SCALABLE COMPUTING
SOFTWARE LABORATORY

SC22
Dallas, TX | hpc accelerates.

LabStor: A Modular and Extensible Platform for Developing High-Performance, Customized I/O Stacks in Userspace

Luke Logan, Jaime Cernuda Garcia, Jay Lofstead*, Xian-He Sun, Anthony Kougkas
Illinois Institute of Technology, * Sandia National Laboratories

ILLINOIS INSTITUTE
OF TECHNOLOGY

Sandia
National
Laboratories

The Explosion of I/O Requirements

A wide diversity of applications

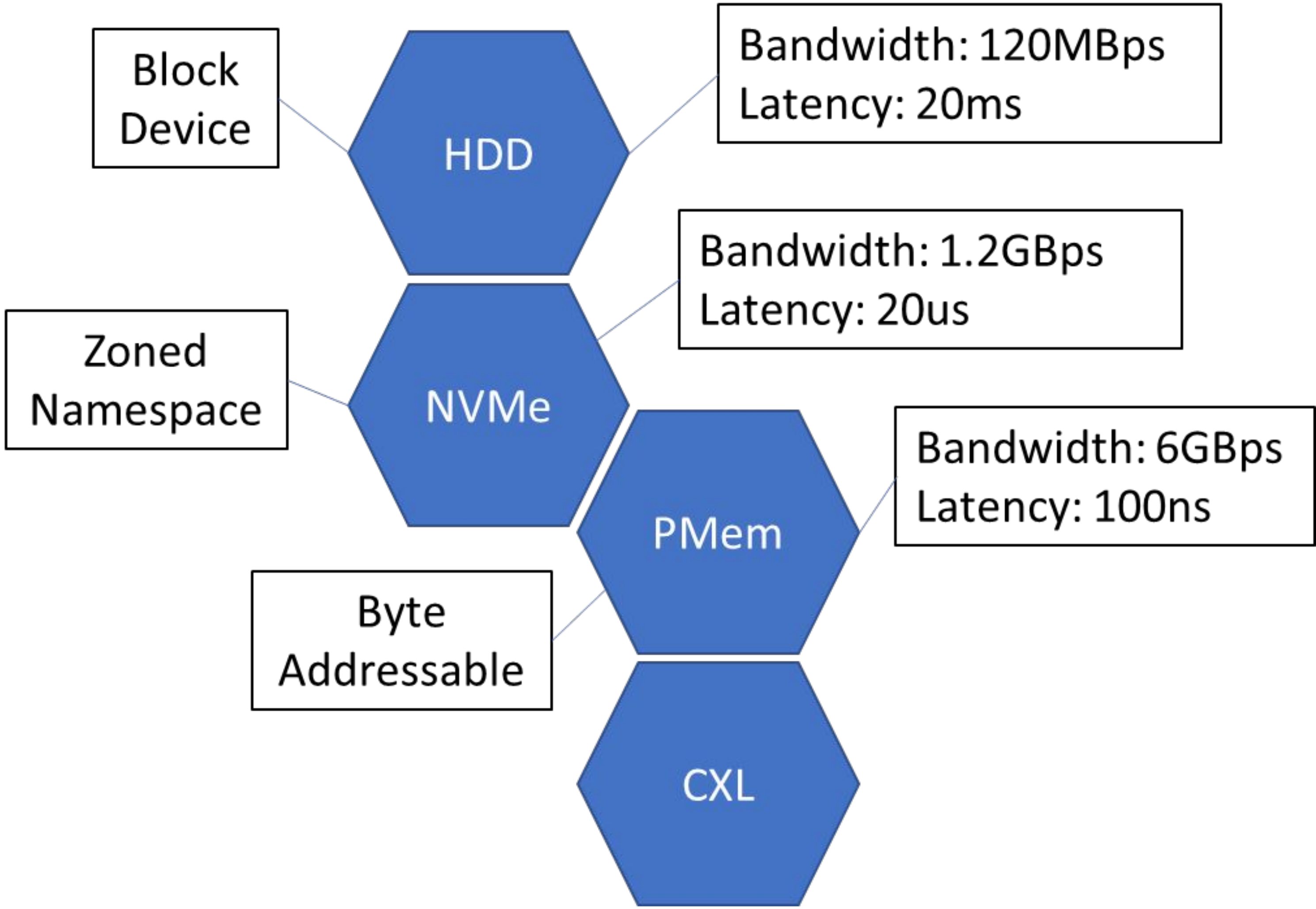
- Machine Learning
- Simulation
- Data Streaming
- AI
- Analysis

Each have varying I/O requirements

- High Bandwidth
- Low Latency
- I/O Interface
- Data Privacy

Rapid Storage Hardware Evolution

- Order of magnitude performance improvement with each new generation
- New interfaces being exposed
- Hardware-specific optimization!



Block Device

HDD

Bandwidth: 120MBps
Latency: 20ms

Zoned Namespace

NVMe

Bandwidth: 1.2GBps
Latency: 20us

PMem

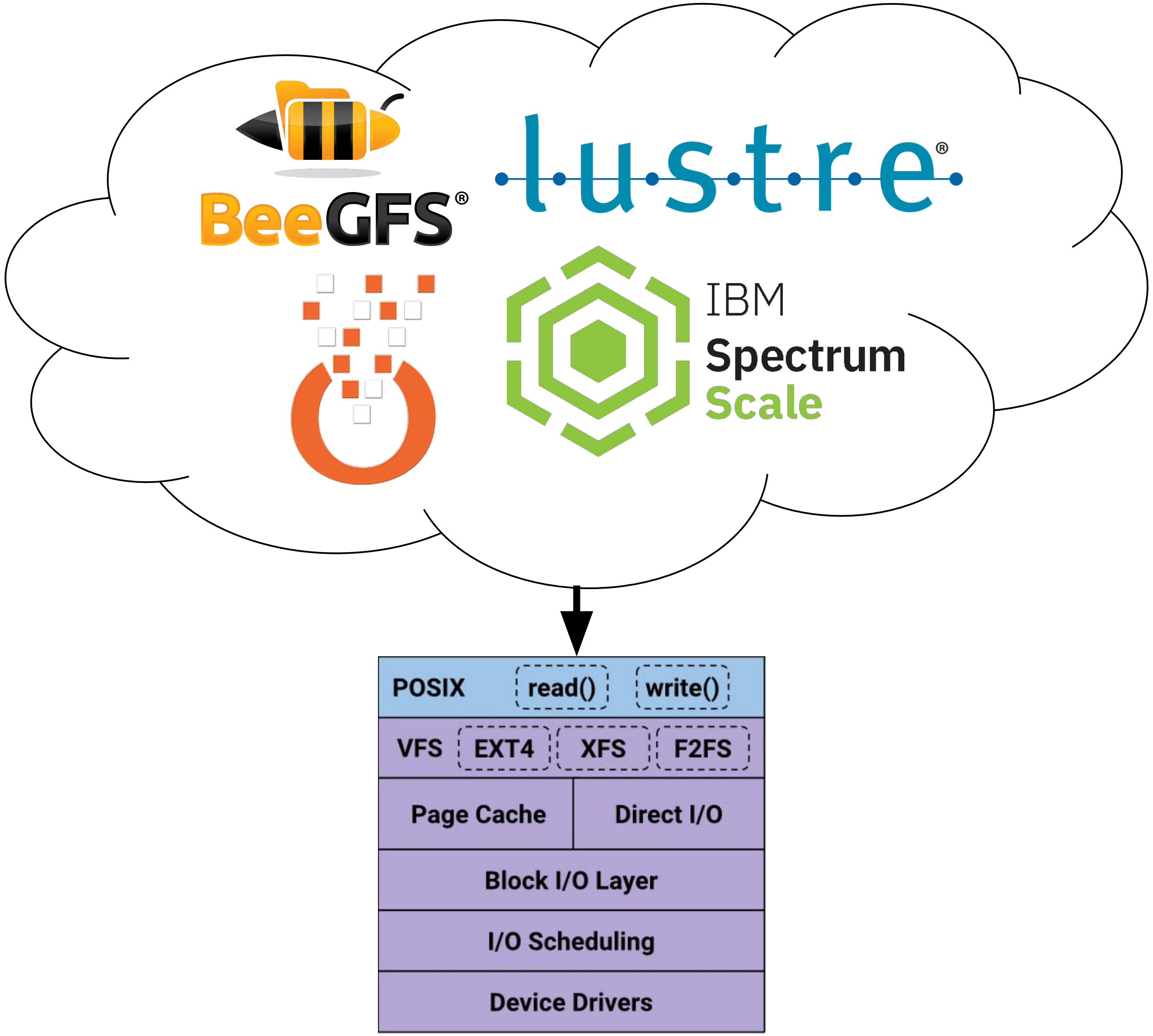
Bandwidth: 6GBps
Latency: 100ns

Byte Addressable

CXL

Parallel Filesystems

- HPC applications rely on parallel filesystems (PFS)
- PFS relies on node-local storage stacks



POSIX read() write()

VFS EXT4 XFS F2FS

Page Cache | Direct I/O

Block I/O Layer

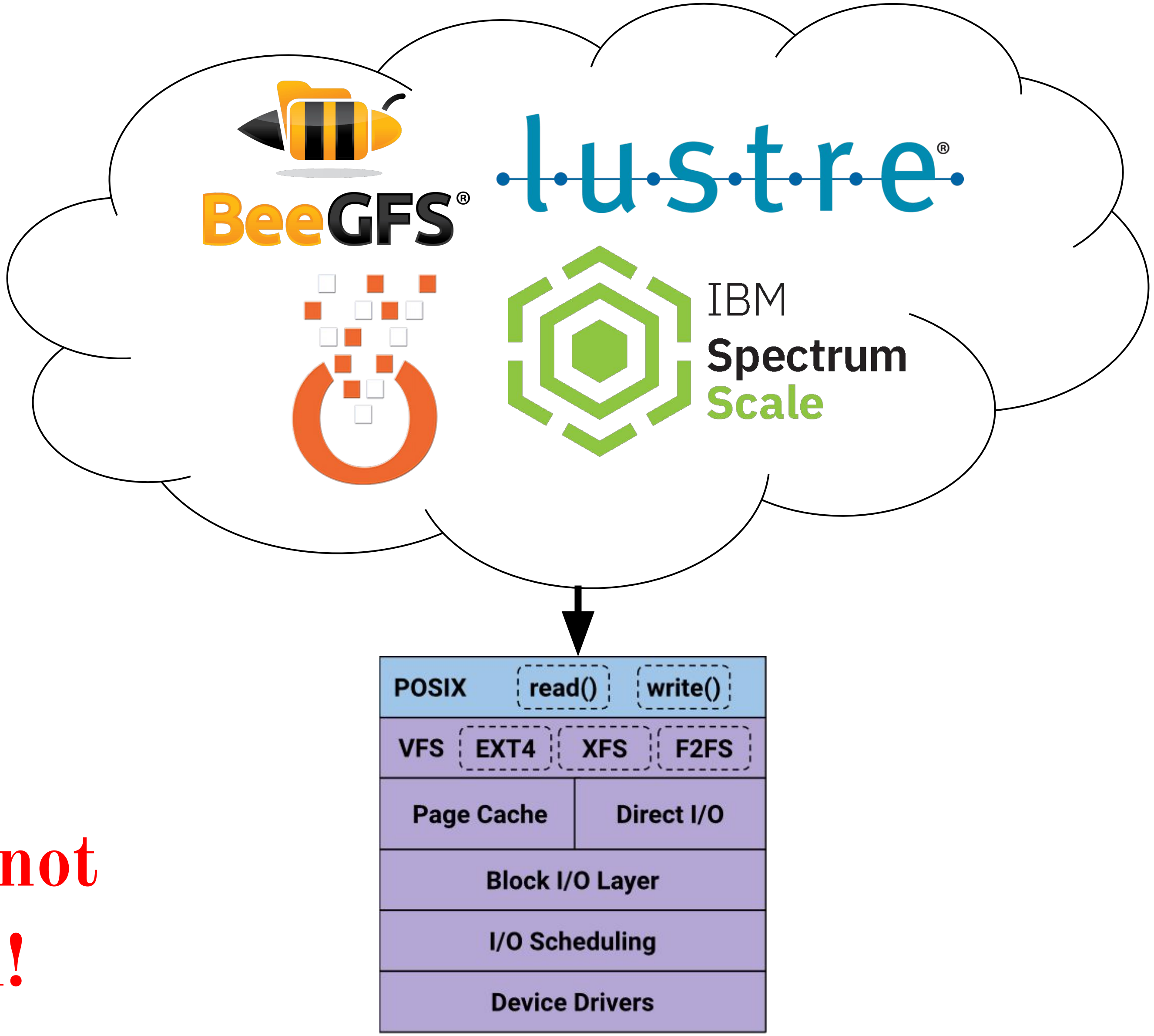
I/O Scheduling

Device Drivers

Parallel Filesystems

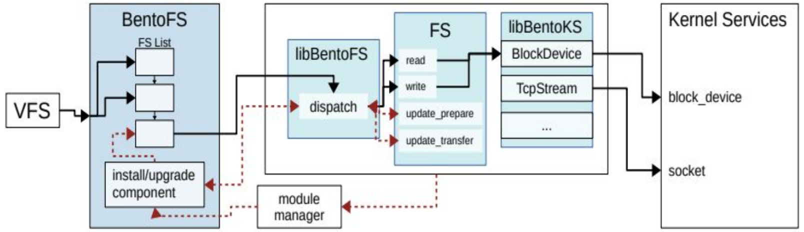
- HPC applications rely on parallel filesystems (PFS)
- PFS relies on node-local storage stacks

Node-local I/O stacks are not adapting rapidly enough!

