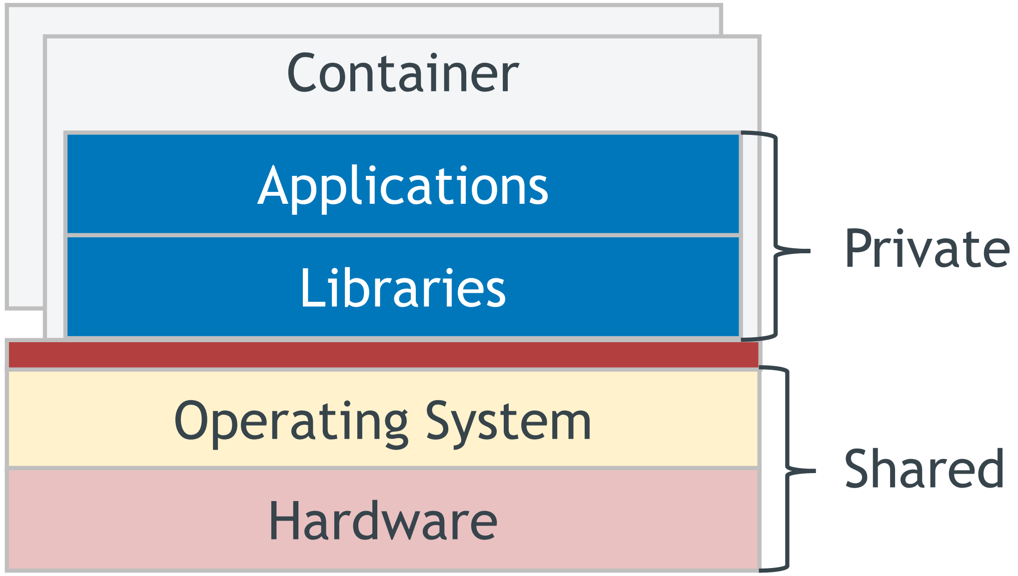
Middleware 2021: 22nd ACM/IFIP International Conference

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

Giorgos Kappes, Stergios V. Anastasiadis
University of Ioannina, Ioannina 45110, Greece

Multitenancy with containers



Containers favor resource utilization

- Low footprint
- Low overhead
- Adjustable resources

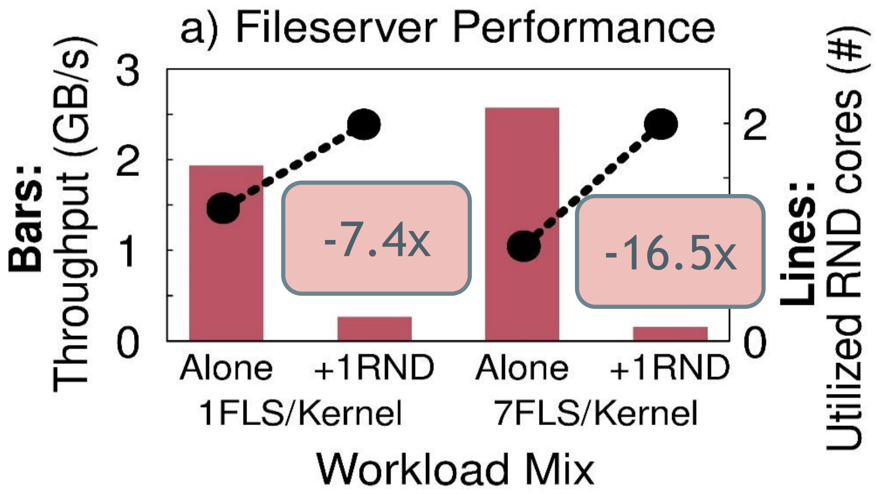
Multitenancy issues due to shared kernel I/O path

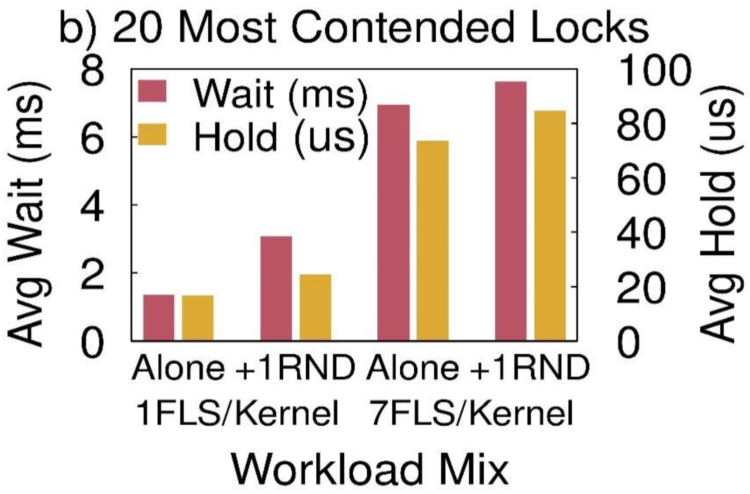
- Low performance isolation
- Weak security isolation & fault containment
- Implicit inefficiencies due to frequent kernel crossings to serve I/O
- Main reasons: Resource contention & inflexible sharing of kernel

Sensitivity to kernel I/O contention

1 (1FLS) or 7 (7FLS) Fileserver on Ceph, 1 (1RND) RandomIO on local ext4 (2 cores per tenant)

Workload colocation causes dramatic performance drop





Kernel utilizes all cores to flush dirty pages

High contention on shared kernel locks

Effective container isolation requires:
explicit allocation of hardware & software resources to each colocated workload

