ACM Symposium on Cloud Computing 2020 (SoCC '20)

A User-level Toolkit for Storage I/O Isolation on Multitenant Hosts

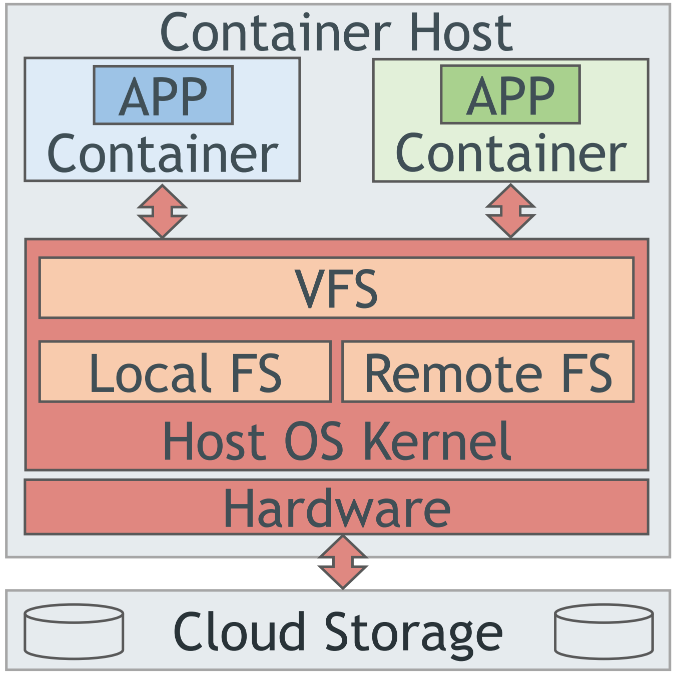
Giorgos Kappes, Stergios V. Anastasiadis

University of Ioannina, Ioannina 45110, Greece

Data-intensive Apps in Multitenant Cloud

Software Containers

- Run in multitenant hosts
- Managed by orchestration systems
- Host data-intensive applications
- Achieve bare-metal performance and resource flexibility



Host OS Kernel

- Serves the containers of different tenants
- Mounts the root and application filesystems of containers
- Handles the I/O to local and network storage devices

Limitations of the Shared Host OS Kernel

1. Unfair use of resources
   - Tenants compete for shared I/O services, e.g., page cache
2. Global configuration rather than per tenant
   - Tenants cannot customize the configuration of system parameters (e.g., page flushing)
3. Software overheads
   - Require complex restructuring of kernel code (e.g., locking)
4. Slow software development
   - Time-consuming implementation and adoption of new filesystems
5. Large attack surface
   - Tenants are vulnerable to attacks/bugs on shared I/O path

Challenges of User-level Filesystems

1. Support multiple processes
   - Connect multiple processes with tenant filesystems at user-level
2. Consistency
   - Coordinate the state consistency across multiple concurrent operations on a shared filesystem
3. Flexible configuration
   - Provide deduplication, caching, scalability across the tenant containers
4. Interface
   - Support POSIX-like semantics (e.g., file descriptors, reference counts)
5. Compatibility with implicit I/O
   - Support system calls with implicit I/O through the kernel I/O path

Motivation: Multitenancy Setup

Tenant

- 1 Container
- 2 CPUs (cgroups v1), 8GB RAM (cgroups v2)

Container host

- Up to 32 tenants

Container application

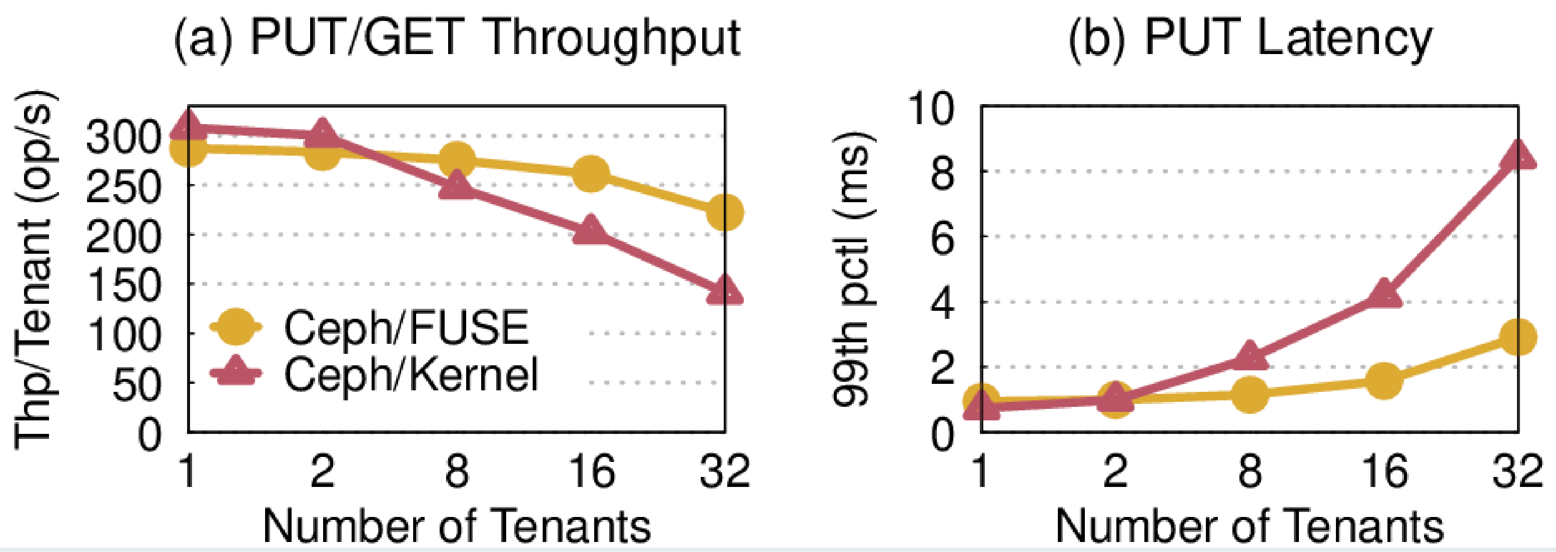
- RocksDB

Shared storage cluster

- Ceph
- Separate root filesystem (directory tree) per container

Motivation: Colocated I/O Contention

RocksDB 50/50 Put/Get



Outcome on 1-32 tenants

- **Throughput (slowdown)** FUSE: up to 23%, Kernel: up to 54%
- **99%ile Put latency (longer)** FUSE: up to 3.1x, Kernel: up to 11.5x

Reasons

- Contention on shared kernel data structures (locks)
- Kernel dirty page flushers running on arbitrary cores

Background on Containers

Lightweight virtualization abstraction that isolates process groups

- Namespaces: Isolate resource names (Process, Mount, IPC, User, Net)
- Cgroups: Isolate resource usage (CPU, Memory, I/O, Network)

Container Image (system packages & application binaries)

- Read-only layers distributed from a registry service on a host

Union filesystem (container root filesystem)

- File-level copy-on-write
- Combines shared read-only image layers with a private writable layer

Application data

- Remote filesystem mounted by the host with a volume plugin, or
- Filesystem made available by the host through bind mount

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)

