, while the unionfs-fuse is a FUSE-based version of Unionfs [3, 54, 56].

We keep the default settings of 5s for the expire interval and 1s for the writeback interval in dirty page flushing. To prevent out-of-memory (OOM) termination, the size of user-level client cache is set to 50% of the pool memory, unless specified differently. We set the max dirty bytes to 50% of the pool RAM in kernel-based Ceph, and to 50% of the client cache in Danaus and FUSE-based Ceph. We repeat the experiments as needed (up to 10 times) to get the half-length of the 95% confidence interval of the primary metric (e.g., throughput) within 5% of the measured average. Although we mostly show the latency, throughput, or timespan, we reached similar conclusions from other metrics, including tail statistics.

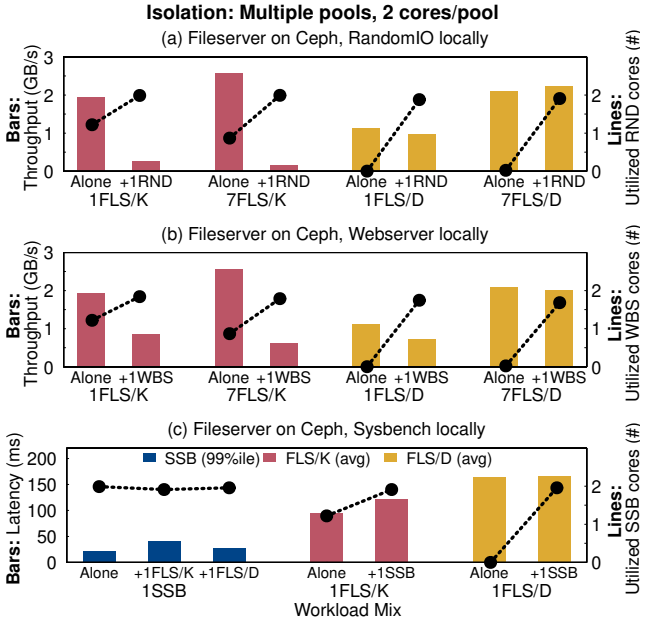

Figure 6: Workload interference. (a) Fileserver throughput (higher is better) and RandomIO core utilization. (b) Fileserver throughput and Webserver core utilization. (c) Sysbench or Fileserver latency (lower is better), and Sysbench core utilization.

6.2 Isolation

We examine the performance interference among 1 or 7 Fileserver (FLS) instances and another workload that is either I/O- or cpu-bound, as summarized in Table 2. Each workload instance runs in a dedicated *container pool* of 2 cores and 8GB RAM. The workloads run at the client machine with the number of enabled cores (4-16) twice the number of running instances (2-8). We run FLS with 5MB mean file size, 1000 files and 120s duration. The dataset is stored on Ceph (without union) through Danaus (D) or the kernel CephFS client (K). Danaus mounts the root filesystem of a container using a private filesystem service with 5GB client cache to hold the entire dataset.

RandomIO Fig. 6a expands Fig. 1a with results from FLS running over Danaus to prevent the performance variability of the kernel. RandomIO (RND) performs random 512B read/write with readahead using 2 threads. It accesses a 1GB file stored through ext4 on 4 local disks (in RAID0). We run 1 or 7 FLS instances on D or K alone or in parallel to 1 RND instance. From the bar chart, 1FLS/D has 41% lower throughput than 1FLS/K. We remind that the execution of RND causes the FLS throughput to drop 7.4x in 1FLS/K+1RND and 16.5x in 7FLS/K+1RND. On the contrary, it only drops 16% in 1FLS/D+1RND and it is even slightly faster in 7FLS/D+1RND. Put differently, D is faster than K by 3.7x-14.4x when FLS is colocated with RND.

Danaus avoids the pressure from RND by handling the FLS I/O with the cores of the pool running the respective FLS instance. This is confirmed by the line chart showing that the cores of the RND pool are utilized less than 2.5% when 1FLS/D or 7FLS/D run alone. Additionally, Danaus reduces the kernel lock contention by running a separate user-level filesystem service per container. In particular, our kernel profiling showed that 1FLS/K+1RND and 7FLS/K+1RND are remarkably delayed at the *i_mutex_key* kernel lock of the superblock struct. Moreover, the core and lock contention together cause the 7FLS/K+1RND to flush 32% fewer dirty pages than 7FLS/K running alone.

Webserver We obtained similar results by running FLS next to the Webserver benchmark (WBS). We configure WBS with 50 threads and 200K files of 16KB mean size on ext4 over 4 local disks (in RAID0). We run 1 or 7 instances of FLS over K or D alone or in parallel to an instance of WBS. From the bar chart of Fig. 6b, the concurrent execution of WBS reduces the kernel FLS throughput by 2.3x in 1FLS/K+1WBS and 4.2x in 7FLS/K+1WBS. Although 7FLS/D alone is 18% slower than 7FLS/K, FLS in 7FLS/D+1WBS is 3.2x faster than 7FLS/K+1WBS. With the WBS inactive, 1FLS/K and 7FLS/K utilize 87-122% the cores of WBS, while 1FLS/D and 7FLS/D only 0.5-2.5% (lines of Fig. 6b). As with RND, we conclude that the kernel FLS performance drops when WBS is running because the kernel cannot utilize any more the cores of WBS to serve FLS.