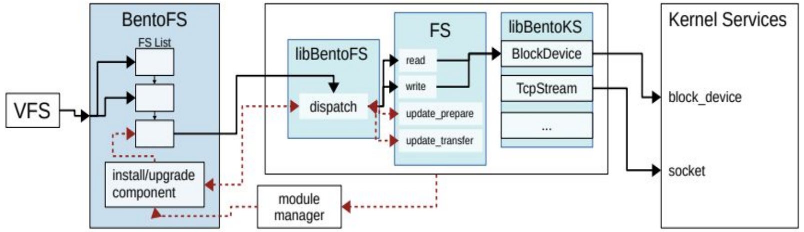


Developing I/O Stacks

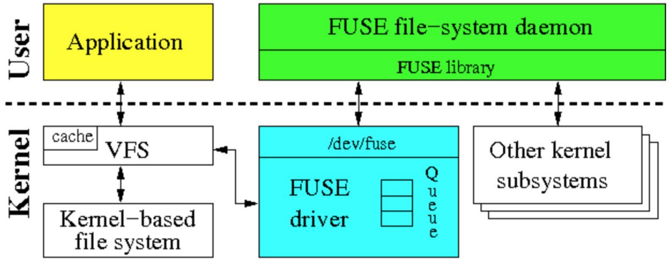
VFS



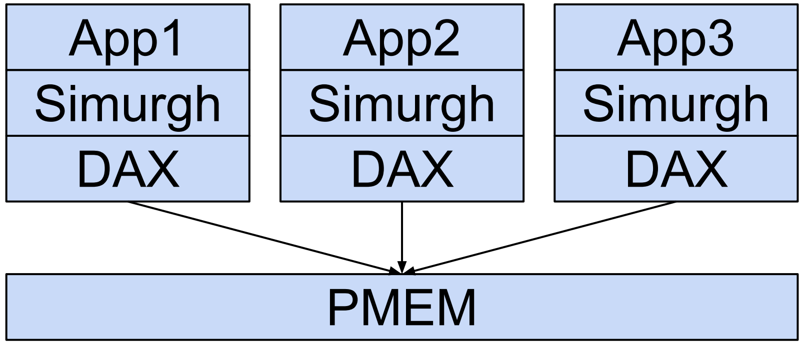
Bento



FUSE



Library I/O Stacks



The Limitations of Existing Platforms

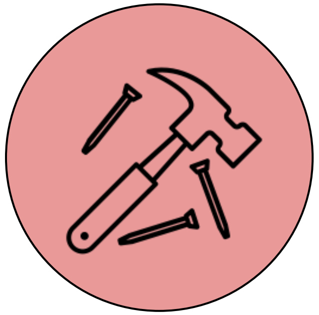
Limited Extensibility

Tight Coupling

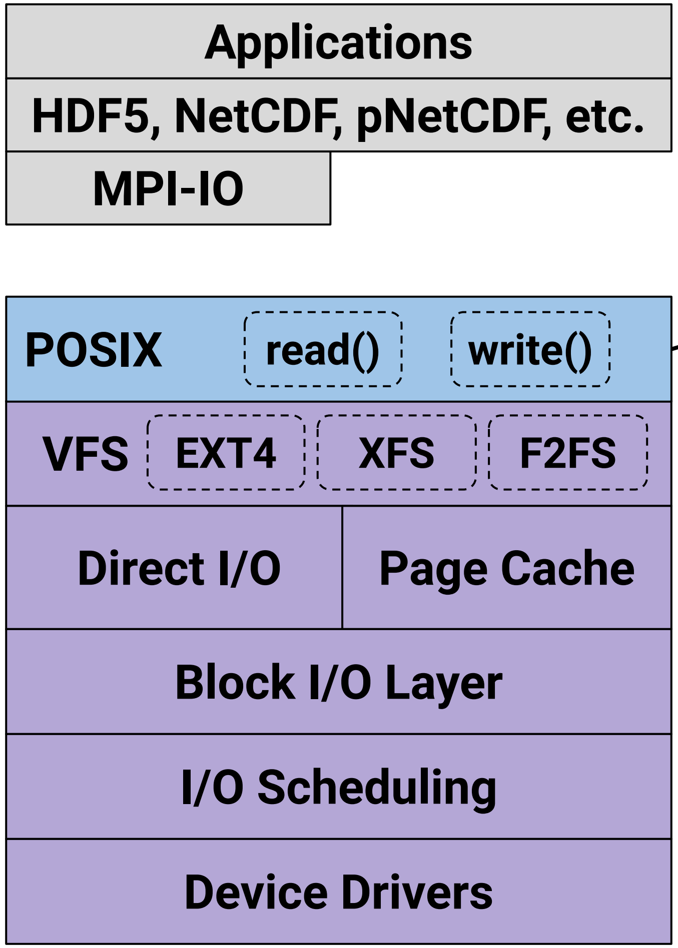
Limited Configurability

Cumbersome Deployments

Limited Extensibility



Development platforms only support the filesystems layer!



Applications

HDF5, NetCDF, pNetCDF, etc.

MPI-IO

POSIX read() write()

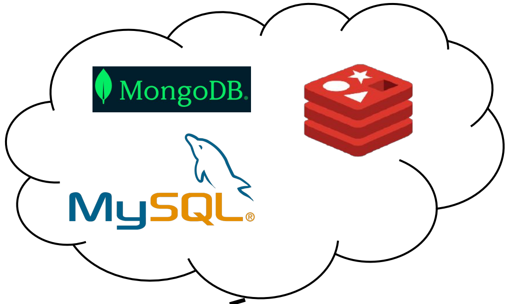
VFS EXT4 XFS F2FS

Direct I/O | Page Cache

Block I/O Layer

I/O Scheduling

Device Drivers



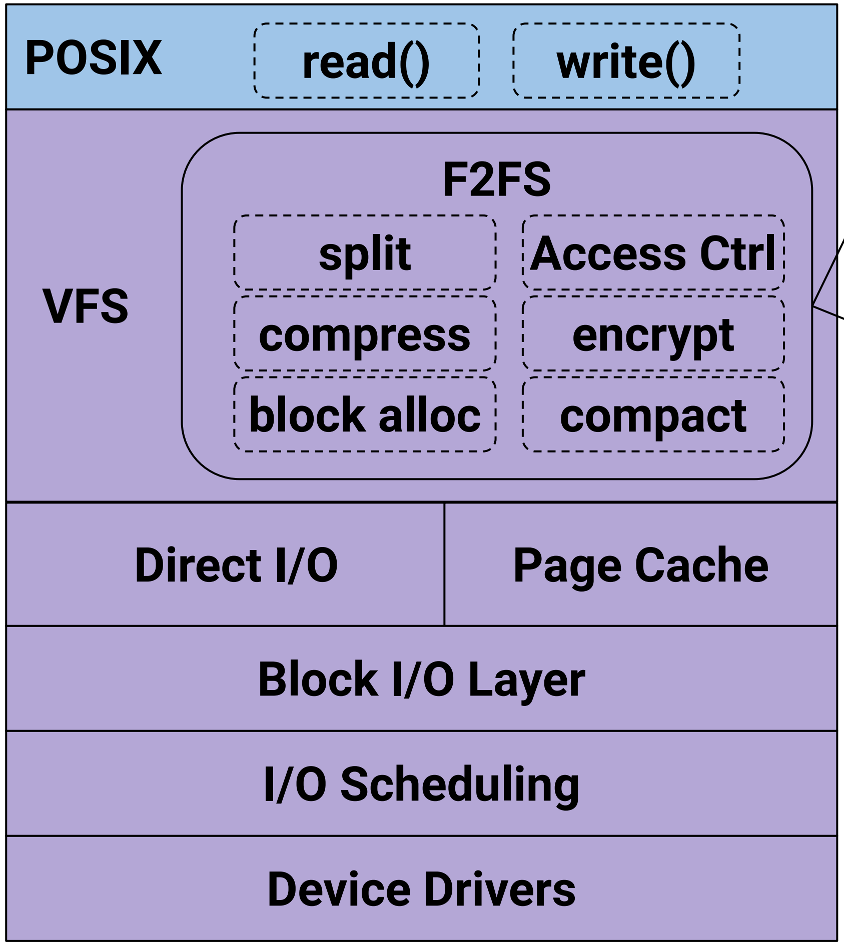
Cannot expose APIs alternative to POSIX

Cannot improve other layers of the I/O stack

Limits performance and feature richness!

Tight Coupling

Development platforms do not promote the single responsibility principle!


POSIX
read()
write()
VFS
F2FS
split
Access Ctrl
compress
encrypt
block alloc
compact
Direct I/O
Page Cache
Block I/O Layer
I/O Scheduling
Device Drivers

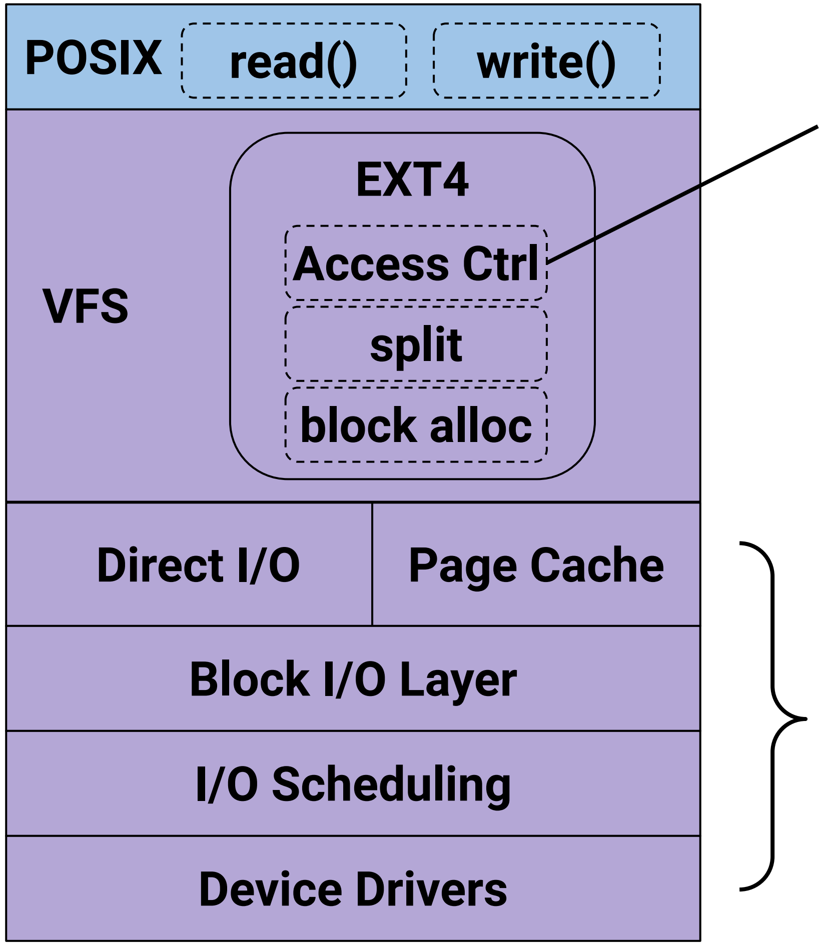
I/O stacks are **tightly coupled** with a large set of features

They cannot be reused by other I/O stacks

Duplicated implementation and debugging effort!

Limited Configurability

I/O stacks are shipped with a fixed set of features!



Some policies are non-negotiable

Some policies are not workload or hardware-optimized

Performance degradation!

Cumbersome Deployments

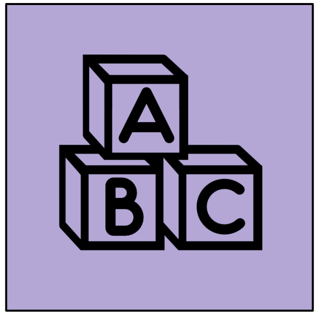
I/O stacks suffer from cumbersome deployment pipelines!

- Ununified namespaces
  - Multiple I/O stacks per program can cause conflict
- Upgrades require reboots and potentially kernel recompilation
- No crash recovery from bugs

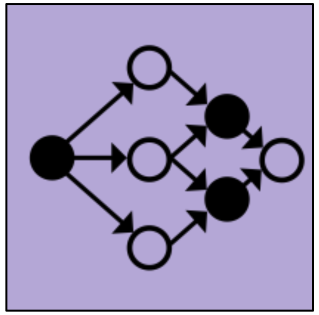
Lowers adoption rates!

