# Oxbow: A Coordinated Architecture for Multi-component File Systems

**Jongyul Kim**, Jaehwan Lee, Inhoe Koo, Peizhe Liu,
Jiyuan Zhang, Junho Ahn, Tianyin Xu, Youngjin Kwon

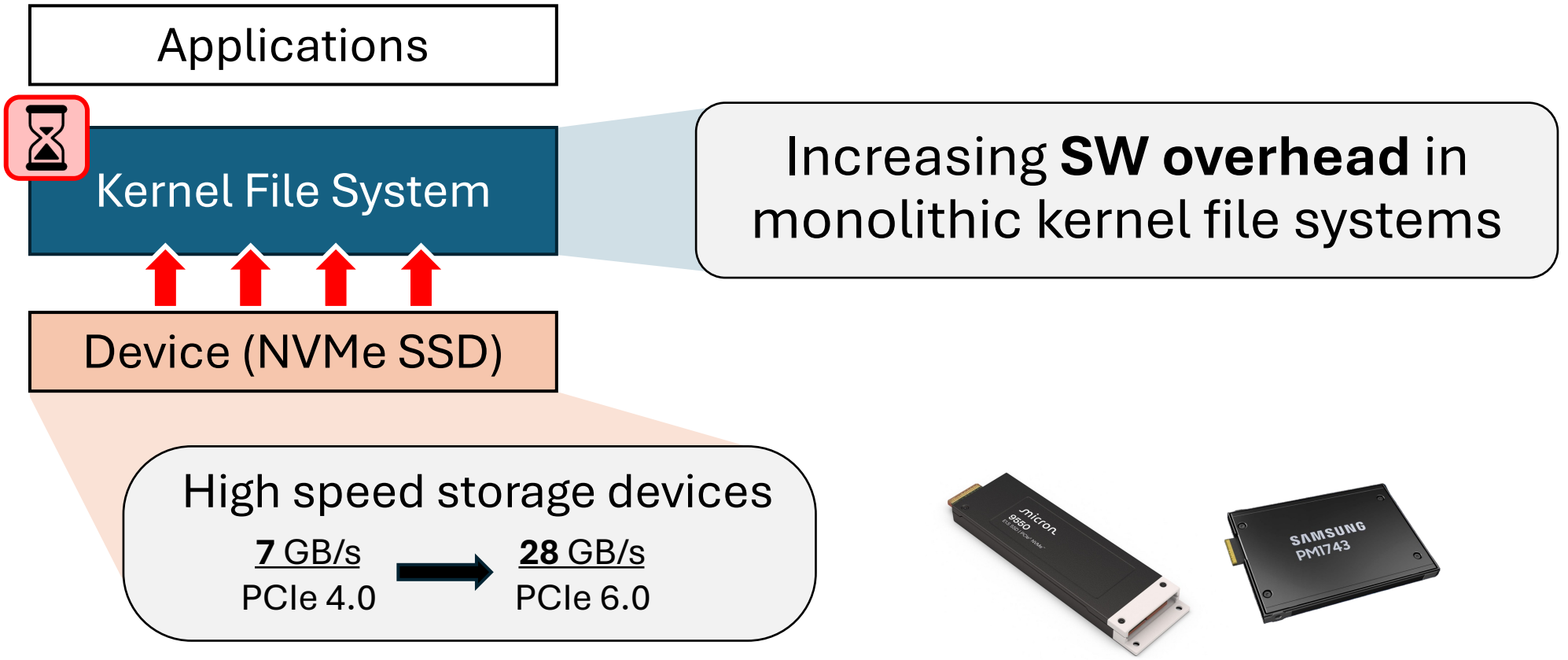




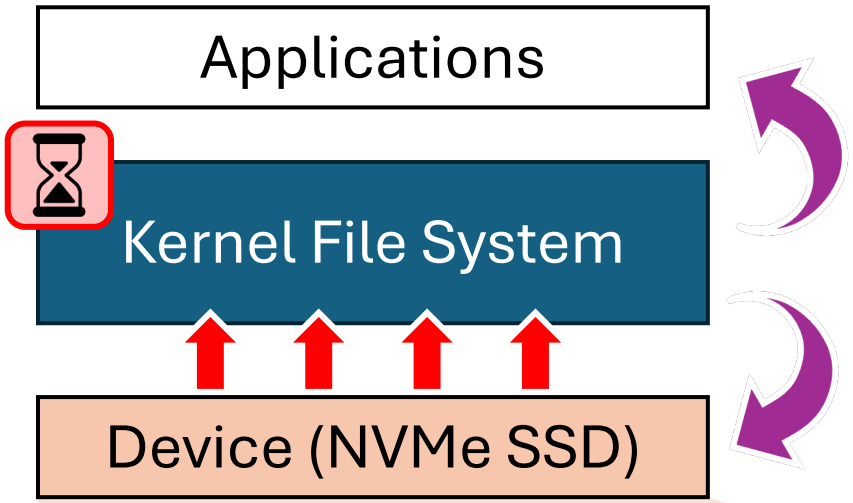


https://github.com/xlab-uiuc/oxbow

# Faster Devices Expose Kernel Bottleneck



# Faster Devices Expose Kernel Bottleneck




**Moving the FS out of the kernel**

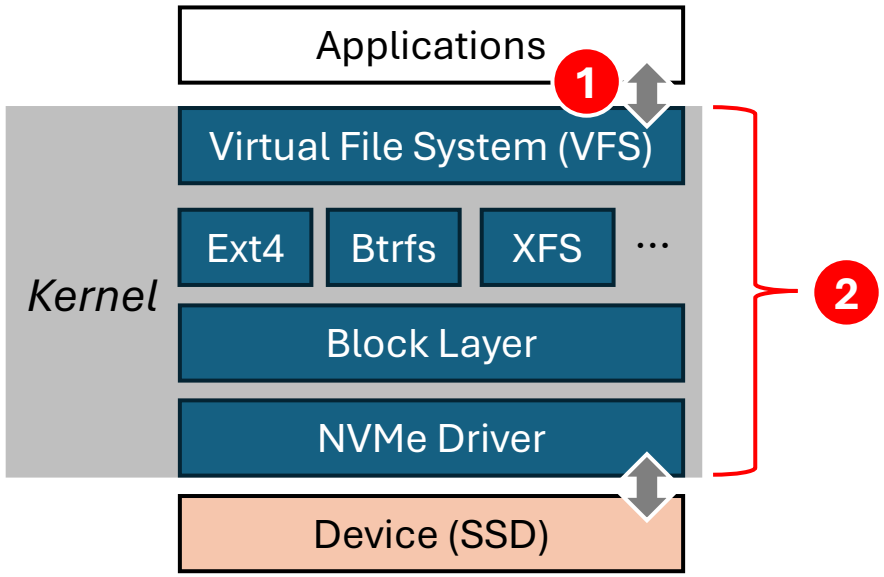
High speed storage devices

**7** GB/s PCIe 4.0 → **28** GB/s PCIe 6.0



# Approach 1: Migration to **User Space**

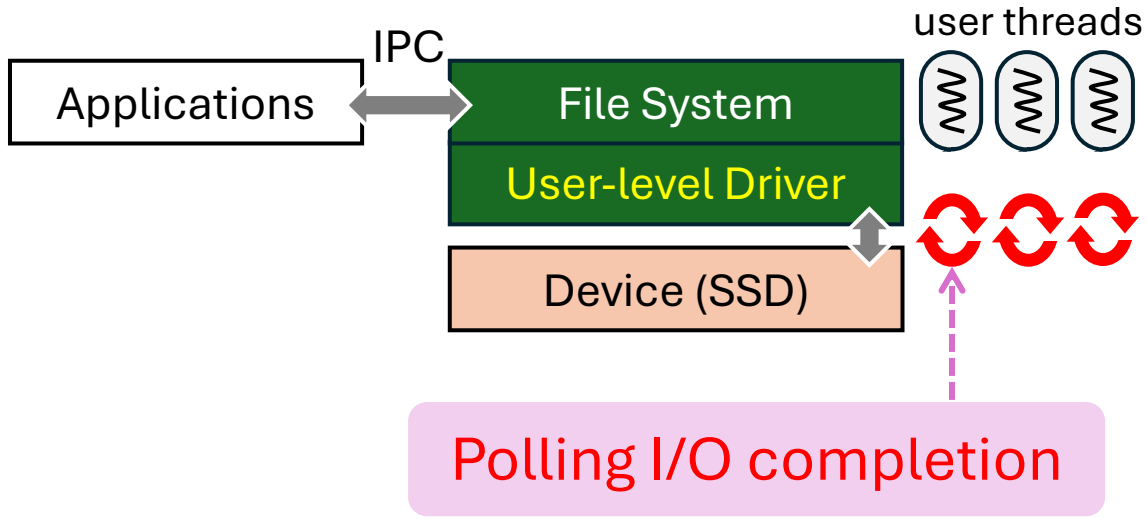
## Monolithic Kernel FS




Kernel software overhead:

1. User-kernel boundary crossing
2. Heavy I/O stack

## User-level FS




Bypasses the kernel with a **user-level driver**

Scales with **user-level threads**

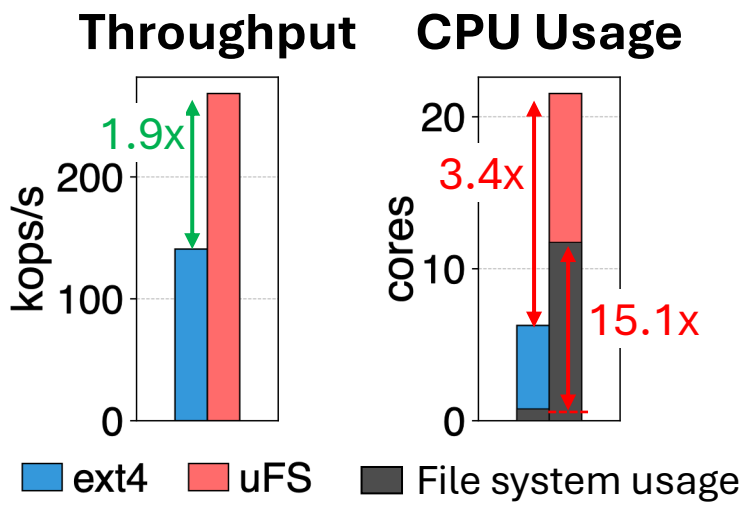
Cannot leverage kernel services

Consumes a lot of CPU cycles

# Approach 1: Migration to **User Space**

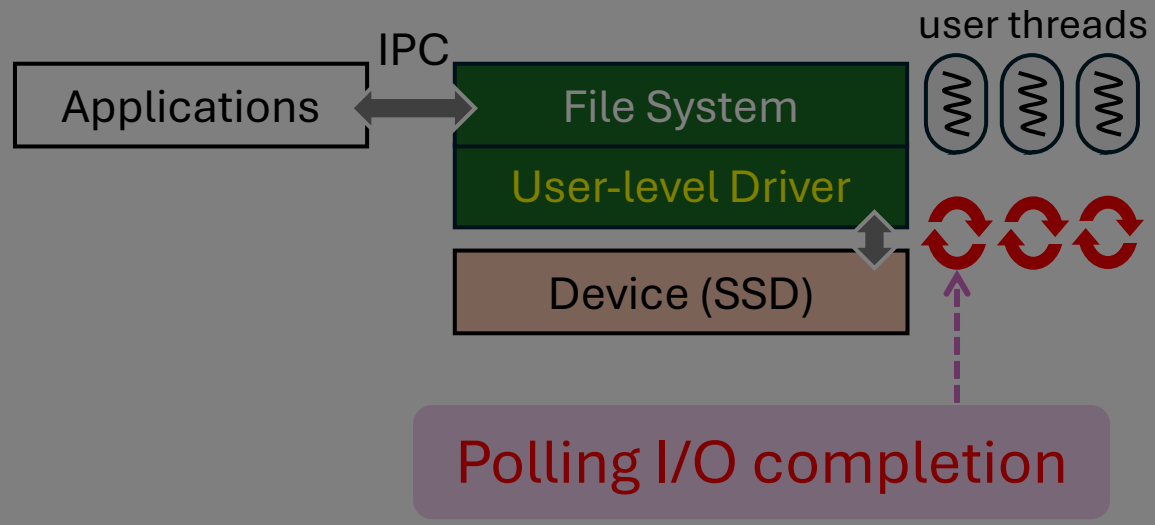
## High CPU Usage of User-level FS

Filebench – varmail, 8 clients




**uFS** [1]: A user-level file system

## User-level FS

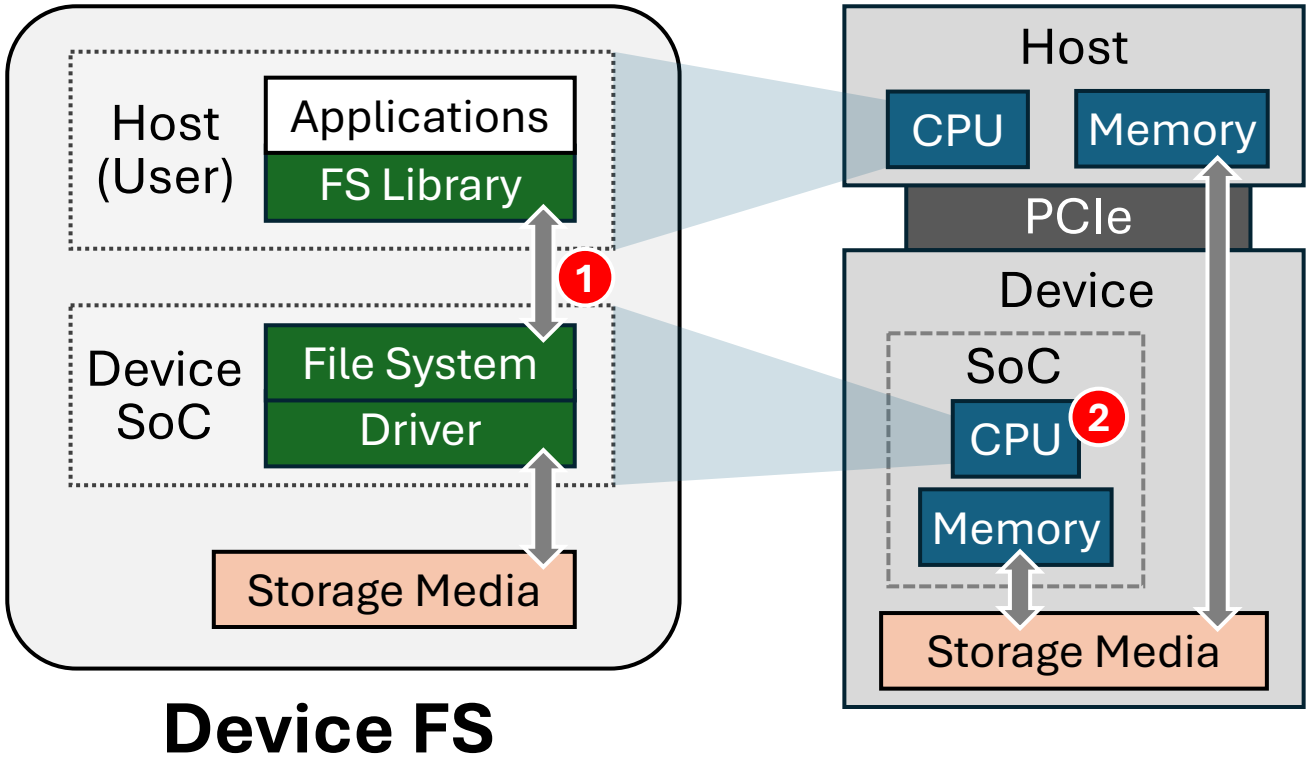



Bypasses the kernel with a **user-level driver**
Scales with **user-level threads**
Cannot leverage kernel services
Consumes a lot of CPU cycles

[1] Scale and Performance in a Filesystem Semi-Microkernel (SOSP'21)