Sysbench Danaus also limits the latency increase (lower is better) across colocated heterogeneous workloads [34]. We consider the Sysbench cpu benchmark (SSB) with 2 threads in prime number calculation using 64-bit integers. In the bar chart of Fig. 6c, we show the 99%ile of the SSB latency and the average FLS latency over Ceph. We run one FLS instance over K or D, alone or in parallel to one SSB instance. The workload interference raises the respective SSB and FLS latency by 93% and 28% in 1FLS/K+1SSB but only 27% and 2% in 1FLS/D+1SSB (3.4x and 14x less). When FLS runs alone, K achieves lower latency than K because it also utilizes the reserved SSB cores (Fig. 6c line chart). In comparison to RND and WBS, SSB affects less intensely the 1FLS/K because it only runs user-level calculations.

In summary, Danaus achieves throughput and latency stability across competing workloads because it serves I/O with less hardware and kernel contention. On the contrary, the kernel-based client causes performance drop up to 16.5x.

6.3 Performance and Efficiency

We consider I/O-bound applications and microbenchmarks over Danaus or different combinations of FUSE and the kernel. We run a workload per container and report the measured performance and consumed resources. Each container runs either independently on a distinct system image (2.7GB), or cloned over a shared read-only lower branch and a private writable upper branch. Although in Section 6.2 we found the kernel-based client to provide reduced isolation among competing container pools, we include it combined with AUFS as a performance baseline from the perspective of a mature kernel-based system (K/K).

6.3.1 Applications. We study RocksDB (v4.5) [11] and Lighttpd (v1.4.45) [8] as representative I/O-intensive applications.

RocksDB (scaleout/scaleup) We run the RocksDB storage engine independently in 32 container pools of the same machine. Each pool is configured with one container over 2 cores and 8GB RAM. The container runs RocksDB with 64MB memory buffer and 2 compaction threads. The pool uses a dedicated Ceph client to mount a

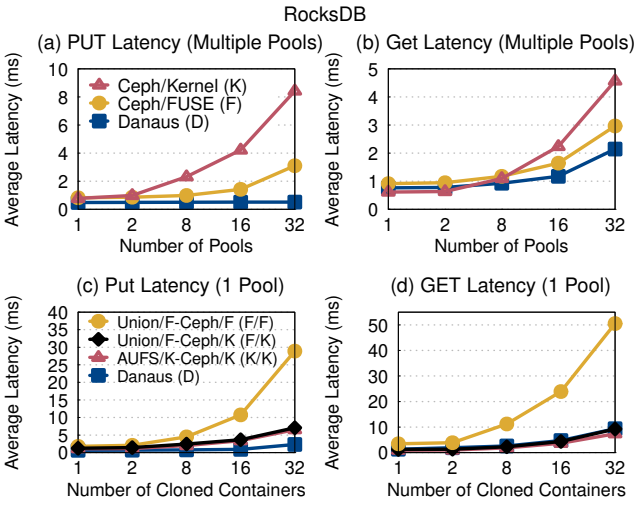

Figure 7: RocksDB. Average latency of RocksDB put and get in scaleout (a,b) and scaleup (c,d) settings of container pools.

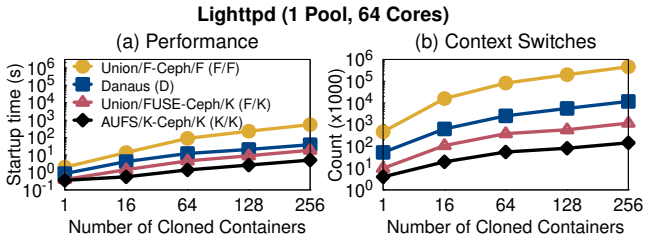

Figure 8: Container startup scaleup. Real time to start 1-256 Lighttpd cloned containers in a single pool over shared client.

private root filesystem holding the files of both the container and RocksDB. The put workload uses 1 thread to insert 1GB random pairs of 9B key and 128KB value. We measured the average put latency of RocksDB over the D, F, or K client for 1-32 pools. In Fig. 7a, D is faster than F and K up to 5.9x and 16.2x (32 pools). A main reason is the intense kernel contention of F and K demonstrated by total kernel lock wait time (not shown) up to 5x and 152x higher over D (1 pool).