Danaus goals

1. Compatibility
   - POSIX-like interface for multiprocess application access
2. Isolation
   - Improve performance isolation & fault containment of data-intensive tenants cohosted on same client machine
3. Efficiency
   - Low utilization of datacenter resources by containers to access their filesystems
4. Flexibility
   - Enable flexible tenant configuration of sharing & caching policies

The Danaus client architecture

Pool: Containers per tenant/machine

- Managed by container engine
- Container image & application data on shared filesystem

Filesystem library

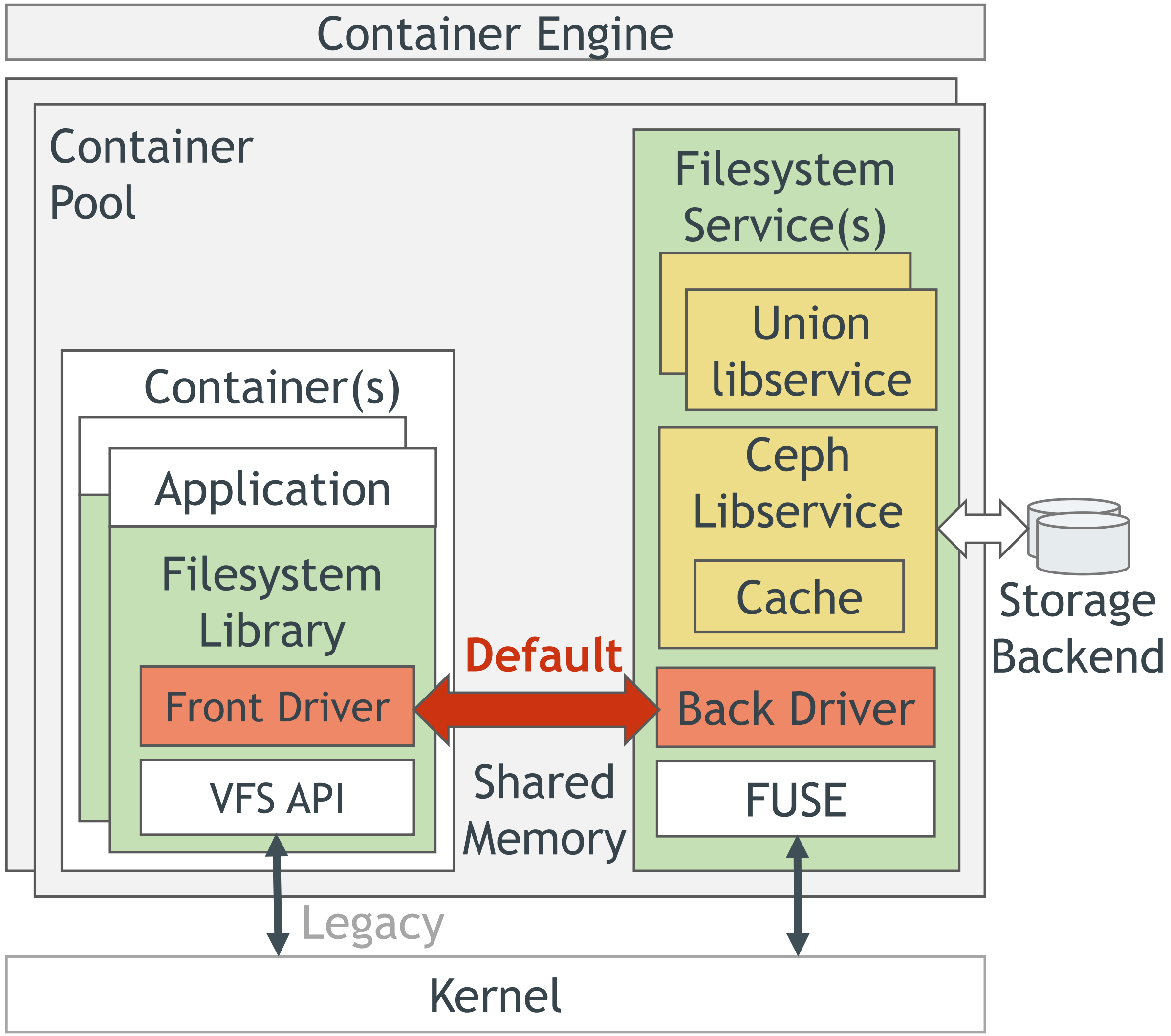
- POSIX API to applications

Filesystem service

- Libservice: user-level I/O function
- Union for container deduplication
- Shared Ceph client with cache for access to network storage