The LabStor Platform

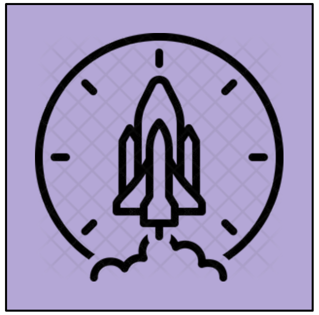
We aim to reinvision the way that I/O stacks are developed and deployed to improve the customizability and code velocity of future I/O stacks



LabMod

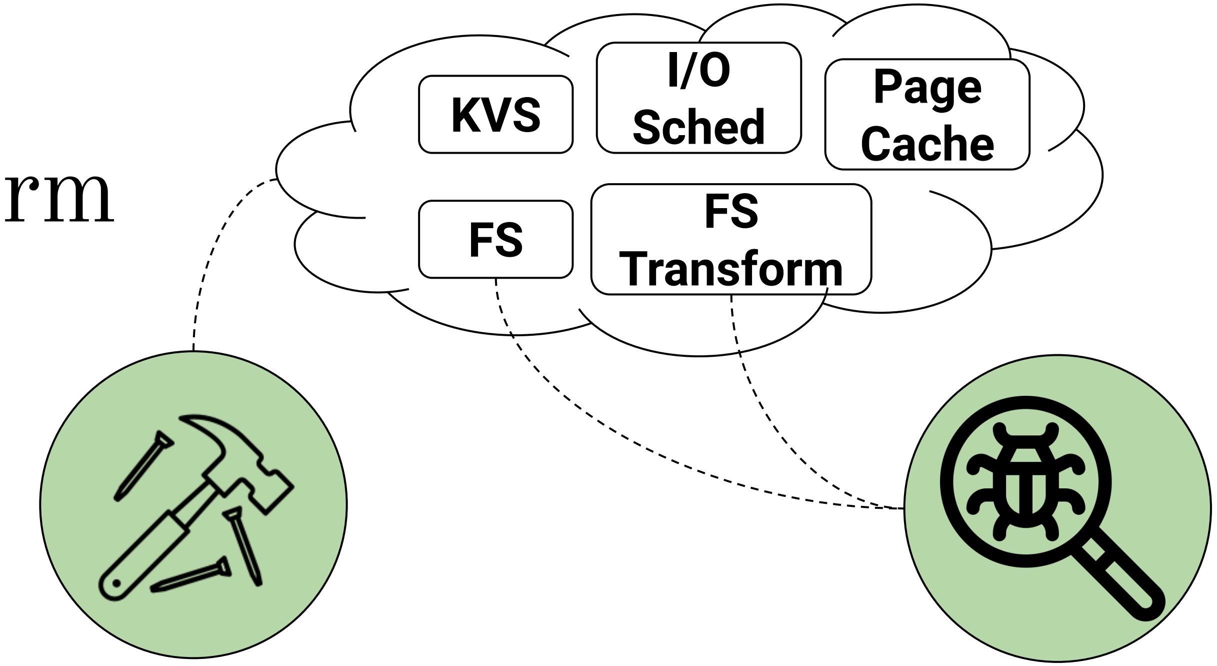


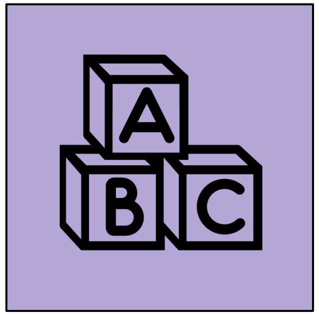
LabStack



LabStor Runtime

The LabStor Platform





LabMod

Generic code object whose type can be chosen by the developer

Promote Extensibility

Provide a wide (and extensible) menu of types for every layer of the stack

Single Responsibility

Provide separate types for filesystems and their features

The LabStor Platform

LabStack

Enable
Configurability

The combination of compatible LabMods into an optimized storage stack

Provide a human-readable schema file format to define LabStacks

The LabStor Platform

LabStor Runtime

Optimize Deployment

Main warehouse and execution engine of LabStor

- Unified namespace for all I/O stacks
- Protocols to upgrade I/O stacks
- Protocols to recover from a crash

Towards fully modular I/O stacks

Towards Modularity: What LabMods Are

Promote Extensibility

Loose Coupling

- An independent, self-contained code object implementing a well-defined, distinct, single-purpose functionality

Full Creative Freedom

Incrementally Upgradeable

Hot Swappable

Stackable

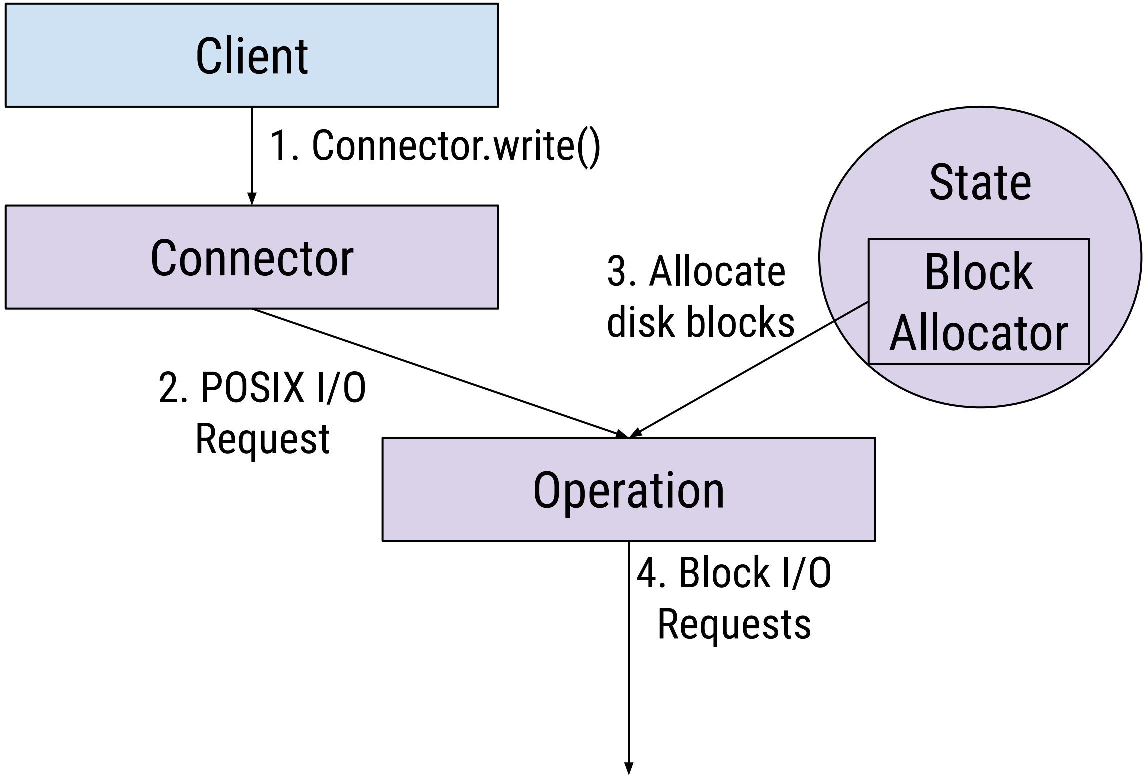
LabMod Components

1. Type
The set of APIs the LabMod implements

2. Connector
Exposes the operation to clients

3. Operation
Functionality of the LabMod

4. State
Internal data required for the operation



LabMod Developer Kit

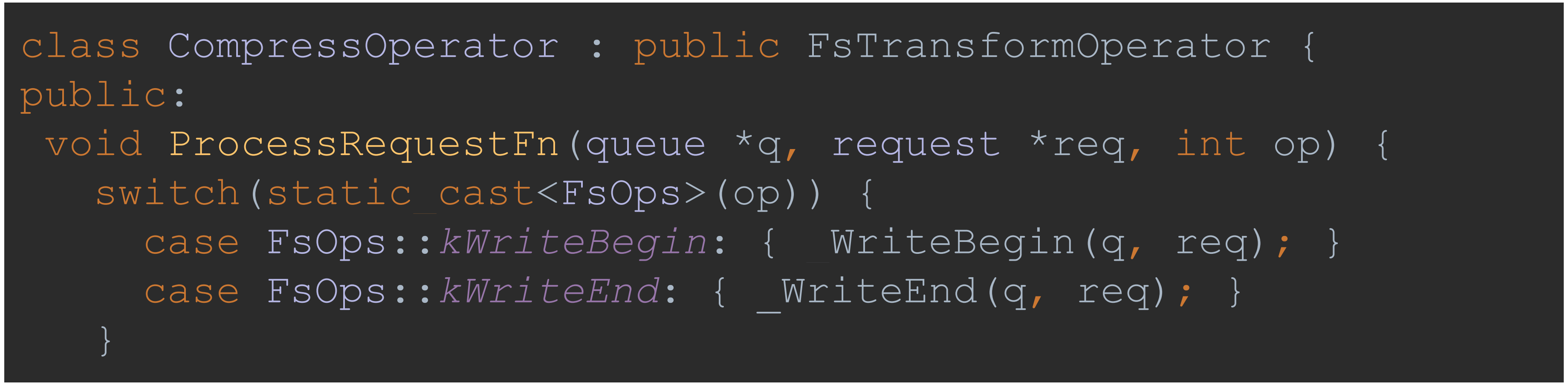
- STL-like data structures and memory allocators
  - Shared-memory compatible
  - Kernel-compatible
- Request queueing API
- Namespace API

LabMod Example: Filesystem Compression (1/3)

```
class CompressConnector : public FsTransformConnector {
public:
 request* WriteBegin(lab::vector<char> &data,
                     ExtendedMetadata &ext_md) {
   auto q = GetQueue();
   auto req = Alloc<fs_request>();
   req->op_ = FsOps::kWriteBegin;
   req->data  << data;
   req->md_ = md;
 }
};
```

- Connectors call LabStor APIs to build & submit requests
- This submits a “kWriteBegin” request

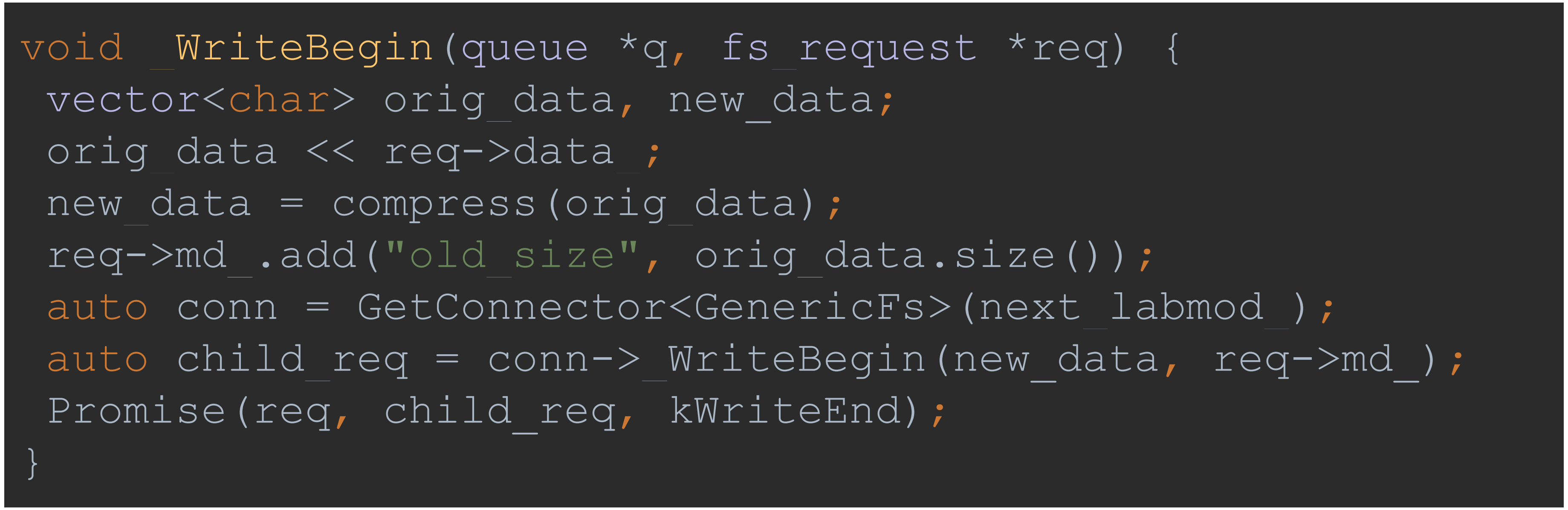
LabMod Example: Filesystem Compression (2/3)



```
class CompressOperator : public FsTransformOperator {
public:
 void ProcessRequestFn(queue *q, request *req, int op) {
   switch(static_cast<FsOps>(op)) {
     case FsOps::kWriteBegin: { _WriteBegin(q, req); }
     case FsOps::kWriteEnd: { _WriteEnd(q, req); }
   }
```

- A worker eventually dequeues the request and calls ProcessRequestFn
- All operators implement ProcessRequestFn
- Routes a request to the proper function

LabMod Example: Filesystem Compression (3/3)