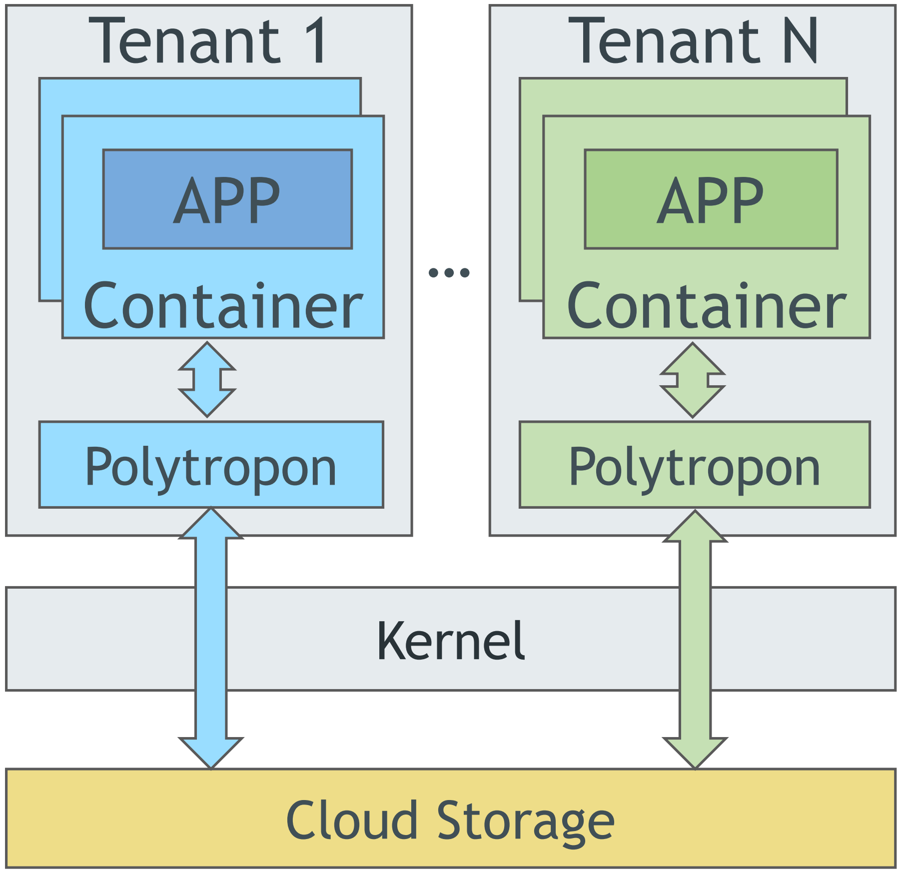
Lightweight hardware virtualization or sandboxing

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

Polytropon: Per-tenant User-level I/O

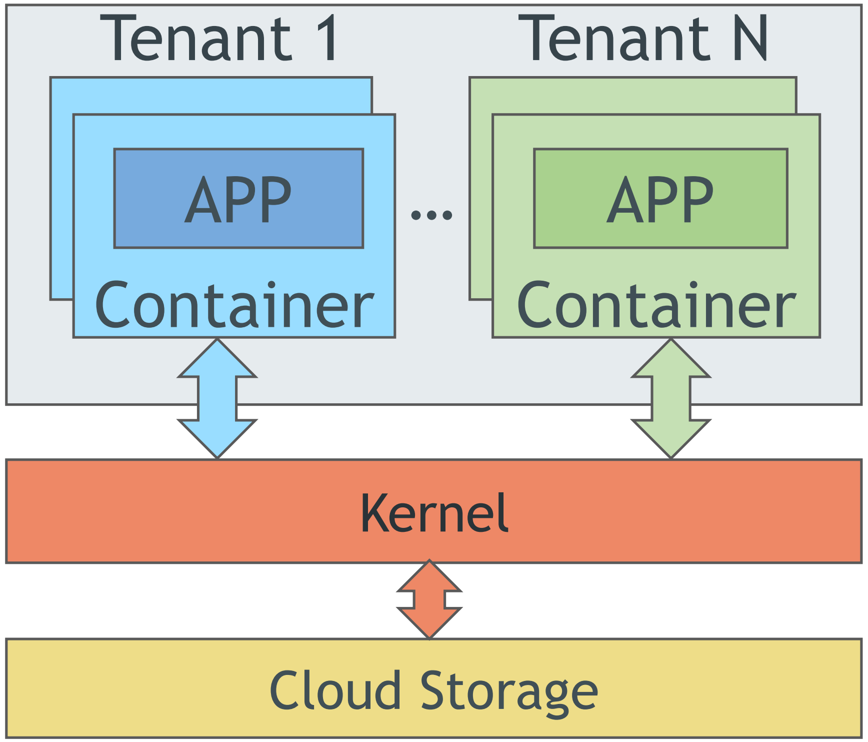
Polytropon

Per tenant user-level filesystems



Legacy

Shared kernel I/O Stack



Design Goals

G1. Compatibility

- POSIX-like interface for multiprocess application access

G2. Isolation

- Tenant containers access their isolated filesystems on a host over dedicated user-level I/O paths

G3. Flexibility

- Cloning, caching, or sharing of container images or application data
- Per-tenant configuration of system parameters (e.g., flushing)

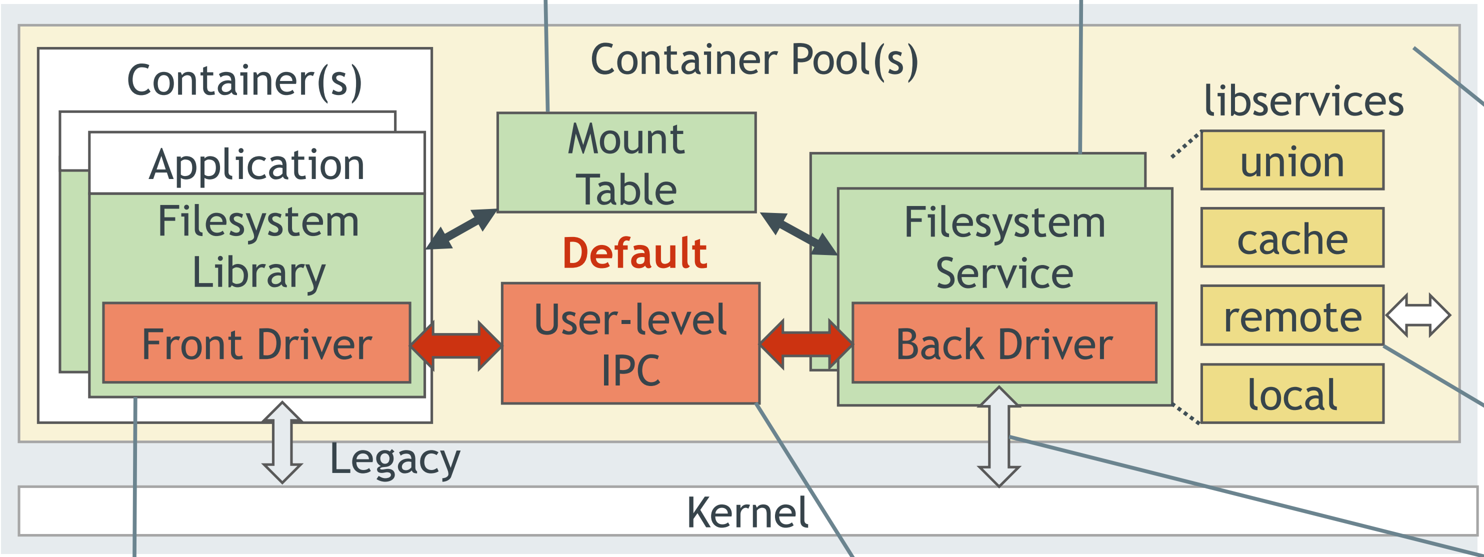
G4. Efficiency

- Containers use efficiently the datacenter resources to access their filesystems