User-level IPC

- Per pool shared memory



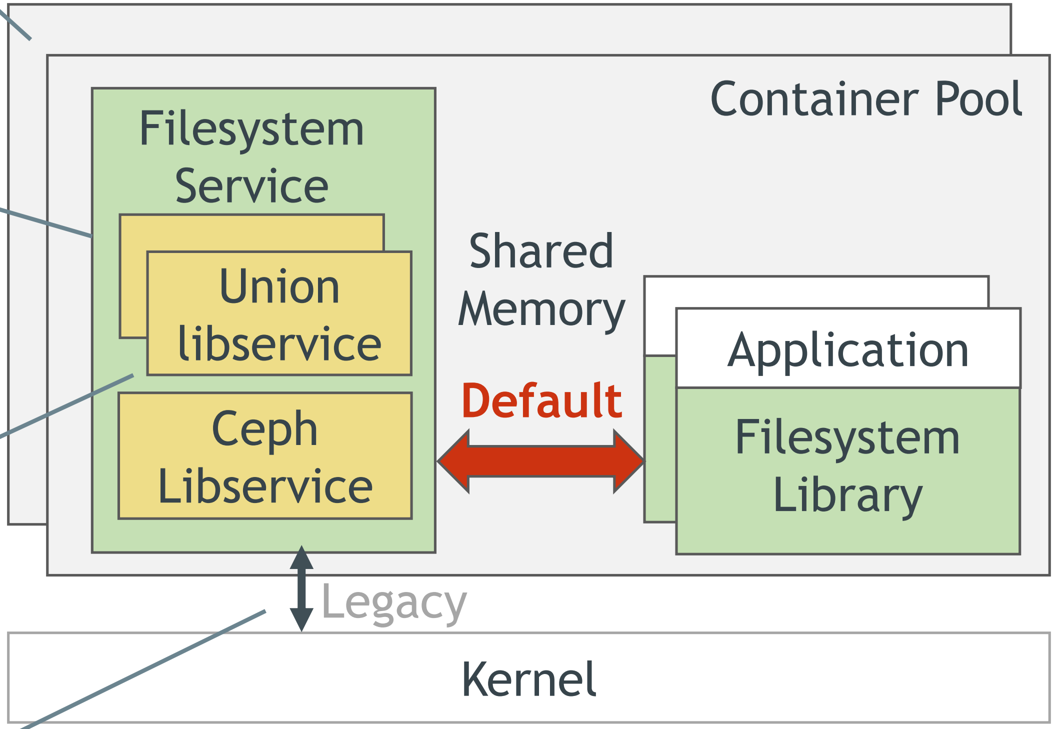
Design principles

Path isolation
Per tenant I/O path

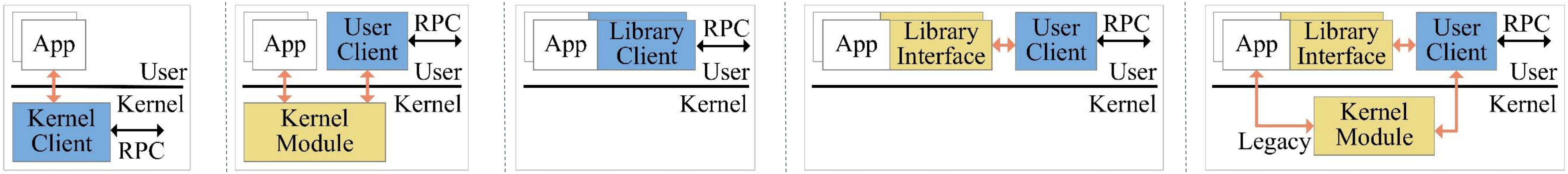
User-level execution
Filesystem and default communication path run at user-level

Filesystem integration
Multiple user-level filesystems interact with each other through function calls

Dual interface
I/O passing from user (default) or kernel level (legacy)



Interface alternatives



(a) Kernel Client

(b) User Client, Kernel Interface

(c) Library Client

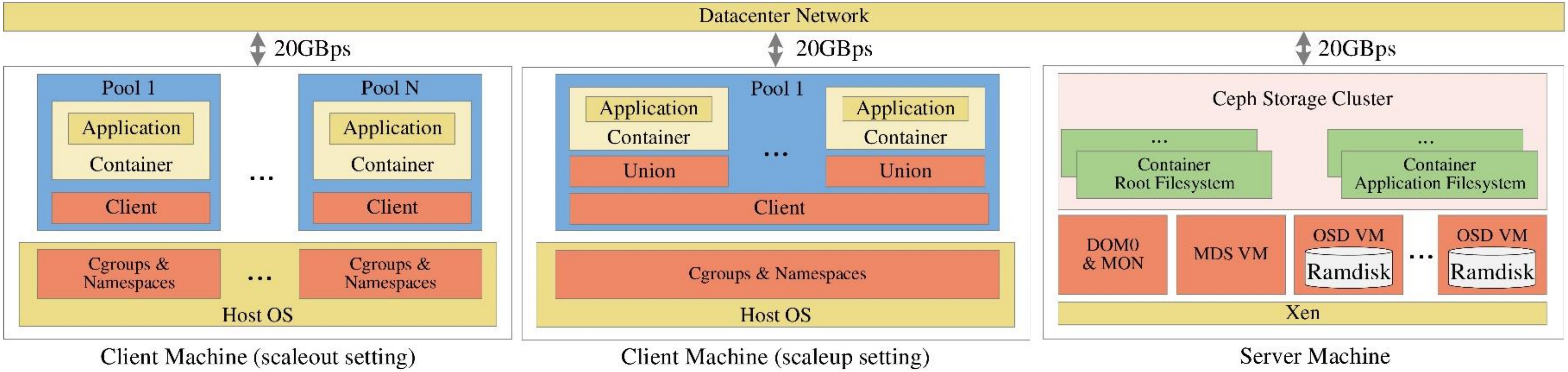
(d) User Client, User Interface (Danaus)

(e) User Client, Dual Interface (Danaus)

Properties

(a) Kernel Client	(b) User Client, Kernel Interface	(c) Library Client	(d) User Client, User Interface (Danaus)	(e) User Client, Dual Interface (Danaus)
Compatibility	Compatibility			Compatibility
	Isolation	Isolation	Isolation	Isolation
Efficiency		Efficiency	Efficiency	Efficiency
			Flexibility	Flexibility

Experimental evaluation setup



2 Servers, each with

- 2 x Quad 16C/16HT Opteron 6378, 256GB RAM
- 2 x 10Gbps Ethernet

Shared Ceph cluster stores container images & application data

- 6 OSDs (2 CPUs, 8GB RAM, 24GB Ramdisk for fast storage)
- 1 MDS, 1 MON (2 CPUs, 8GB RAM)

