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, Chicago, USA, **Sandia National Laboratories, Albequerque, USA*
llogan@hawk.iit.edu, jcernudagarcia@hawk.iit.edu, gflofst@sandia.gov, sun@iit.edu, akougkas@iit.edu

Abstract— **Traditionally, I/O systems have been developed within the confines of a centralized OS kernel. This led to monolithic and rigid storage systems that are limited by low development speed, expressiveness, and performance. Various assumptions are imposed including reliance on the UNIX-file abstraction, the POSIX standard, and a narrow set of I/O policies. However, this monolithic design philosophy makes it difficult to develop and deploy new I/O approaches to satisfy the rapidly-evolving I/O requirements of modern scientific applications. To this end, we propose LabStor: a modular and extensible platform for developing high-performance, customized I/O stacks. Single-purpose I/O modules (e.g, I/O schedulers) can be developed in the comfort of userspace and released as plug-ins, while end-users can compose these modules to form workload- and hardware-specific I/O stacks. Evaluations show that by switching to a fully modular design, tailored I/O stacks can yield performance improvements of up to 60% in various applications.**

Index Terms—**Clouds and Distributed Computing, Programming Frameworks and System Software**

I. Introduction

HPC systems are increasingly suffering from a storage bandwidth bottleneck. Compute performance is increasing at a much faster rate than storage bandwidth. In this environment, new storage technologies are being deployed that offer much higher bandwidth with low latency, but the I/O stack through the Linux kernel imposes serious overheads. Approaches such as demonstrated in pMEMCPY [1] show it is possible to adapt to the native interfaces and dramatically increase bandwidth. Unfortunately, these are one-off solutions for each different technology. To truly achieve all available bandwidth, the design philosophy of the Linux I/O stack must be reconsidered.

Parallel filesystems (PFS), such as Lustre [2], and other distributed storage systems (e.g., key-value stores, (No)SQL databases) [3] rely on native Linux filesystems (e.g., ext4) as their backend on each storage server. This is because they provide a well-tested, secure, and familiar interface to a diverse set of storage hardware devices [4], [5]. However, this reliance imposes limitations [3], [6], [7] on all I/O operations. Specifically, Linux filesystems suffer from three main issues regarding I/O system development and performance: a) Severe performance degradation caused by workload-agnostic policies (e.g., I/O scheduling, caching); b) Significant software overhead for low-latency devices [4], [8]–[11] caused by frequent context switching and data copying [5], [12]; c) Limited expressiveness caused by the UNIX file and the strong POSIX-compliance (e.g., transactions are difficult to program using the UNIX file representation [3]) forcing other data representations to be translated into a file. Since PFS offer a higher-level view of the data on top of multiple local storage servers, the I/O stacks used locally on each storage node must be optimized to improve performance and customizability of the the distributed layer.

This work seeks to re-invision the way that I/O stacks are developed and deployed in order to provide per-workload and per-hardware optimizations. To achieve this, we have to overcome several obstacles. First, the methodologies used to implement I/O systems must be substantially improved to provide high-velocity development. Developing within the kernel should be avoided since it is limited in functionality, does not support upgrades without service interruptions, and is generally complex and time consuming [13]–[16]. Monolithic design patterns [17] should also be avoided since it slows development speed, testing is harder, and code-reuse is limited [18], [19]. All layers of the I/O stack, including I/O interfaces, must be modular and hence easily programmable [20]–[26]. Second, wide support for both traditional and modern storage hardware must be maintained. Development platforms which support only one type of hardware should be avoided since they increase assumptions, enforce dependencies, limit portability, and lower adoption rate [27], [28]. It is unrealistic to expect re-development of device drivers for all storage mediums current or future. Third, flexible and tunable access control mechanisms must be provided. Strict protocols enforced globally and holistically should be avoided since they have limited guarantees [29]–[31], are not always required [32]–[34], and impose significant performance costs [10], [35]. Fourth, high-performance must be ensured. Restricting device access behind long I/O paths should be avoided as they cause software overheads [36]–[38], limit expressiveness [26], [39], [40], and dismiss hardware-specific optimizations [41]–[43]. The ability to directly access hardware using its native APIs should be provided [32]–[34], [44], [45]. Fifth, end-users should find it easy to integrate newly-developed I/O stacks. Requiring the individual deployment of multiple, independently developed I/O systems through interception should be avoided, as this requires expertise to resolve conflicts in symbol resolution, hinders productivity, and invites erroneous outcomes [18], [46], [47]. Modular and programmable stacks hold promise to address all these challenges and have tremendous impact on both the developers and the end users of I/O systems.

To address the limitations of rigid, monolithic designs, we propose LabStor: a modular, extensible, and high-performance I/O platform that can be customized and extended to meet the needs of different environments. LabStor introduces the following contributions:

1) **LabMods**: single-purpose, self-contained code objects which can be developed in the comfort of userspace and released as plug-ins. LabMods can represent any functionality, such as I/O scheduling, caching, access control, or filesystems, and can provide any interface, including POSIX, key-value store, SQL, etc.
2) **LabStacks**: A combination of LabMods made by end-users to

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

form optimized, situation-specific I/O stacks. They are defined in a human-readable schema file.
3) **LabStor Runtime**: The system for deploying, querying, debugging, upgrading and executing LabStacks. It provides various utility scripts and a userspace process/thread scheduling framework for balancing LabStack work with application work.
4) **Implementation**: LabStor comes equipped with a variety of LabMods, including Driver LabMods, which enable out-of-the-box support for developing new I/O stacks for different hardware types.

Currently, no framework exists that provides high-velocity development, hardware diversity, and manageability, while also providing I/O interface diversity and performance. LabStor's I/O stacks and interfaces can be uniquely customized and rapidly developed to support a wide variety of workloads in both HPC and Cloud environments, all with the improved I/O performance and manageability required by modern systems. While this approach may appear to limit itself to a single node or an exclusive storage system, consider burst buffers or other fast storage tiers. In many cases, such as in ORNL's Summit machine and LLNL's upcoming El Capitan, node-local storage system extensions could take advantage of these designs to maximize overall storage performance. Gestalt machine designs [48] may use persistent memory as either extended DRAM or node-local storage on a job-by-job, node-by-node basis. These hardware deployments could take strong advantage of customized LabStacks for managing the node-local portion of the storage hierarchy. In that context, we explore LabStor as an approach to take best advantage of storage devices in a supercomputer in this tier. Through the use of a new filesystem and key-value store, LabStor showcases performance improvements of up to 60% under different workloads in different environments when compared with existing approaches. *LabStor is not a single I/O stack, rather it is a comprehensive platform that facilitates the rapid development and composition of new workload- and hardware-specific I/O stacks.*

II. Background and Related Work

In recent years, there has been a growing interest in exploring changes and improvements to the traditional UNIX I/O stack motivated by: a) hardware innovation in low-latency storage (e.g., NVMe [12] and Persistent Memory (PMEM) [49]) which demand leaner I/O stacks, and b) the increasingly diverse I/O requirements of modern workloads which necessitates high-velocity development and customization of the I/O stack.

A. *Shifting I/O from Kernel to Userspace*

Traditionally, storage devices are exposed to the users via the OS kernel, which acts as an authoritative source of trust. Applications interact indirectly with storage devices using I/O system calls which involves context switching between user processes and the kernel. However, this approach generates a rigid software stack [18], [19], [26], [39], [40] that has been shown to demonstrate significant overheads [4], [10], [12], [33], [37], [45]. To ease some of the burden from the kernel, alternative approaches offer shorter paths to the hardware.

Hardware APIs: The Storage Performance Development Kit (SPDK) [33] is an ©Intel open-source project that provides a set of tools and libraries for writing userspace storage applications for NVMe devices. SPDK leverages the NVMe specification to map the PCI Base Address Register (BAR) into user applications. Similarly, LightNVM [34] is a userspace driver for OpenChannel SSDs that enables the programming of the Flash Translation Layer (FTL) and submission of I/O requests to OpenChannel SSDs entirely in userspace. DAX [32] is an I/O mechanism that allows development of PMEM I/O systems in userspace. To get the lowest latency, DAX treats PMEM devices as byte-addressable memory avoiding normal filesystem block I/O conventions. By mapping the PMEM devices into the application address space, DAX allows direct interaction with PMEM via CPU load/store operations.