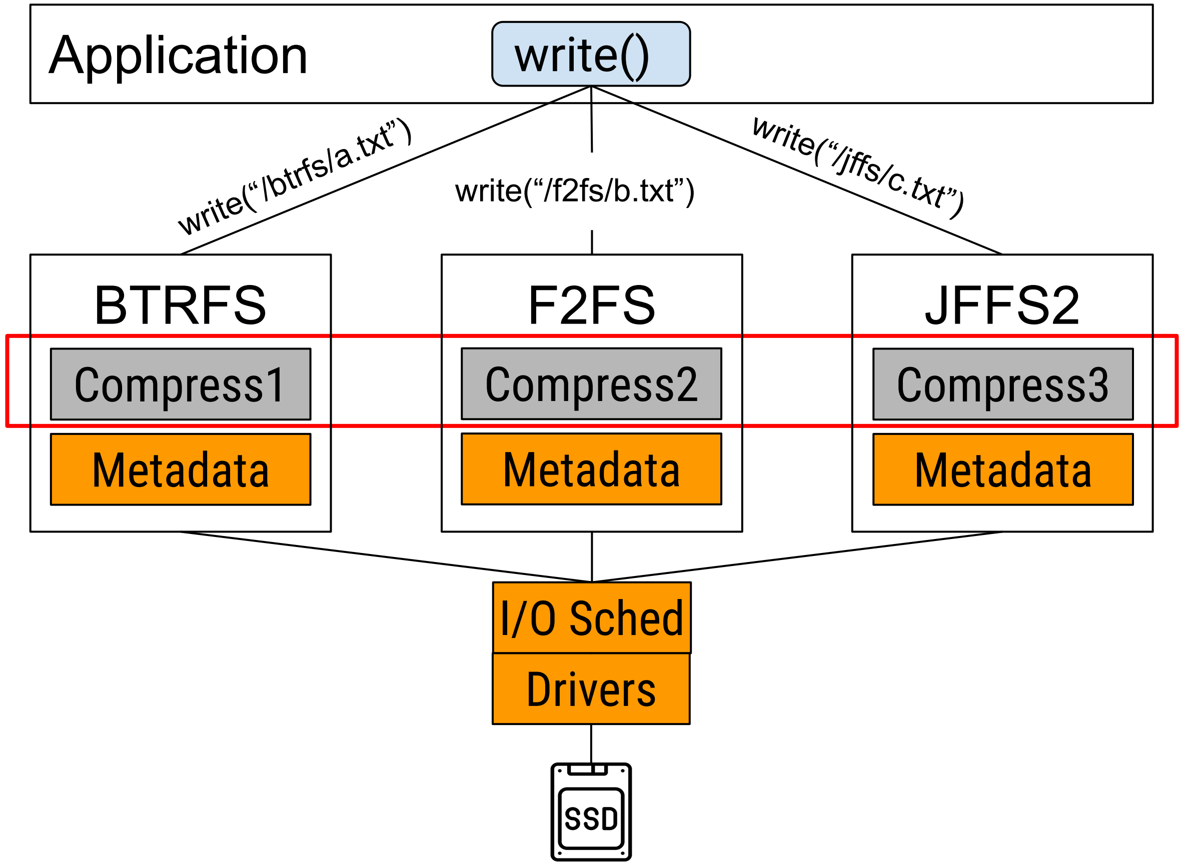
```
void _WriteBegin(queue *q, fs_request *req) {
 vector<char> orig_data, new_data;
 orig_data << req->data_;
 new_data = compress(orig_data);
 req->md_.add("old_size", orig_data.size());
 auto conn = GetConnector<GenericFs>(next_labmod_);
 auto child_req = conn->_WriteBegin(new_data, req->md_);
 Promise(req, child_req, kWriteEnd);
}
```

- The main functionality of the labmod
- Compresses the input data and stores the uncompressed size in metadata
- Promise will asynchronously call kWriteEnd after child_req completes

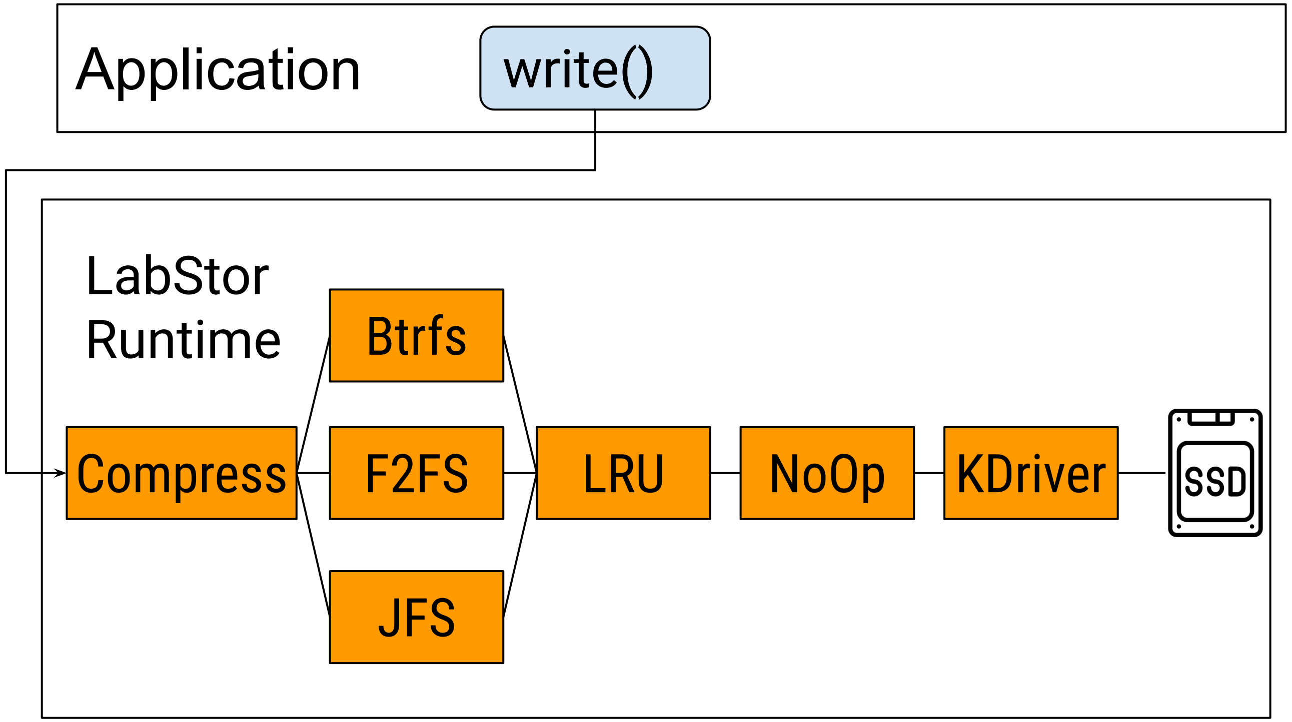
Why does single responsibility matter?

The Benefit of Single Responsibility

- Many Linux filesystems provide transparent compression
- Very similar implementations



The Benefit of Single Responsibility



Total LOC for Compression

Kernel LOC	LabStor LOC
3756	327

- The same compression module can be reused across filesystems in LabStor
- 10x less code needs to be written
- Less debugging and implementation effort!

Towards Composable I/O Services

The LabStack

Enable
Configurability

- A user-defined combination of compatible LabMods into a single I/O system
- LabStacks can be mounted using a human-readable name (e.g., “fs::/a”)

Pick & Choose

Choose only the LabMods required by the I/O stack

Dynamic Modification

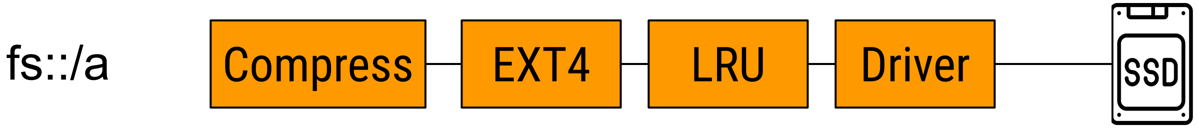
Dynamically modify to adapt to changing I/O requirements

Multiple Paths, Same Data

Provide different views over the same content (e.g., different I/O APIs)

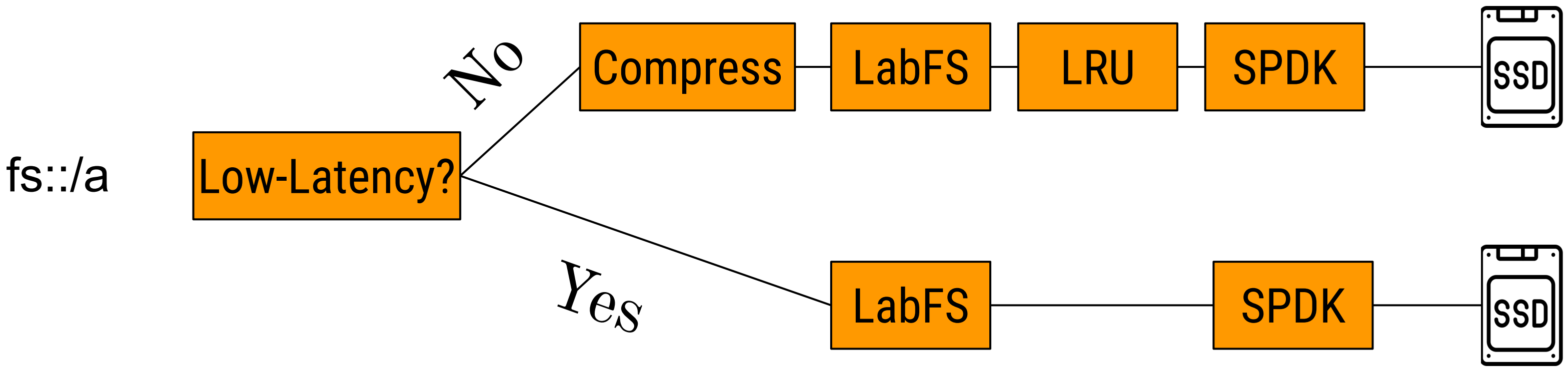
What does composability enable that was not previously possible?

Examples: I/O Specialization (1)



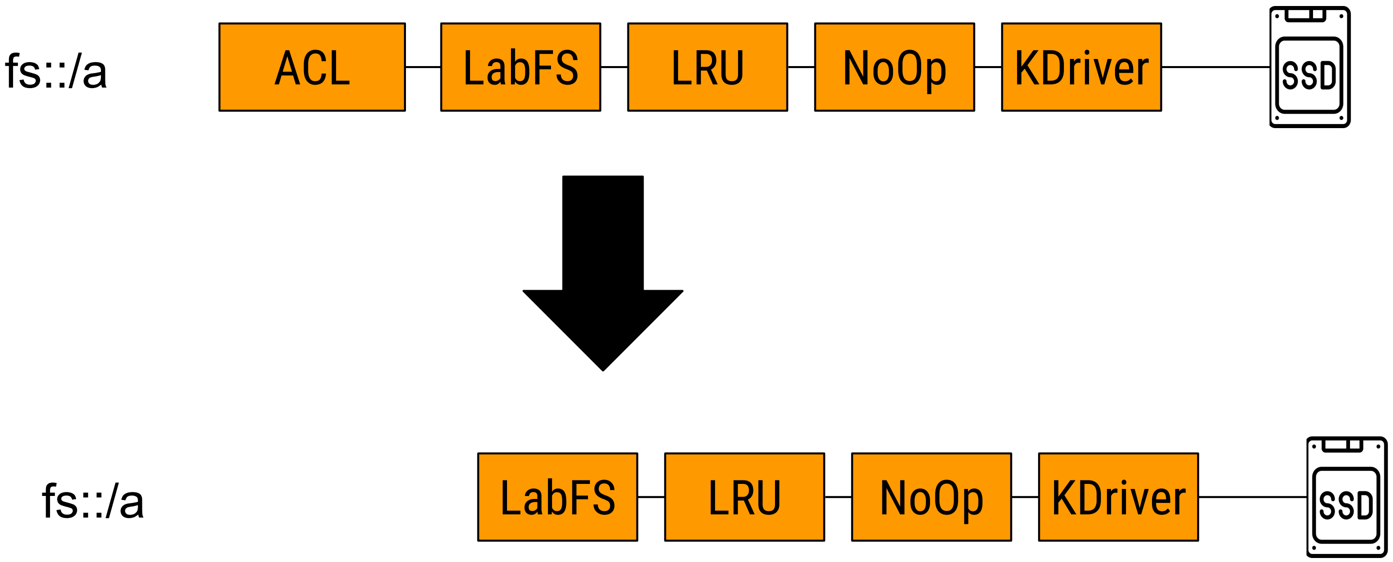
- Typically, a filesystem will have one rigid set of features
- E.g., compression attempted on every I/O access

Examples: I/O Specialization (2)