# Approach 2: Migration to **Devices**

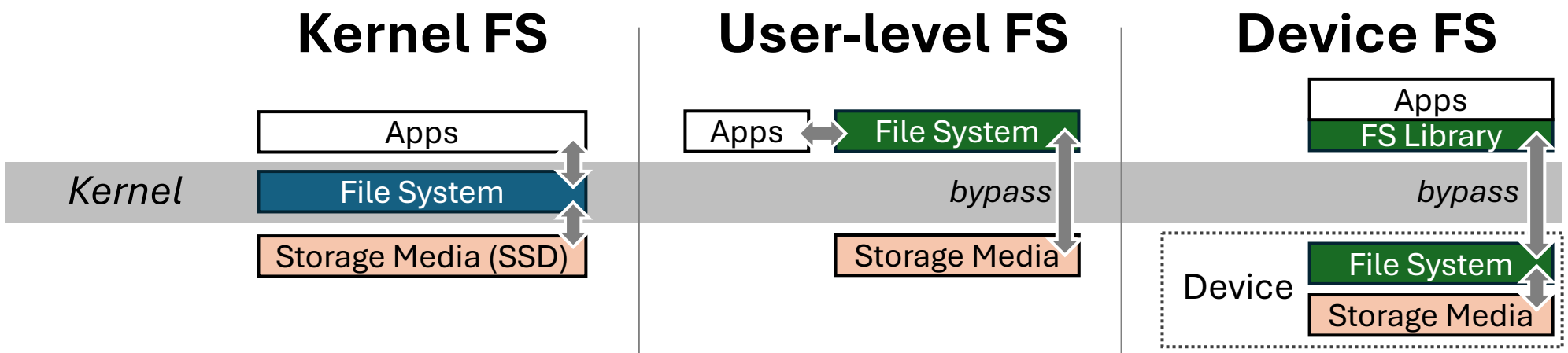
Saving host CPU via device offloading




Offloading overhead:

1. PCIe crossing (host ↔ device)
2. Inferior device CPU

# Tradeoffs among File System Architectures



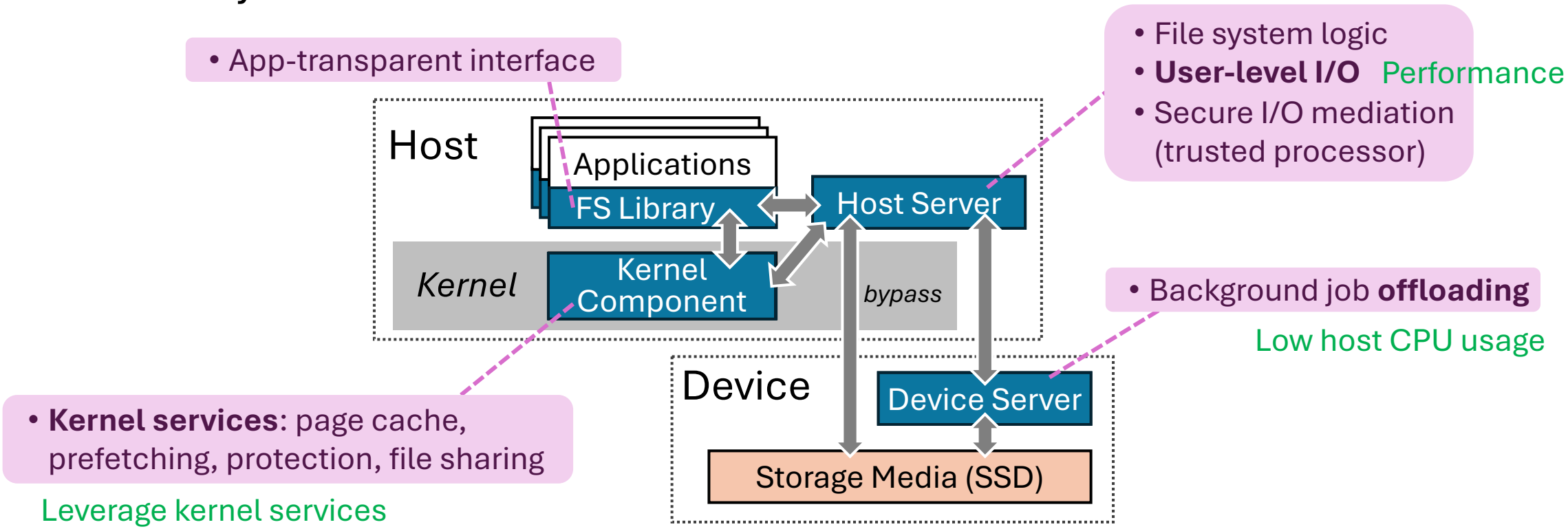

| | Kernel FS | User-level FS | Device FS |
|---|---|---|---|
| **Performance** | **Poor** *kernel crossing* | **Good** *fast I/O, user threads* | **Poor** *PCIe crossing, slow CPU* |
| **Low CPU Usage** | **Good** *interrupt-based I/O* | **Poor** *polling-based I/O* | **Good** *use device CPU* |
| **Kernel Services** | **Good** *cache, prefetch, sharing* | **Poor** *none* | **Poor** *none* |