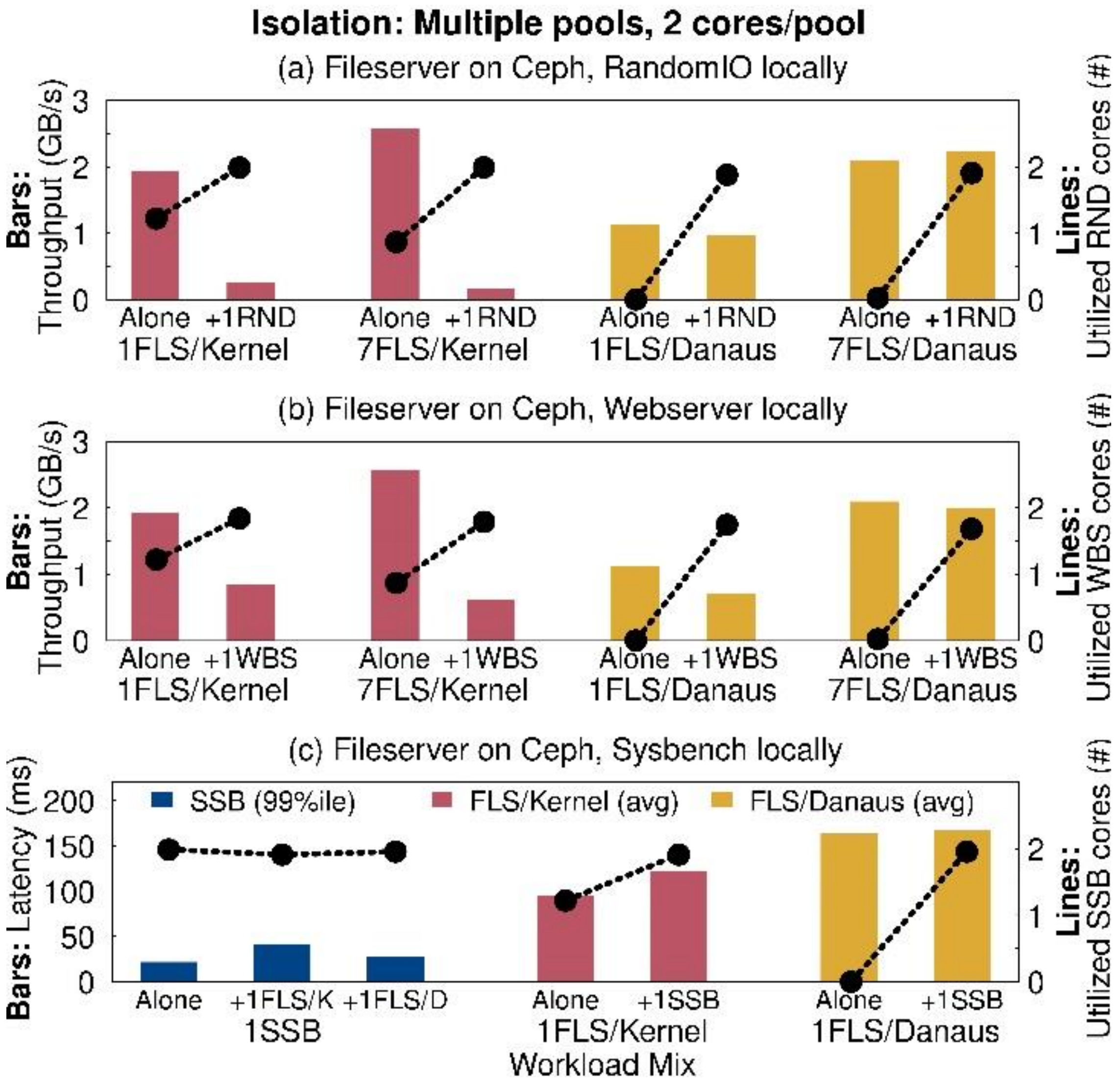
Workload interference

Workloads

- 1 or 7 Fileserver, 1 RandomIO
- 1 or 7 Fileserver, 1 Webserver
- 1 Fileserver, 1 Sysbench

Outcome

- Kernel: up to 16.5x throughput drop of Fileserver, up to 93% raise of Sysbench 99%ile latency
- Danaus: throughput & latency stability, lower performance when standalone but higher when colocated, lower CPU utilization