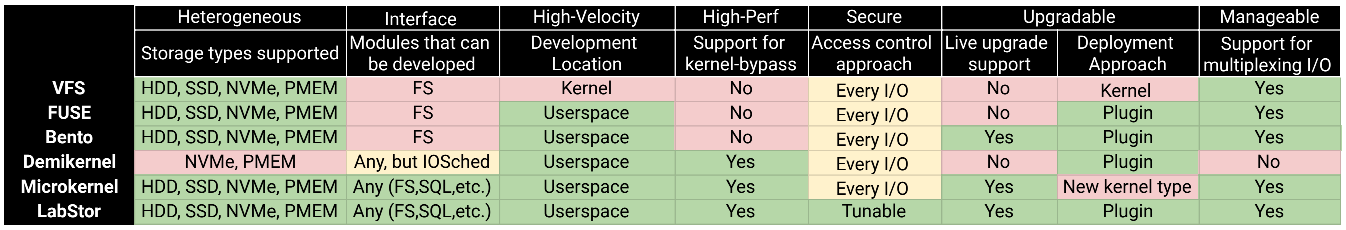
B. *Developing I/O Stacks in Userspace*

The Linux I/O stack is a constantly evolving entity. With every new update of the kernel new changes and optimizations get included [13]. Yet, many works have discussed the limitations of the Linux I/O stack and have proposed modifications to it. For example, many I/O schedulers have been proposed to better handle specific workloads [20]–[24]. Some examples include Blk-switch [20] and Bulk I/O Dispatch (BID) [25]. Typically, developing the Linux I/O stack requires direct kernel modification and recompilation. This approach limits the pace of development since custom enhancements or modifications are hard to install, manage, and share. To mitigate this, the kernel allows the development of kernel modules (KM), pieces of code that can be loaded and executed inside the kernel. There are several types of KMs that allow customization of the I/O stack such as I/O schedulers, page fault handlers, filesystems, and char/block device drivers. Filesystems are implemented through the *Virtual Filesystem (VFS) Layer* and are then inserted into the kernel. Yet, kernel programming is complex, difficult to debug and limited by its complexity [13], [14], [47]. New I/O system development methodologies have been proposed that leverage the Linux I/O stack while enabling its development and customization in userspace.

Filesystems in Userspace (FUSE): FUSE [47] enables users to implement filesystems in userspace through the use of *upcalls*. When a user executes an I/O system call, the request is passed to the FUSE VFS kernel module, which then forwards the I/O request to the userspace filesystem implementation. This approach enables high-velocity filesystem development, but incurs a significant performance sacrifice due to the addition of multiple context switches [13], [14]. Works such as Direct FUSE [16] and XFUSE [14] have proposed improvements to FUSE to reduce the number of context switches. However, none of these works allow for development of alternative representations to files, alternative interfaces to POSIX, and still inherit the overheads of the kernel I/O stack.

Injecting User Code into the Kernel: Bento [13] is a platform which enables filesystems to be developed in userspace and interpreted in the kernel. This enables the use of well-established kernel functionality such as permission checking and device drivers while also avoiding FUSE overheads. It also improves manageability (e.g., support for live upgrades). Bento makes use of a VFS kernel module that is capable of running pre-compiled Rust-based filesystem implementations submitted by the user and to which I/O operations are directed. Thus, the Bento framework drastically improves the velocity of filesystem development. However, Bento suffers the same limitations as FUSE, but with less context switching cost.

Microkernels: A microkernel, such as MINIX3 [50] or SeL4 [51], is a minimalistic kernel architecture where the majority of OS services are provided as libraries developed in userspace, including device drivers and I/O systems. This provides enhanced development speed when compared to monolithic kernels (e.g., Linux), as most programming is done outside the kernel. However, microkernels are not used in production computing centers. As of November 2021, all Top500 supercomputing centers run Linux [52]. This is because Linux has a massive community of developers and is easily installed



	Heterogeneous	Interface	High-Velocity	High-Perf	Secure	Upgradable		Manageable
	Storage types supported	Modules that can be developed	Development Location	Support for kernel-bypass	Access control approach	Live upgrade support	Deployment Approach	Support for multiplexing I/O
VFS	HDD, SSD, NVMe, PMEM	FS	Kernel	No	Every I/O	No	Kernel	Yes
FUSE	HDD, SSD, NVMe, PMEM	FS	Userspace	No	Every I/O	No	Plugin	Yes
Bento	HDD, SSD, NVMe, PMEM	FS	Userspace	No	Every I/O	Yes	Plugin	Yes
Demikernel	NVMe, PMEM	Any, but IOSched	Userspace	Yes	Every I/O	No	Plugin	No
Microkernel	HDD, SSD, NVMe, PMEM	Any (FS,SQL,etc.)	Userspace	Yes	Every I/O	Yes	New kernel type	Yes
LabStor	HDD, SSD, NVMe, PMEM	Any (FS,SQL,etc.)	Userspace	Yes	Tunable	Yes	Plugin	Yes

Fig. 1. Comparison of different I/O stack design philosophies

on any machine architecture. To utilize microkernels, HPC centers would have to uproot their infrastructure and manually determine the set of modules required for their systems to run, including drivers, which is both difficult and expensive. For these reasons, maintaining compatability with Linux is a must.

Semi-Microkernels: Various works, particularly in the networking domain [17], [53], have proposed combining the micro- and monolithic- kernel approaches, termed semi-microkernel (SMK). Applications communicate with the SMK using interprocess-communication. In these works, the SMK process runs in userspace, providing critical OS services entirely in userspace. Filesystems as Processes [54] discusses the performance and security requirements of developing filesystems while bypassing the kernel. uFS [15], a SMK filesystem dedicates CPU resources to handle filesystem threads to optimize latency on NVMes. To the best of our knowledge, the only work which applies this approach to I/O is uFS, which is a specific filesystem implementation, not a platform for developing and managing I/O systems. It does not discuss development of other layers of the I/O stack, developing alternative representations to files, or the deployment and upgrading of I/O systems.

Library I/O Stacks: Development of I/O stacks as loadable libraries directly linked into applications has also been presented. Demikernel [55], [55] splits I/O operations into a control path and a data path. Control operations are directed into a protected OS kernel. The data path is implemented by LibOS, an abstraction over the interface of the kernel-bypass device, and all I/O and network operations are handled by it. The same design pattern is followed by Nova [4] and SplitFS [10]. Another work, Simurgh [56], proposes a PMEM filesystem which implements all I/O functionality within a single library while protecting against buggy, but not necessarily malicious, code. Similarly to the SMK, these works primarily discuss the implementations of independent I/O systems. They are developed in isolation and can conflict when deployed simultaneously, cannot be upgraded while applications are running, are only used for storage where kernel-bypass accelerators exist, and require users to manage multiplexing concurrently-deployed library I/O systems.

C. Motivation

In summary, the following observations motivate the design and implementation of this work. First, there is a clear trade-off between I/O performance and storage features. Systems that execute I/O operations in the kernel enjoy the wide support for hardware and the provided secure enclave at the cost of performance and limited configurability. On the other hand, systems that execute I/O in userspace recover some performance and enhance customizability at the cost of limited hardware support, manual implementations of security semantics, and often troublesome deployments. Second, there has been a clear trend directed at the programmability of I/O systems. With works noting the difficulty of upgrading and debugging kernel code, leading to potential reduction of innovation and a clear increase in security issues and bugs from unknown and unpatched vulnerabilities. Finally, over the last couple decades, there is a growth in I/O interfaces deviating from the typical UNIX file abstraction (e.g., POSIX files) including key-value stores, NOSQL databases, log stores, etc. However, currently all these interfaces remain subservient to the file abstraction, requiring a translation to a file, and enforcing these systems to follow the underlying assumptions of the file interface (i.e., POSIX compliance). The translations and semantic compliance can lead to significant performance degradation.