Toolkit Overview

Mount Table
Routes container I/O to filesystems

Filesystem Service
Provisions the private/shared filesystems of containers



Container Pool
Collection of containers
Per tenant / Machine

Composable libservices
User-level storage functions

Filesystem Library
Provides filesystem access to processes

Optimized User-level IPC
Optimized queue and data copy for fast I/O transfers

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

Filesystem Service

Purpose

- Handles the container storage I/O in a pool

libservice

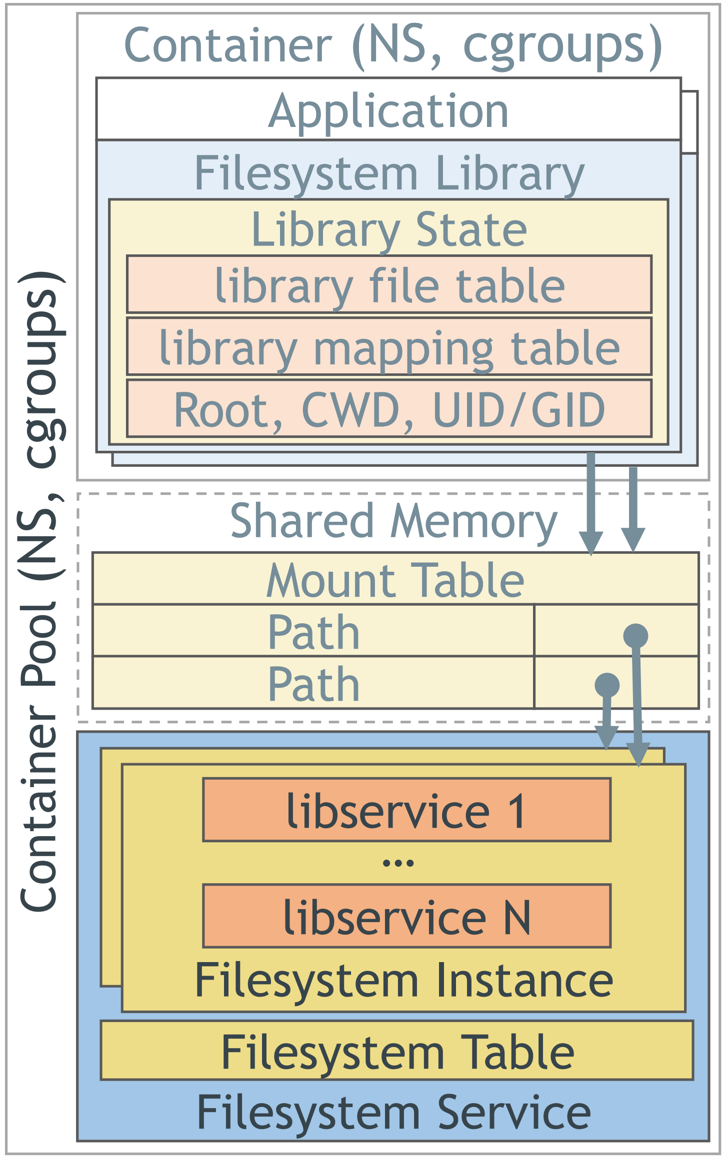
- Standalone user-level filesystem functionality
- Implemented as a library with POSIX-like interface
- Network filesystem client; local filesystem; block volume; union; cache with custom settings

Filesystem Instance

- Mountable user-level filesystem on mount path
- Collection of one or multiple libservices

Filesystem Table

- Associates a mount path with a filesystem instance



Mount Table

Purpose

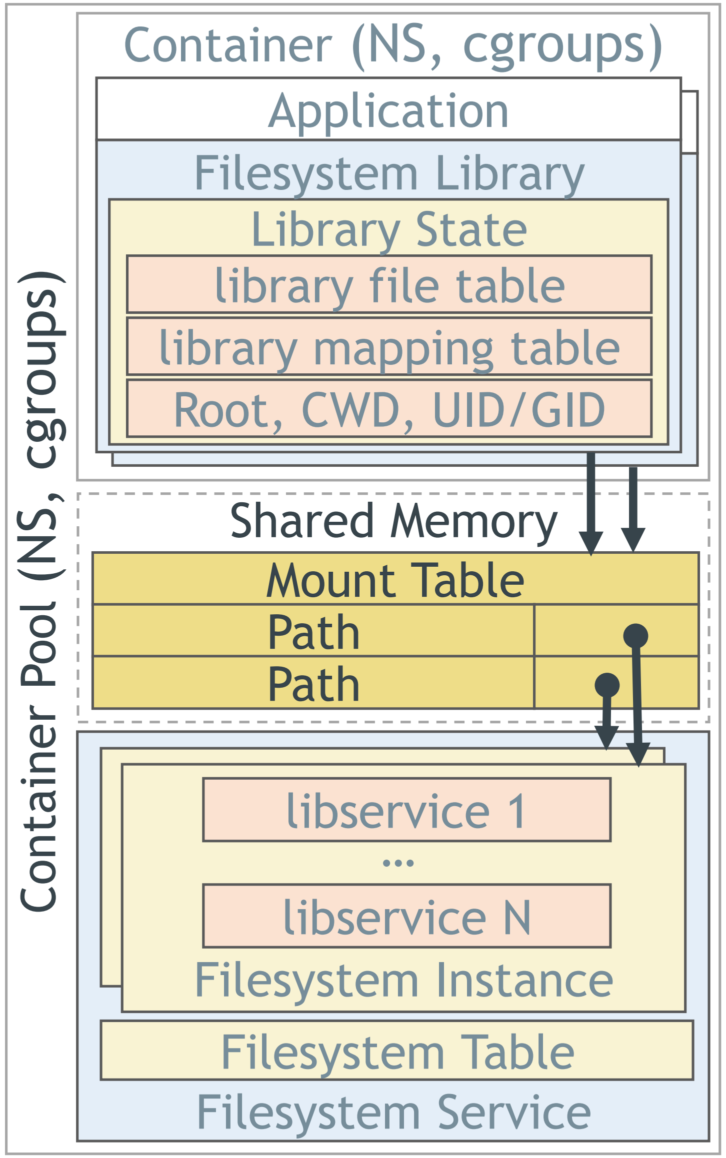
- Translates a mount path to a serving pair of filesystem service & filesystem instance