Data-intensive applications: RocksDB

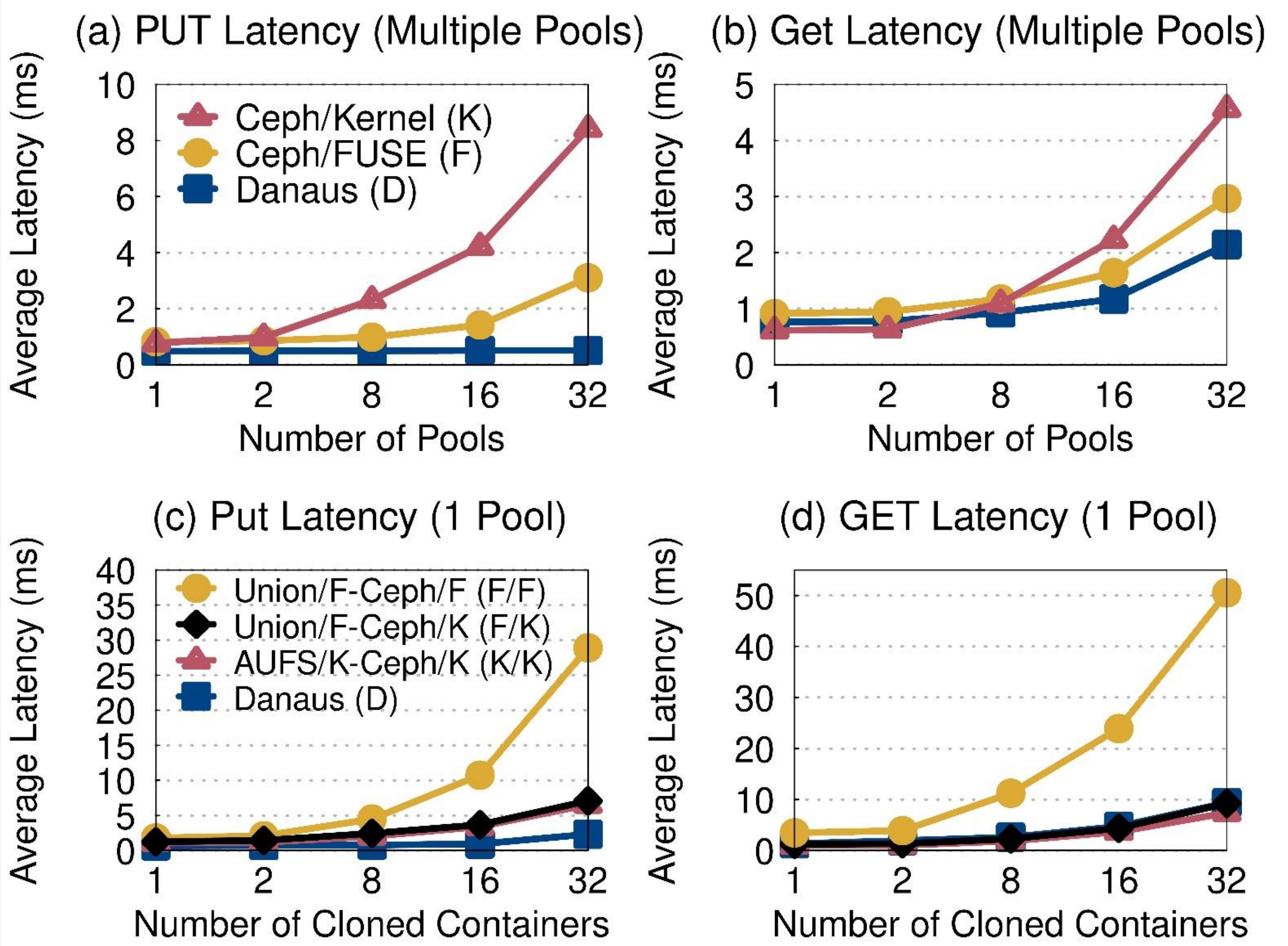
Scaleout (1 Container/Pool)

- Danaus: stable & lower latency than Kernel (up to 16.2x) & FUSE (up to 5.9x)
- FUSE & Kernel: face intense kernel lock contention

Scaleup (up to 32 Containers)

- Danaus: lower put latency than Kernel & FUSE
- Danaus: lower get latency than FUSE, comparable with Kernel

RocksDB (Container: 2 cores, 8GB RAM)



Lessons learned

Shared kernel causes performance interference on containers

- Sources: lock contention, aggressive hardware resource allocation

Container images & data on shared filesystem

- On-demand file transfers during runtime, native data sharing

Functionality & execution separation improves isolation

- Explicit allocation of hardware & software resources to tenants

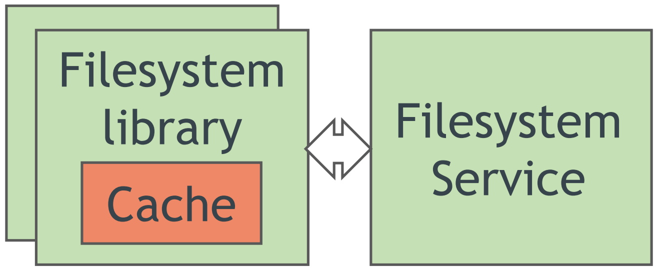
Per tenant user-level client for decentralization & concurrency

- User-level client may be refactored more easily than kernel-level

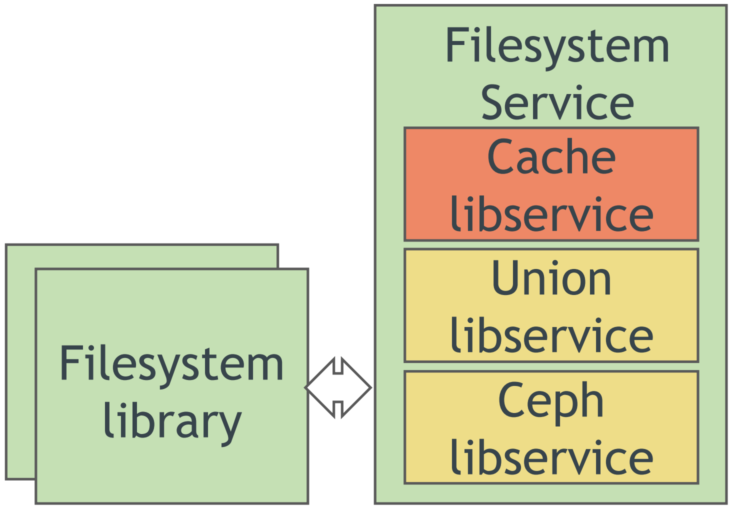
Throughput & latency stability of user-level I/O access & handling