To address these issues, LabStor presents a platform for developing I/O stacks in userspace, which provides high performance, configurability, and development velocity, while maintaining the hardware diversity, manageability and security that comes from kernel-level I/O stacks. At the same time, LabStor leverages its configurability to provide users with the ability to untether themselves from the file interfaces by implementing their own memory-to-disk data mappings. Figure 1 showcases a comparison of related work and their design goals.

III. The LabStor Platform

LabStor is an I/O platform designed to provide developers with the ability to implement highly performant, custom, and modular I/O stacks entirely in userspace. With LabStor, users can create and customize all I/O policies (e.g., I/O scheduling), utilize a diverse set of storage devices (including modern hardware such as PMEM [32] or NVMe [33], [34]), push updates and new features in the I/O stack without significant service interruption, and maintain security semantics. Most importantly, all this functionality comes without the cumbersome, often difficult, kernel programming since LabStor brings the I/O stack entirely in usespace. In addition, LabStor liberates users from the restrictive POSIX file interface, allowing alternative representations to be mapped directly to the hardware by enabling direct access to device drivers from userspace. LabStor pushes towards full modularization of the stack by introducing the following: a) the LabStor Module (LabMod): an object that contains code implementing well-defined functionality (e.g., device drivers, block I/O layer, page caches, I/O schedulers, and filesystems). b) the LabStor Stack (LabStack): a set of rules governing a composition of multiple LabMods that implement an entire I/O stack. c) the LabStor Runtime: the management and communication infrastructure to execute and deploy LabStacks.

A. Towards Fully Modular I/O Stacks

Traditional I/O stacks used in HPC have been historically developed within the kernel, which provides a well-tested, well-documented, secure and uniform interface to a wide variety of storage hardware. However, this approach followed monolithic designs that led to fragmentation in the storage space in terms of innovation, features, performance, and semantic richness. Further, monolithic designs suffer from a lack of policy decisions (e.g., I/O schedulers), limited expressiveness, minimal code re-use, and cumbersome development pipelines. Such problems are exacerbated in distributed I/O stacks where functionality and/or semantics are often duplicated. For instance, a PFS may have implemented access control while the underlying storage stack of each participating node already incorporates such semantics [3]. To alleviate some of these limitations, recent efforts [13] have demonstrated the benefit of developing I/O systems outside the boundaries of the kernel. However, these approaches impose assumptions such as the UNIX file abstraction and the strict POSIX standard, limiting expressiveness, policy decisions, and configurability. More effort is required to make it easier for developers to deliver new features fast, reliably, and portably.

LabStor, as a development platform, promotes the Single Responsibility Principle where code reusability, scalability, and speedy development is achieved by loosely coupled components, organized in smaller code bases, allowing easier debugging, faster deployments, and interoperability between I/O subsystems. To boost innovation, LabStor introduces the concept of the *LabMod*: an independent, self-contained code object implementing a well-defined, distinct, single-purpose functionality. LabMods can be independently developed, maintained, and released as plug-ins. LabMods can be interchanged, stacked, and incrementally-upgraded.

Fundamentally, LabMods are comprised of four elements: type, operation, state, and connector. The type is the set of APIs the LabMod implements. The operation takes a well-defined input, processes the input, and produces a well-defined output. State is internal data stored within the LabMod required for the operation's success. Lastly, the connector exposes the LabMod operation to applications. Connectors and operators can live in two separate address spaces, in the same address space, or even in kernel space. For example, a client may call a POSIX LabMod and use its connector to pass a POSIX I/O request as input to a filesystem operation. The operation then converts the I/O request to one or more block I/O requests by allocating disk blocks using the allocator stored in the filesystem's state. The filesystem then outputs the block I/O requests to the next LabMod in a stack (e.g., I/O scheduler, page cache, compression).

LabMods provide developers full creative freedom over their functionality and interface. For example, LabMods can be built for filesystems, (No)SQL databases, I/O schedulers, compression, drivers, etc. However, to enable the LabMod properties of upgradeability, stackability, and interchangeability, developers must implement specific APIs. For instance, to enable live upgrades, LabMods must implement the `StateUpdate` API, which copies state from the old LabMod into the new. As another example, to capture performance counters of LabMods, developers must implement the `EstProcessingTime` and `EstTotalTime` APIs. While high development speed is important, providing developers with the ability to test, probe and debug their code in an easy manner is just as important. LabStor provides a debugging mode that allows LabMods to be run in isolation and supports existing tools such as GDB or Valgrind to fully test their individual LabMods before deploying them in production. Lastly, LabStor comes equipped with various LabMods for interacting with storage and managing the complexities of multiplexing I/O systems.
Driver LabMods: To minimize kernel programming while maintaining compatibility with a wide variety of storage, LabStor provides an enhanced, extensible alternative to Linux's VFS and block layer for interacting with storage devices entirely in userspace. Unlike the Linux block layer, which is limited to the block device abstraction [20], LabStor provides users and developers a comprehensive menu of storage hardware APIs through the use of Driver LabMods. This includes LabMods which expose the Linux kernel's multi-queue device driver hardware queues directly to developers, bypassing several layers of the kernel stack, in addition to various LabMods supporting userspace I/O mechanisms (e.g., SPDK, DAX), which may provide APIs other than block (e.g., zoned namespace and queues). From these fundamental LabMods, all other layers of an I/O stack (e.g., scheduling, caching, etc.,) now become programmable in userspace. Additionally, as LabStor runs primarily in userspace, LabMods can utilize well-established libraries, including data structure libraries (e.g., Boost) and machine learning libraries (e.g. mlpack [57]). This allows developers to make use of an extensive pre-exisitng code base that is already well tested and documented. For example, time series analysis can be used to predict characteristics of future I/O requests to reduce seek penalties on HDDs or decide which pages to evict from the page cache.
Management LabMods: One design goal of LabStor is to provide an expressive and manageable I/O platform. Users should be able to express their I/O requirements through the API that best fits their design (i.e., file, key-value, SQL). This is achieved through the concept of a Generic LabMod which exposes an I/O interface (e.g., Generic Filesystem receives POSIX calls). Generic LabMods are in charge of creating I/O requests and forwarding them to the appropriate I/O system that implements these calls. Generic LabMods are loaded into clients using `LD_PRELOAD`, enabling seamless support for legacy applications and the development of new I/O interfaces. To multiplex I/O systems without user intervention or code changes, Generic LabMods manage state that is common among I/O systems of a particular type, similar to the VFS. LabStor currently provides two Generic LabMods: a) *GenericFS*, which manages the allocation of file descriptors and the routing of I/O requests to the proper filesystem (FS) implementation, and b) *GenericKVS*, which only does the latter for key-value stores (KVS).

B. Towards Composable I/O Services

Modern scientific computing consists of a wide variety of workloads such as simulation, data analytics, and machine learning applications, with many distinct and often conflicting I/O requirements [58]. It is unrealistic to expect a single static I/O stack to efficiently satisfy this variety of I/O requirements, including semantics and targeted performance characteristics. A software defined storage approach brings hope to a new era of performant I/O systems tailored to match the I/O demands of large-scale scientific applications. To this end, LabStor introduces *LabStacks*: a user-defined combination of compatible LabMods into a single I/O system. LabStor provides common capabilities such as device drivers, I/O schedulers, caching, and filesystems as building blocks in the form of LabMods, while introducing new and exotic ideas, such as configurable consistency or ML-driven cache eviction algorithms. Using LabStacks, users can construct customized, workload-specific, and hardware-optimized I/O services. For example, an application which generates large amounts of compressible data (e.g., VPIC [59]) to an NVMe could benefit from a LabStack which contains a compression LabMod and SPDK Driver LabMod.

LabStacks are defined in a specification file which includes the following attributes: a) a mount point, which is a human-readable path (e.g., /home/user); b) a set of governing rules, such as priority hints and execution method; and c) a DAG of LabMods, where each vertex contains the LabMod name, LabMod UUID, attributes for initialization, and a set of other outputs (e.g. other LabMods), represented as a list of LabMod UUIDs and LabStack mount points. The LabMod UUID is a human-readable name which represents a unique instance of a LabMod. The execution method determines whether the Labstack DAG is executed synchronously (directly in the client application) or asynchronously (divided among threads, allowing stages to be pipelined).