No single architecture delivers all three

# Our Approach: A Coordinated Architecture

Combine user, kernel, and device components and coordinate them

- Each layer does what it does best

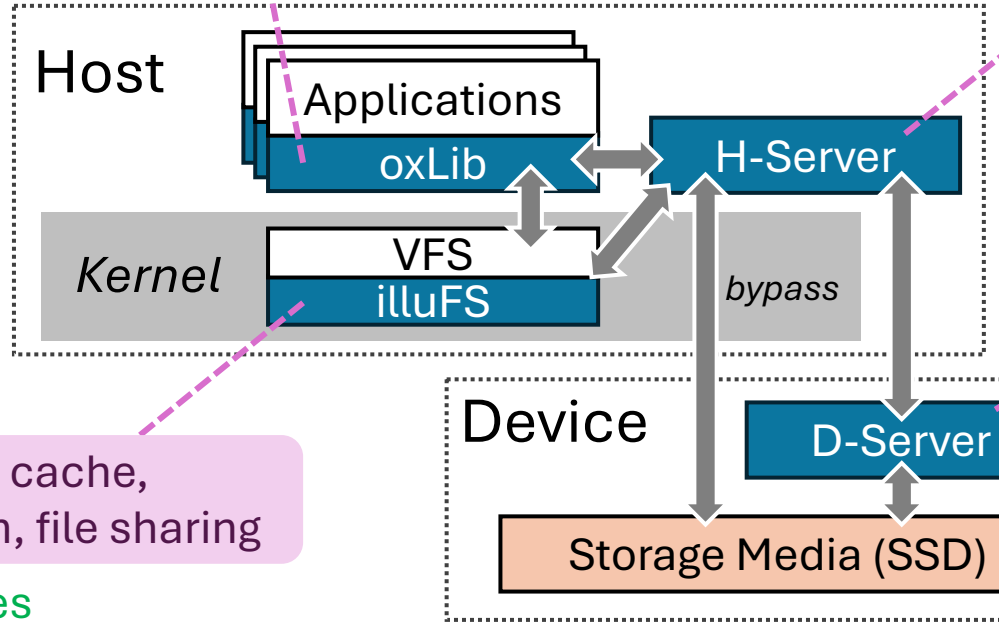


# Our Approach: A Coordinated Architecture

Combine user, kernel, and device components and coordinate them

- Each layer does what it does best

**Oxbow File System**




- App-transparent interface
- File system logic
- **User-level I/O** Performance
- Secure I/O mediation (trusted processor)
- Background job **offloading**

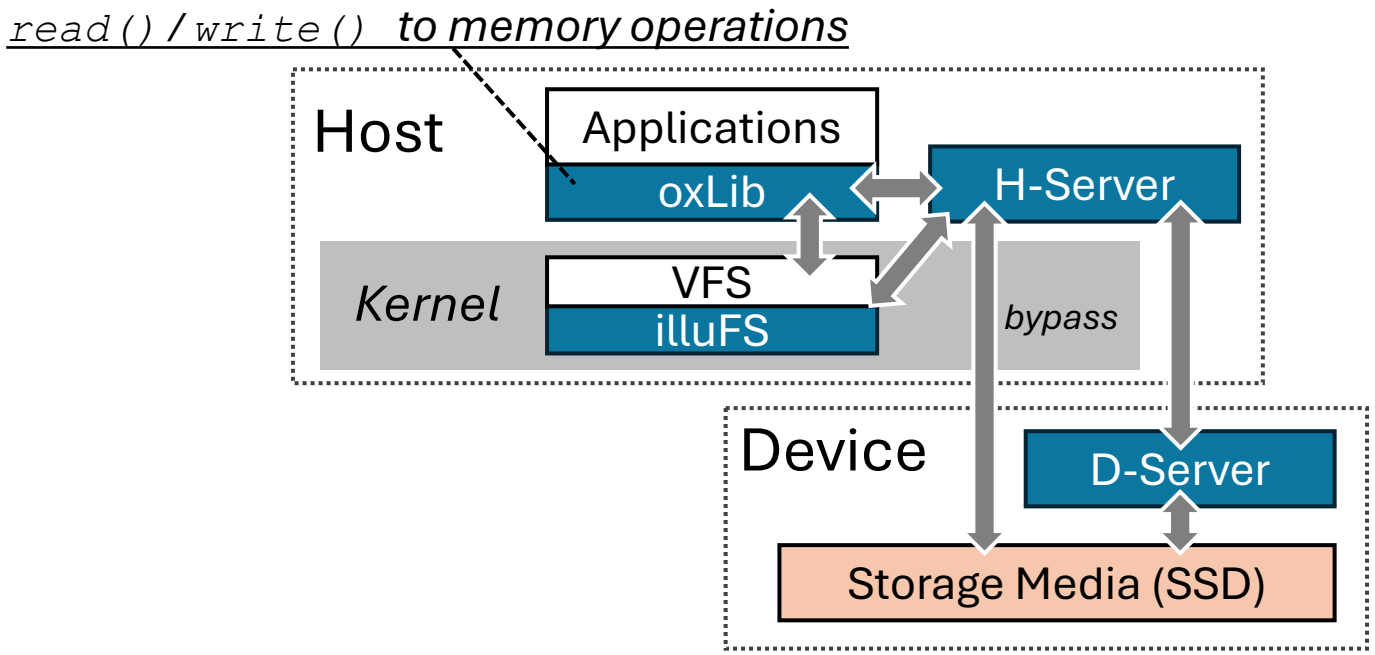
Low host CPU usage

- **Kernel services**: page cache, prefetching, protection, file sharing

Leverage kernel services

# Oxbow Design: Asymmetric I/O Paths

Which operations require kernel involvement?

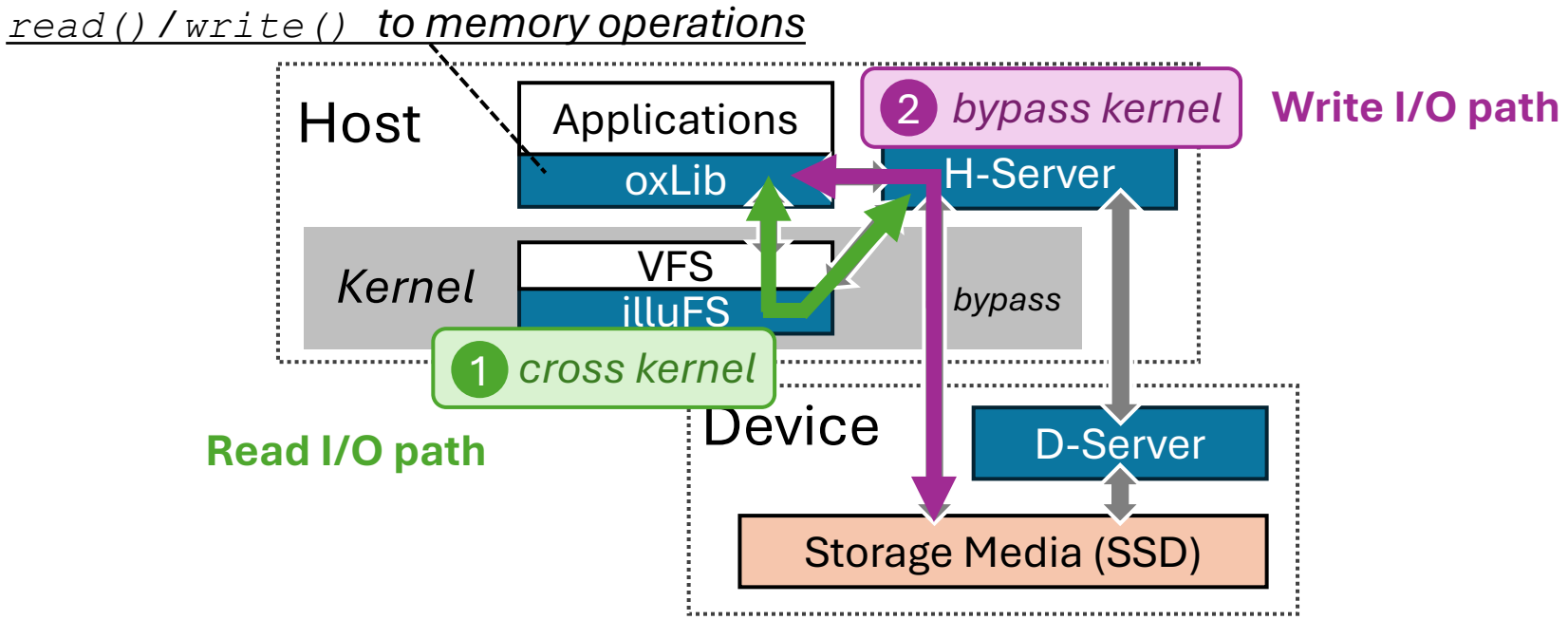


# Oxbow Design: Asymmetric I/O Paths

Which operations require kernel involvement?