Structure

- Hash table located in pool shared memory

Mount request: mount path, FS type, options

- Search for longest prefix match of the mount path
- Full match: Filesystem instance already mounted
- Partial match & Sharing: New filesystem instance with shared libservices in matching filesystem service
- Otherwise: New filesystem service



Filesystem Library

Purpose

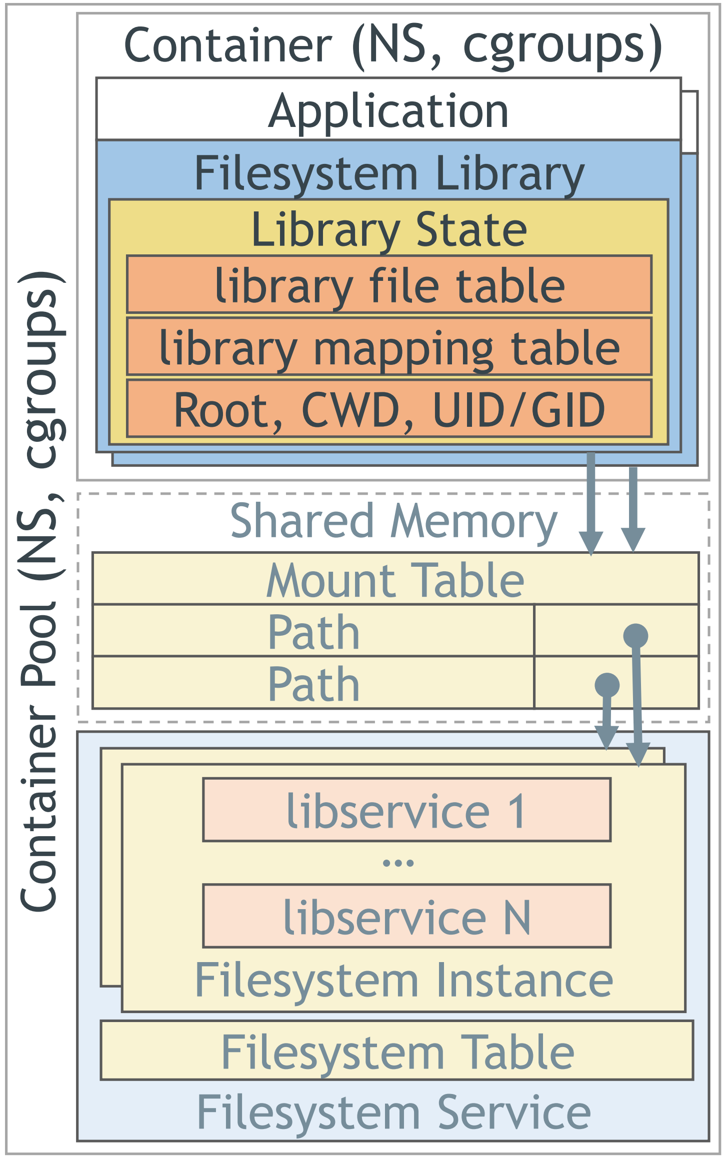
- Provides filesystem access to processes

State management

- Private part (FS library): Open file descriptors, user/group IDs, current working directory
- Shared part (FS service): Filesystem instance, libservice file descriptors, file path, reference count, call flags

Dual interface

- Preloading: Unmodified dynamically-linked apps
- FUSE path: Unmodified statically-linked apps



File Management

Service Open File

- Shared state across applications

Library Open File

- Private state in application process

Libservice File Descriptor

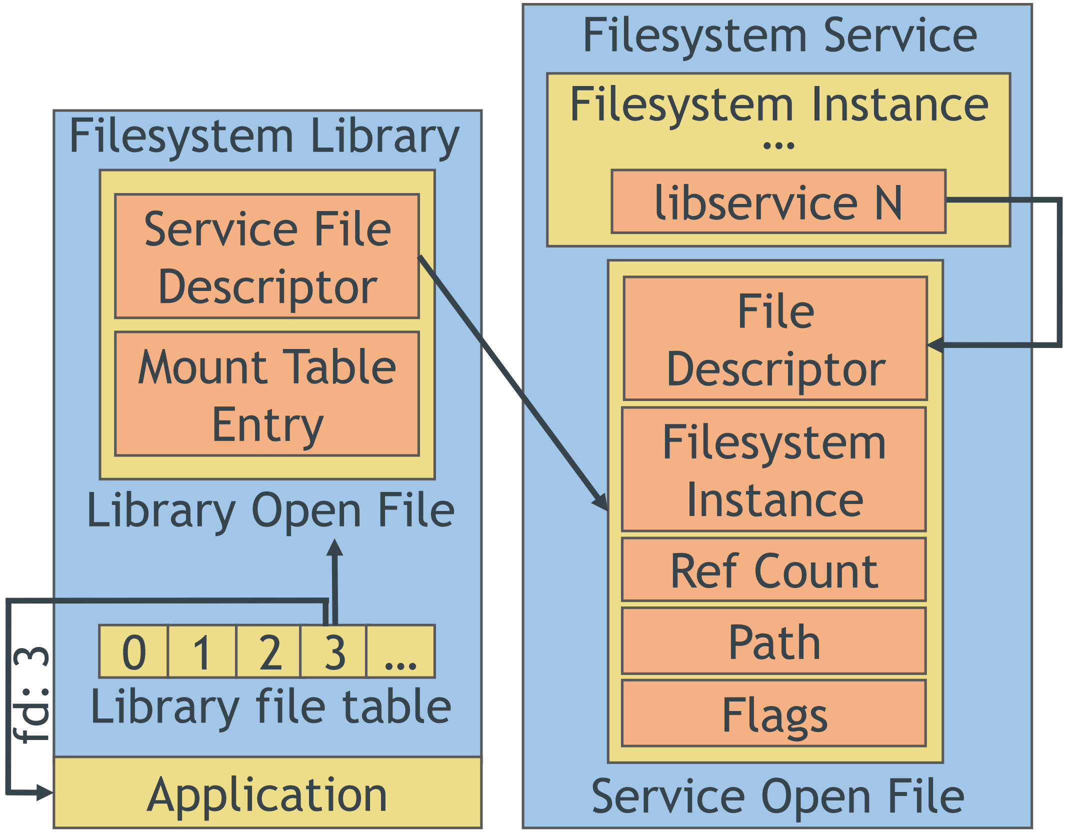
- File descriptor returned by a libservice

Service File Descriptor

- Memory address of a Service Open File

Library File Descriptor

- Index of Library Open File
- Returned to the application



Other Calls

Process Management

- fork/clone: modified to correctly handle shared filesystem state
- exec: preserve a copy of library state in shared memory, retrieved by the library constructor when the new executable is loaded

Memory Mappings

- mmap: emulated at user level by mapping the specified address area and synchronously reading the file contents

