APSys 2020 - Tsukuba Japan

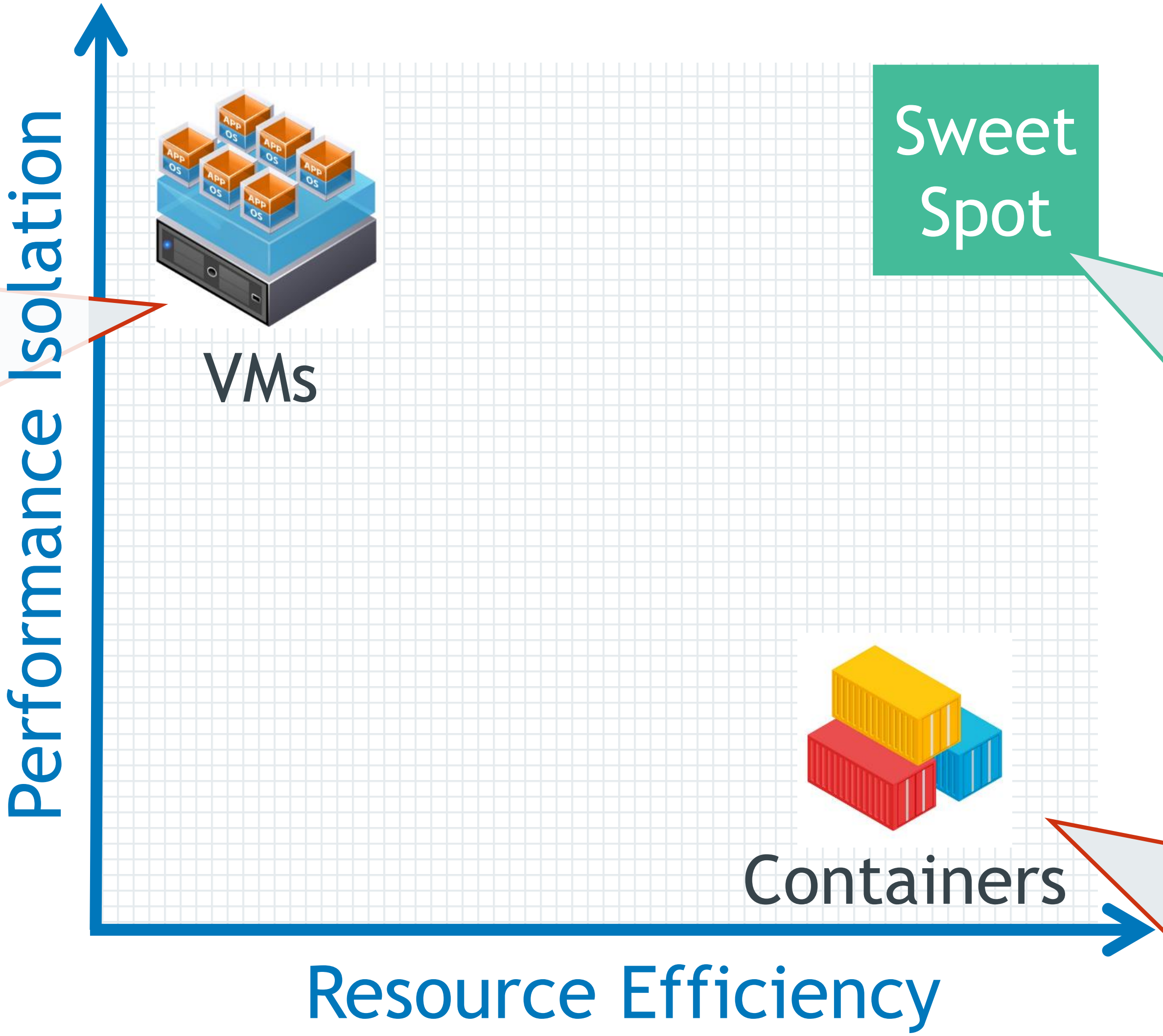
Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation

Giorgos Kappes, Stergios Anastasiadis

University of Ioannina, Greece

Isolation or Efficiency?

High overhead
High footprint
Static resources
Strong isolation



Low overhead
Low footprint
Adaptive resources
Strong isolation

Low overhead
Low footprint
Adaptive resources
Weak isolation

Serving Data-Intensive Applications

User facing; Processing vast amounts of data; Variable demands

- E.g., Key-value stores, interactive applications, real-time big data

Predictable performance

- Latency-sensitive
- Strict Service-Level Objectives (tail latency)

Sensitive to interference

- Tail latency increases with load

How to achieve high resource efficiency?