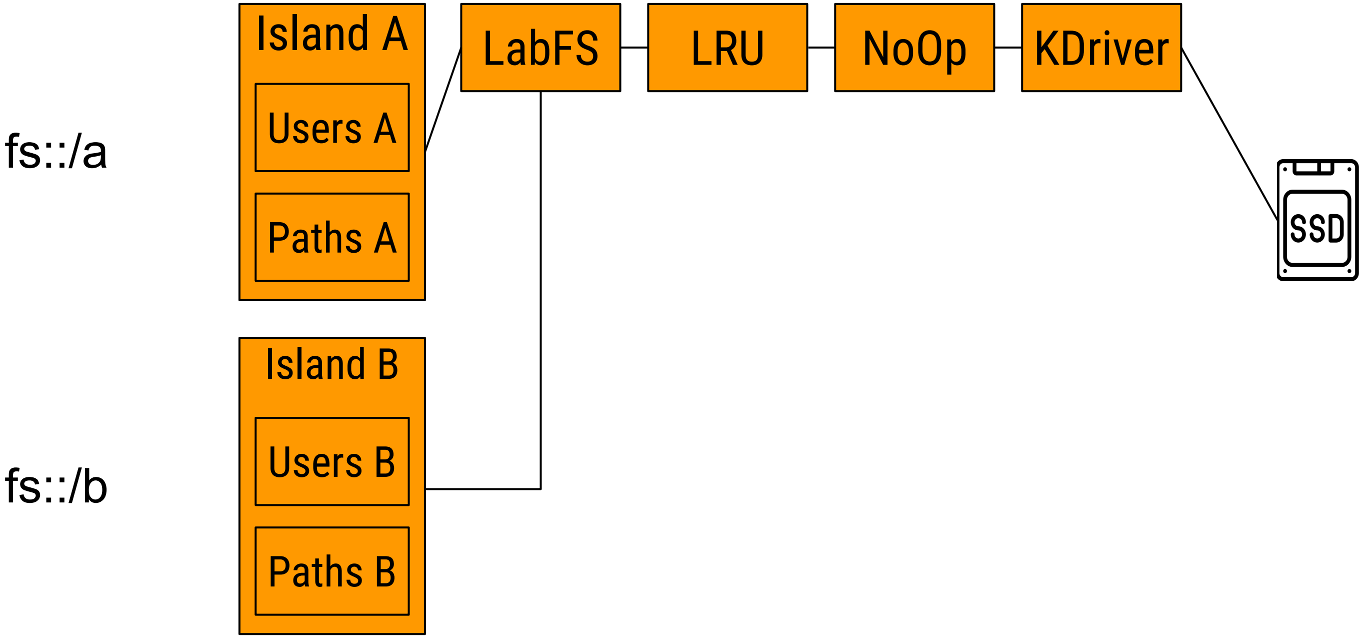
- Can provide different paths which have different optimizations
- High-bandwidth requests get compression + caching
- Low-latency requests get stored ASAP

Examples: Tunable Access Control (1)



- Authentication is typically required on every request in Linux
- Can completely disable authentication by removing that LabMod
- Useful when hardware is dedicated to a user

Examples: Tunable Access Control (2)



fs::/a

Island A

Users A

Paths A

fs::/b

Island B

Users B

Paths B

LabFS

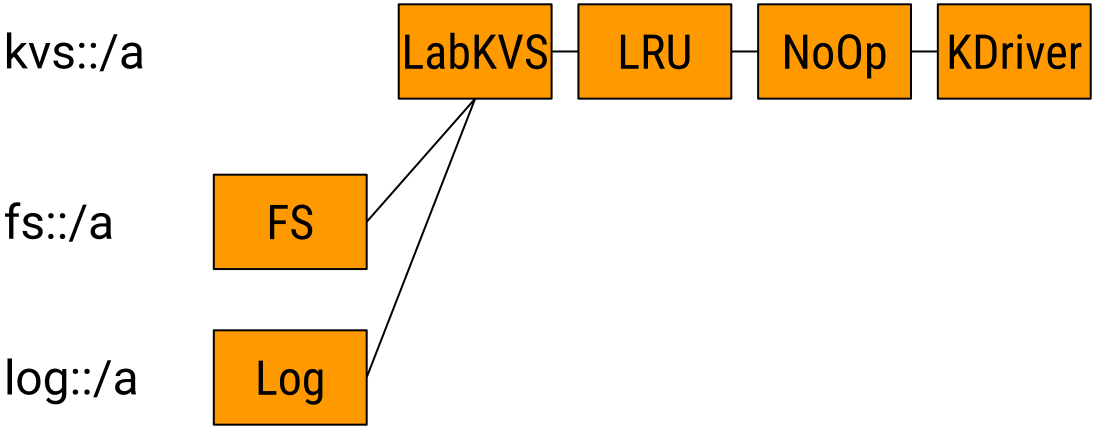
LRU

NoOp

KDriver

- Alternative approaches to authentication can also be created
- fs::/a and fs::/b give users access to different “data islands”

Examples: Interface Convergence & Diversity



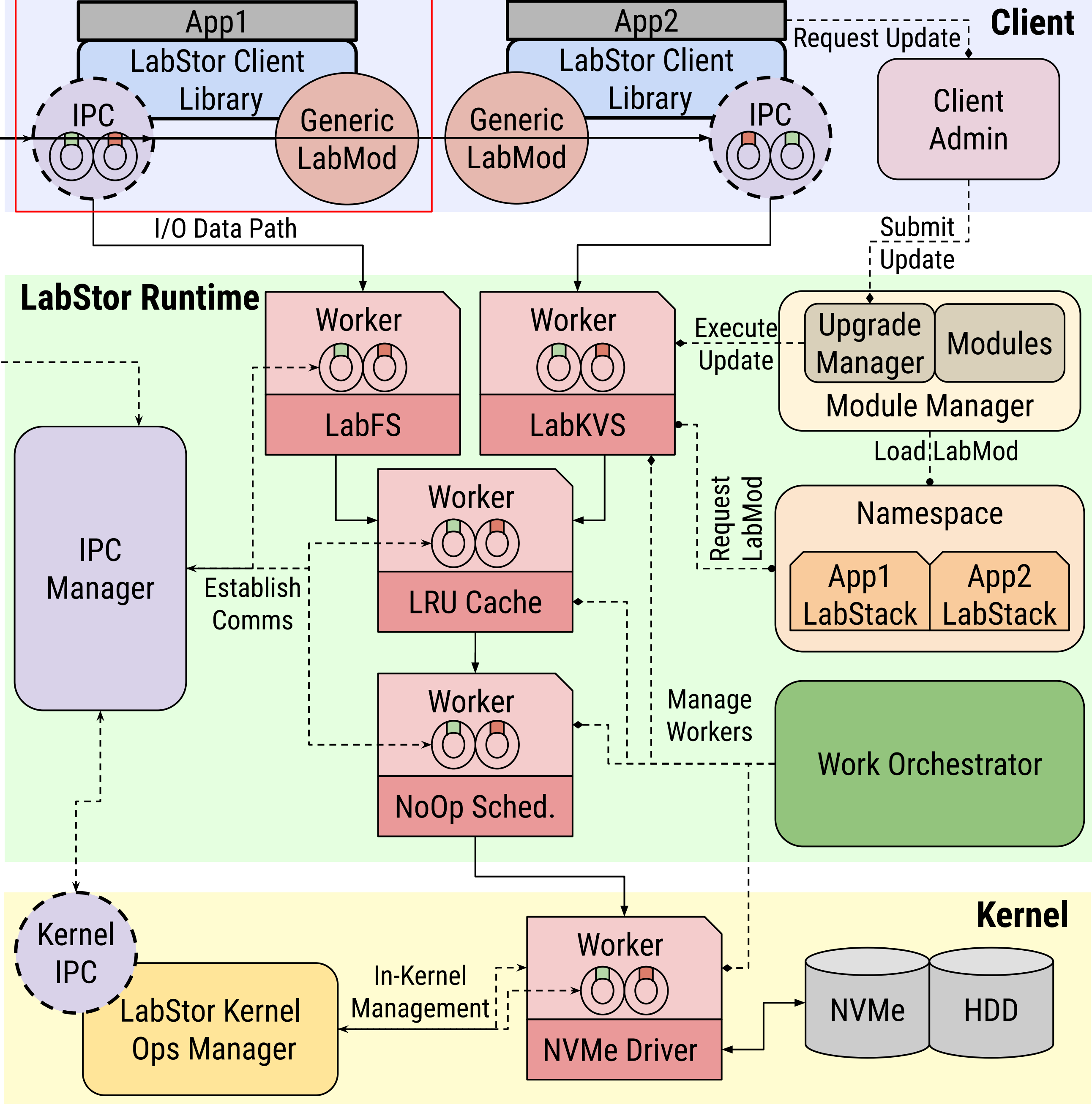
- Provide alternative APIs and data representation to POSIX
- Expose different APIs over the same content
- Data stored as objects, but accessed using either log or filesystem APIs

A Powerful Alternative to the Linux Stack

How does LabStor execute LabStacks?

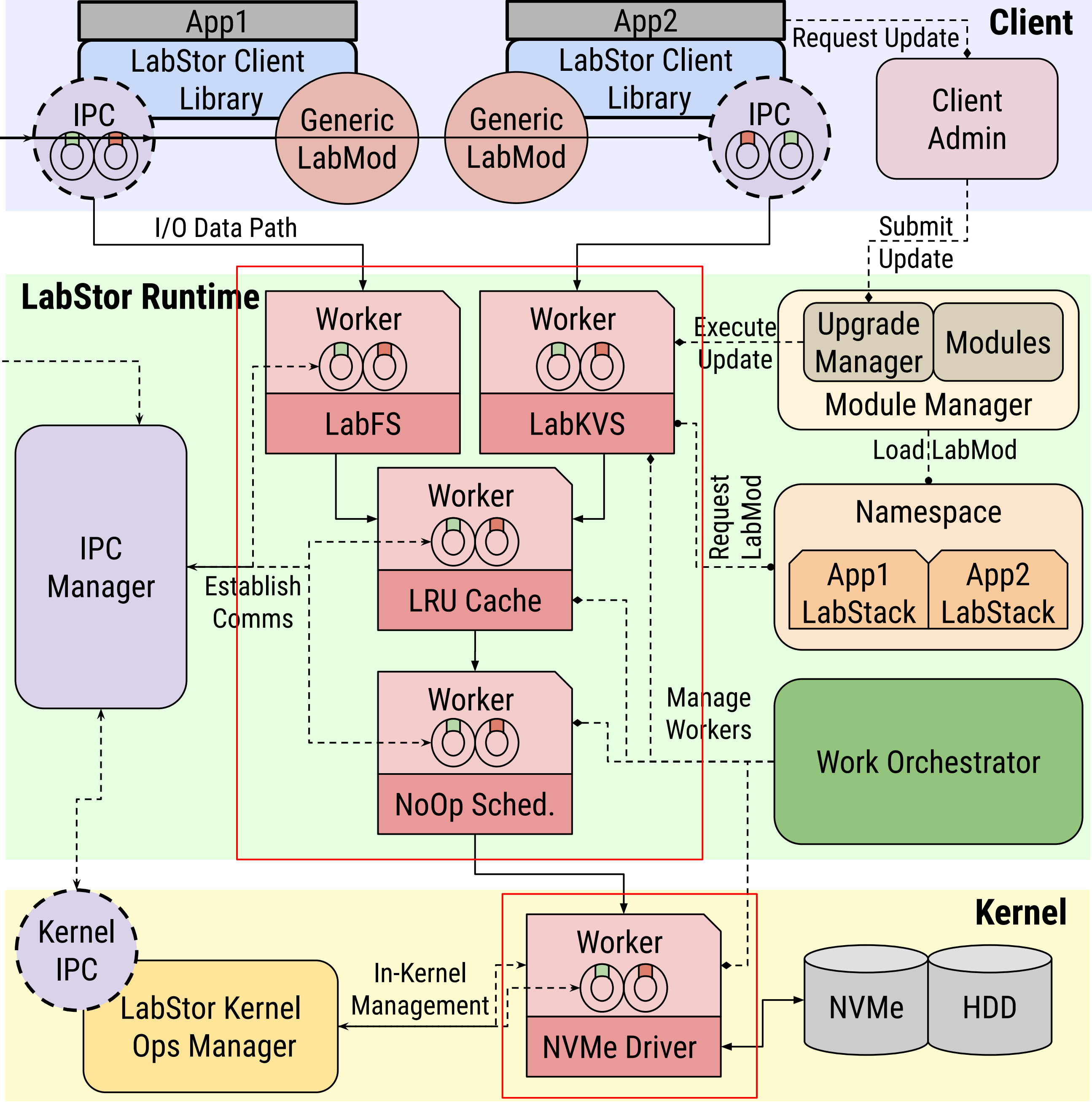
Executing LabStacks (1)

1. Client initially registers with the Runtime
2. Create shared-memory queues between client & runtime
3. Client loads the connector of "Generic LabMod" from the Namespace
4. Call the connector to place an I/O request in the queue