Library Functions

- The libc API supported using the fopencookie mechanism

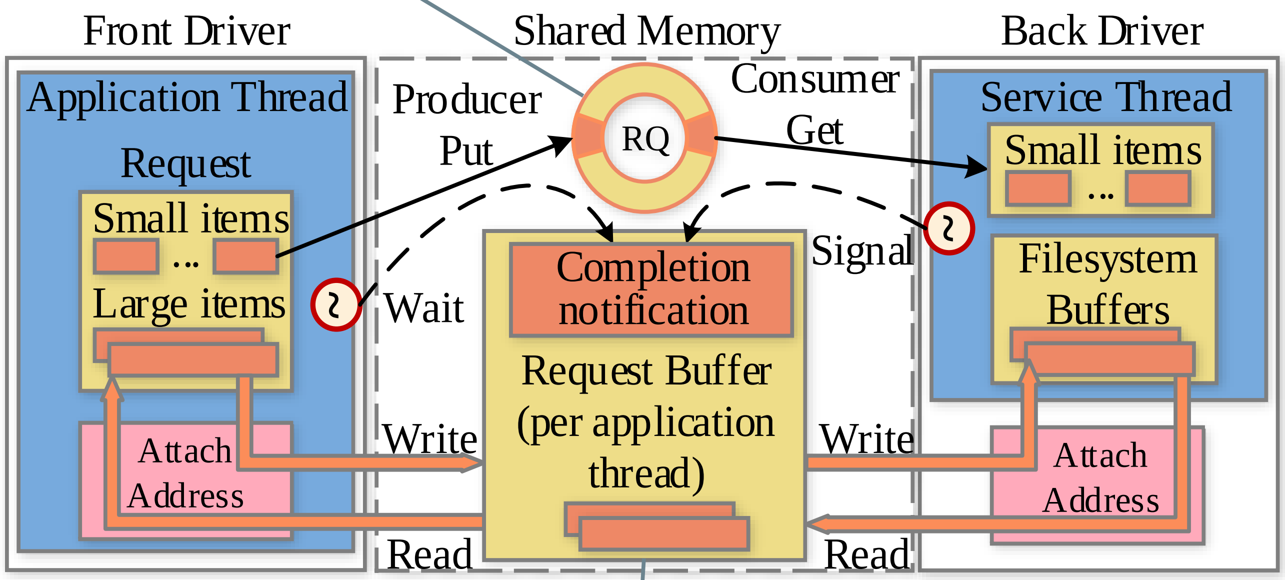
Asynchronous I/O

- Asynchronous I/O supported with code from the musl library

Interprocess Communication

Request Queue

Communication of I/O requests; distinct queue per core group



Request Buffer

Communication of completion notification and large data; distinct per application thread

[FD]: Front Driver [BD]: Back Driver

1. [FD] Copy large data on request buffer
2. [FD] Prepare a request descriptor
3. [FD] Insert the request descriptor to a request queue
4. [FD] Wait for completion on the request buffer
5. [BD] Retrieve the request descriptor and the request buffer
6. [BD] Process the request, copy the response to the request buffer
7. [BD] Notify the front driver for completion
8. [FD] Wake up and copy the response to the application buffer

Relaxed Concurrent Queue Blocking (RCQB)

Idea

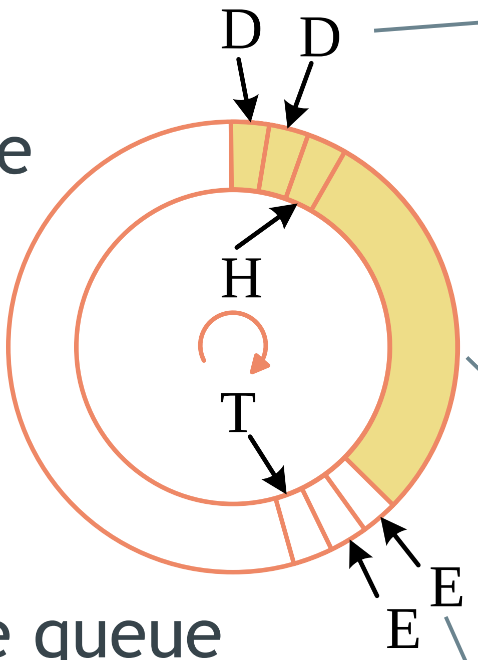
- 1st Stage: Distribute operations sequentially
- 2nd Stage: Let them complete in parallel potentially out of FIFO order

Goals

- High operation throughput
- Low item wait latency in the queue

Implementation

- Fixed-size circular buffer



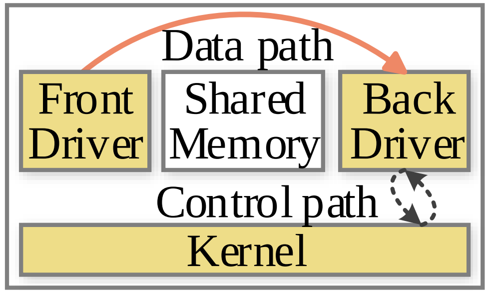
Dequeue operation:
1. Allocate a slot sequentially with fetch-and-add on head
2. Lock the slot for dequeue, remove the item, unlock the slot

Dequeuers follow the enqueuers

Enqueue operation:
1. Allocate a slot sequentially with fetch-and-add on tail
2. Lock the slot for enqueue, add the item, unlock the slot

Cross-Memory Attach (CMA)

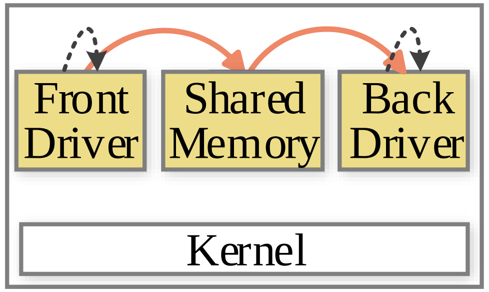
- Copy data between process address spaces with zero kernel buffering



(a) CMA (1 Copy)

Shared-Memory Copy (SMC) with libc memcpy

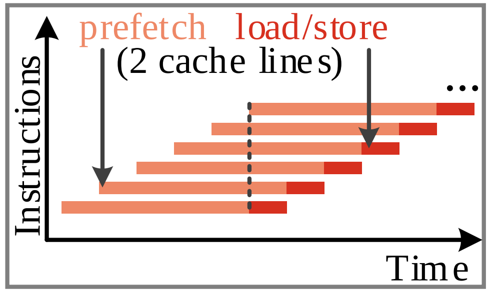
- Copy data from source to shared memory buffer
- Copy data from shared memory buffer to target



(b) SMC / SMO (2 Copies)

Shared-Memory Optimized (SMO) pipeline of 2 stages

- One time: Non-temporal prefetch of 10 cache lines
- 1st Stage: Non-temporal prefetch of 2 cache lines
- 2nd Stage: Non-temporal store of 2 cache lines



(c) SMO (Copy pipeline)

Pool Management

Container engine

- Standalone process that manages the container pools on a host

Isolation

- Resource names: Linux namespaces
- Resource usage: cgroups v1: CPU, network, cgroups v2: memory

Storage drivers

- Mount container root & application filesystems

Pool start

- Fork from container engine process, create & join the pool’s namespaces

Container start

- Fork from container engine process, inherit pool’s namespaces, or create new namespaces nested underneath

Prototype Supported Filesystems

Polytropon (user-level path and execution)

- Root FS: Union libservice over a Ceph client libservice
- Application FS: Ceph client libservice