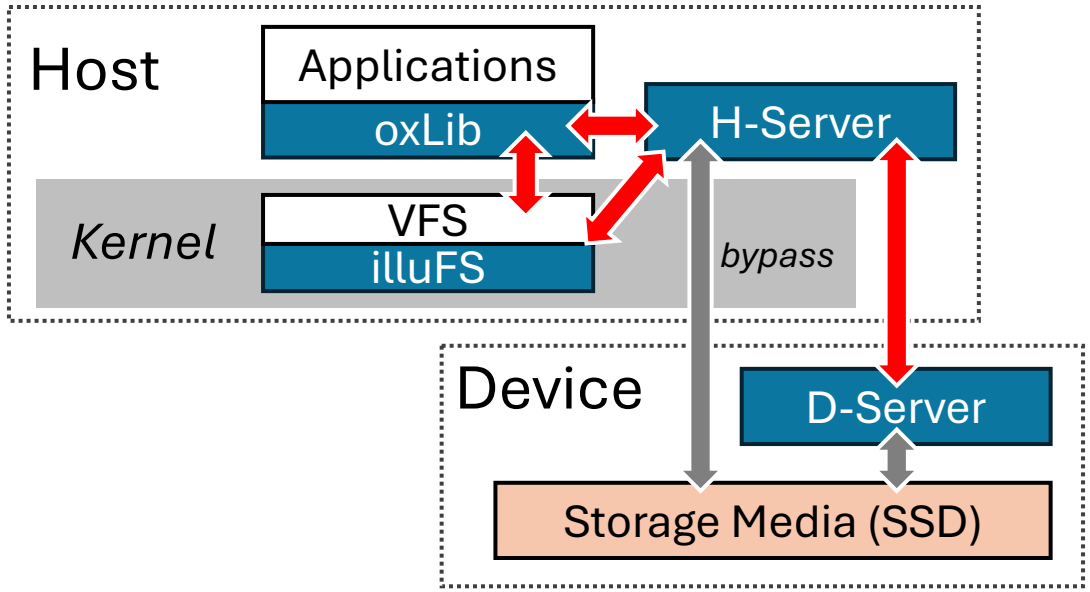
→ **Read path**: Leverage kernel's readahead

→ **Write path**: Fast I/O with user-level I/O



# Oxbow Design: Minimize Disaggregation Tax

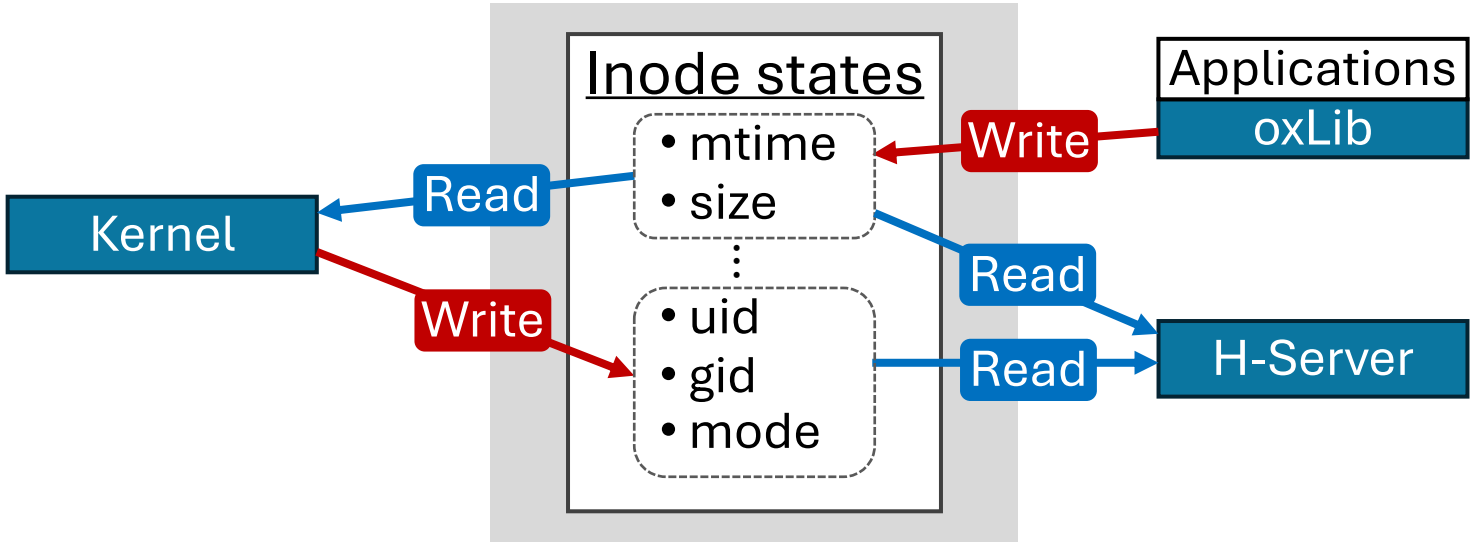
Challenge: Communication, synchronization and copy overhead