LabStacks can be initialized on a mount point like any other I/O stack. To do so, users register their LabStacks within a LabStack

Namespace by simply passing the specification file to the (overloaded) `mount` command. First, `mount` will ensure that all LabMods in the DAG are instantiated by populating a Module Registry, a key-value store where keys are LabMod UUIDs and values are the LabMod instances. A LabMod is only instantiated if its UUID did not exist in the registry. Next, the set of rules are parsed to properly configure the runtime of the LabStor platform. Lastly, a validation step is performed to verify that the LabMods are compatible and then the LabStack is inducted into the LabStack Namespace. LabStack DAGs can be modified dynamically after mounting via the `modify.stack` command, which enables the insertion/removal of vertices. LabMod instances in the Module Registry can also be hot swapped live via `modify.mods`. For example, swapping one LabMod I/O scheduler for another.

LabStacks bring many benefits into the complex space of data management, as shown in the below examples. Any I/O stack is a path (or a projection) to the contents kept in storage. As a result, one can generate multiple views of the same data by deploying several LabStacks on top of the same device. This leads to a plethora of useful features.

Tunable Access Control: I/O stacks are responsible for providing data access control mechanisms. Traditionally, this has been achieved with user groups and globally-applied permission checks. Modern research has shown the benefits of tunable access control where data is exposed based on varying degrees of sensitivity. However, this is difficult to implement in a monolithic design and is typically offered as a non-negotiable feature. With LabStor's modular approach, one can deploy multiple LabStacks over the same content, each with varying degrees of access. Permission LabMods inside the stack can implement islands of data that are viewable by different actors. Not only is this easily implemented, but it can also dynamically change if the operator of the I/O service chooses so.

Interface Convergence: A growing theme in recent years is the hyper-convergence between different I/O interfaces [60], [61]. Modern workloads demand the support for multiple data access interfaces which puts pressure to system developers to widen the support for diverse APIs. This has led to two distinct approaches. Either the user must adapt their code to the provided interface or translation middleware layers are employed, elongating the data path and imposing additional software overheads. In LabStor, data can be accessed by seamlessly multiplexing LabStacks. For example one with POSIX interface and another with get/put/delete over the same content.

Dynamic Semantics Imposition: Application I/O patterns can be highly variable [62]–[65], requiring live changes to the I/O stack to maximize performance. However, this requirement is limited by the rigidity of traditional I/O stacks. LabStacks enable the ability to dynamically impose new semantics into the I/O stack by allowing LabMods to be inserted/removed while applications are still running. For instance, a compression LabMod can be added to a LabStack for a period of time, or an I/O scheduler can be swapped for another depending on the current I/O traffic.

Active Storage: Offloading data-intensive tasks to storage has been shown to benefit the end user both in terms of performance and ease-of-use [59], [64]. For example, instead of applications manually employing compression before persisting data, a LabStack could contain a Compression LabMod which transparently compresses the data before writing to storage. Compression happens asynchronously and in batches which further boosts performance.

Decentralized I/O System Designs: Various works [4], [10], [35] have demonstrated the benefits of eliminating centralized authorities from the I/O path for latency-sensitive I/O requests. One approach is to decouple metadata and data operations, enabling security for metadata and increased performance for data operations. In LabStor this can be done by using two separate LabStacks: one for metadata that asynchronously executes in a separate runtime, and another for data that synchronously executes at the client using Driver LabMods such as SPDK. State required for the data operations (e.g., block allocations) can be stored in shared memory between the two LabStacks. Fully decentralized designs execute both metadata and data LabStacks client-side, improving latency (but at a cost to security).

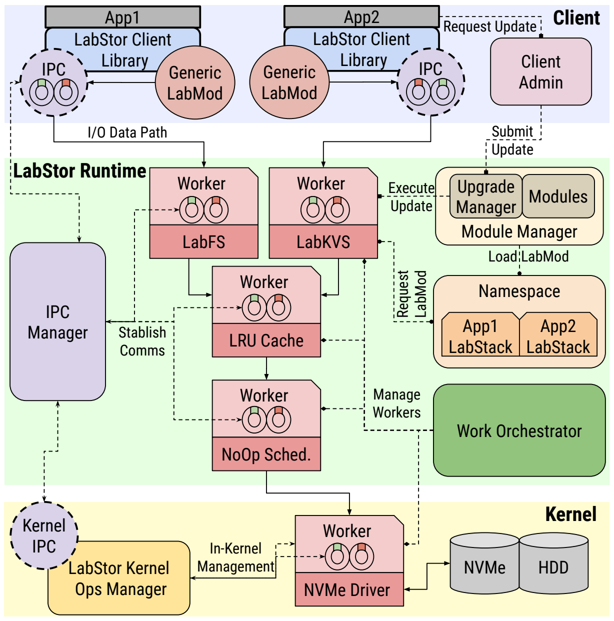


Fig. 2. Component Diagram

C. A Powerful Alternative to the Linux Stack

The LabStor Runtime is the primary warehouse and execution engine of LabStacks. It runs in a separate address space from the clients in order to provide security and high-performance. The Runtime is also responsible for the loading, storage, execution and upgrading of LabStacks and LabMods. Figure 2 demonstrates the architecture of the Runtime which consists of the following components:

Kernel Operations (KO) Manager: to achieve extensive hardware support and manage all in-kernel operations, the KO Manager is implemented as a Linux kernel module which manages the deployment of Driver LabMods and the communication with them.

Inter-Process Communication (IPC) Manager: to communicate between the Client Library, Runtime and LabMods, the IPC Manager provides a highly-performant and secure infrastructure based on shared memory and a queuing system. The LabStor client initially connects to the LabStor Runtime through a UNIX domain socket, providing process credentials to the LabStor Runtime, which can be used for authentication. Similarly, the IPC Manager connects to the Kernel Operations Manager through a netlink socket.

Module Manager: to enable live upgrades, deployment, and querying of LabMods, the Module Manager maintains a Module Registry (a hashmap in shared memory) that holds all instantiated LabMods and their entrypoints. It also holds a queue, where requests for module upgrades are stored. The Module Manager implements several protocols including centralized and decentralized upgrade which are executed when invoked by the admin thread to process upgrades.

LabStack Namespace: to create, hold and multiplex LabStacks, the Namespace maintains a semantic shared-memory key-value store that holds LabStack as DAGs.

Workers: to provide security and reduce overhead caused by context switching, clients can send requests to workers which execute in a separate address space from the client. Workers receive requests by polling request queues and process the requests by querying the LabStack Namespace and Module Manager for the required LabMods. Workers execute either in the Runtime (using pthreads), in kernel space (using kthreads), or directly in the client thread. Workers also periodically monitor LabMods to get get performance metrics, useful to work orchestration policies.

Work Orchestrator: to ensure high performance and resource utilization, the Work Orchestrator manages the assignment of request queues to workers and the assignment of workers and application processes to CPU cores. In order to manage workers executing in-kernel, the orchestrator communicates with the LabStor Kernel Module to spawn, freeze, and terminate kernel threads.

1) Efficient and Secure Inter-LabMod Communication: To enable zero-copy, high-performance communication between LabMods, LabStor relies on shared memory. However, one consequence of this design is that faulty and malicious processes have more potential to cause harm to other running processes due to the nature of shared memory. Care must be taken into providing privilege to only the processes that need visibility to the shared memory. In addition, memory needs to be shared between kernel and user space, which requires specialized memory sharing techniques that are not provided by higher-level shared-memory libraries, such as Boost.

To provide these shared memory capabilities, a novel shared memory LabMod (ShMemMod) is provided by LabStor, which uses `vmalloc` for allocating regions of shared memory and `remap_pfn_range` for mapping the allocated regions into a user's address space. The memory can only be mapped by specific processes which have been granted access to the memory by the LabStor Runtime, enabling both high-performance and security, even among processes launched by the same user.