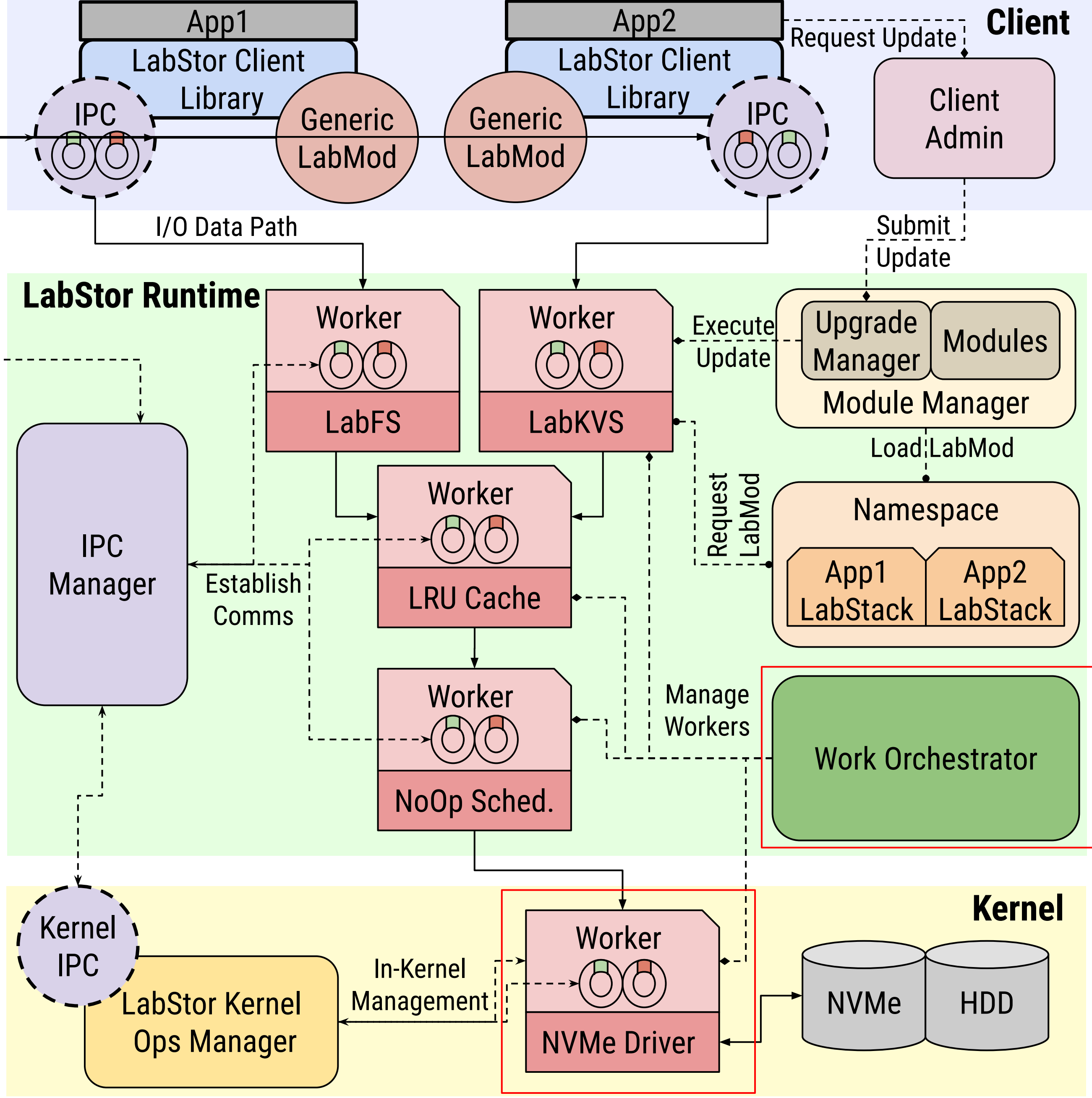
Executing LabStacks (2)

- The runtime spawns or registers workers
- Eventually process the queues
- Workers can execute either in kernel or userspace
  - Re-use kernel functionality



Executing LabStacks (3)

- Work orchestrator assigns queues to workers
- Can place multiple queues on a single worker
    - Helpful if requests are latency-sensitive



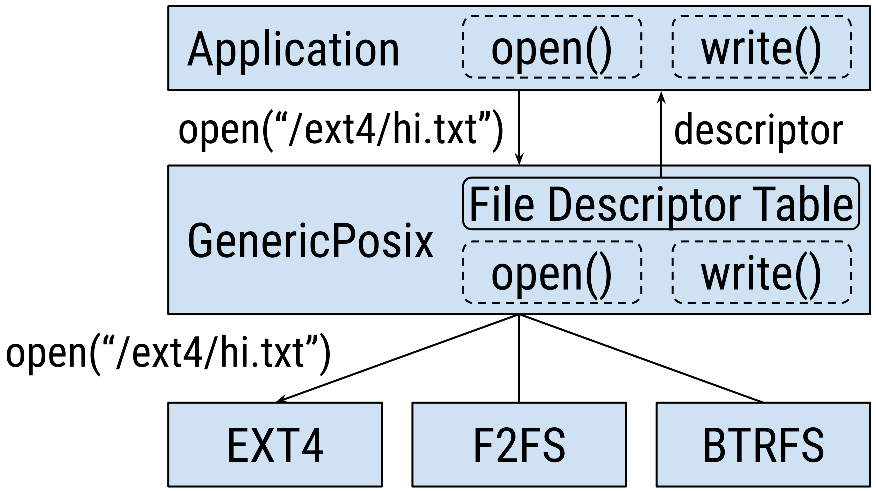
How to deploy LabStacks?

Deployment

- LabStacks are mounted in the LabStack Namespace using a utility script (mount.labstack)
- After mounting, there are three ways of accessing it:
  - **Native API** (GetConnector)
  - **API Interception** (Generic LabMods)
  - System calls (VFS)

Generic LabMods & Unified Namespace

Intercept	**Route**	**Reuse**
Intercept I/O for a particular API (e.g., POSIX)	Many implementations of the same API at the same time	Manages state common between I/O systems of a particular type



LabStack NS

Key	**LabStack**
/ext4	EXT4
/f2fs	F2FS
/btrfs	BTRFS

Generic LabMods & Unified Namespace

Intercept

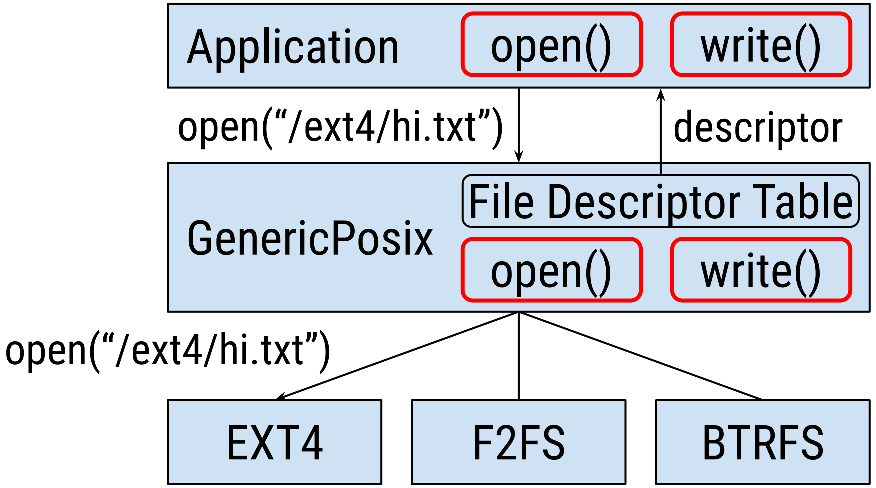
Intercept I/O for a particular API (e.g., POSIX)

Route

Many implementations of the same API at the same time

Reuse

Manages state common between I/O systems of a particular type



LabStack NS

Key	**LabStack**
/ext4	EXT4
/f2fs	F2FS
/btrfs	BTRFS

Generic LabMods & Unified Namespace

Intercept

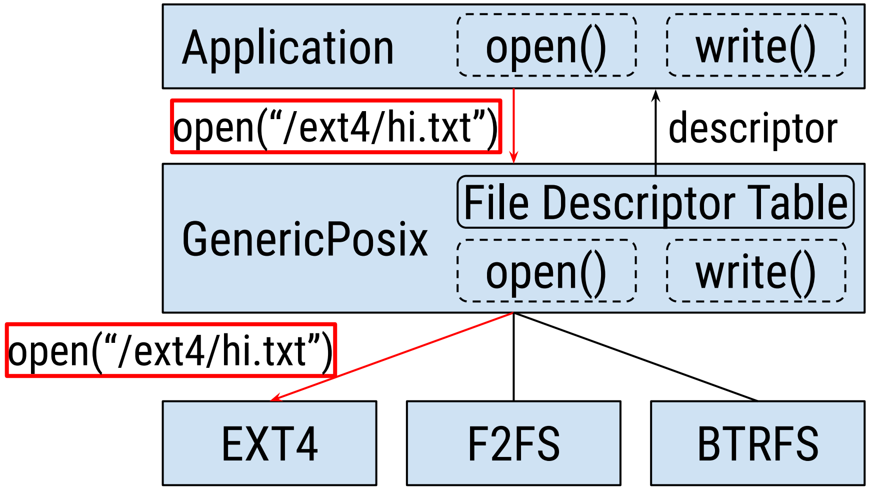
Intercept I/O for a particular API (e.g., POSIX)

Route

Many implementations of the same API at the same time

Reuse

Manages state common between I/O systems of a particular type



LabStack NS

Key	**LabStack**
/ext4	EXT4
/f2fs	F2FS
/btrfs	BTRFS

Generic LabMods & Unified Namespace

Intercept

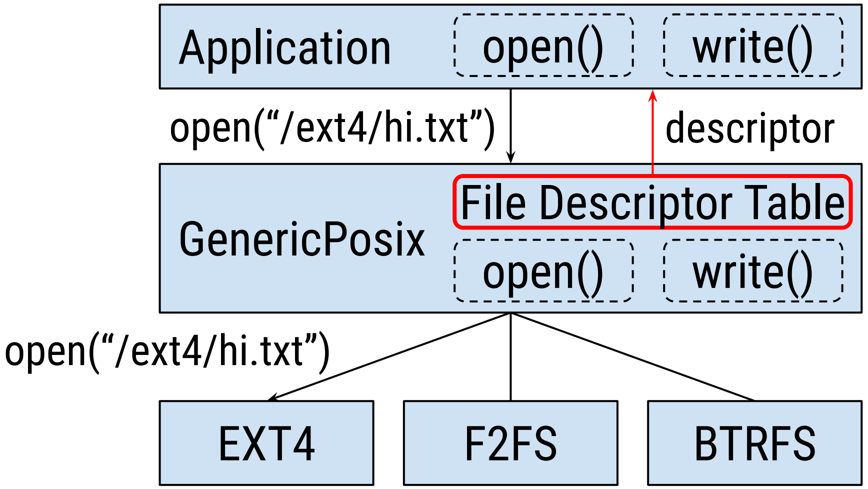
Intercept I/O for a particular API (e.g., POSIX)

Route

Many implementations of the same API at the same time

Reuse

Manages state common between I/O systems of a particular type



Application

open()

write()

open("/ext4/hi.txt")

descriptor

GenericPosix

File Descriptor Table

open()

write()

open("/ext4/hi.txt")

EXT4

F2FS

BTRFS

LabStack NS

Key	**LabStack**
/ext4	EXT4
/f2fs	F2FS
/btrfs	BTRFS

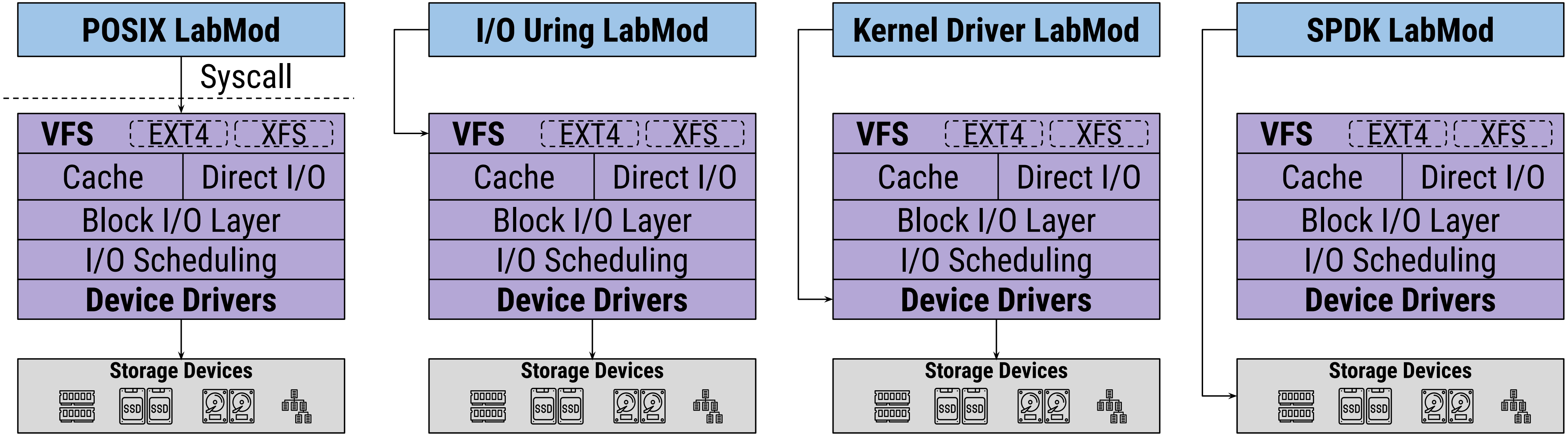
Other deployment considerations

- **Crash recovery:** what happens if a LabMod was buggy and crashed the Runtime?
    - Data structures in shared memory, and can be recovered after a crash
- **Upgrade protocols:** how to update LabMods after deployment?
    - Request queues are paused, and all pending operations will be completed
    - The upgrades are then processed