- Dynamic resource allocation
- Workload collocation

Container Resource Isolation

Limit the container resource view and usage

- Private resources: assigned exclusively to tenants
- Shared resources: limit enforcement, accounting
- Isolation: A tenant should only consume its assigned resources

Namespaces: Isolate resource names

- **Process:** Process IDs
- **Mount:** Mount Points
- **IPC:** SysV IPC, Message Queues
- **User:** User and Group IDs
- **Net:** Net Devices, stacks, ports

Cgroups: Isolate resource usage

- **CPU:** CPU, Cpuset controllers
- **Memory:** Memory controller
- **I/O:** IO Controller
- **Network:** net_cls (class), net_prio (priority) controllers

Multitenancy Setup

Tenant

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

Container Host

- Up to 32 tenants

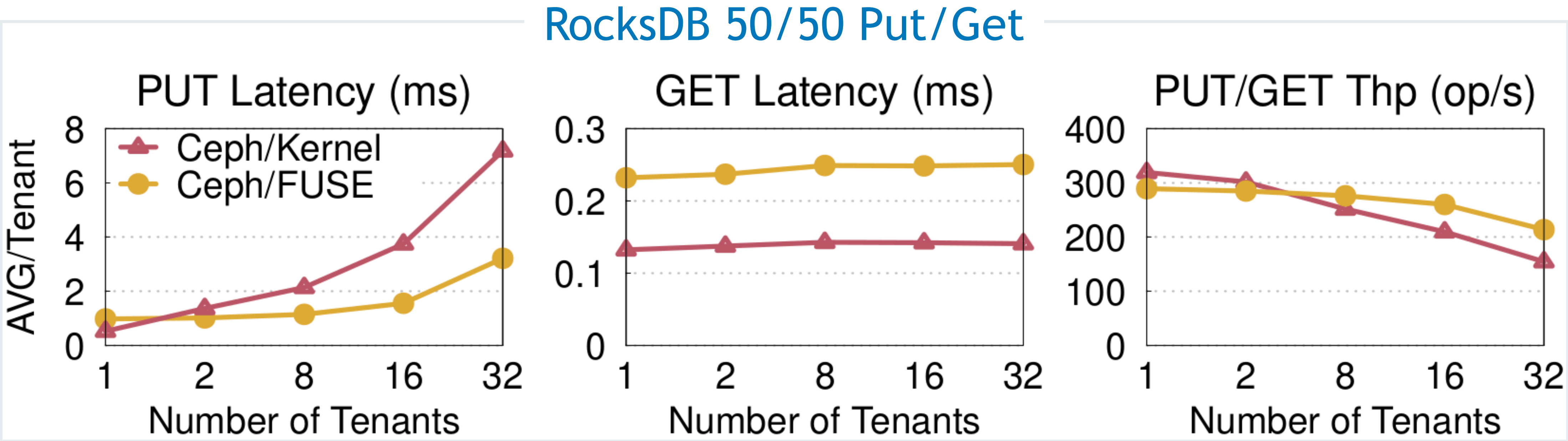
Container Application

- RocksDB

Shared Storage Cluster

- Ceph
- Per container root directory trees

Motivation: Collocated I/O Contention



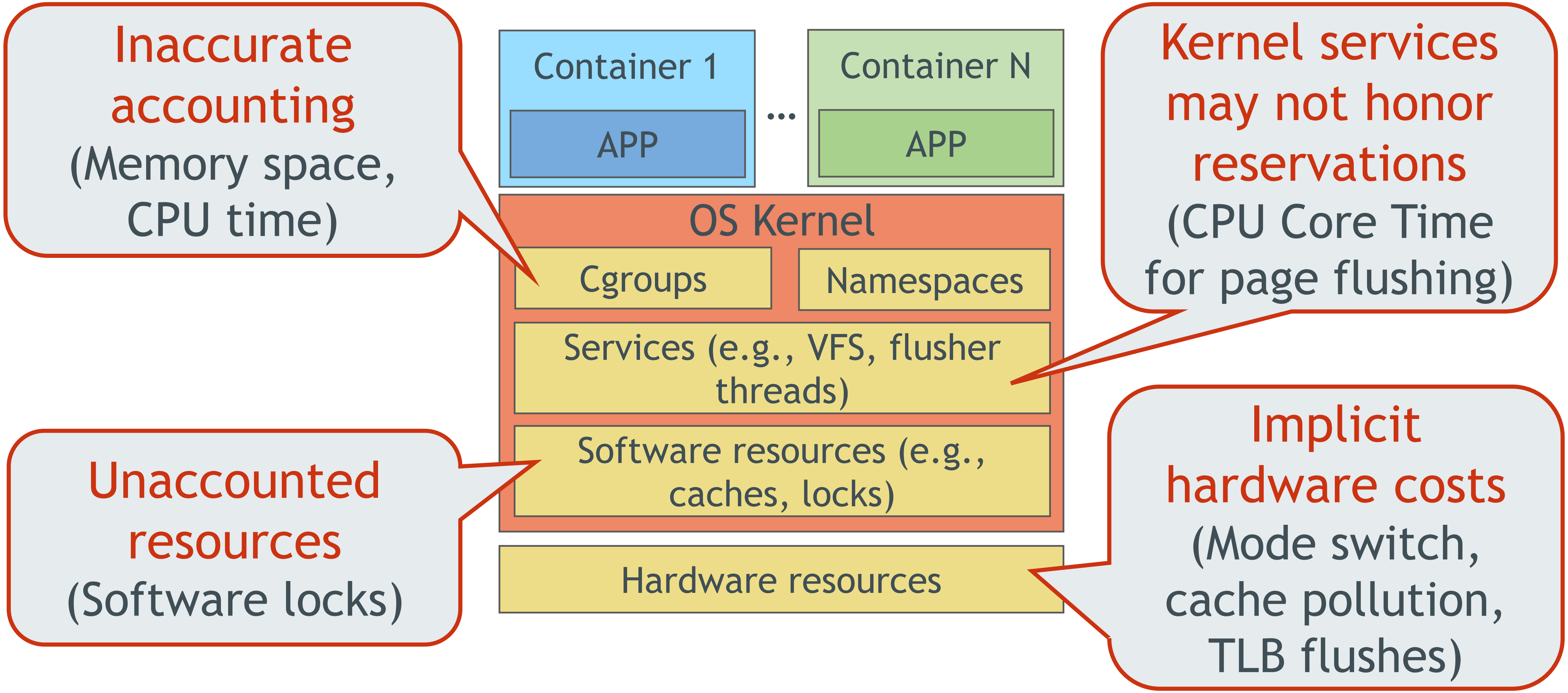
Outcome

- Workload collocation: severe performance variability & slow down

Reasons

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

I/O Multitenancy Issues



Existing Solutions

Kernel structure partitioning

- Performance overheads from static partitioning