In another experiment, we run an out-of-core read-intensive workload per pool. Each container populates RocksDB with 8GB using random 128KB puts before reading back 8GB with random gets. In Fig. 7b, D is faster than F and K up to 1.4x and 2.2x (32 pools). Correspondingly, the total kernel lock wait time of F and K (not shown) is up to 2.8x and 17.8x that of D (32 pools). Moreover, D reduces the cpu activity (incl. IO wait) and memory consumption of K up to 9.5x and 2.6x. *In scaleout, Danaus reduces the lock contention and resource consumption by running and accessing at user level a distinct client per container.*

We also explored a scaleup setting running multiple cloned containers in a single pool. Each container runs a private RocksDB instance. A container mounts its root filesystem through a private filesystem instance consisting of a distinct union filesystem and a *shared* Ceph client. The Ceph client lets the containers share the lower read-only layer of their root filesystem. We measured the RocksDB latency across D, F/F, F/K and K/K (Table 1). With respect to put latency, D is faster than F/F, F/K and K/K up to 12.6x, 3.9x and 3.6x, respectively (Fig. 7c). In contrast, the get workload (Fig. 7d) leads to mixed results with D up to 5.4x faster than F/F at 32 clones but up to 2x slower than K/K at 2 clones.

Serving all the clones with a single client in scaleup cancels the scaleout decentralization and concurrency. Still, D achieves substantial latency benefit over the other systems in put, and over F/F in get.

Lighttpd (scaleup) In a single pool, we measure the time to start a number of cloned containers with Lighttpd waiting for requests. The I/O traffic is generated by the exec call to start the initial command, the mmap calls to fetch the dynamic libraries, and the user-level calls to prepare the application files. In Danaus, this scaleup workload exercises mostly the (legacy) kernel path to read data and less the (default) user-level path to write data. The kernel-based AUFS (K/K) with CephFS (K/K and F/K) lead to higher performance in comparison to the user-level unionfs-fuse with cephfs-fuse (F/F) or libcephfs (D).

With respect to the real time to start 1-256 instances, D is up to 8.8x slower than K/K and 2.9x than F/K (Fig. 8a). From kernel profiling we confirmed that the workload is read-intensive and D uses the (legacy) FUSE path to fetch the dynamic libraries. In comparison to F/F that uses similar Unionfs and Ceph client code, D reduces considerably the startup time (2.3-14.2x) and cpu activity (up to 10.5x). The improved performance and efficiency of D over F/F is explained in part by the 9-39x fewer context switches (top two lines of Fig. 8b).

In the examined real-world applications, Danaus achieves lower latency and resource consumption than the other systems in scaleout or put scaleup, and over F/F in get scaleup. In legacy-dominated I/O, the more mature K shortens the startup time, while D is up to 14.2x faster than F/F.

6.3.2 Microbenchmarks. We measure the performance and resource consumption of two Filebench workloads and a custom-developed one. We consider scaleout and scaleup settings of data served by root filesystems mounted from Ceph network storage through components based on Danaus, FUSE or the kernel.

Seqwrite/Seqread (scaleout) We generate sequential writes with Seqwrite in 1-32 pools. Each pool of 2 cores and 8GB RAM mounts a private root filesystem from Ceph through D, F or K. We configure Seqwrite with 1GB file size, 16 threads, and 120s duration. Seqwrite generates I/O activity in the entire path from the application to the backend servers. In Fig. 9 (top), the throughput of D and F is up to 2.8x higher than K, while the I/O wait cpu time of K is up to 90x that of F. One reason is that K spends three orders of magnitude more time for kernel lock waiting (most notably *i_mutex_dir_key* and *i_mutex_key*). We also found that K handles I/O with unallocated cores whose number decreases at more pools.

In a different experiment, we generate sequential reads with Seqread configured similarly to Seqwrite. The workload stresses the local path to the client cache fully holding the accessed file.

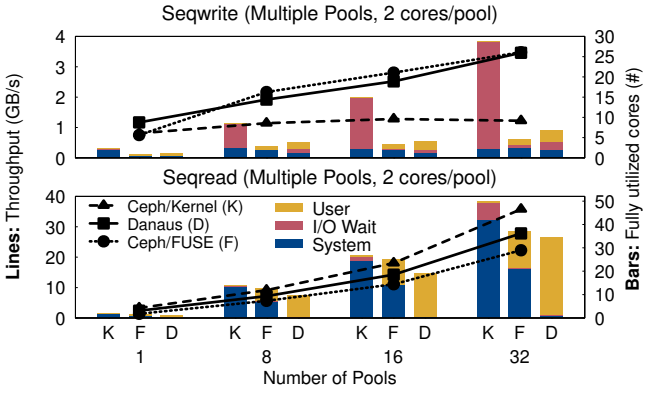

Figure 9: Sequential I/O scaleout. Performance and core utilization of the Filebench Seqwrite/Seqread at multiple pools.

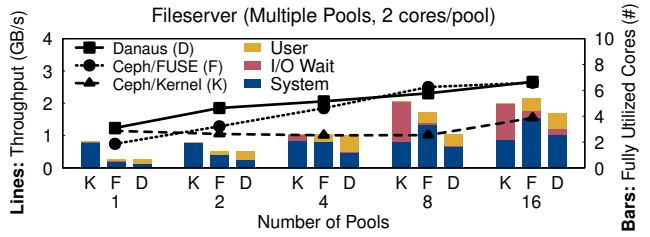

Figure 10: Random I/O scaleout. Performance and core utilization of the Filebench Fileserver at multiple pools.