- Performance of workloads insensitive to competing resource demands

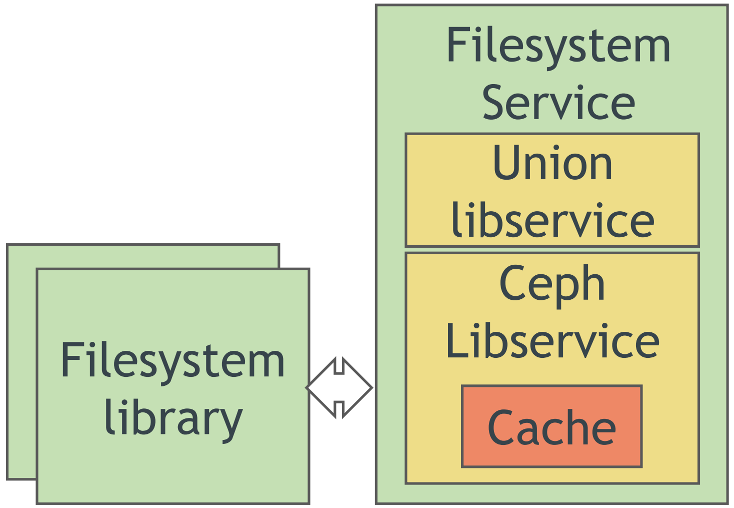
Caching



Cache at filesystem library



Cache at filesystem service as separate libservice

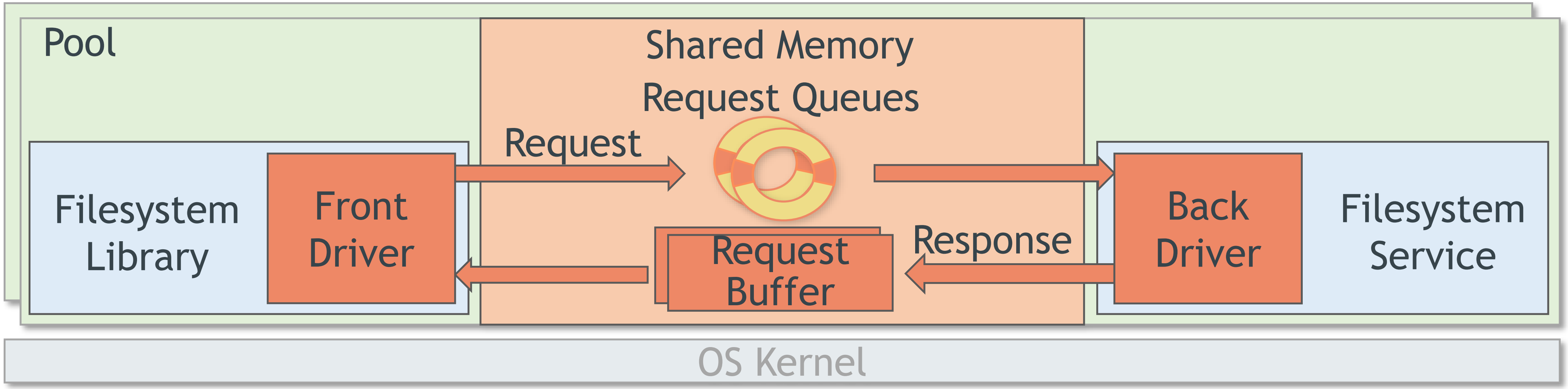


Cache at filesystem service in backend client; Union-based deduplication on top (Danaus)

Consistency of Danaus

- At write return, the written data/metadata has reached the client cache & is visible by subsequent reads to the same client
- CephFS consistency policy propagates the write to other clients

Interprocess communication



User level

- Front driver at filesystem library; back driver at filesystem service
- Minimize mode switches, CPU cache stalls

Per pool data structures

- Utilize shared memory

Request Queue

- I/O requests + small data
- Distinct queue per core group

Request Buffer

- Large data + completion notification
- Distinct per application thread

Pool management

Container engine

- User-level daemon that manages the container pools on a host

Resource reservation and isolation

- Resource usage: cgroups v1: cpu & network, cgroups v2: memory
- Resource names: Linux Namespaces

Storage options

- Danaus
- Backend client: Kernel-based Ceph or FUSE-based Ceph
- Union filesystem: Kernel-based AUFS or FUSE-based unionfs-fuse

Kernel-based mounts through VFS

- Different kernel filesystem instance per kernel mount
- Different user-level FUSE process per FUSE mount

Prototype implementation

Filesystem library: dynamic library preloaded to applications

- POSIX-like API, replaces Kernel VFS
- Functions for synchronous & asynchronous I/O, processes, threads, sockets, pipes, memory mappings

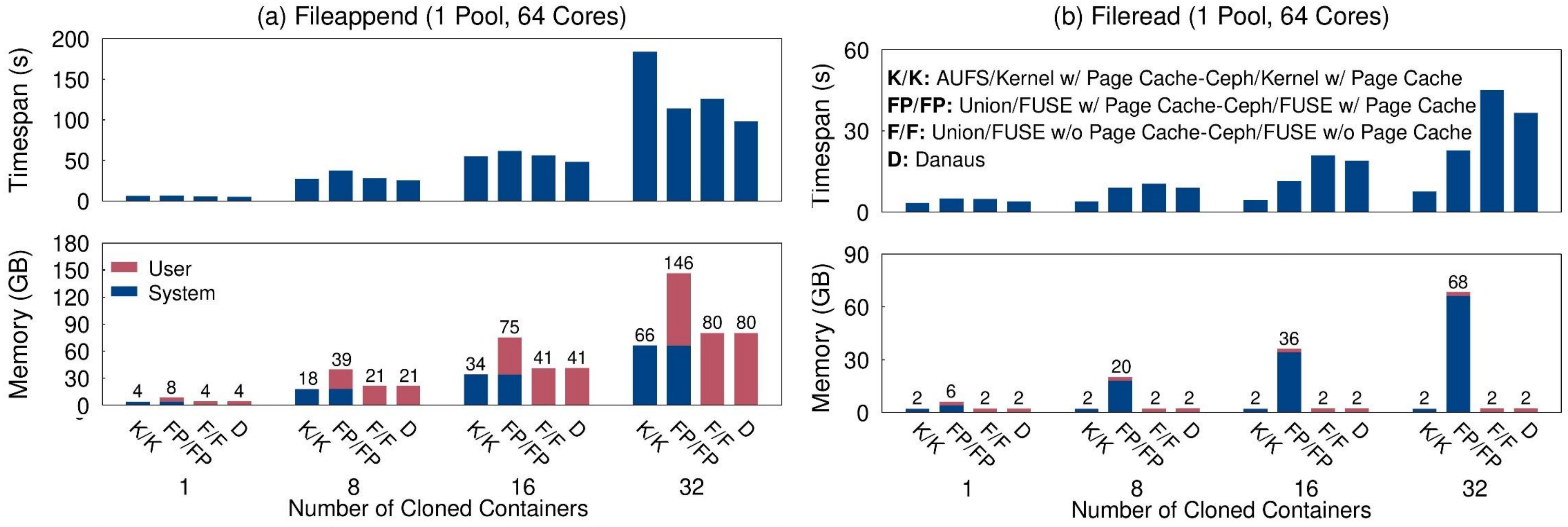
Filesystem service: standalone per-pool process