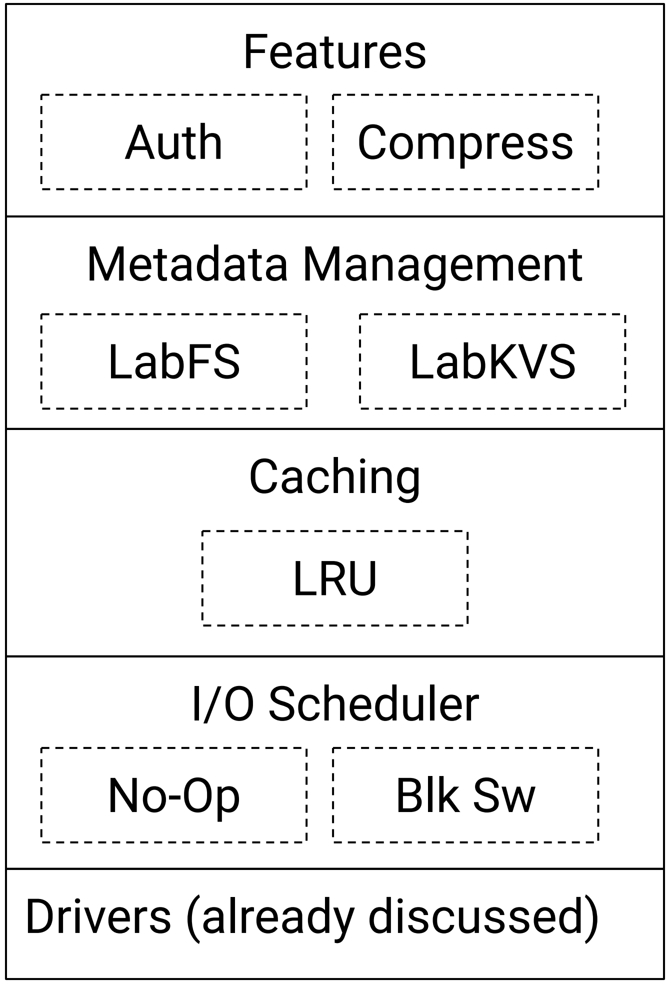
Provided LabMods

Diverse Storage Driver Layer

- Various ways to interact with hardware
- Tradeoffs between hardware-generality and performance



Our Prototype LabMods



Features

Auth

Compress

Metadata Management

LabFS

LabKVS

Caching

LRU

I/O Scheduler

No-Op

Blk Sw

Drivers (already discussed)

Evaluations

Testbed

- Chameleon Cloud
- Storage hierarchy node
- NVMe (Intel P3700, 2TB)
- SSD (Intel SSDSC2BX01, 1.6TB)
- HDD (Seagate ST600MP0005, 600G)
- RAM (512GB)
- CPU: 24 core / 48 threads
  - 2x Intel(R) Xeon(R) CPU

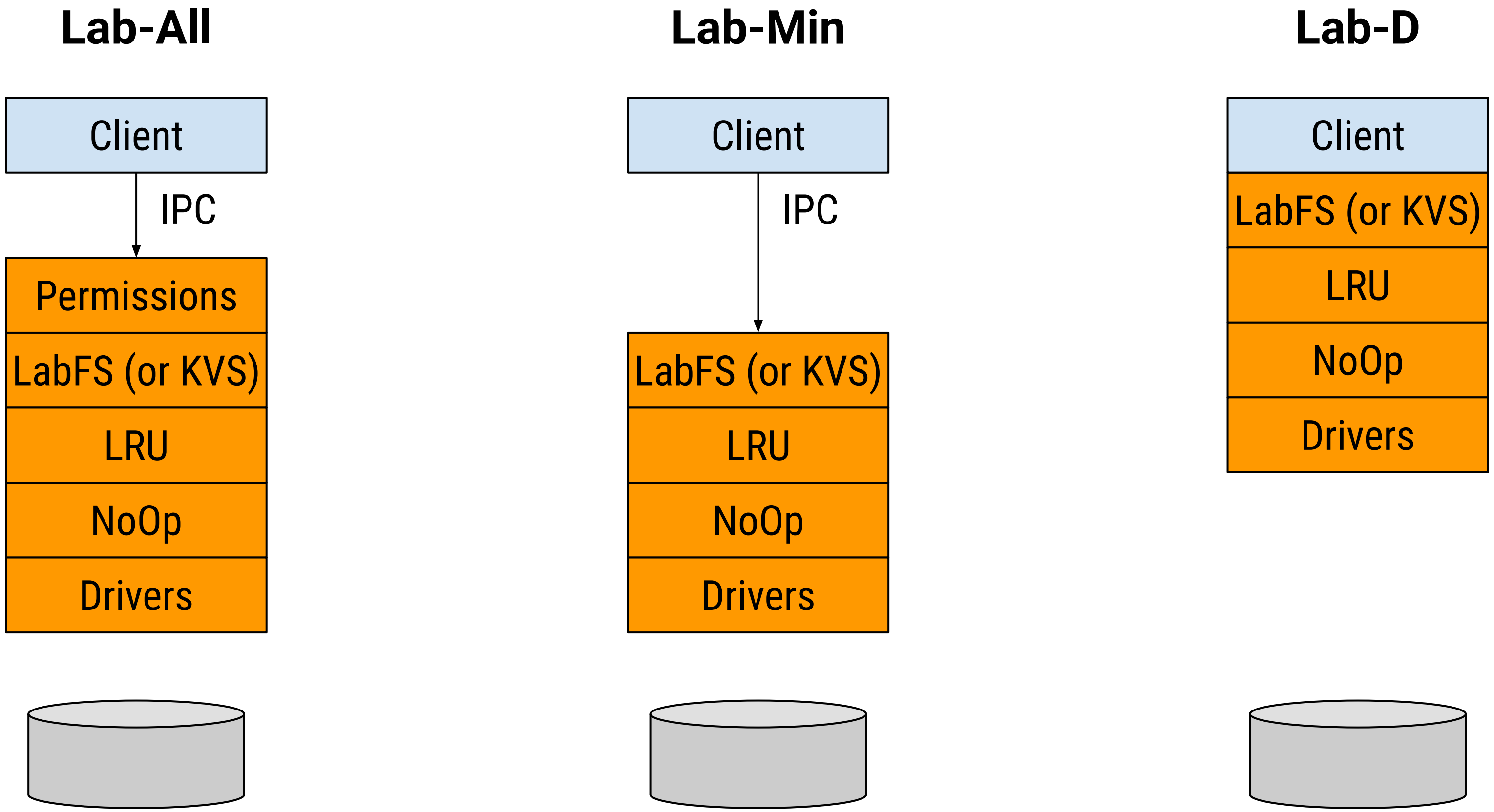
Software

- Ubuntu 20.04, kernel 5.4
- FIO 3.28, FxMark, LABIOS, Filebench 1.4.9.1

Evaluation Objectives

Modularity	**Customization**	**I/O Expressiveness**	**Design Correctness**
The choice of modules have major performance impacts	I/O stacks should be more customized to the workload and environment	The best I/O interface should be chosen to store data	LabStor can execute I/O stacks without sacrificing resource utilization or performance

Experimental Setup: LabStacks



Lab-All

Client

IPC

Permissions

LabFS (or KVS)

LRU

NoOp

Drivers

Lab-Min

Client

IPC

LabFS (or KVS)

LRU

NoOp

Drivers

Lab-D

Client

LabFS (or KVS)

LRU

NoOp

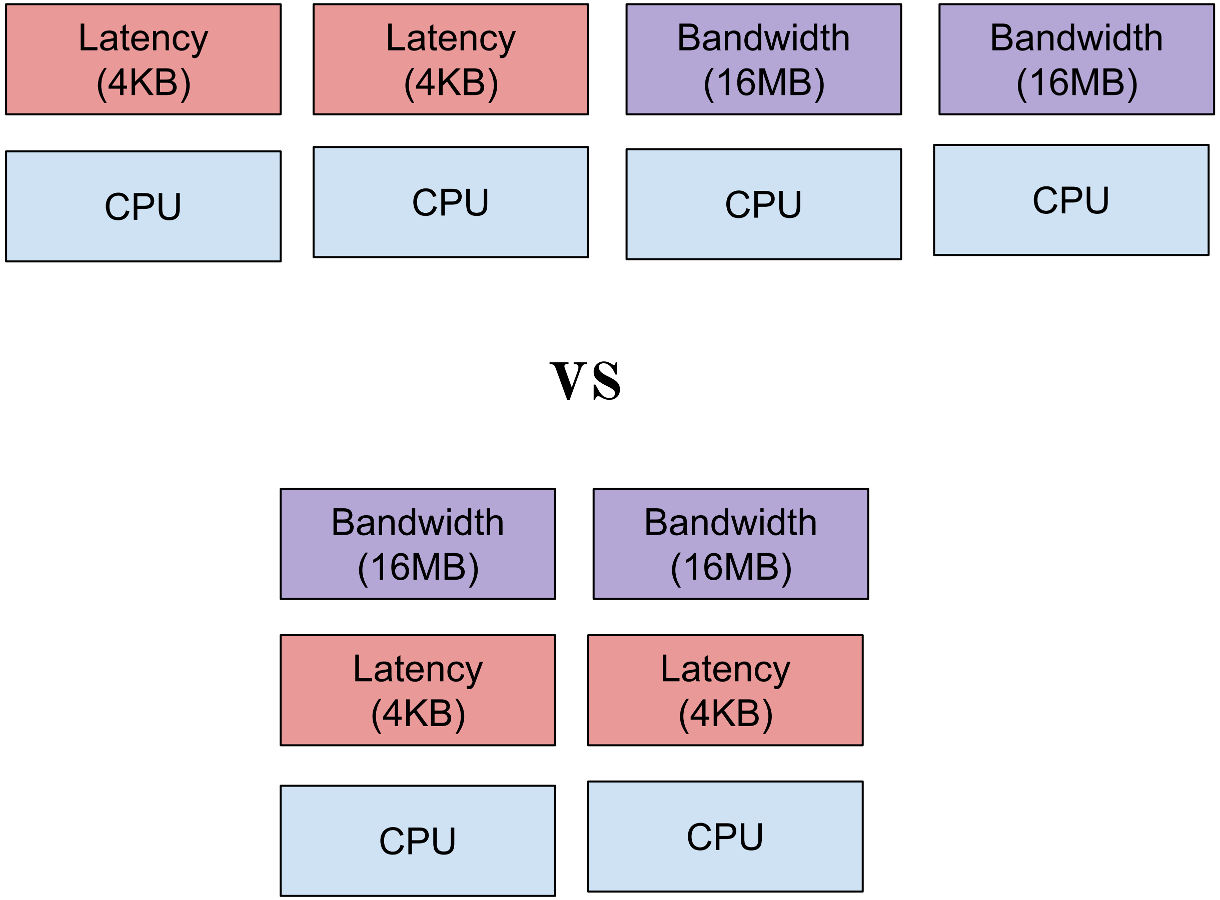
Drivers

What is the performance difference between LabStor and other development platforms?

What is the performance benefit of having a configurable I/O stack?

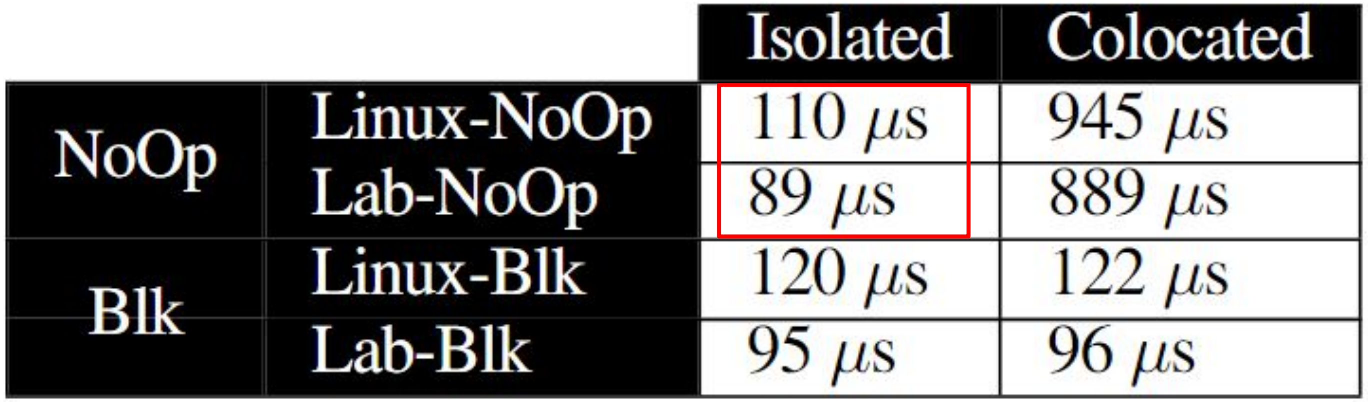
Developing and Customizing I/O Policies

- Two ioscheds
  - No-Op
  - Blk-Switch
- Two workloads:
  - Latency: synchronous 4K requests
  - Bandwidth: synchronous 16MB requests
- Compare when the workloads are isolated and colocated