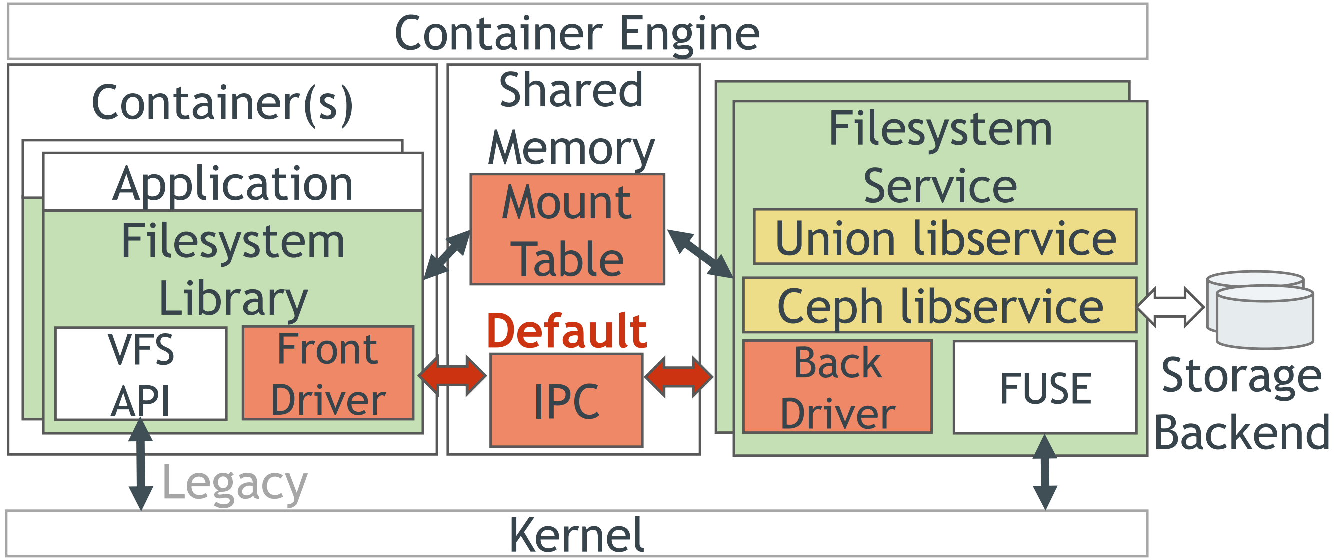
Kernel (kernel-level path and execution)

- Root FS: Kernel-level AUFS over kernel-level CephFS
- Application FS: Kernel-level CephFS

FUSE (kernel-level path and user-level execution)

- Root FS: FUSE-based UnionFS over FUSE-based Ceph client
- Application FS: FUSE-based Ceph client

Danaus as a Polytropon Application



Dual Interface

- Default: Shared memory IPC
- Legacy: FUSE

Filesystem Instance

- Union libservice (optional)
- Ceph libservice

Union libservice

- Derived from unionfs-fuse
- Modified to invoke the libservice API instead of FUSE

Ceph libservice

- Derived from libcephfs

Experimental Evaluation Setup

2 Servers

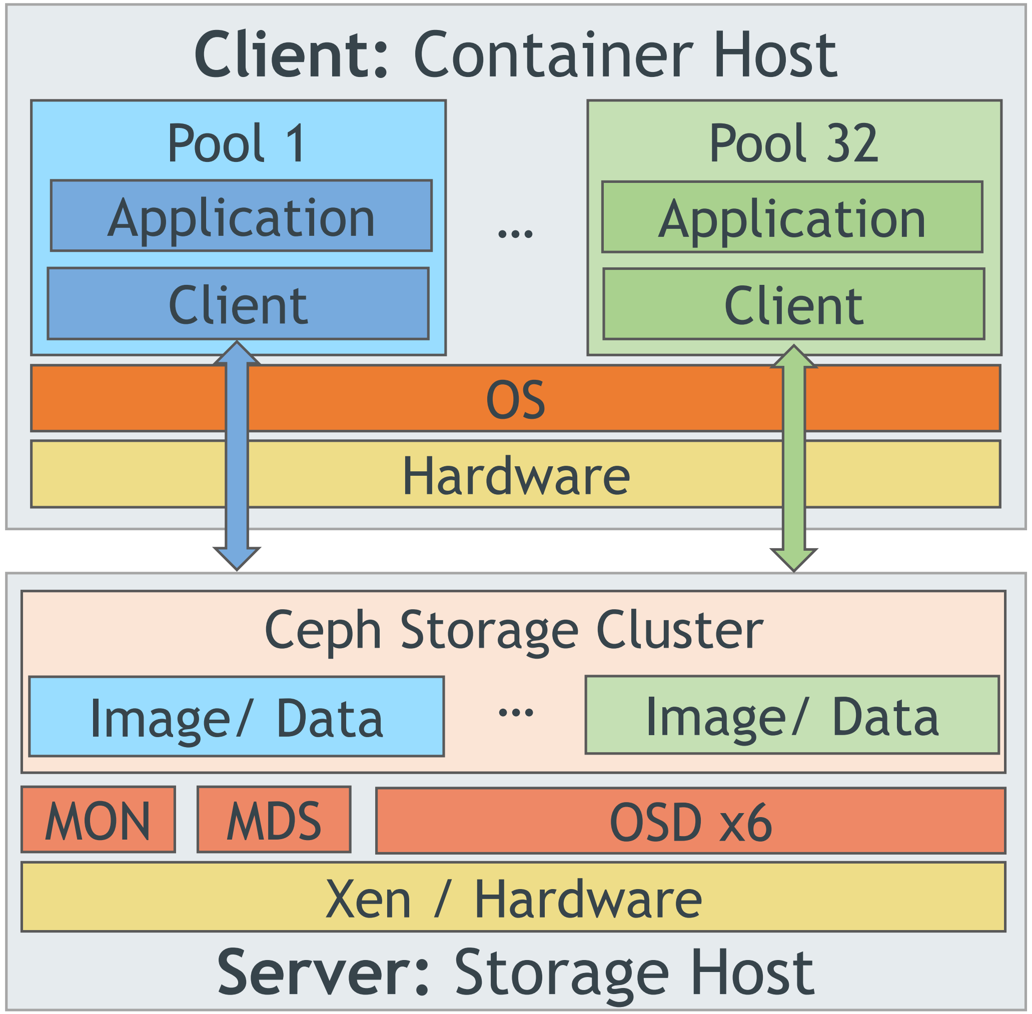
- 64 Cores, 256GB RAM
- 2 x 10Gbps Ethernet
- Linux v4.9