Another communication primitive used by the IPC manager are Queue Pairs (QPs). They can be stored in shared-memory or private memory. QPs have a number of properties. `Primary queues` are the queues where clients initiate requests. `Intermediate queues` hold requests spawned as a result of another request. Typically, primary queues are stored in shared memory and intermediate queues are stored in private memory. In addition to this, queues can be marked as ordered or unordered. Ordered queues must be processed in sequence on a single worker. Unordered queues can be processed by multiple workers.

2) Live Upgradable and Hot-Pluggable LabMods: To support the ability to upgrade and hot-plug LabMods without service interruption, the Module Manager implements two live upgrade protocols: centralized and decentralized. This is because LabMod operators can exist either in the Runtime's address space or in the client's address space. Due to space, we only detail the centralized case. The centralized protocol updates the Runtime, whereas the decentralized updates all running clients. In both cases, an upgrade request is submitted to the Module Manager using the `modify.mods`. The upgrade request contains the LabMod name to upgrade, a path to the updated code, and the upgrade type.

Centralized Upgrade: An upgrade request is initially placed in the upgrade queue in the Module Manager. The `Runtime Admin` will periodically (every t ms, configurable by the user) poll for updates. If there are any, the Module Manager will mark all of the primary queues as `UPDATE_PENDING`. Eventually, workers processing primary queues will acknowledge the `UPDATE_PENDING` state by setting the flag `UPDATE_ACKED`. Intermediate queues will continue processing until all events have been completed. When all primary queues have been paused and all intermediate requests have completed, the Module Manager will process all upgrades. For each upgrade, it will traverse every LabMod in the Module Registry and call the `StateUpdate` API for each LabMod of this type, which transfers the state from the old LabMod to the new. `StateUpdate` can be modified by developers to tailor the upgrade process to their needs. After all upgrades have been processed, the Module Manager will unmark all the primary queues and requests will begin flowing through the system again.

3) Crash Recovery: To provide security and resource management capabilities, LabStacks have the option to execute within the the LabStor Runtime. However, as LabStacks can contain untrusted and potentially buggy code, there is potential for the Runtime to crash. Without proper management, crashes such as this can cause severe setbacks, as potentially long-running applications would have to be restarted. To prevent this, the LabStor Runtime can be restarted and repaired while applications continue to run. To communicate with the Runtime, clients place requests in request queues and poll for completion using `Wait`. If the LabStor Runtime crashes, `Wait` will eventually detect that the Runtime is offline and wait for it to be restarted by the administrator (for a configurable period of time). If restarted, the LabStor client library in each process will iterate over the LabStack Namespace, invoke the `StateRepair` API in each LabMod, and then continue.

4) Load Balancing & Scheduling: The Work Orchestrator (WO) is a userspace process and thread scheduling framework, similar to the concept of FlexSC [66]. FlexSC proposes a specific policy to dedicate CPU cores to handle latency-sensitive system call work to avoid context switching, whereas LabStor provides a general framework to make different policies. LabStor WOs typically should make two considerations. First, LabStacks can generate both latency-sensitive and compute-intensive requests, requiring intelligent division of labor. In addition the I/O traffic on systems can be dynamic (e.g., checkpoint-restart workloads), requiring intelligent scaling of CPU resources to provide efficient resource utilization. To manage both of these challenges, the WO defines a `rebalance` operation, which takes as input n queues and m workers. This operation is called in two cases: first, when a new client connects; and second, every t ms, configurable by the user. To decommission a worker, the WO reassigns all request queues from the worker. Workers can be oversubscribed to a core by placing request queues on workers pinned to the same core. When a worker hasn't processed a request for a period of time (in μs, also configurable), the worker will yield to avoid busy waiting for an entire WO epoch. Load balancing queues has a long and deep history of research dating back more than 30 years, and many algorithms have been proposed (e.g., Markov Chains [67] and supermarket model [68]) for solving the `rebalance` problem. For this reason, the WO is also modular.

LabStor currently provides a *dynamic work orchestration policy*, which aims to minimize the number of workers and context switches, while keeping performance loss under a configurable threshold and optimizing for latency-sensitive requests. First, rebalance divides the queues into two groups based on the maximum expected processing time of the request (`EstProcessingTime` API in the LabMod) and the number of queued requests: latency-sensitive queues (LQs) and computational queues (CQs). The queues are then partitioned among workers, where LQs are placed on a subset of workers, and CQs on another subset. With this partitioning, a modified Knapsack problem is solved where multiple sacks (the workers) aim to fit all items (the queues) such that each sack has equal weight (total estimated processing time of the queue). The partitioning with the fewest number of workers within the threshold is selected. Workers processing latency-sensitive requests are dedicated to CPU cores in order to avoid interference caused by context switching.

D. Deployment Model

Installing LabStor: LabStor can be downloaded from github (https://github.com/scs-lab/labstor) and compiled/installed using CMake. After compilation, a kernel module (the LabStor Kernel Ops Manager), the LabStor Runtime, various utility commands (mount.stack, modify.stack, etc.), and a series of LabMods (libraries) are presented to the user. The kernel module is inserted into the kernel using `insmod`. After this, the Runtime needs to be started. Trusted users can modify the Runtime configuration YAML, which contains information such as LabMod locations and work orchestration policies.
Installing LabMods: Once developed, users can install LabMods into LabMod repos, which are directories searched by the Runtime to determine the set of available LabMods. Repos can be defined in the LabStor configuration file and adjusted during runtime using the `mount.repo` and `unmount.repo` commands. These commands are unprivileged and can be executed by any user. A configurable maximum number of repos per-user is defined in the LabStor configuration file. A LabMod repo which is owned by the same user as the LabStor Runtime is considered trustworthy by default. Untrusted LabMods (which could potentially contain buggy or malicious code) can still be employed and debugged; however, the execution of these LabMods must be in a separate address space from the Runtime in order to ensure security.
Mounting LabStacks: After installing the LabMod repos, users can define LabStacks. LabStacks are defined as YAML files, which contain a DAG of LabMods and a set of governing rules, including a set of users which have authority to modify the LabStack. To mount the LabStack, users can call `mount.stack`. While access control is provided primarily by the LabMod implementation, LabStor provides mechanisms to prevent untrusted users from unduly modifying or executing other LabStacks. The maximum length of a LabStack is configurable.
Modifying LabStacks and Upgrading LabMods: After mounting, a LabStack can be modified dynamically by uploading an updated YAML file of the LabStack, where the differences between the original and updated LabStacks will be applied. Code upgrades for LabMods can be speicified by adding an upgrade request to the desired LabMod in the LabMod DAG. LabStack modifications and upgrades can be rejected for unprivileged users during mounting.
Application-Side: In order for applications to interact with LabStacks, LabStor provides a client library which is used to communicate with the Runtime to mount, modify, query, and execute LabStacks.

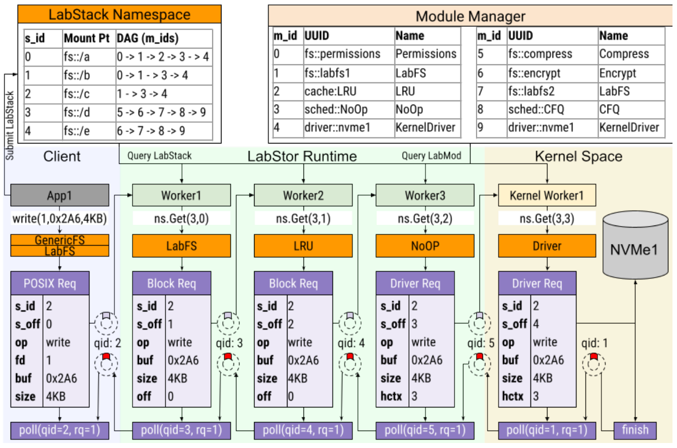
E. Use Cases

The current release of LabStor provides example LabMod implementations of both a filesystem and a key value store, respectively called LabFS and LabKVS. Developers can implement the same APIs to create their own.