- High engineering effort to refactor the entire kernel
- E.g., IceFS (OSDI ‘14), Multilanes (FAST ‘14)

Dynamic resource allocation

- Hardware resources only (e.g., CPU, RAM)
- No guarantee for fair allocation of system services (page flushing)
- E.g., PARTIES (ASPLOS ‘19)

Lightweight hardware virtualization

- Virtualization overheads, static resource allocations
- E.g., LightVM (SOSP ‘17), X-Containers (ASPLOS ’19)

The libservices unified framework

Goals

- Isolation: Tenant resource utilization limited by reservations
- Elasticity: Dynamic resource allocation
- Efficiency: Low virtualization cost
- Compatibility: Unmodified applications

libservices

- User-level storage functions derived from existing I/O libraries
- Build complex filesystem services for the client and server

Key concepts

- Same design pattern at client and server
- Dynamic provisioning of storage systems per tenant
- User-level storage services over reserved resources

Dynamic Storage Provisioning

Storage System

- Client-Server Architecture

Application Filesystem

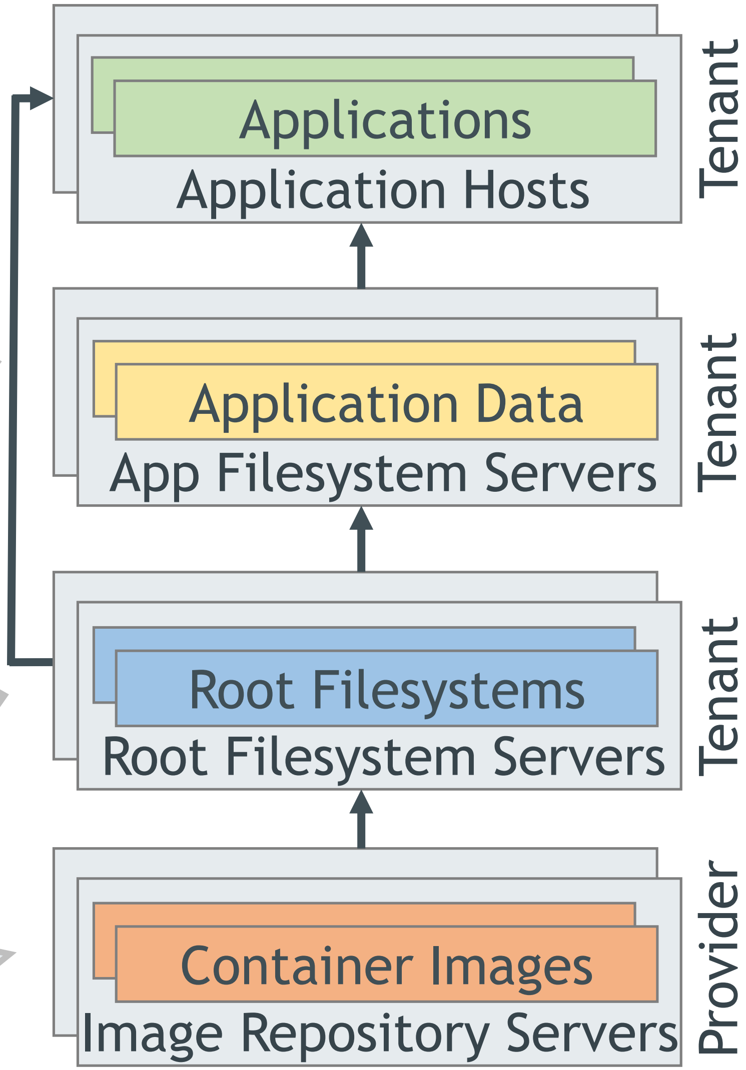
- Stores application data
- Serves tenant applications

Root Filesystem

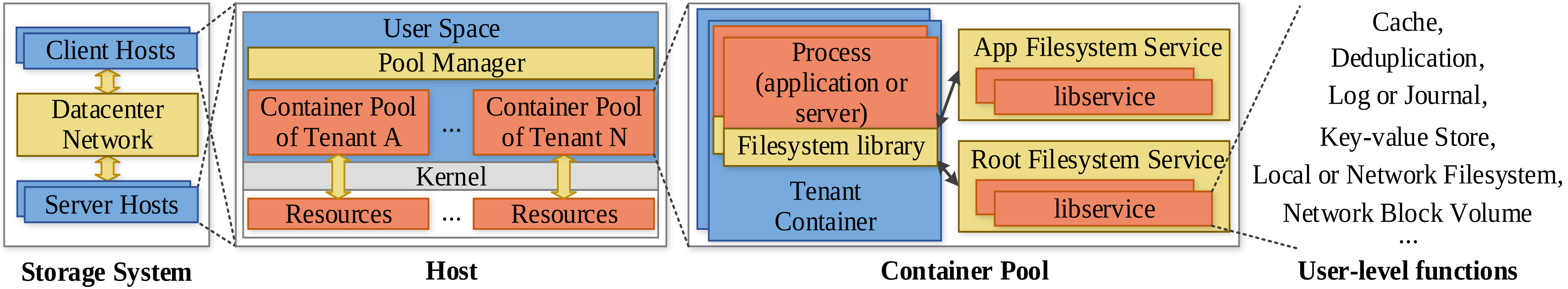
- Stores container root filesystems
- Serves Application containers & Application Filesystem servers