Developing and Customizing I/O Policies

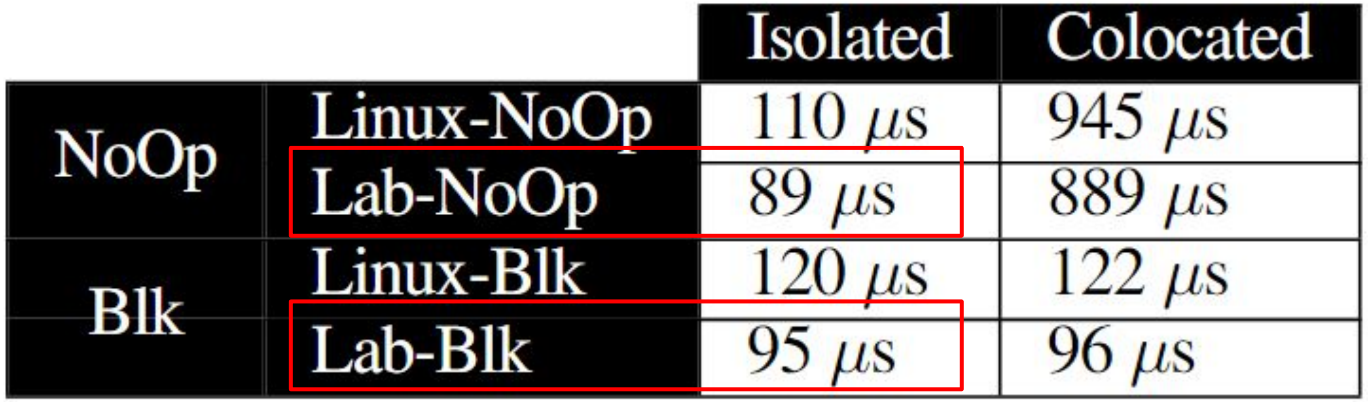
- LabStor's NoOp and Blk-Switch are 20% faster than their in-kernel counterpart
- LabStor bypasses significant in-kernel overheads



		Isolated	Colocated
NoOp	Linux-NoOp	110 μs	945 μs
	Lab-NoOp	89 μs	889 μs
Blk	Linux-Blk	120 μs	122 μs
	Lab-Blk	95 μs	96 μs

Developing and Customizing I/O Policies

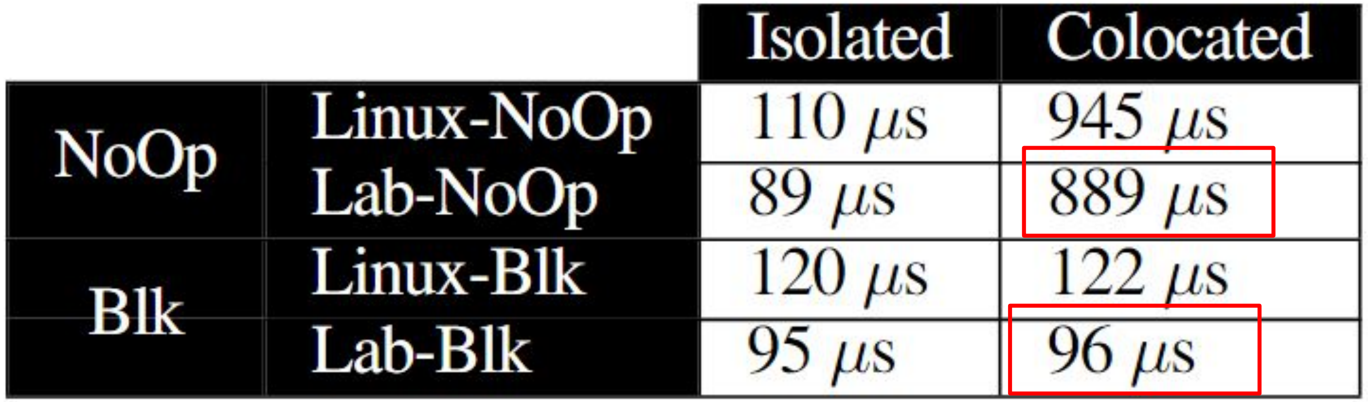
- No-Op is 8% faster than blk-switch when there is no colocation
- Blk-sw has overhead due to additional code logic



		Isolated	Colocated
NoOp	Linux-NoOp	110 μs	945 μs
	Lab-NoOp	89 μs	889 μs
Blk	Linux-Blk	120 μs	122 μs
	Lab-Blk	95 μs	96 μs

Developing and Customizing I/O Policies

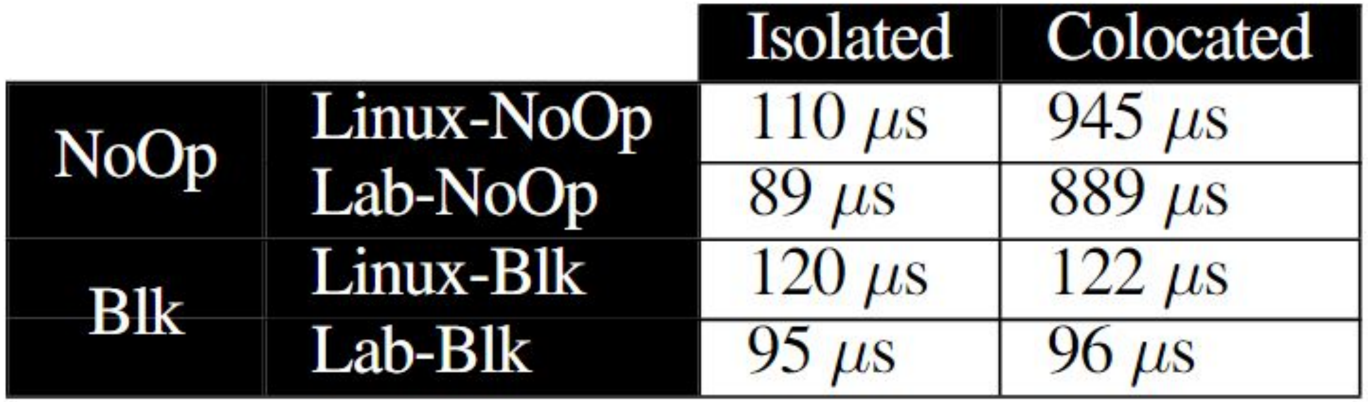
- Blk-switch is 10x faster than No-Op when there is colocation
- Routes latency-sensitive requests to separate queues, reducing starvation



		Isolated	Colocated
NoOp	Linux-NoOp	110 μs	945 μs
	Lab-NoOp	89 μs	889 μs
Blk	Linux-Blk	120 μs	122 μs
	Lab-Blk	95 μs	96 μs

Developing and Customizing I/O Policies

- Both policies have pros and cons in different circumstances

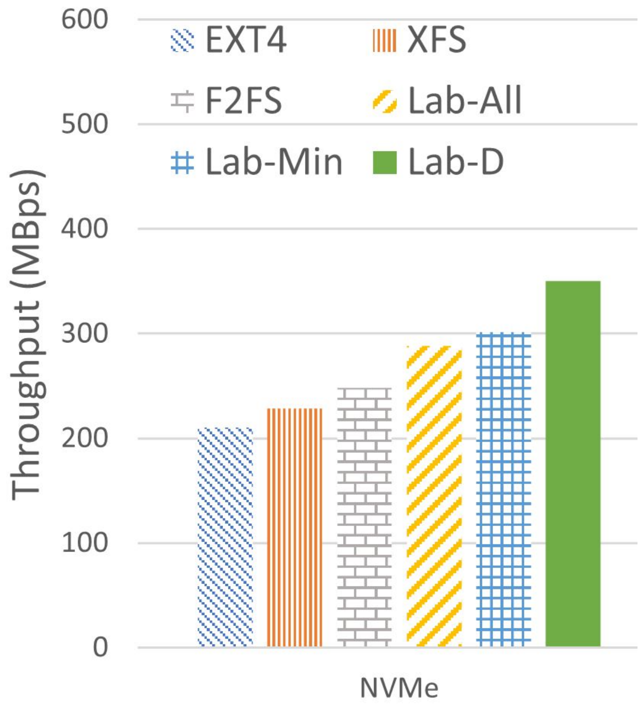


		Isolated	Colocated
NoOp	Linux-NoOp	110 μs	945 μs
	Lab-NoOp	89 μs	889 μs
Blk	Linux-Blk	120 μs	122 μs
	Lab-Blk	95 μs	96 μs

What is the benefit of enabling more than just POSIX filesystems to be developed?

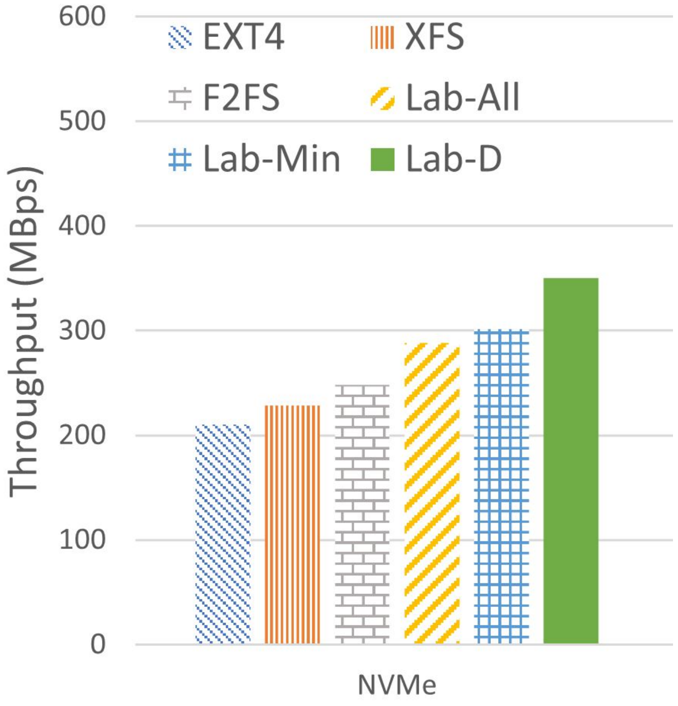
I/O Expressiveness

- Labios: a distributed storage system used to bridge the gap between different I/O stacks
- POSIX vs Key-Value API
- Labios generates 8KB I/Os and stores using different stacks



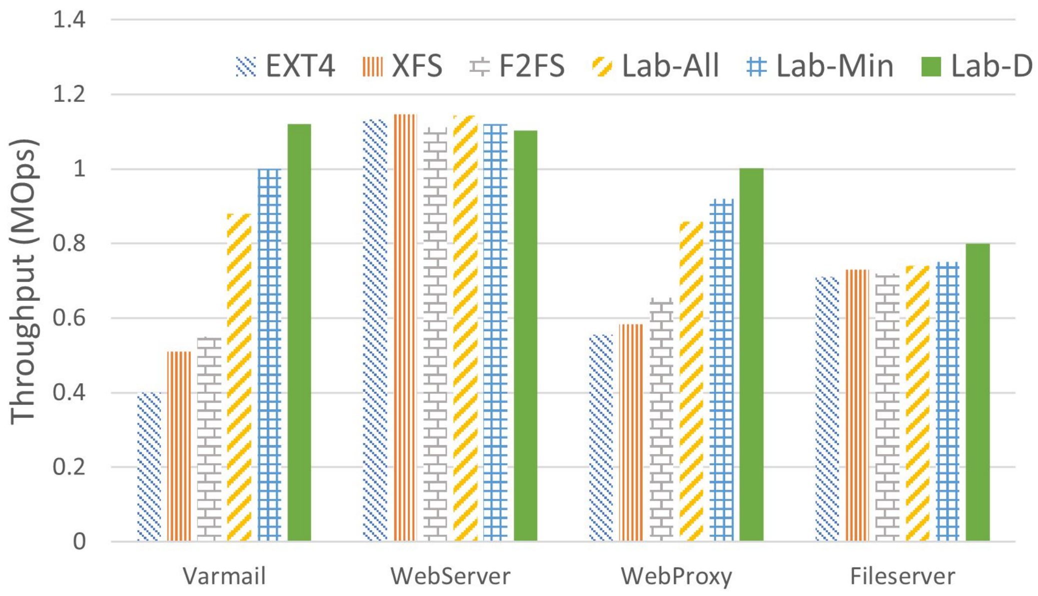
I/O Expressiveness

- LabKVS outperforms all I/O stacks for various use cases
- This is because KVS reduces # of syscalls from 4 down to 1, significantly reducing software overhead
- Providing new interfaces to storage can provide substantial benefits



How does increased modularity improve real-world programs?

Filebench



- We run a full real-workload over various LabStacks
- We find that different LabStor configurations yield different performance
  - Except webserver, which performs large-sequential I/O
- Can save up to 40% on performance by choosing only the required labmods

Conclusion

Conclusions

LabStor: a platform for developing high-performance, customized I/O stacks in userspace

Modular

Provide expressive, customizable and high-velocity I/O stacks

Composable

Provide the ability to upgrade and manage I/O stacks in an easy and efficient manner

Performant

Up to 60% gains under various workloads and devices

SCALABLE COMPUTING
SOFTWARE LABORATORY

ILLINOIS INSTITUTE
OF TECHNOLOGY

NSF

Chameleon

Sandia
National
Laboratories