An Example POSIX Filesystem: Many legacy applications depend on POSIX. For compatability, LabStor provides LabFS, a POSIX-compliant filesystem. LabFS is a log-structured, crash-consistent filesystem which provides specific optimizations for NVMe and PMEM and provenance tracking. It uses a scalable per-worker block allocator, which evenly divides device blocks among the pool of workers, initially defined in the Runtime configuration. Workers can steal from one another if more space is needed. If the number of workers decreases, free blocks of the decommissioned workers are assigned to running workers. If new workers are added, they will steal a (configruable) number of blocks from the other workers. LabFS uses a per-worker log for tracking metadata operations (file creations, etc.). As opposed to storing inodes and bitmaps on-disk as traditional FSes do, LabFS only stores the log and reconstructs inodes in-memory by traversing the log. All inodes are stored in a hashmap to maximize scalability. In addition, LabFS can be configured to bypass the Runtime for metadata and data operations to further reduce latency.
An Example Key-Value Store: LabKVS is similarly designed to LabFS; however, LabKVS implements a put/get/remove API, which creates keys and stores data using a single syscall, as opposed to the three (open-modify-close) required by POSIX.
A real deployment use case: a custom file system stack: Figure 3 showcases a real use case of LabStor. In this example, a series of asynchronous LabStacks implementing full file system I/O stack are deployed on a machine. The example makes use of the GenericFS and the LabFS labMods, but the same logic could apply for other file system implementations.

Initially, various LabStacks, including "fs::/b", have been mounted in the LabStack Namespace (shown in Figure 3). A client app is then launched with the GenericFS LabMod to intercept POSIX calls. The GenericFS LabMod can be LD_PRELOADed. The client first connects to the Runtime using the LabStor Client Library and IPC Manager. Eventually, the app calls `open()` to create the file "fs::/b/hi.txt", which is intercepted by GenericFS. GenericFS determines the LabStack corresponding to this path by querying the LabStack Namespace. First, it checks if "fs::/b/hi.txt" is in the Namespace. Since it is not, it checks if the parent directory ("fs::/b") is. Since it is (s_id: 2), GenericFS loads the LabStack DAG and allocates a file descriptor (fd). GenericFS then queries the Module Registry for the first LabMod in the DAG (LabFS) and calls its connector. The connector constructs a request containing the fd, path remainder ("/hi.txt"), LabMod ID (1), and LabStack ID (2); queries the IPC Manager for a queue pair (QP, qid: 1); places the request in the submission queue (SQ); and then synchronously waits for the request to complete by polling the completion queue (CQ). Eventually, Worker 1, which had been assigned this queue by the Work Orchestrator, polls the I/O request from the SQ. The worker loads LabFS from the Module Registry and passes the I/O request and QP to its operator. The LabFS operator processes the `open`, which creates an inode and then places a completion request in the CQ. The client receives this completion request, internally stores the mapping between fd 1 and LabStack 2 in a table, and then returns the fd back to the app.

After the open, the client writes 4KB of data to fd 1. GenericFS intercepts the write, checks its internal fd table, and determines that LabStack 2 should execute the command. The LabStack is loaded from the Namespace and the first LabMod in the DAG is loaded from the Module Registry. The LabMod's connector is called, which constructs a POSIX I/O request and places it in a queue pair (QP). Within the Runtime, Worker 1 polls this request, loads LabFS from


LabStack Namespace

s_id	Mount Pt	DAG (m_ids)
0	fs::/a	0 -> 1 -> 2 -> 3 - > 4
1	fs::/b	0 -> 1 - > 3 -> 4
2	fs::/c	1 - > 3 -> 4
3	fs::/d	5 -> 6 -> 7 -> 8 -> 9
4	fs::/e	6 -> 7 -> 8 -> 9

Module Manager

m_id	UUID	Name	m_id	UUID	Name
0	fs::permissions	Permissions	5	fs::compress	Compress
1	fs::labfs1	LabFS	6	fs::encrypt	Encrypt
2	cache:LRU	LRU	7	fs::labfs2	LabFS
3	sched::NoOp	NoOp	8	sched::CFQ	CFQ
4	driver::nvme1	KernelDriver	9	driver::nvme1	KernelDriver

POSIX Req

s_id	2
s_off	0
op	write
fd	1
buf	0x2A6
size	4KB

Block Req

s_id	2
s_off	1
op	write
buf	0x2A6
size	4KB
off	0

Block Req

s_id	2
s_off	2
op	write
buf	0x2A6
size	4KB
off	0

Driver Req

s_id	2
s_off	3
op	write
buf	0x2A6
size	4KB
hctx	3

Driver Req

s_id	2
s_off	4
op	write
buf	0x2A6
size	4KB
hctx	3

Fig. 3. LabStack Example

the Module Registry, and passes the I/O request to its operator. The operator then performs block allocation and forwards a block I/O request to the next stage in the DAG. This is done by quering the LabStack Namespace for the LabStack DAG; determining the next set of LabMods to forward the request to (only the LRU LabMod in this case); constructing Block I/O requests destined for the next LabMod; loading a QP from the IPC Manager; and then placing the requests in the QP. The LabFS operator will asynchronously poll for the completion of the LRU request, allowing Worker 1 to continue processing other requests, without stalling for the LRU event to complete. This pattern of asynchronous message passing and polling continues until all events have completed.

F. Implementation Details

LabStor is implemented in 18K lines of C/C++ code and has been tested on kernel 5.4. LabStor uses pthreads to create and pin threads. LabStor comes equipped with a full I/O stack, including LabMods for page caching (LRU), tunable consistency guarantees, permissions checking, compression, I/O scheduling policies, and I/O systems (LabFS & LabKVS).

Security: Currently, our prototype enforces security by allowing only authorized root users to modify LabMods and LabStacks, similar to how mount commands are managed in Linux.

Fork/Clone/Execve: To maintain compliance with POSIX, GenericFS intercepts the `clone` and `execve` system calls. Both intialize new adress spaces and require open file descriptors (`fd`) to be shared after the creation of a new address space. When called, the IPC Manager will disconnect from the LabStor Runtime, re-connect to establish new shared-memory queue pairs, and then send a message to the LabStor Runtime requesting that all file descriptors from the parent process be copied to the newly created process. `fork` is implemented on top of `clone`. For `execve`, open `fd` state is copied to the LabStor Runtime and is reloaded upon completion.

Kernel Driver LabMod: Most of Linux's storage drivers (e.g., HDD, SATA SSD, NVMe) follow the Multi-Queue (MQ) I/O path. LabStor exposes these drivers through three APIs: `submit_io_to_hctx`, which submits an I/O request directly to a hardware dispatch queue, `submit_io_blk`, which submits an I/O request using the standard Linux block layer, and `poll_completions`, which polls for I/O completion events on poll-based storage devices (e.g., NVMe). For MQ device drivers, `blk_mq_alloc_request_hctx` and `blk_mq_try_issue_directly` are used to place I/O requests directly in the hardware dispatch queues (hctx) of the driver. These functions are maintained by the Linux community, and have been present since kernel 4.8 in 2016 [69]. For non-MQ device drivers (e.g., PMEM drivers), we use the `submit_bio` function.

Kernel Modifications: LabStor currently re-implements the function `blk_mq_try_issue_directly` (amounting to 215 LOC) to avoid kernel recompilation, as this function is not public. This could be fixed by either making this function public or extending the `bio` structure to designate the hctx where a request should be placed. Either method would require minimal code change (roughly 5 LOC).

G. Discussion and Considerations

LabStor limitations: LabStor filesystems cannot be used for booting the main OS; only kernel-level filesystems can be used for this. The current prototype of LabStor does not provide support for networking, although it can be extended to do so. Labstor is intended to avoid compiling a custom kernel, although our code has only been tested on kernel 5.4, and may result in compilation issues on other kernels. LabStor is also designed with the assumption that the machine is many-core (>2 cores) and has a 64-bit address space.

Re-implementation Overhead: Within LabStor, developers can choose to re-architect the entire I/O stack to achieve much higher performance than the traditional kernel stack. However, for situations where it is more desireable to rely on the already-tested policies provided by the kernel, LabMods built on top of kernel APIs such as I/O uring can be used to inherit some of the kernel's functionality.