Image Repository

- Stores and distributes container images



User-level Storage Framework



Container Pool

- Collection of Containers
- Per tenant / machine

Pool Manager

- Manages pool resources
- Per machine

Filesystem Service

- Collection of user-level I/O services per tenant

Filesystem Library

- Storage access to applications at user level

Libservice

Standalone user-level storage function, e.g.,

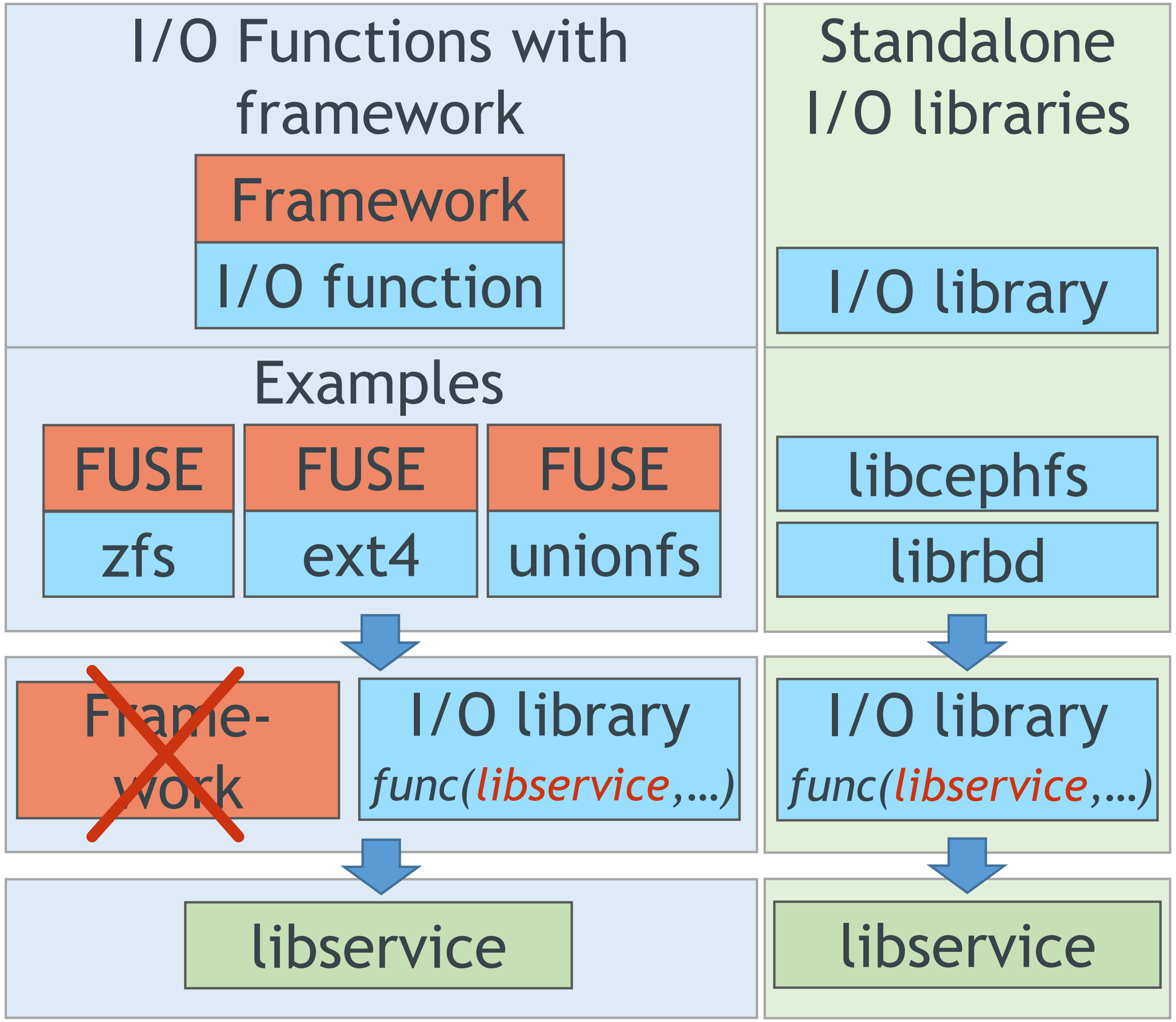
- Network filesystem client
- Local filesystem
- Block Volume
- Cache
- Deduplication
- Log
- Key-Value store