Shared Ceph Storage Cluster

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

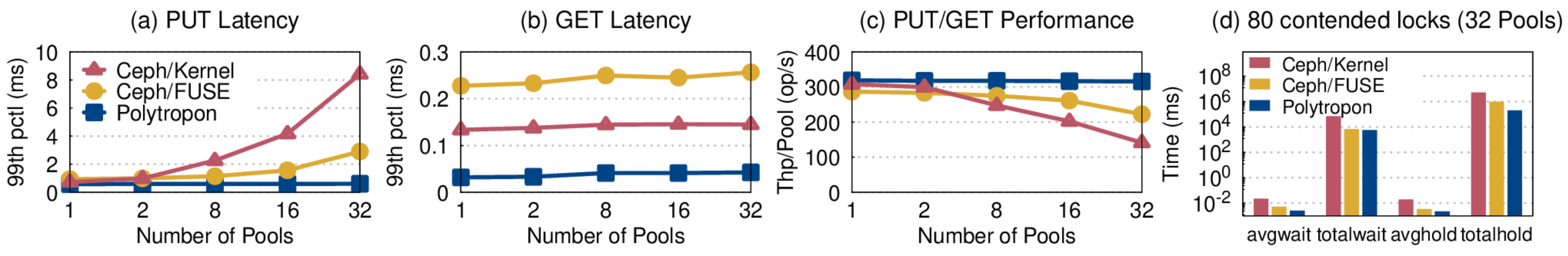
Container Pool

- 1 Container
- Cgroups v1 (CPU), v2 (Memory)



Data-Intensive Applications: RocksDB

RocksDB 50/50 Put/Get



Polytropon achieves faster I/O response & more stable performance

- **Put latency (longer)** FUSE: up to 4.8x, Kernel: up to 14x
- **Get latency (longer)** FUSE: up to 4.2x, Kernel: up to 7.2x
- **Throughput (slowdown)** FUSE: up to 23%, Kernel: up to 54%

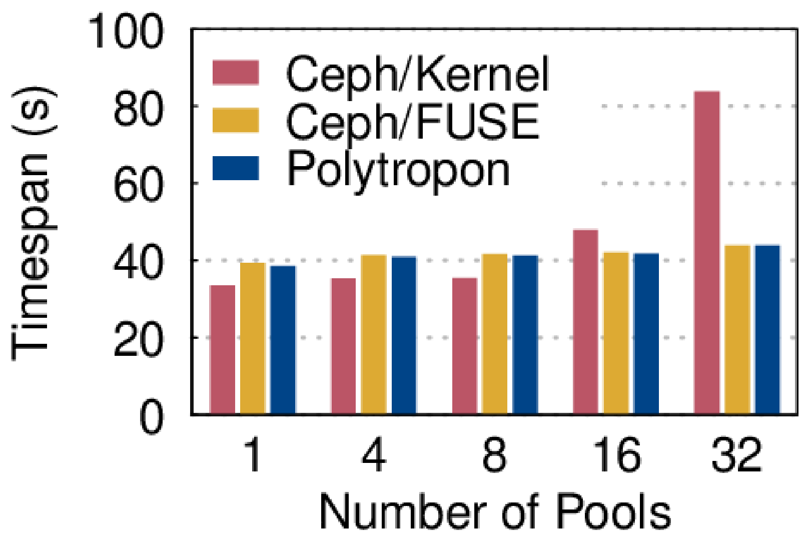
FUSE and Kernel client face intense kernel lock contention

Data-Intensive Applications: Gapbs, Source Diff

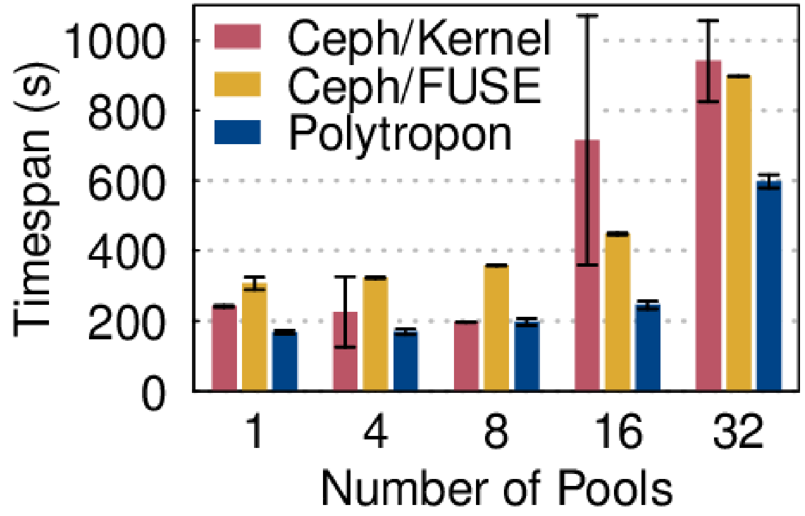
Workload: BFS on directed graph (1.3GB)

Read Intensive

Gapbs - BFS



Source diff



Workload: Difference between Linux v5.4.1 & v5.4.2

Read & Metadata Intensive

Gapbs: Polytropon and FUSE keep the timespan stable regardless of pool count

- **Timespan (longer)** Kernel: up to 1.9x
- Kernel client slowed down by wait time on spin lock of LRU page lists

Diff: Polytropon offers stable performance, the I/O kernel path causes delays

- **Timespan (longer)** FUSE: up to 1.9x, Kernel: up to 2.9x
- Kernel I/O causes substantial performance variability: 32.6x higher std

Cloned Containers

1 Pool of

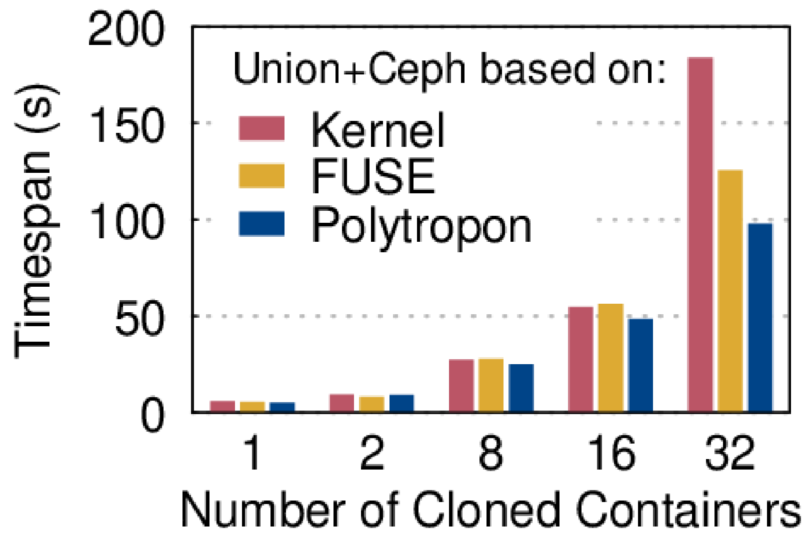
- 64 Cores
- 200GB RAM

Cloned Containers

- Separate root filesystem (Union libservice): a writable branch over a read-only branch
- The branches are accessible over a shared Ceph libservice

