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
The Explosion of I/O Requirements

A wide diversity of applications

- Simulation
- Machine Learning
- 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!
Parallel Filesystems

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

App1
Simurgh
DAX

App2
Simurgh
DAX

App3
Simurgh
DAX

PMEM

Luke Logan

llogan@hawk.iit.edu

LabStor: A modular platform for developing custom I/O stacks

llogan@hawk.iit.edu
The Limitations of Existing Platforms

- Limited Extensibility
- Tight Coupling
- Limited Configurability
- Cumbersome Deployments
## Limited Extensibility

### Development platforms only support the filesystems layer!

### Cannot expose APIs alternative to POSIX

### Cannot improve other layers of the I/O stack

### Limits performance and feature richness!

### Applications
- HDF5, NetCDF, pNetCDF, etc.
- MPI-IO

<table>
<thead>
<tr>
<th>POSIX</th>
<th>read()</th>
<th>write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFS</td>
<td>EXT4</td>
<td>XFS</td>
</tr>
<tr>
<td>Direct I/O</td>
<td>Page Cache</td>
<td></td>
</tr>
<tr>
<td>Block I/O Layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O Scheduling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Drivers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Tight Coupling

I/O stacks are **tightly coupled** with a large set of features. They cannot be reused by other I/O stacks.

Development platforms do not promote the single responsibility principle!

<table>
<thead>
<tr>
<th>POSIX</th>
<th>read()</th>
<th>write()</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F2FS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>split</td>
<td>Access Ctrl</td>
</tr>
<tr>
<td></td>
<td>compress</td>
<td>encrypt</td>
</tr>
<tr>
<td></td>
<td>block alloc</td>
<td>compact</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Direct I/O</th>
<th>Page Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block I/O Layer</td>
<td></td>
</tr>
<tr>
<td>I/O Scheduling</td>
<td></td>
</tr>
<tr>
<td>Device Drivers</td>
<td></td>
</tr>
</tbody>
</table>

**Duplicates 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

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

I/O stacks suffer from cumbersome deployment pipelines!

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

Promote Extensibility

Single Responsibility

Generic code object whose type can be chosen by the developer

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

Provide separate types for filesystems and their features
The LabStor Platform

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

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

**Promote Extensibility**

**Loose Coupling**

- 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

---

**Diagram:**
- **Client**
  - 1. Connector.write()
- **Connector**
  - 2. POSIX I/O Request
- **Operation**
  - 3. Allocate disk blocks
- **State**
  - 4. Block I/O Requests
  - Block Allocator
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)

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

```cpp
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;
  }
};
```
LabMod Example: Filesystem Compression (2/3)

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

```cpp
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); } 
        }
    }
```
LabMod Example: Filesystem Compression (3/3)

- 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

```c
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);
}
```
Why does single responsibility matter?
The Benefit of Single Responsibility

- Many Linux filesystems provide transparent compression
- Very similar implementations
The Benefit of Single Responsibility

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

### Total LOC for Compression

<table>
<thead>
<tr>
<th></th>
<th>Kernel LOC</th>
<th>LabStor LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Btrfs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2FS</td>
<td>3756</td>
<td>327</td>
</tr>
<tr>
<td>LRU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoOp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KDriver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JFS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LabStor: A modular platform for developing custom I/O stacks

Luke Logan

llogan@hawk.iit.edu
Towards Composable I/O Services
The LabStack

- 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)

- 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**
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**

<table>
<thead>
<tr>
<th>Key</th>
<th>LabStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ext4</td>
<td>EXT4</td>
</tr>
<tr>
<td>/f2fs</td>
<td>F2FS</td>
</tr>
<tr>
<td>/btrfs</td>
<td>BTRFS</td>
</tr>
</tbody>
</table>