Filesystem Service

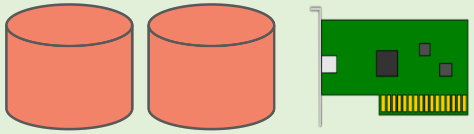
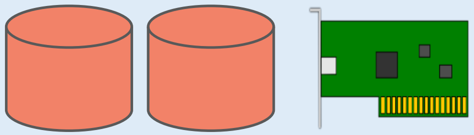
- Stack or tree of libservices
- Requests pass through libservices from top to bottom

Building Libservices

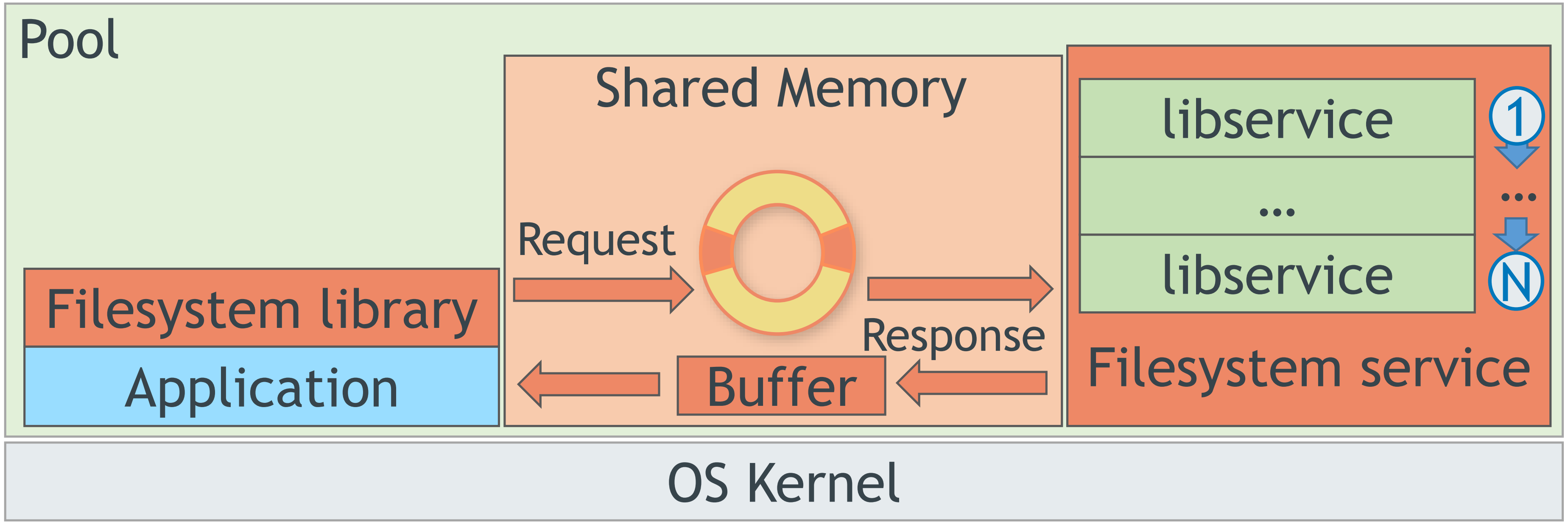
1. Use existing I/O component
   - I/O function & framework
   - Standalone I/O library
2. Create standalone library
   - Separate I/O function from framework, global deps
3. Port I/O library to libservice interface
   - Libservice object first parameter to I/O functions



Libservice Functions

Client	Server
Serves applications, pool managers, orchestration systems	Stores data and metadata on local storage devices
Union — File deduplication Cache — Data/metadata Caching Local Filesystem, Network Block Volume, Network Filesystem — Remote storage access	Log or Journal — Crash recovery Key-value Store — Metadata storage Cache — Data/metadata caching Deduplication — Block deduplication Local filesystem — Persistent storage
 Local storage and network devices	 Local storage and network devices

Interprocess Communication



User level

- Minimize mode switches & CPU cache stalls

Per pool shared memory

- Circular queues for requests
- Shared buffers for responses

Resources and Devices

Resource reservation

- Guarantee resource limits (CPU, RAM, Net, I/O)

Resource management

- Resource tracking and process accounting
- Dynamic resource allocation based on reservations & utilization

Device management

- Protected operation of local devices

Our approach

- Kernel Cgroups for accounting of user-level processes
- Possible to manage the network & storage devices at user level

Example Storage Systems

Root Filesystem (boot application & storage containers)

- Client: Network FS with cache (CephFS); Union FS (AUFS)
- Server: Local journaled FS (ext4); Key-value store (RocksDB)