From Fig. 9 (bottom), K is faster than D up to 37%, and D faster than F up to 75% (1 pool). Unlike F and K, D predominantly runs at user level and uses negligibly the kernel (bar chart). With user-level profiling, we found that the concurrency of D is limited by the *client_lock* lock [27]. This is a global lock of libcephfs used by D, but not by the kernel locking of K. From preliminary experiments, removing the global lock improves the Danaus concurrency but requires refactoring libcephfs, which is beyond our current scope. *In summary, at pool scaleout, D is faster than F and K in sequential write, but slower than K in cached sequential read.*

Fileserver (scaleout) We experiment with Fileserver in scaleout of multiple pools. Each pool of 2 cores and 8GB RAM mounts a root filesystem through a private D, F or K client. Fileserver running on Ceph generates a random read/write workload that stresses the entire path to backend storage. We run 1-16 pools on the client host and measure the total throughput of the pools. From Fig. 10 (line chart), D achieves 2.7GB/s at 16 pools with an advantage of 1.7x over F at 1 pool and 2.3x over K at 8 pools. The amount of resources consumed by the three clients at the server side is comparable (not shown), but K generates up to 22x higher I/O wait cpu time at the client side (Fig. 10, bar chart).

Thus, in a random read/write workload at pool scaleout, D achieves substantial performance improvement over K at increasing number of pools, and over F at smaller scales.

Fileappend/Fileread (scaleup) We run multiple cloned containers inside a single pool of 64 cores and 200GB memory, leaving

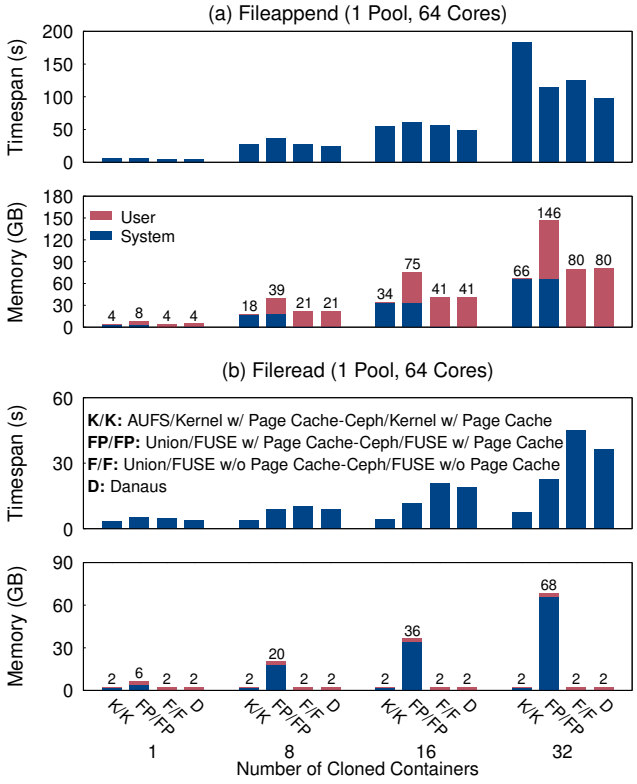

Figure 11: Sequential I/O scaleup. In a single pool, we measure the timespan (lower is better) and maximum memory to start a number of concurrent containers running the Fileappend (a) or Fileread (b) benchmark, and wait until they all finish.

64GB RAM to the host system. The root filesystem of each container is mounted through a private union filesystem and a shared Ceph client to provide a common read-only lower branch. We measure the time to launch the containers, run a high-data/low-metadata workload and wait until they all finish.

Fileappend opens a single 2GB file in `O_WRONLY|O_APPEND` mode, writes 1MB and closes it. The generated I/O is approximately 50/50 read/write because the file is copied to the upper layer by the copy-on-write union. In Fig. 11a (top), D tends to achieve shorter timespan (lower is better) in comparison to the other systems and especially K/K by up to 46% in 32 containers. In Fig. 11a (bottom), the maximum memory required by K/K, F/F and D increases linearly with the number of containers, while the page-cache usage by the FUSE-based Ceph client almost doubles it in FP/FP.

In a different experiment, Fileread opens a 2GB file in `O_RDONLY` mode, reads the entire file in 1MB blocks, and closes it. In Fig. 11b (top), K/K achieves 1.2-4.9x shorter timespan than D, but up to 7.4x higher client cpu activity (not shown). In Fig. 11b (bottom), F/F requires the same memory size as D but with 11-23% longer timespan. Although FP/FP achieves shorter timespan than D, it occupies up to 30x more memory. The excessive memory usage of FP/FP is caused by caching of the union and Ceph client in page

cache, and of the Ceph client at user level. Bypassing the page cache at either the FUSE union or the FUSE client still keeps the memory usage at least twice that of F/F.

Therefore, in pool scaleup with cloned containers over union and shared client, Danaus and FUSE are faster than the kernel in mixed read/write, but slower in sequential read.