**Application**
- open() (e.g., open("/ext4/hi.txt"))
- write()

**GenericPosix**
- open() (e.g., open("/ext4/hi.txt"))
- write()
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**

<table>
<thead>
<tr>
<th>Key</th>
<th>LabStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ext4</td>
<td>EXT4</td>
</tr>
<tr>
<td>/f2fs</td>
<td>F2FS</td>
</tr>
<tr>
<td>/btrfs</td>
<td>BTRFS</td>
</tr>
</tbody>
</table>

Application

open("/ext4/hi.txt")

descriptor

**GenericPosix**

open("/ext4/hi.txt")

**File Descriptor Table**

- **EXT4**
- **F2FS**
- **BTRFS**

- open() /write()
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

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

Application
open() write()
open("/ext4/hi.txt") descriptor

GenericPosix
open() write()
open("/ext4/hi.txt")

File Descriptor Table

EXT4 F2FS 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

**Application**
open() write()

open("/ext4/hi.txt") descriptor

**GenericPosix**
open() write()

open("/ext4/hi.txt")

**File Descriptor Table**

**EXT4** **F2FS** **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

<table>
<thead>
<tr>
<th>Features</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Auth</td>
<td></td>
</tr>
<tr>
<td>Compress</td>
<td></td>
</tr>
</tbody>
</table>

### Metadata Management

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LabFS</td>
<td></td>
</tr>
<tr>
<td>LabKVS</td>
<td></td>
</tr>
</tbody>
</table>

### Caching

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td></td>
</tr>
</tbody>
</table>

### I/O Scheduler

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Op</td>
<td></td>
</tr>
<tr>
<td>Blk Sw</td>
<td></td>
</tr>
</tbody>
</table>

**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**: The choice of modules have major performance impacts.
- **Customization**: I/O stacks should be more customized to the workload and environment.
- **I/O Expressiveness**: The best I/O interface should be chosen to store data.
- **Design Correctness**: 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

<table>
<thead>
<tr>
<th></th>
<th>Isolated</th>
<th>Colocated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NoOp</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux-NoOp</td>
<td>110 $\mu$s</td>
<td>945 $\mu$s</td>
</tr>
<tr>
<td>Lab-NoOp</td>
<td>89 $\mu$s</td>
<td>889 $\mu$s</td>
</tr>
<tr>
<td><strong>Blk</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux-Blk</td>
<td>120 $\mu$s</td>
<td>122 $\mu$s</td>
</tr>
<tr>
<td>Lab-Blk</td>
<td>95 $\mu$s</td>
<td>96 $\mu$s</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Isolated</th>
<th>Colocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux-NoOp</td>
<td>110 μs</td>
<td>945 μs</td>
</tr>
<tr>
<td>Lab-NoOp</td>
<td>89 μs</td>
<td>889 μs</td>
</tr>
<tr>
<td>Blk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux-Blk</td>
<td>120 μs</td>
<td>122 μs</td>
</tr>
<tr>
<td>Lab-Blk</td>
<td>95 μs</td>
<td>96 μs</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Isolated</th>
<th>Colocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOp</td>
<td>110 µs</td>
<td>945 µs</td>
</tr>
<tr>
<td>Linux-NoOp</td>
<td>89 µs</td>
<td>889 µs</td>
</tr>
<tr>
<td>Lab-NoOp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blk</td>
<td>120 µs</td>
<td>122 µs</td>
</tr>
<tr>
<td>Linux-Blk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lab-Blk</td>
<td>95 µs</td>
<td>96 µs</td>
</tr>
</tbody>
</table>
Developing and Customizing I/O Policies

- Both policies have pros and cons in different circumstances

<table>
<thead>
<tr>
<th></th>
<th>Isolated</th>
<th>Colocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOp</td>
<td>110 µs</td>
<td>945 µs</td>
</tr>
<tr>
<td></td>
<td>89 µs</td>
<td>889 µs</td>
</tr>
<tr>
<td>Lab-NoOp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blk</td>
<td>120 µs</td>
<td>122 µs</td>
</tr>
<tr>
<td></td>
<td>95 µs</td>
<td>96 µs</td>
</tr>
<tr>
<td>Lab-Blk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
THANK YOU