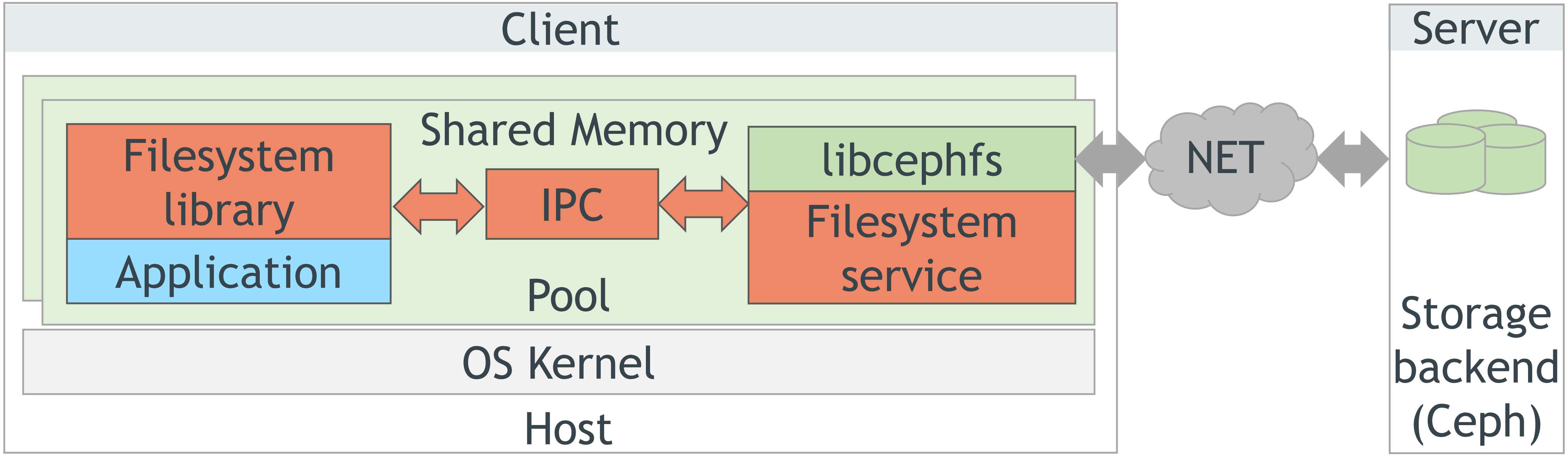
Application Filesystem (serve applications)

- Client: Network FS with cache (CephFS)
- Server: Local journaled FS (ext4); Key-value store (RocksDB)

Container Image Storage (image repository)

- Client: Network FS with cache (NFS)
- Server: Local FS with cache & deduplication (ZFS)

Early Prototype: Client per tenant



Provision the client side of the root filesystem storage system

- Filesystem service: libcephfs libservice (network client and cache)
- Filesystem library: preloaded to applications (LD_PRELOAD)
- IPC: User-level shared-memory

Test Setup

2 Servers

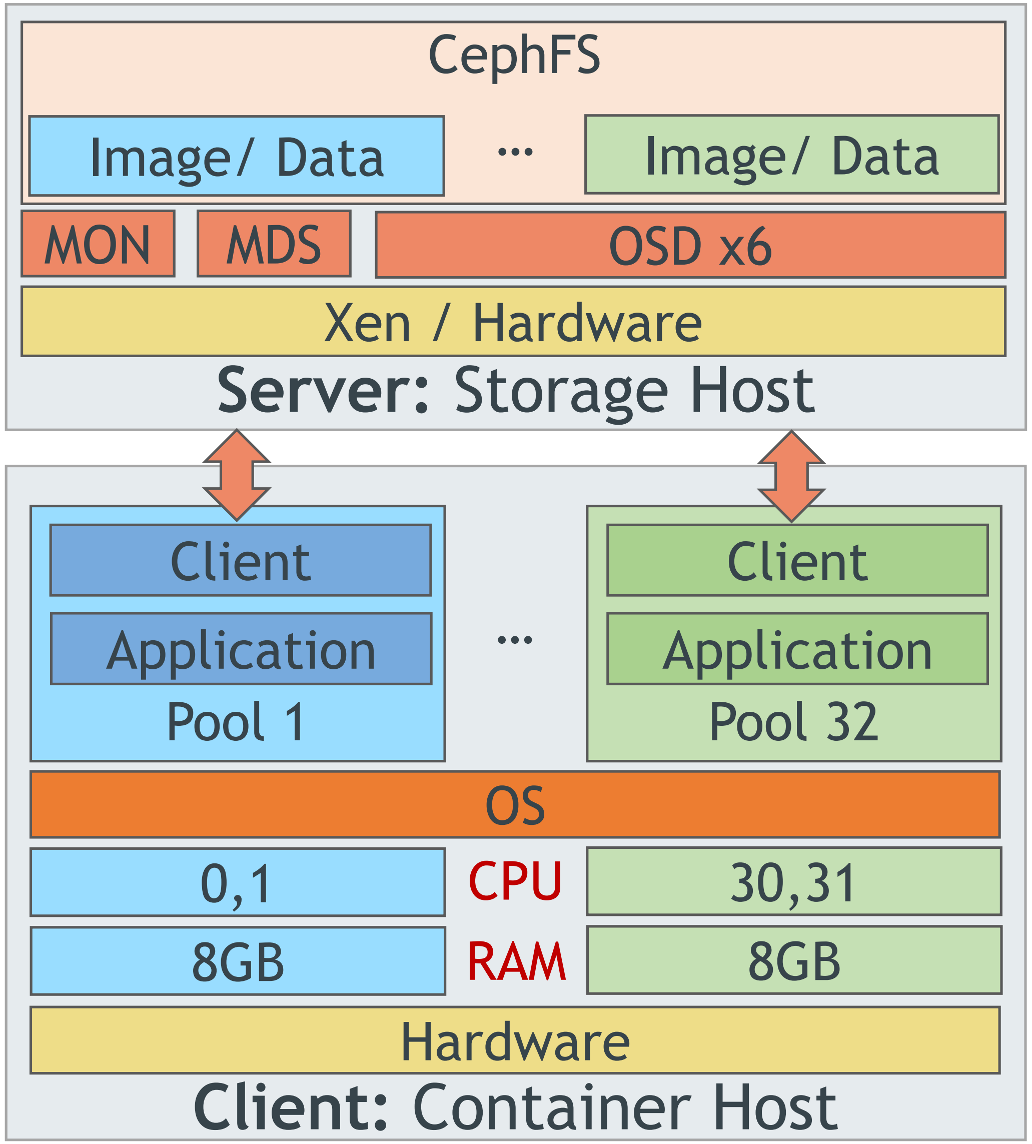
- 64 Cores, 256GB RAM
- 2 x 10Gbps Ethernet
- Linux v5.4.0