7 RELATED WORK

Libservices [38] is an abstraction of per-tenant user-level shared services that can be used to provision the container image registries and root or application filesystems in cloud environments. Based on libservices, the Polytropon user-level toolkit was previously introduced to implement the storage components of multitenant hosts, including the API library, message queues, data transfers and filesystem services [39]. In the present work, we build Danaus based on libservices and Polytropon. We motivate our work from the inefficiencies of current container storage systems. Subsequently, we specify our principles and goals, justify the interface, cache and consistency properties of Danaus, and evaluate the isolation, efficiency and performance in cloned or independent containers. Next, we summarize and differentiate the related work in filesystems, virtualization, user-level I/O, and microkernels.

Filesystems IceFS disentangles the physical structures of a local filesystem and demonstrates independent performance and reliability in virtual machines running over the same local filesystem and applications sharing the same storage servers of a distributed filesystem. Danaus addresses the complementary problem of isolating the distributed filesystem clients of colocated containers [47]. Slacker relies on snapshot/clone operations of an NFS shared network filesystem to efficiently serve container images through the Docker daemon, and uses a bitmap inside the host kernel to deduplicate the client cache [32]. TotalCOW uses block address caching and data comparison to avoid I/O and cache inefficiencies of container images [76]. Both systems target the efficient shared image deduplication of container clones over block-based volumes lacking native support for container isolation or file-level sharing.

The PolarFS distributed filesystem achieves low latency in cloud databases with lightweight network and I/O stacks [21]. The Docker storage drivers show performance differences depending on the workload I/O [65, 77]. The registry service handles a pull-intensive workload that benefits from container image caching [17]. Wharf obviates the image registry by placing the image layers on a shared NFS storage backend [81]. Danaus takes advantage of data sharing at both the client host and the storage backend to improve the container I/O efficiency.

Virtualization Operating systems and hypervisors require substantial tuning (e.g., swappiness [51], cgroup [70]) to overcome the performance interference among colocated data-intensive containers or virtual machines (VMs) [20, 30, 60]. Serverless functions run on containers to hide the server management, but suffer from I/O performance degradation when colocated on the same VM [73]. The network processing has been isolated across containers through novel kernel resource accounting (Iron [40]), or offloading over message queues to a kernel thread on dedicated core (IsoStack [59]).

System isolation is improved with complex component partitioning of the hypervisor (LightVM [48]) or kernel (VirtuOS [52]). FlexSC introduced the exception-less system calls to improve multicore processor efficiency in system-intensive workloads [62]. MultiLanes uses file-backed virtualized block devices and partitions the VFS data structures to reduce the container contention over a shared local filesystem [36]. Heracles uses processor hardware and Linux kernel methods to colocate latency-critical and best-effort tasks [46]. The above approaches improve isolation with hardware-based or kernel-based partitioning instead of the user-level functionality per container pool of Danaus. The gVisor sandbox limits the system API attack vector with an isolated user-space kernel, but currently at substantial I/O performance cost [7, 78].

User-level I/O A library intercepts the application I/O calls and communicates with a user-level router to virtualize the container network (SlimSocket [83], FreeFlow [41]). In non-cache-coherent multicores, an application communicates over RPC with the Hare local filesystem, but accesses the buffer cache through shared memory [31]. Aerie and SplitFS implement local filesystems for storage class memory with user-level libraries [35, 71]. Arrakis provides direct device access to applications through hardware virtualization [53]. Graphene executes multi-process applications using shared POSIX abstractions over libOS [67]. DAFS enabled local file sharing over RDMA with a user-level non-POSIX API [26]. The SFS toolkit constructed user-level loopback servers accessed by in-kernel NFS clients [49]. Danaus isolates the container filesystem I/O of multiple tenants at the client host rather than virtualizing the fast local I/O devices.

Microkernels Fault isolation is achieved by loading a distrusted module in a logically separate portion of the address space [72]. The Raven kernel implemented threads, IPC, and device management at user level without kernel data copying [57]. The Cache Kernel relied on user-mode application kernels to manage kernel objects [22]. The exokernel securely exported all hardware resources to enable physical resource management in untrusted library operating systems [29]. Danaus targets the container I/O isolation on shared storage rather than the I/O management in microkernels.

8 LESSONS LEARNED

Our experience from building and evaluating Danaus has been very positive in achieving our goals with manageable complexity and demonstrating improved isolation, efficiency and performance over existing representative systems.

Kernel contention The operating-system kernel of a host can become a hotspot to the colocated containers. The two main sources of this misbehavior are the lock contention of the shared data structures and the aggressive hardware resource allocation by the kernel to serve the requests of the colocated tenants.

Images and data on shared filesystem Danaus provides an integrated approach to serve both the *root images* and *application data* with sharing configuration options enabled through a union filesystem per container and a common client.

Serving the root images of the containers directly from a distributed filesystem replaces costly image downloads to the host with on-demand file transfers during runtime. The union filesystems deduplicate the cloned images to save space and bandwidth at the storage clients and servers.