# Oxbow Design: Minimize Disaggregation Tax

Challenge: Communication, synchronization and copy overhead

→ *1) Shared-ownership* metadata with shared memory

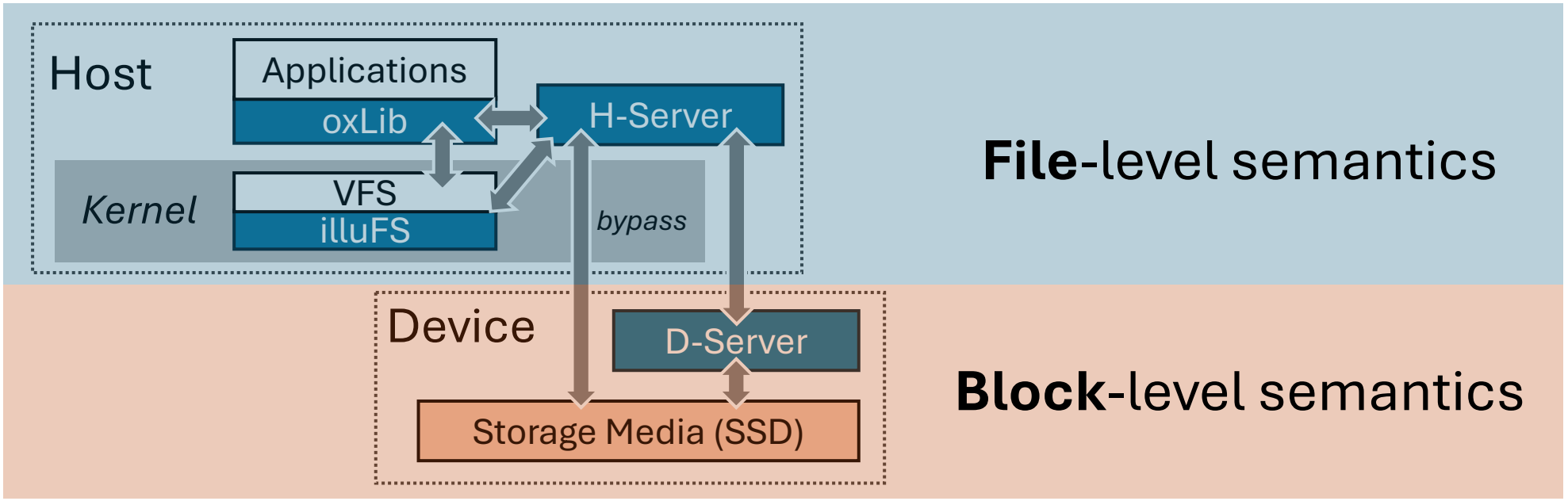



→ Minimal synchronization, zero metadata copies

# Oxbow Design: Minimize Disaggregation Tax

Challenge: Communication, synchronization and copy overhead

→ 2) Semantic decoupling



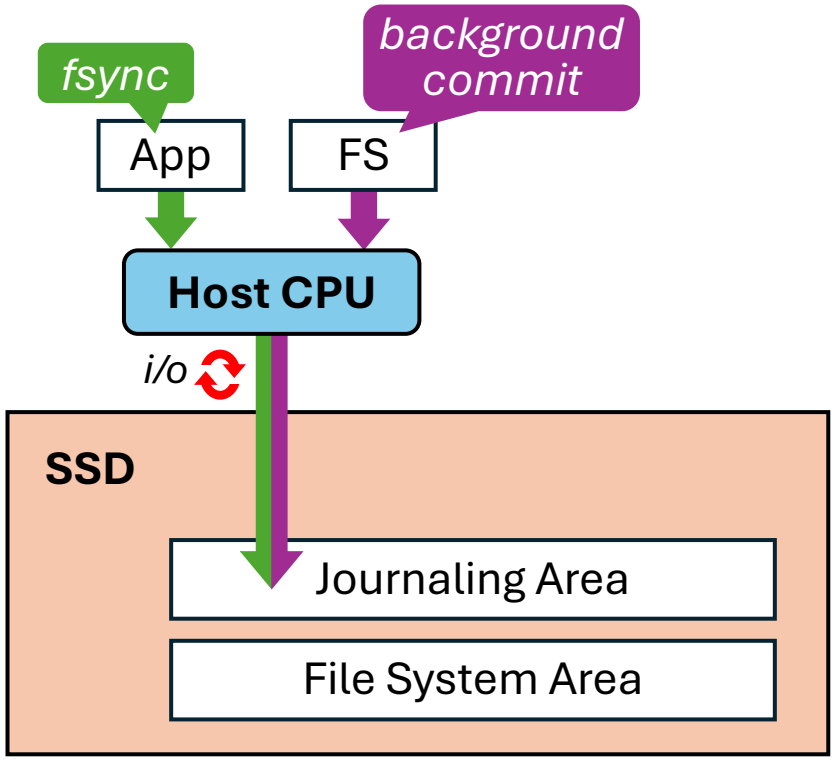

→ Minimal communication between host and device

# Oxbow Design: Redesign Persistence Path

Offload CPU intensive background I/O → *Background journaling* (↱ Commit)

Challenge: Crash-consistency dependencies slow down fsync

**Current Journaling File Systems**



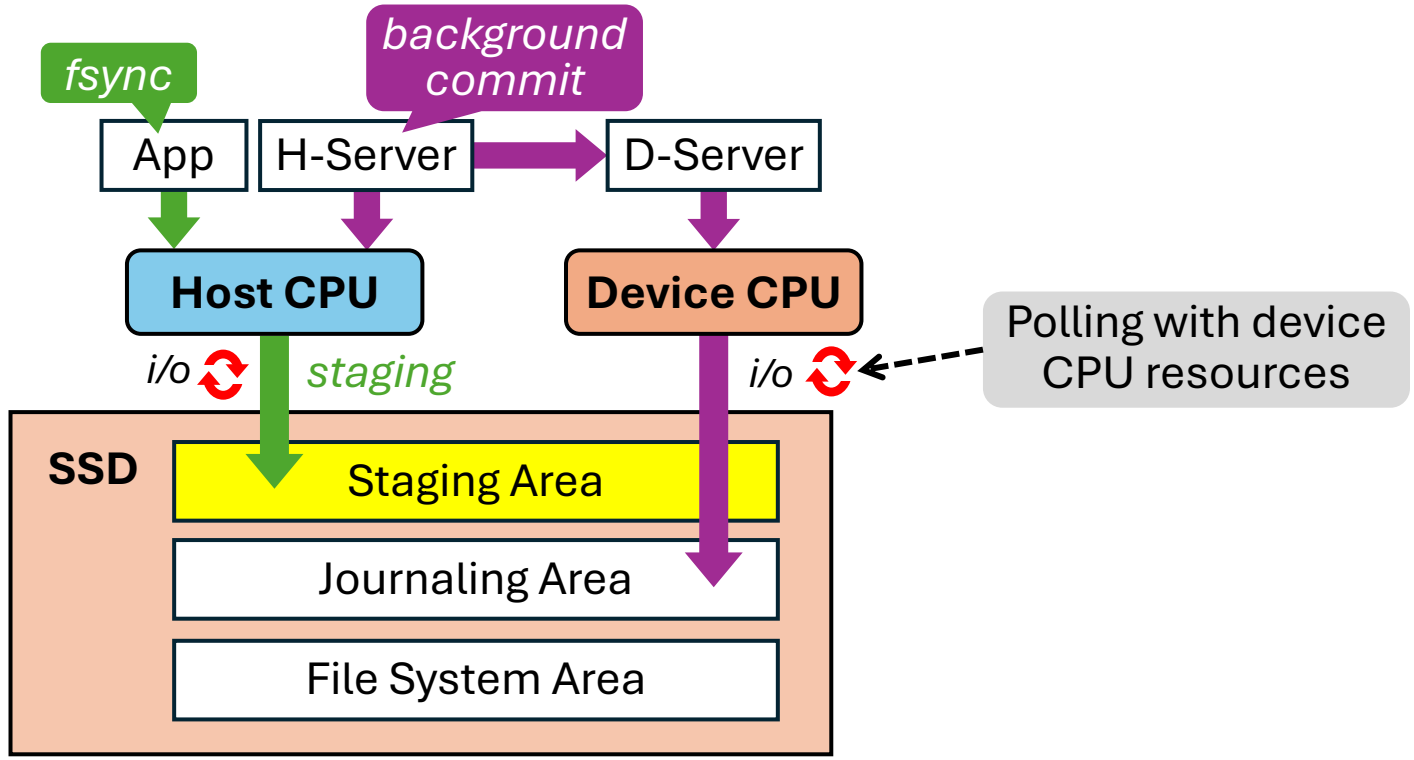
# Oxbow Design: Redesign Persistence Path

Offload CPU intensive background I/O → *Background journaling* (↱ Commit)