Shared CephFS

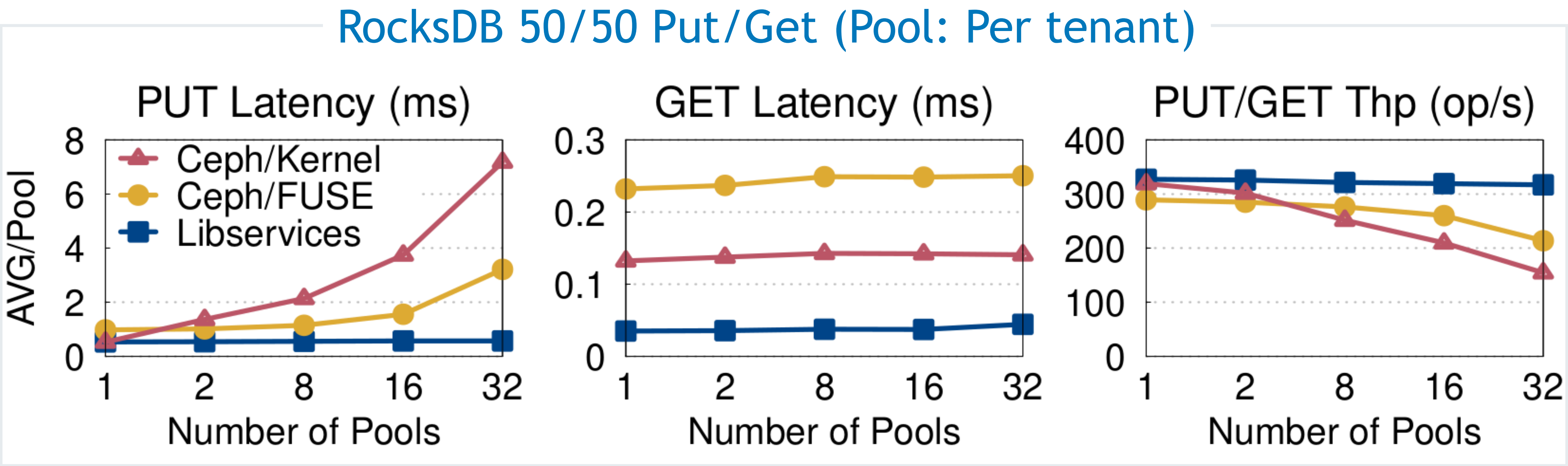
- 6 OSDs (2 CPUs, 8GB RAM, 24GB Ramdisk for fast storage)
- 1 MDS, 1 MON (2 CPUs, 8GB RAM)

Container Pool

- 1 Container
- 2 CPUs (Cgroup v1 - cpuset)
- 8 GB RAM (Cgroup v2 - memory)



I/O Workload Collocation



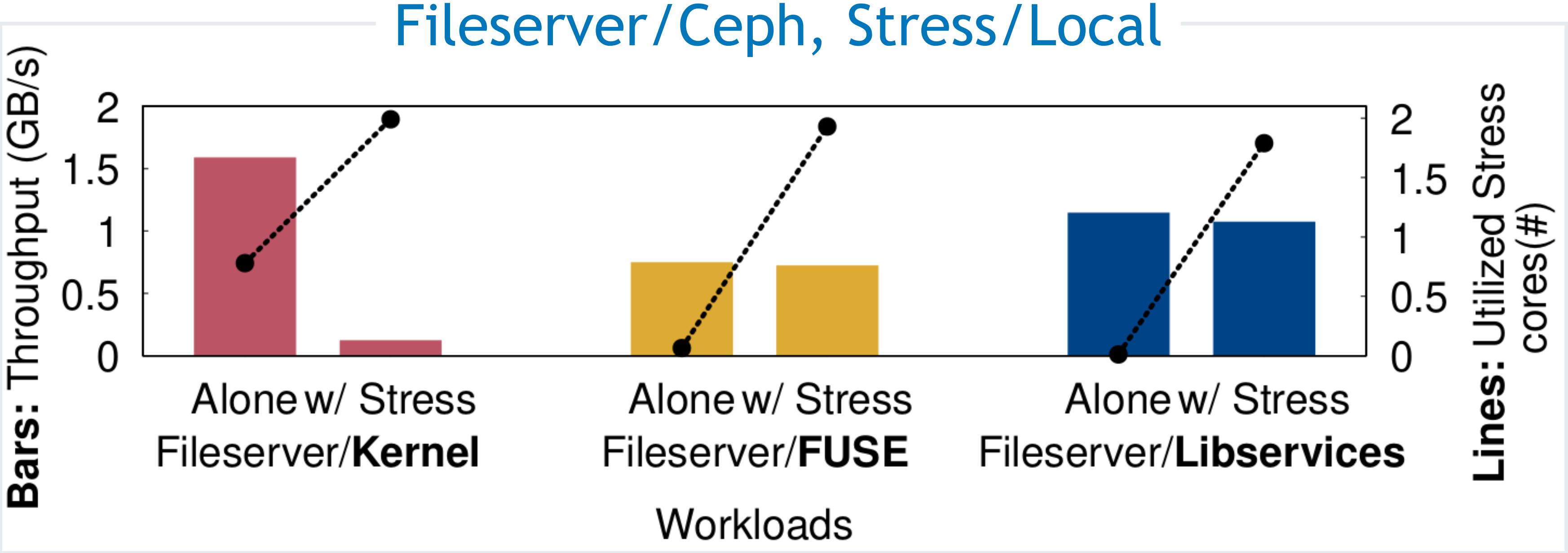
Libservices achieve faster I/O response & stable performance