Fetching the application data from a shared distributed filesystem enables native data sharing across the collaborating containers. If multiple containers of the same tenant share the same filesystem client at a host, they save memory and bandwidth resources.

Functionality and execution separation Improved performance and fault isolation is possible by serving each tenant with separate functionality on a host. The user-level execution of the filesystem client and the interprocess communication permits to serve each tenant with exclusively reserved memory and processor resources.

Client implementation The user-level client may experience contention if implemented with coarse-grain locks accessed by multiple containers. The problem is less intense when each of the colocated tenants uses its own client. As it avoids complex kernel dependencies, the user-level client is expected to require less effort to be refactored with fine-grain locking for increased concurrency.

Throughput and latency stability We experimentally confirmed that the throughput and latency of I/O-intensive workloads served at user level is relatively insensitive to competing resource-demanding workloads. The performance of kernel-served workloads may be higher when running alone, but it drops dramatically next to competing neighbors.

Scaleout and scaleup In scaleout workloads with random I/O or sequential writes, Danaus has performance advantage and comparable or improved efficiency over the kernel-based systems (including FUSE) due to the decentralization of the client across the different tenants.

In scaleup workloads, Danaus maintains performance advantage over the kernel-based systems in write or mixed requests, while in read requests, Danaus is faster or more efficient over the FUSE-based systems.

9 LIMITATIONS AND FUTURE WORK

The container runtimes already support a plethora of storage drivers and volumes plugins for the container root and application filesystems [2, 5]. *Danaus introduces a user-level client architecture that provides isolation and efficiency for both these types of container filesystems in a single framework.*

In our future work, we plan to port Danaus to production orchestration systems. Moreover, we are also interested to explore the applicability of the Danaus client in per-tenant storage provisioning for serverless function computations. The partitioning of host resources across different pools trades the resource utilization for improved isolation. We leave for future extension of our framework the dynamic reallocation of underutilized resources (e.g., memory) combined with service quality guarantees.

Ideally, we should provide native end-to-end multitenant isolation at the host, the network and the storage backend [16]. Such an extension could help adopt the latest developments in datacenter and storage protocols on per-tenant basis and further strengthen fault tolerance [44]. It is an interesting problem that we leave for future work the restructuring of the storage servers and the integration of the filesystem services with a user-level network stack [58].

The Danaus prototype was made possible by the open-source clients of Ceph at the kernel and user level. However, the proposed architecture generally applies to scalable filesystems (e.g., Gluster [4], Lustre [9]). Danaus does not prevent the copy-on-write of entire files when they are modified. As future work, the backend servers and the client cache should support file deduplication at the block level (Slacker adopts image block-level deduplication in the kernel-based client [32]). Danaus could conveniently facilitate the container migration between hosts through the shared network filesystem [80].

10 CONCLUSIONS

The kernel I/O handling penalizes the container performance because it causes contention at hardware and software resources. Danaus serves separately each tenant with a user-level filesystem service that integrates a union filesystem with a distributed filesystem client and local caching. We handle I/O with the private resources of a container pool and avoid the costly locking of the shared kernel. Accordingly, the user-level lock contention in scaleup is partly addressed by running per-tenant filesystem services. At reduced memory and cpu utilization, Danaus achieves higher performance (14.4x) and lower latency (16.2x) than a mature kernel-based client, most notably in scaleout or write-intensive scaleup. Overall, our user-level client architecture improves the isolation and efficiency of container I/O at multitenant hosts over an unmodified stock system kernel.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments that helped us improve the final version.

REFERENCES

[1] 2019. Are remote / network filesystems supported as upperdir? https://github.com/containers/fuse-overlayfs/issues/141.
[2] 2019. Manage data in Docker. https://docs.docker.com/storage/.
[3] 2020. AUFS. https://en.wikipedia.org/wiki/Aufs.
[4] 2020. Gluster. https://www.gluster.org/.
[5] 2020. Kubernetes Concepts: Storage. https://kubernetes.io/docs/concepts/storage/.
[6] 2021. Amazon Web Services. https://aws.amazon.com/.
[7] 2021. gVisor. https://github.com/google/gvisor.
[8] 2021. Lighttpd fly light. https://www.lighttpd.net/.
[9] 2021. Lustre. lustre.org.
[10] 2021. Microsoft Azure. https://azure.microsoft.com.
[11] 2021. RocksDB. rocksdb.org.
[12] 2021. Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.
[13] 2021. Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/.
[14] 2021. Use the BTRFS storage driver. https://docs.docker.com/storage/storagedriver/btrfs-driver/.
[15] Sungyong Ahn, Kwanghyun La, and Jihong Kim. 2016. Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems. In *USENIX HotStorage Workshop* (Denver, CO). USENIX Association, Berkeley, CA, 111–115.
[16] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO). USENIX Association, Berkeley, CA, 233–248.
[17] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *USENIX Conference on File and Storage Technologies* (Oakland, CA). USENIX Association, Berkeley, CA, 265–278.
[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA). USENIX Association, Berkeley, CA, 689–703.