Applicability outside HPC: In this paper, we explore the benefits of LabStor in the context HPC. However, our approach also has impacts to multi-tenant clouds and virtual machines. One future work could be to apply LabStor as an I/O hypervisor to avoid repetition of the I/O path in a VM while also maintaining isolation guarantees. Additionally, within LabStor, new I/O scheduling and work orchestration policies can be implemented, which factor the I/O behavior of running applications, service-level objectives, and core oversubscription in their scheduling decisions.

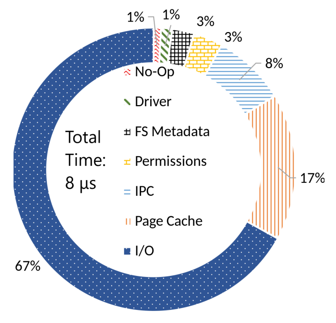
IV. Evaluations

Testbed: We ran all of our experiments in Chameleon [70] using the storage hierarchy appliance, equipped with NVMe (Intel P3700; 2TB), SSD (Intel SSDSC2BX01; 1.6TB) HDD (Seagate ST600MP0005; 600GB), and 512 GiB of RAM. It contains 2xIntel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz for a total of 24 cores and 48 threads.

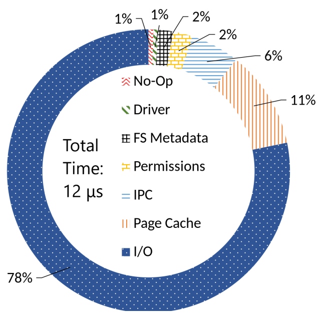
Software: We use Ubuntu 20.04 with kernel 5.4. To emulate PMEM, we modify the OS bootloader [43]. For synthetic benchmarks we use FIO 3.28 and FxMark [71]. For non-synthetic evaluations, we use LABIOS [60] as an object store and FileBench-1.4.9.1 [72] for generating I/O traffic. We present the average of 5 runs. For the following evaluations, we define the following LabStacks: a) *Lab-All*: permissions checks, LRU cache, NoOp sched, Kernel_Driver, async_exec_mode. b) *Lab-Min*: removes permissions. c) *Lab-D*: remove permissions, sync_exec_mode.

A. Internal Evaluations

I/O Stack Anatomy: To understand the impact software has on I/O operations and quantify the overheads imposed by the filesystem implementation, we run a test where we read/write 4KB of data from/to an NVMe drive using LabFS. We capture the amount of time spent in different LabMods on the data path. A LabStack, resembling that of a traditional I/O stack, is configured to use LabFS, permissions checking, No-Op I/O scheduling, LRU Page Cache, and the Kernel Driver LabMod. The LabStor Runtime uses a single worker to process



(a) 4KB write NVMe



(b) 4KB read NVMe

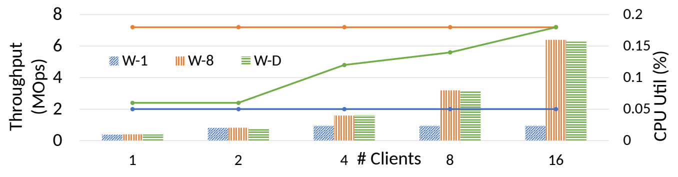
Fig. 4. I/O stack anatomy

the request. Overall, it can be seen from Figure 4(a) I/O takes the most time as expected. Software amounts to 34% of the overall time. The page cache takes 17% of time due to data copying. Next, inter-process communication using shared memory takes 8.4% of time. Since the Runtime is on a separate core, the request needs to be fetched from another core's cache or directly from DRAM, both of which are expensive. Filesystem metadata management and permissions checking each take 3% of time due to memory allocations and data structure manipulations (block allocation, logging block in inodes, querying the inode hashmap). The No-Op I/O scheduler only amounts to about .5% of I/O time, as it only keys a request to a hardware queue. Lastly, the driver amounts to approximately 1% of I/O time. The results are similar for reads. Overall, it is apparent that for each feature added to the I/O path, a performance penalty is paid. While certain penalties (e.g., caching) can be bypassed in a typical I/O stack, certain costs are applied uniformly to all I/O operations and are non-negotiable. Maintaining security semantics, for example, requires overheads of 3% for permissions checking and 8.4% for ensuring applications cannot directly modify metadata, protecting it behind costly IPCs. Providing more flexibility on the I/O path can result in significant performance gains.

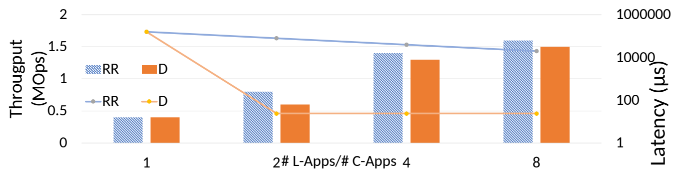
TABLE I
LabMod live upgrade performance in seconds

#Upgrades	0	256	512	1024
Centralized	29.08	30.21	32.536	34.338
Decentralized	29.08	30.51	33.56	35.81

Live Upgrade: In this evaluation, we measure the amount of service interruption caused by updating a component in the LabStack during runtime. We run an application which messages a dummy module 100,000 times using a single thread. Roughly 20 seconds after the app is launched, the dummy module is upgraded. We measure the total running time of the app with a centralized upgrade and with a decentralized upgrade III-C2. We vary the number of upgrade requests and report the application's running time in seconds. From Table I, it can be observed that the cost of a live upgrade was roughly 5ms which does not noticeably impact the application's running time until thousands of upgrade requests are queued (an addition of 5 seconds after 1024 upgrades). There are three main factors which impact the performance of an upgrade: First, the storage medium the upgrade is located on; Second, the cost of loading the update into memory; Third, the cost of transferring state. In this case, the dummy module is 1MB and located on an NVMe. The I/O cost accounted for the majority of time spent in the upgrade process. The state needed to be transferred was simply a few bytes of pointers. Overall, this is a substantial improvement over a reboot per update, which in Chameleon Cloud,



(a) CPU Allocation



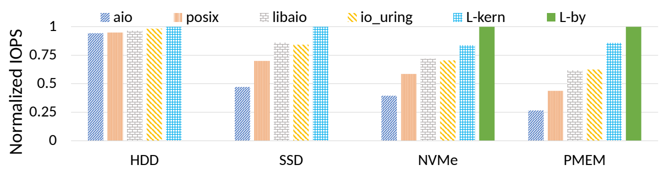
(b) Request Partitioning

Fig. 5. Dynamic Work Orchestration

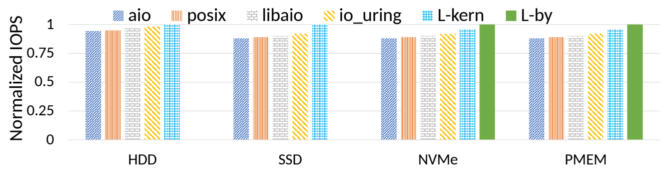
takes roughly 300 seconds – five orders of magnitude slower.

Work Orchestration: Dynamic CPU Allocation: In this evaluation, we measure the ability of the Work Orchestrator to dynamically adjust to changes in load. We run a workload where each client thread randomly writes 1GB of data with 4KB request sizes and vary the number of clients (between 1 and 16). The LabStack tested uses no-op scheduling with Kernel Driver LabMod over NVMe. We compare three worker configurations: 1 worker, 8 workers, and a dynamic number of workers. We measure IOPS and average CPU utilization. From Figure 5(a), it can be observed that when there are only 2 clients, a single worker is able to saturate the load without performance loss. However, after 4 clients, the single worker is overloaded and IOPS decrease by 50%. While 8 workers achieve maximum performance, it has 25% higher CPU utilization than the dynamic policy, which only requires 4 cores. At 16 clients, both the dynamic and 8-worker configurations achieve similar performance and CPU utilization. Overall, it is observed that dynamic work orchestration can balance resource utilization without sacrificing performance.