- **Put latency (longer)** FUSE: up to 5.6x, Kernel: up to 12.6x
- **Get latency (longer)** FUSE: up to 3.9x, Kernel: up to 6.7x
- **Throughput (slowdown)** FUSE: up to 1.5x, Kernel: up to 2.1x

Contention Sensitivity

2 Containers of

- 2 Cores
- 8GB RAM
- Fileserver or Stress with rand I/O



Outcome

- Kernel client sensitive to contention
- Throughput drops up to 12.9x when collocated with Stress

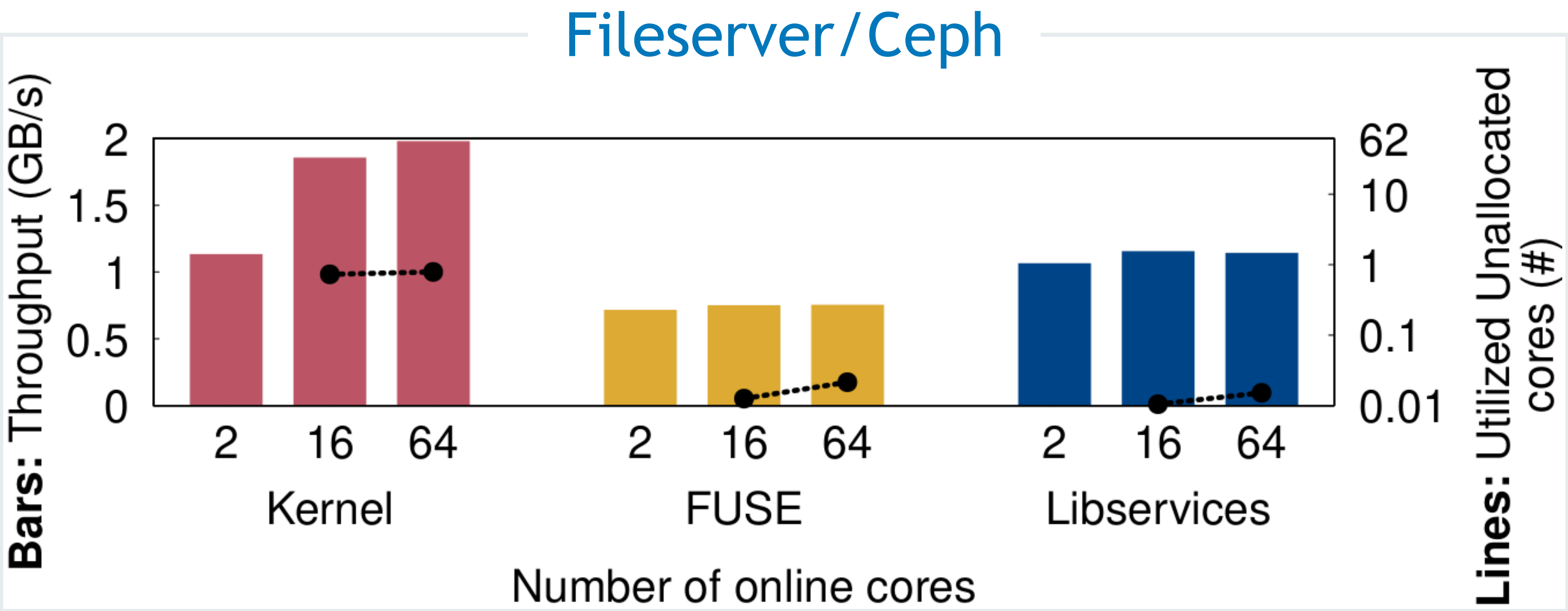
Reason

- Kernel I/O utilizes all cores (dirty page flushing)

Violation of Resource Limits

1 Container of

- 2 Cores
- 8GB RAM
- Fileserver



Outcome

- The Kernel client increases performance up to 75% because it utilizes unallocated cores
- Kernel flusher threads run on non allocated cores

Conclusions & Future Work

The Problem: Performance variability from shared Kernel I/O

- Lack of accounting; Aggressive resource utilization

Our Solution: Libservices Framework

- Performance isolation combined with high efficiency
- I/O performance isolation by handling container I/O at user level
- Same design pattern for the client and server of a storage system
- Dynamic provisioning of container storage systems

Future Work

- Dynamic readjustment of allocated resources (e.g., memory)
- Network and storage device management at user level
- Resource scheduling services at user level (e.g., Cgroups)