[19] Eric W. Biederman. 2006. Multiple Instances of the Global Linux Namespaces. In *Ottawa Linux Symposium* (Ottawa, Canada). 101–112.

[20] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (May 2016), 50–57.

[21] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. In *Proc. VLDB Endowment.* 1849–1862.

[22] David R. Cheriton and Kenneth J. Duda. 1994. A Caching Model of Operating System Kernel Functionality. In *USENIX Symposium on Operating Systems Design and Implementation* (Monterey, CA). USENIX Association, Berkeley, CA.

[23] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Transactions on Computer Systems* 32, 4, Article 10 (Jan. 2015), 47 pages.

[24] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3, 5 (Sep-Oct 2016), 54–62.

[25] Jeff Darcy. 2011. Building a Cloud File System. *USENIX; login:* 36, 3 (June 2011), 14–21.

[26] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. 2003. The Direct Access File System. In *USENIX Conference on File and Storage Technologies* (San Francisco, CA). USENIX Association, Berkeley, CA, 175–188.

[27] Patrick Donnelly. 2018. break client_lock. https://tracker.ceph.com/issues/23844.

[28] Jake Edge. 2018. Measuring container security. *LWN.net* (Dec. 2018).

[29] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *ACM Symposium on Operating Systems Principles* (Copper Mountain, CO). ACM, New York, NY, 251–266.

[30] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. IBM Research Report RC25482, IBM Research Division.

[31] Charles Gruenwald, III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Hare: a file system for non-cache-coherent multicores. In *ACM European Conference on Computer Systems* (Bordeaux, France). ACM, New York, NY, 30:1–30:16.

[32] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX Conference on File and Storage Technologies* (Santa Clara, CA). USENIX Association, Berkeley, CA, 181–195.

[33] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference* (Renton, WA). USENIX Association, Berkeley, CA, 489–504.

[34] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX Annual Technical Conference* (Boston, MA). USENIX Association, Berkeley, CA, 519–532.

[35] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *ACM Symposium on Operating Systems Principles* (Huntsville, ON, Canada). ACM, New York, NY.

[36] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. 2014. MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores. In *USENIX Conference on File and Storage Technologies* (Santa Clara, CA). USENIX Association, Berkeley, CA, 317–329.

[37] Antti Kantee. 2009. Rump File Systems: Kernel Code Reborn. In *USENIX Annual Technical Conference* (San Diego, CA). USENIX Association, Berkeley, CA, 201–214.

[38] Giorgos Kappes and Stergios V. Anastasiadis. 2020. Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation. In *ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan). ACM, New York, NY, 33–41.

[39] Giorgos Kappes and Stergios V. Anastasiadis. 2020. A User-Level Toolkit for Storage I/O Isolation on Multitenant Hosts. In *ACM Symposium on Cloud Computing* (Virtual Event, USA). ACM, New York, NY, 74–89.

[40] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating Network-based CPU in Container Environments. In *USENIX Symp. Networked Systems Design and Implementation* (Renton, WA). USENIX Association, Berkeley, CA, 313–328.

[41] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *USENIX Symp. Networked Systems Design and Implementation* (Boston, MA). USENIX Association, Berkeley, CA, 113–126.

[42] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. 2019. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *USENIX Annual Technical Conference* (Renton, WA). USENIX Association, Berkeley, CA, 15–31.

[43] Mark Lamourine. 2015. Storage Options for Software Containers. *USENIX ;login:* 1 (Feb. 2015), 10–14.

[44] James Larisch, James Mickens, and Eddie Kohler. 2018. Alto: Lightweight VMs Using Virtualization-Aware Managed Runtimes. In *Intl Conference on Managed Languages & Runtimes* (Linz, Austria). ACM, New York, NY, 8:1–8:7.

[45] Andrew Leung, Andrew Spyker, and Tim Bozarth. 2018. Titus: Introducing Containers to the Netflix Cloud. *Commun. ACM* 61, 2 (Feb. 2018), 38–45.

[46] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *ACM/IEEE Intl Symposium on Computer Architecture* (Portland, OR). ACM, New York, NY, 450–462.

[47] 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 *USENIX Symp. Operating Systems Design and Implementation* (Broomfield, CO). USENIX Association, Berkeley, CA, 81–96.

[48] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *ACM Symposium Operating Systems Principles* (Shanghai, China). ACM, New York, NY.

[49] David Mazières. 2001. A Toolkit for User-Level File Systems. In *USENIX Annual Technical Conference* (Boston, MA). USENIX Association, Berkeley, CA, 261–274.

[50] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. 2019. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *USENIX Annual Technical Conference* (Renton, WA). USENIX Association, Berkeley, CA, 587–602.

[51] Rina Nakazawa, Kazunori Ogata, Seetharami Seelam, and Tamiya Onodera. 2017. Taming Performance Degradation of Containers in the Case of Extreme Memory Overcommitment. In *IEEE International Conference on Cloud Computing* (Honolulu, HI). IEEE, Los Alamitos, CA.

[52] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *ACM Symposium on Operating Systems Principles* (Farminton, PA). ACM, New York, NY, 116–132.