- Ceph libservice as distributed fs client derived from libcephfs
- Union libservice as union filesystem derived from unionfs-fuse

Container filesystems

- Separate filesystem instances consisting of
  - Private or shared Ceph libservice + (optional) Private Union libservice

Sequential I/O scaleup with cloned containers



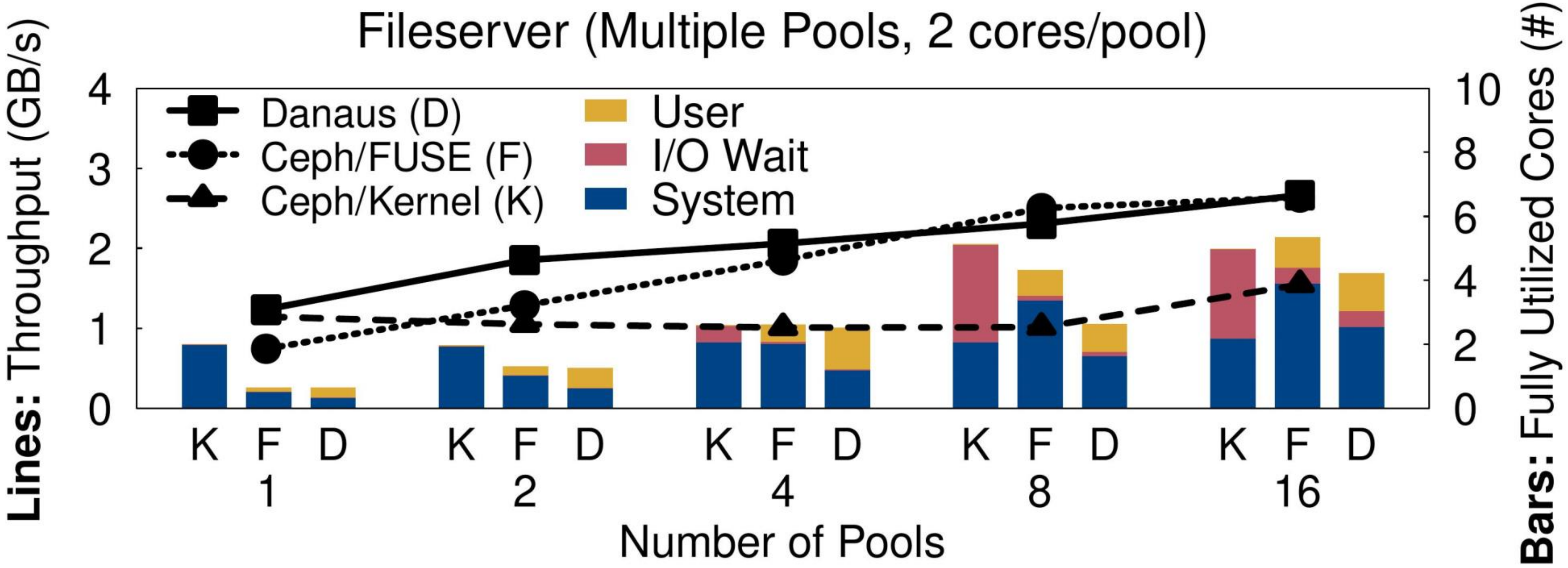
Fileappend (append 1MB to single 2GB file – 50/50 read/write)

- Handling communication & filesystem service at user-level improves performance
- Danaus: up to 46% shorter timespan, comparable memory with kernel

Fileread (read 2GB file in 1MB blocks)

- Concurrency of Danaus limited by coarse-grained Ceph client lock
- FUSE with page cache occupies up to 30x more memory than Danaus

Random I/O scaleout



Danaus achieves better performance than Kernel and FUSE

- Workload: Filebench fileserver
- Danaus is up to 2.3x faster than Kernel
- Danaus is up to 1.7x faster than FUSE