Workload:
1. Open a cloned 2GB file,
2. append 1MB,
3. close it

Fileappend



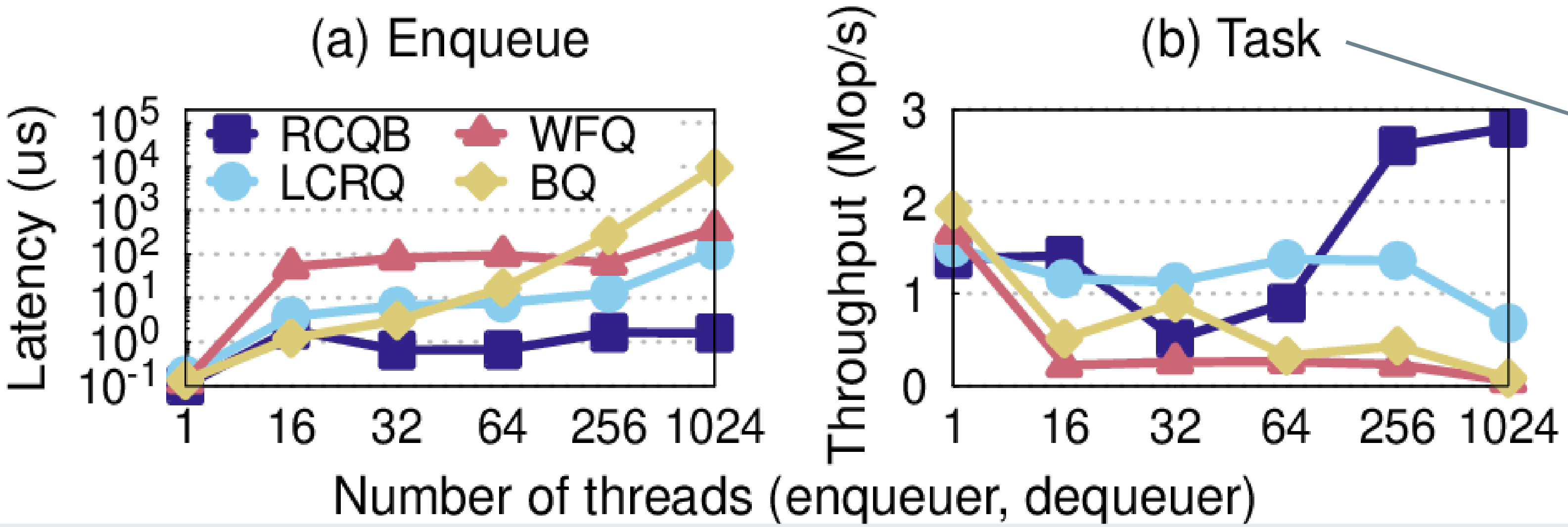
50-50 read/write

Handling both the communication and filesystem service at user-level improves performance

- Fileappend: Opens a cloned 2GB file, appends 1MB, closes it
- **Timespan (longer)** FUSE: up to 28%, Kernel: up to 88%

Concurrent Queues

Closed system, separate threads for enqueuer & dequeuer



Task: Rate at which enqueuers receive completions by dequeuers

1 Pool of

- 64 Cores

RCQB achieves lower average enqueue latency & higher task throughput due to parallel completion of enqueue and dequeue operations

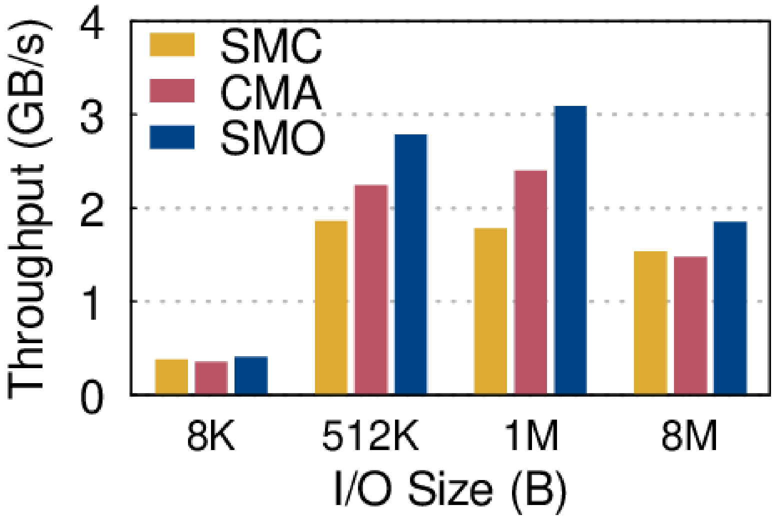
- **Average enqueue latency (longer)** LCRQ: up to 77x, WFQ: up to 246x, BQ: up to 5881x
- **Task throughput (lower)** LCRQ: up to 4x, WFQ: up to 34x , BQ: up to 52x

IPC Performance

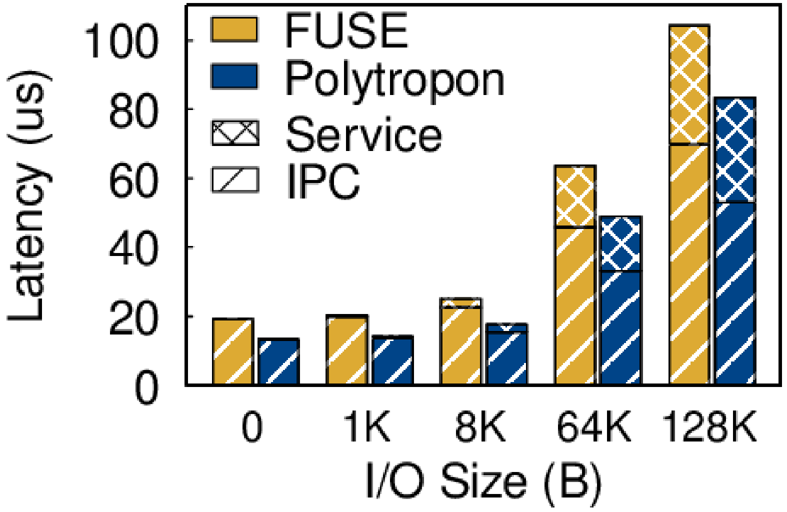
1 Pool of

- 8 Cores
- 32GB RAM

Seqread/Ceph (Polytropon)



File I/O RAM



The SMO pipelined copy improves data transfer

- SMO is 66% faster than SMC and 29% faster than CMA

Handling the IPC at user-level makes Polytropon faster than FUSE

- FUSE: 32-46% longer to serve reads due to 25-46% higher IPC time

Conclusions & Future Work

Problem: Software containers limit performance of data-intensive apps

- Storage I/O contention in the shared kernel of multitenant hosts

Our Solution: The Polytropon toolkit

- User-level components to provision filesystems on multitenant hosts
- Optimized IPC with Relaxed Concurrent Queue & pipelined memory copy

Benefits

- Combine multitenant configuration flexibility with bare-metal performance
- I/O isolation by letting tenants run their own user-level filesystems
- Scalable storage I/O to serve data intensive containers

Future Work

- Native user-level support of most I/O calls (e.g., mmap, exec)
- Support network block devices & local filesystems with libservices

Process Management

Fork, Clone

- FS service: increases the reference count of opened files in parent
- FS library: invokes the native fork, replicates library state from parent to child

Exec

1. Create copy of library state in shared memory with process ID as possible key
2. Invoke the native exec call, load the new executable, call the FS library constructor
3. The FS library constructor recovers the library state from the copy

Memory Mappings

mmap(addr, length, prot, flags, fd, offset)

1. maddr = mmap(addr, length, prot, MAP_ANONYMOUS, -1, 0)
2. Create a Memory Mapping & add it to the Memory Mapping Table
3. polytropon_pread(fd, maddr, length, offset)
4. Increase backing file reference counter
5. return maddr