[53] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO). USENIX Association, Berkeley, CA, 1–16.

[54] Radek Podgorny. [n. d.]. unionfs-fuse. https://github.com/rpodgorny/unionfs-fuse.

[55] Shaya Potter and Jason Nieh. 2010. Apiary: Easy-to-use Desktop Application Fault Containment on Commodity Operating Systems. In *USENIX Annual Technical Conference* (Boston, MA). 103–116.

[56] David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok. 2006. Unionfs: User- and Community-Oriented Development of a Unification File System. In *Ottawa Linux Symposium* (Ottawa, Canada). 357–370.

[57] D. Stuart Ritchie and Gerald W. Neufeld. 1993. User Level IPC and Device Management in the Raven Kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium.* USENIX Association, Berkeley, CA, 111–126.

[58] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *USENIX Annual Technical Conference* (Boston, MA). USENIX Association, Berkeley, CA, 101–112.

[59] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. 2010. IsoStack - Highly Efficient Network Processing on Dedidated Cores. In *USENIX Annual Technical Conference* (Boston, MA). USENIX Association, Berkeley, CA, 61–74.

[60] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *ACM/IFIP/USENIX Intl Middleware Conference* (Trento, Italy). ACM, New York, NY, 1:1–1:13.

[61] Balbir Singh and Vaidyanathan Srinivasan. 2007. Containers: Challenges with the memory resource controller and its performance. In *Ottawa Linux Symposium* (Ottawa, Canada). 209–222.

[62] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX Symp. Operating Systems Design and Implementation* (Vancouver, BC). USENIX Association, Berkeley, CA, 33–46.

[63] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security Symposium* (Baltimore, MD). USENIX Association, Berkeley, CA, 1423–1439.

[64] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. 2013. Virtual Machine Workloads: The Case for New Benchmarks for NAS. In *USENIX Conf. File and Storage Technologies* (San Jose, CA). USENIX Association, Berkeley, CA, 307–320.

[65] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In Search of the Ideal Storage Configuration for Docker Containers. In *IEEE Intl Workshops on Foundations and Appplications of Self* Systems* (Tucson, AZ). IEEE, Los Alamitos, CA.

[66] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:* 41, 1 (2016), 6–12. https://github.com/filebench/filebench/wiki.

[67] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *ACM European Conference on Computer Systems* (Amsterdam, The Netherlands). ACM, New York, NY, 9:1–9:14.

[68] Uresh Vahalia. 1995. *UNIX Internals: The New Frontiers.* Prentice Hall Press, USA.

[69] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *USENIX Conference on File and Storage Technologies* (Santa Clara, CA). USENIX Association, Berkeley, CA, 59–72.

[70] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *ACM European Conference on Computer Systems* (Bordeaux, France). ACM, New York, NY, 18:1–18:17.

[71] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *ACM European Conference on Computer Systems* (Amsterdam, The Netherlands). ACM, New York, NY, 14:1–14:14.

[72] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles* (Asheville, NC). ACM, New York, NY, 203–216.

[73] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *USENIX Annual Technical Conference* (Boston, MA). USENIX Association, Berkeley, CA, 133–146.

[74] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA). USENIX Association, Berkeley, CA, 307–320.

[75] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as Processes. In *ACM Symposium on Cloud Computing* (Carlsbad, CA). ACM, New York, NY, 199–211.

[76] Xingbo Wu, Wenguang Wang, and Song Jiang. 2015. TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers. In *ACM Asia-Pacific Workshop on Systems* (Tokyo, Japan). ACM, New York, NY, 15:1–15:7.

[77] Qiumin Xu, Manu Awasthi, Krishna T. Malladi, Janki Bhimani, Jingpei Yang, and Murali Annavaram. 2017. Performance Analysis of Containerized Applications on Local and Remote Storage. In *IEEE Intl Conf on Massive Storage Systems and Technology* (Santa Clara, CA). IEEE, Los Alamitos, CA.

[78] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *USENIX Workshop on Hot Topics in Cloud Computing* (Renton, WA). USENIX Association, Berkeley, CA.

[79] Daniel Zahka, Brian Kocoloski, and Kate Keahey. 2019. Reducing Kernel Surface Areas for Isolation and Scalability. In *International Conference on Parallel Processing* (Kyoto, Japan). ACM, New York, NY, Article 41, 10 pages.

[80] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, Dean Hildebrand, and Douglas Thain. 2017. Wharf: Sharing Docker Images across Hosts from a Distributed Filesystem (Poster). In *Intl Conf for High Performance Computing, Networking, Storage and Analysis* (Denver, CO). ACM, New York, NY.

[81] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit Warke, and Dean Hildebrand. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *ACM Symposium on Cloud Computing* (Carlsbad, CA). ACM, New York, NY, 174–185.

[82] Zhenyun Zhuang. 2016. Don't Let Linux Control Groups Run Uncontrolled. *LinkedIn Engineering* (Aug. 2016).

[83] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA). USENIX Association, Berkeley, CA, 331–344.