Conclusions

Kernel I/O handling penalizes container performance

- Contention on hardware & software resources

Danaus: Isolation & efficiency for container root filesystems and data

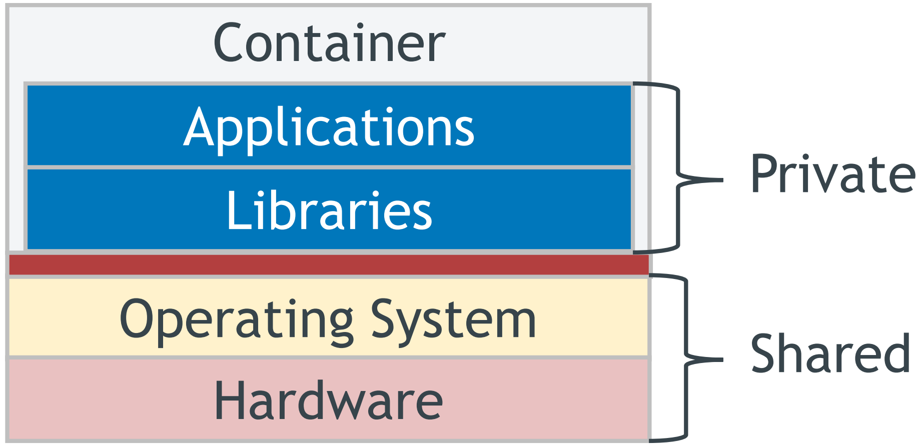
- Isolate storage I/O paths of different tenants
- Serve tenants with distinct clients running & accessed at user-level
- Integrate union filesystem with distributed filesystem client at user-level
- Handle I/O with reserved resources of tenant, avoid kernel contention

Future work

- Port Danaus to production orchestration systems
- Dynamic reallocation of underutilized resources (e.g., memory)
- End-to-end multitenant isolation
- Integrate user-level network software stack to Danaus

Backup

Multitenancy with containers



Containers favor resource utilization

- Low footprint
- Low overhead
- Adjustable resources

Multitenancy issues due to shared kernel I/O path

- Low performance isolation
- Weak security isolation & fault containment
- Implicit inefficiencies due to frequent kernel crossings to service I/O
- Resource duplication

Main reasons

- Resource contention & inflexible sharing of kernel

Sensitivity to kernel I/O contention

Workload

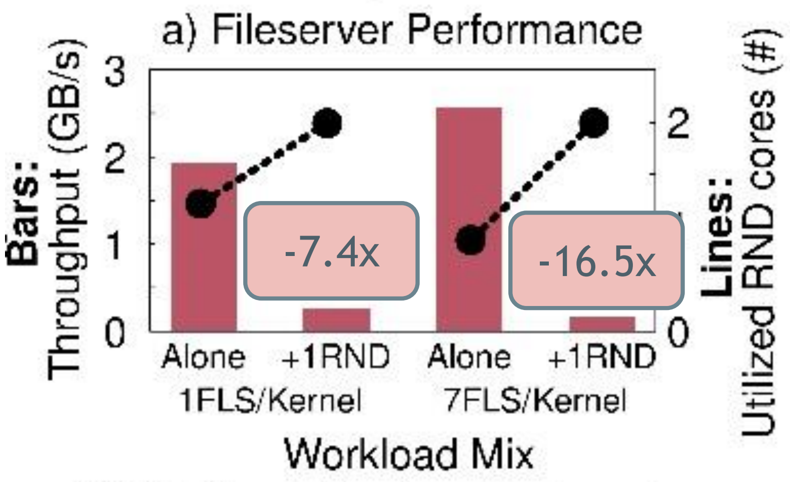
- 1 or 7 Fileserver containers with 2 cores
- 1 RandomIO container with 2 cores
- Fileserver data on Ceph accessed through kernel client, RandomIO data on local ext4 partition

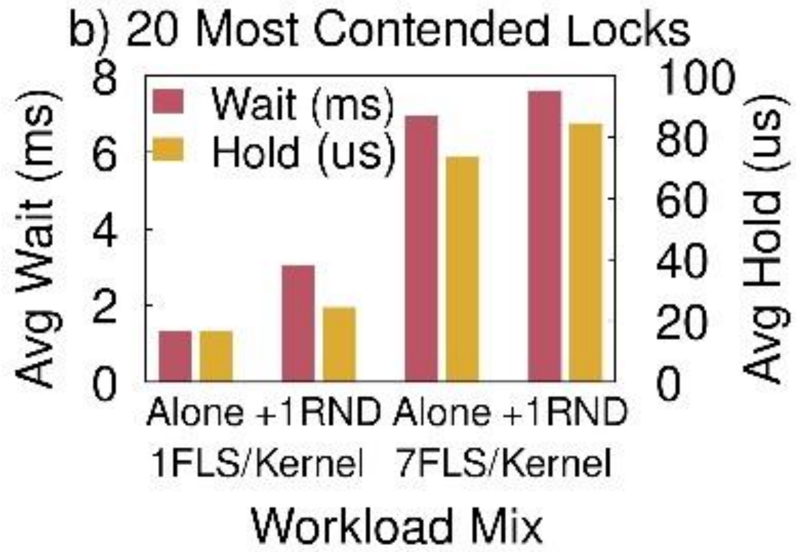
Performance drop due to workload colocation

- Fileserver throughput drops up to 16.5x
- Kernel utilizes all host cores to flush dirty pages
- High contention on shared kernel locks

Effective container isolation requires

- Explicit allocation of hardware & software resources to each collocated workload





Existing Solutions

User-level filesystems with kernel-level interface

- May degrade performance due to user-kernel crossings
- E.g., FUSE, ExtFUSE (ATC'19), SplitFS (SOSP'19), Rump (ATC'09)

User-level filesystems with user-level interface

- Lack multitenant container support
- E.g., Direct-FUSE (ROSS'18), Arrakis (OSDI'14), Aerie (EuroSYS'14)

Kernel structure partitioning

- High engineering effort for kernel refactoring
- E.g., IceFS (OSDI'14), Multilanes (FAST'14)

Lightweight hardware virtualization or sandboxing

- Target security isolation; incur virtualization or protection overhead
- E.g., X-Containers (ASPLOS '19) , Graphene (EuroSys '14)