Challenge: Crash-consistency dependencies slow down fsync

→ 1) Split journaling

**Oxbow**
**Split Journaling**

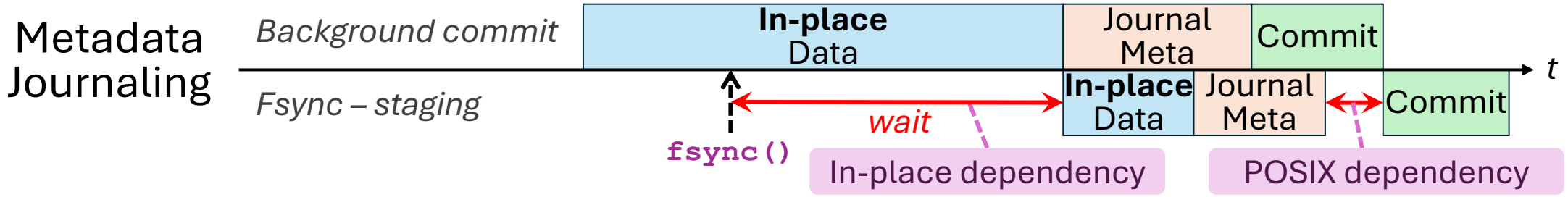


# Oxbow Design: Redesign Persistence Path

Offload CPU intensive background I/O → *Background journaling*

Challenge: Crash-consistency dependencies slow down fsync

→ 2) Data journaling with self-contained staging

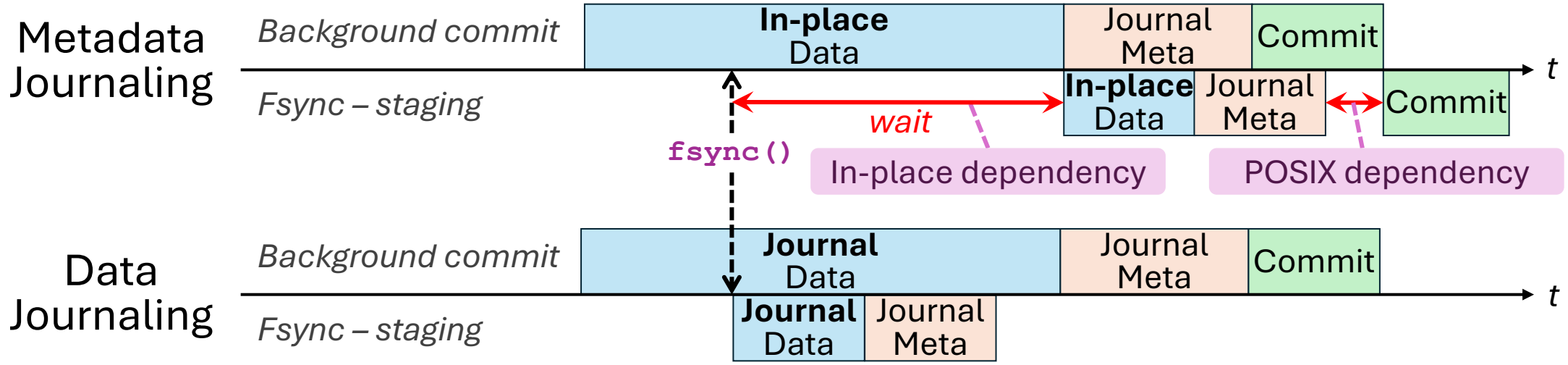


# Oxbow Design: Redesign Persistence Path

Offload CPU intensive background I/O → *Background journaling*

Challenge: Crash-consistency dependencies slow down fsync

→ 2) Data journaling with self-contained staging

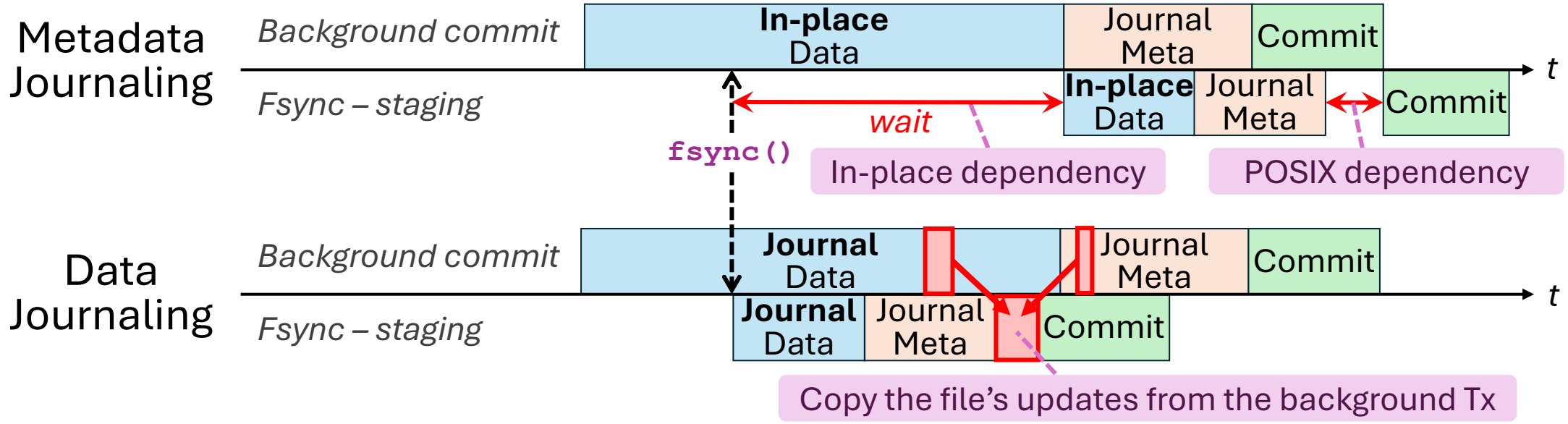


# Oxbow Design: Redesign Persistence Path

Offload CPU intensive background I/O → *Background journaling*

Challenge: Crash-consistency dependencies slow down fsync

→ 2) Data journaling with self-contained staging




→ Commit offloaded, fsync stays fast

# Evaluation

# Hardware Specification

## Host

- 2 x Xeon Gold 5218 16 core CPU
- 128 GB DRAM

## Computational Storage Device (Smart SSD)

- Emulated with DPU (Data Processing Unit)
  - NVMe-oF with target offload
- NVIDIA Bluefield-2 DPU
  - 8 x ARM A72
  - 16 GB DRAM
- Samsung PM1735 NVMe SSD
  - SR-IOV
  - Namespace sharing

# Experimental Setup

## Oxbow

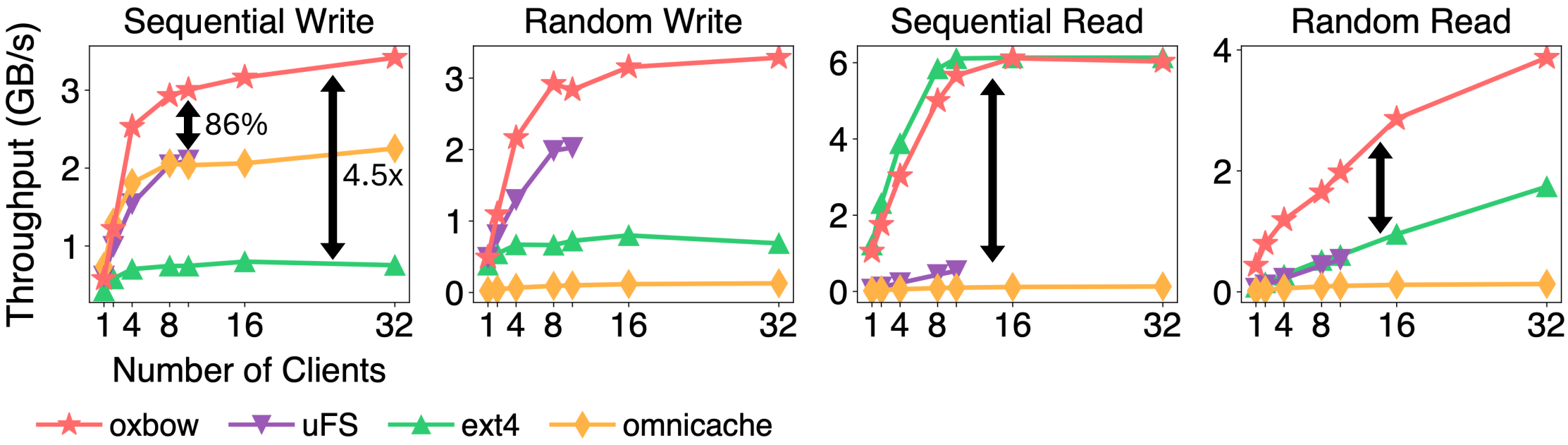
- 53K LOC in C
- File system logic: lwext4 [1]

| Kernel FS | User-level FS | Device FS |
|---|---|---|
| **Ext4**<br>• Data journaling | **uFS**<br>• Metadata journaling<br>• Zero-copy with application modification | **OmniCache-mod** [2]<br>• Ext4 with metadata journaling and direct I/O |

[1] https://github.com/gkostka/lwext4
[2] OmniCache: Collaborative Caching for Near-storage Accelerators (FAST'24)

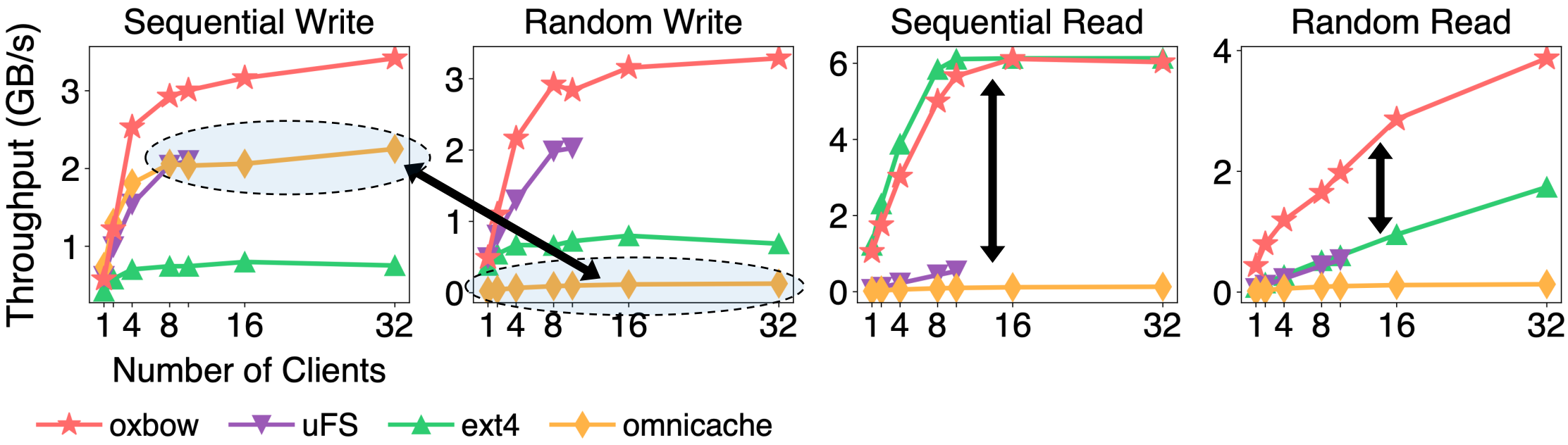
# Throughput and Scalability

Microbenchmark: 4KB I/O to 2GB of private files, fsync follows after 512 MB of writes

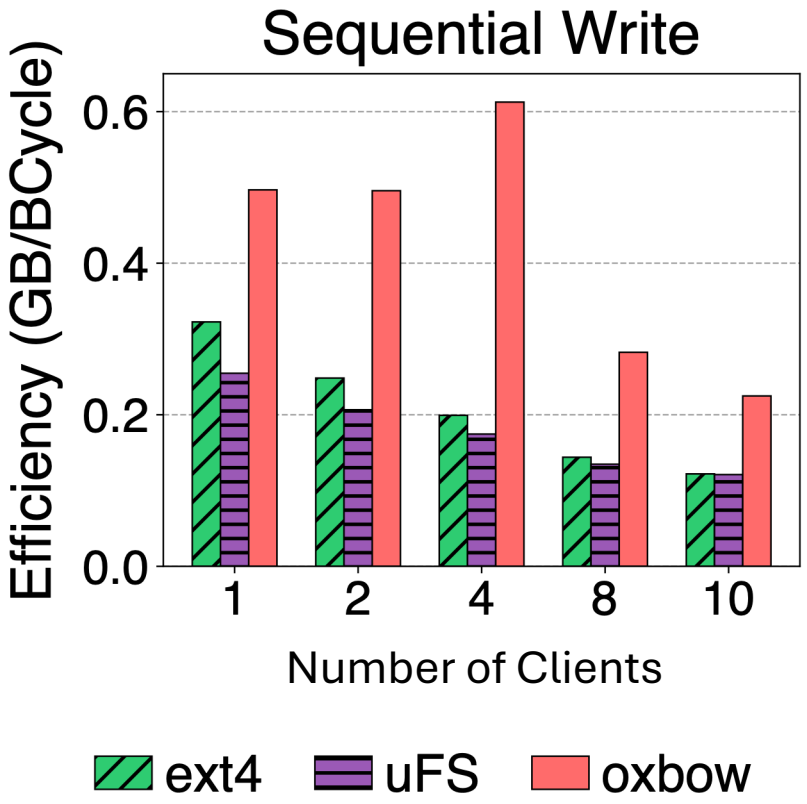


# Throughput and Scalability

Microbenchmark: 4KB I/O to 2GB of private files, fsync follows after 512 MB of writes



# Host CPU Efficiency (Throughput Microbenchmark)




**CPU Efficiency**
= data processed per host CPU cycle

(higher is better)

**Ext4**: low throughput
**uFS**: high CPU utilization

**Oxbow** performs better while using less host CPU with device offloading

# Conclusion

**Problem**: Single-tier file systems force a trade-off

- User-level, kernel, and in-device file systems each inherit their tier's weaknesses along with its strengths

**Oxbow**: Coordinate, don't choose – place each operation where it is best served

- Asymmetric I/O path
- Shared-ownership metadata and semantic decoupling
- Split Journaling for offloading

**Results**: All three goals simultaneously

- Performance – user-level I/O → Write throughput up to **4.8x Ext4 / +86% over uFS**
- Leverage kernel services → Sequential read throughput up to **18.5x uFS**
- Save host CPU usage – offloading → **–55% host CPU, up to 3.9× throughput-per-CPU**