Work Orchestration: Request Partitioning: In this evaluation, we measure how queue scheduling policies impact performance. To do this, we deploy two LabStacks: latency-sensitive (L) and compressor (C). The L-LabStack contains an LRU cache, No-Op I/O scheduler, and uses the Kernel Driver LabMod as a backend. The C-LabStack adds compression. We run a metadata-intensive workload (L-App) which creates 5,000 files per-thread over the L-LabStack, and a large I/O workload (C-App) which writes 128GB of data per-thread with request sizes of 32MB through the C-LabStack. Both the number of L-App and C-App threads are fixed at 8. We vary the number of Runtime workers to be between 1 and 8. We compare two work orchestration policies: round-robin (RR) and dynamic and measure the average latency of the L-App and bandwidth of the C-App. From Figure 5(b), it can be observed that the RR policy achieves the highest bandwidth in each test. For this policy, all workers are processing the C-App, since queues are evenly divided among workers. Since the processing time of L-Apps is about $3\mu s$, whereas the compression takes roughly $20ms$, bandwidth is impacted negligibly. RR also scales linearly with the number of clients, since all workers process the C-App. However, in general, the RR policy causes severe latency penalty, as the L-App must wait for multiple $20ms$ compressions before being processed. LabStor's dynamic scheduling, however, sends the L-App queues to separate workers from the C-App queues, significantly improving latency, but at a cost to bandwidth. As the number of workers increases, the bandwidth cost drops from 30% down to 6%,



(a) Random Write (4kB, 1 thread)



(b) Random Write (128kB, 1 thread)

Fig. 6. Storage API performance. IOPS are normalized.

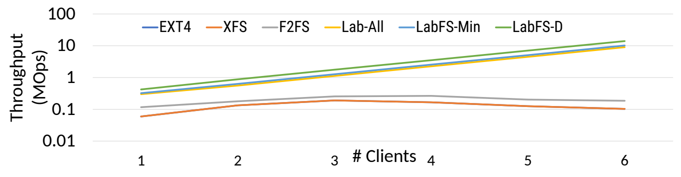


Fig. 7. Metadata throughput

since the L-Apps are all placed on the same worker. Overall, it can be concluded that the Orchestrator can balance requests with varying computational complexities fairly among a given set of resources.

B. Stress Tests

Storage Interface Performance: In this evaluation, we measure the performance of LabStor's storage LabMods compared to traditional kernel-based approaches for interacting with storage in userspace. For these tests, LabStacks consisting only of DAX, SPDK or Kernel Driver LabMods are compared to using POSIX I/O, POSIX AIO, libaio, and I/O Uring to write directly to device files (e.g., /dev/nvme0n1). We repeat all tests for various storage hardware, including PMEM, NVMe SSD, SATA SSD and SATA HDD. We measure end-to-end application performance. We present the results on Figure 6. It can be observed that the I/O paths provided by Labstor outperform existing solutions significantly on low-latency device types (SSD, NVMe, PMEM). Unlike POSIX syscalls, LabStor uses shared-memory queues as opposed to context switches, which avoids software overhead and is friendlier to CPU caches. POSIX AIO suffers additional overhead due to the cost of context switching to the AIO thread, amounting up to 60-70% overhead on NVMe and PMEM. libaio and I/O Uring obtain significantly improved performance over typical POSIX syscalls, as they avoid context switching. However, for 4KB I/Os, the Kernel Driver LabMod outperforms all other solutions by at least 15% by avoiding context switches and data copies on NVMe. The SPDK LabMod outperforms the Kernel Driver LabMod by an additional 12% on NVMe by avoiding the complex allocation of structures required by the Kernel Driver. As the request size increases, the benefits of the reduced software overhead decrease. By the 128KB size, performance difference between SPDK and POSIX is roughly 6%. Overall, it is apparent that different interfaces to storage can have significant impacts on performance depending on the workload and device type. Simply relying on POSIX to build storage systems and incurring the costs of the kernel's stack can yield dramatic latency costs of up to 60%.

Metadata Throughput: In this evaluation, we measure the overhead

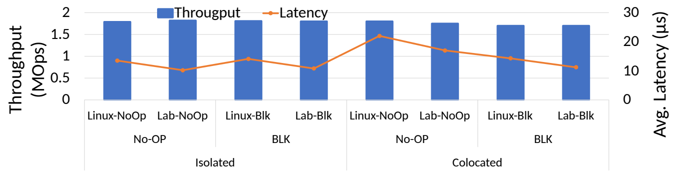


Fig. 8. Performance of different I/O Schedulers

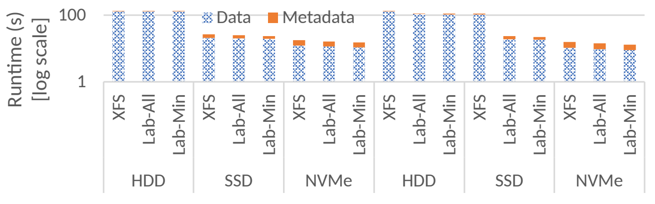
of various LabStor configurations on metadata performance: a common workload in various HPC applications [73]–[75]. We compare three configurations of LabFS to various I/O systems (EXT4, XFS, F2FS) on workloads which stress file creation using FxMark [71]. We vary the number of client threads to be between 1 and 24. The LabStor Runtime is configured with 16 workers. From Figure 7, it can be seen that all LabFS configs outperform the alternatives by up to 3x in the single-threaded case while also maintaining scalability in each of the scenarios. For the single-thread case, LabFS-All primarily outperforms the alternatives due to a reduction in context switches caused by syscalls. When the permissions LabMod is removed from the LabStack, performance improves by an additional 7%. Finally, removing the centralized authority entirely improves performance by an additional 20% in the single-threaded case by removing IPCs. As the number of clients increases, LabFS scales well since LabFS stores all files in a single hashmap, which supports insert, rename, and delete operations with minimal contention and overhead. LabFS's block allocator divides blocks among workers, also minimizing contention. However, the kernel filesystems scale very poorly, as they use locking in order to ensure the correctness of their data structures. Overall, simply relying on kernel filesytems and the VFS as backends for metadata-sensitive workloads can result in severe performance penalties due to software overheads and locking. New high-performance I/O systems can be built within LabStor using its lightweight IPC mechanisms and scalably managed by its Runtime. In addition, by giving end-users more choice of what functionality (e.g., permissions) is truly required by their I/O systems, software overhead can be further reduced.

Developing & Customizing I/O Policies: In this evaluation, we demonstrate the power of the LabStor platform to develop low-level I/O policy decisions while maintaining high performance. To do this, we integrate the No-Op and blk-switch [20] I/O schedulers into LabStor and compare against their in-kernel counterparts. No-op maps I/O requests to device queues based on the CPU core the request originated. Blk-switch takes into consideration the load emplaced on a queue. We deploy two applications: throughput-bound (T-App) and latency-bound (L-App). The T-App produces 64KB random writes per-thread with an I/O depth of 32. The L-App produces 4KB random writes per-thread (I/O depth 1). Both apps run for a period of 1 minute. Both the T-App and L-App have 8 threads, and the LabStor Runtime is configured to have 8 workers. We measure average and P99 latency when the L-Apps and T-Apps are isolated and colocated.

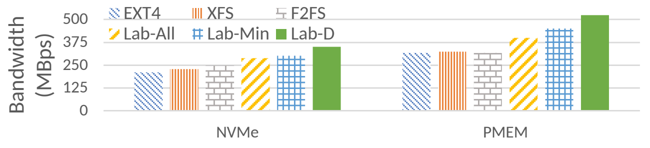
TABLE II
P99 LATENCY OF LINUX AND LABSTOR SCHEDULERS

		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

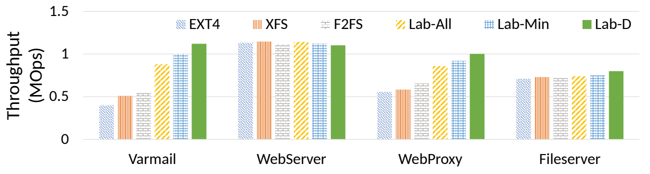
Figure 8 shows that when the processes are isolated, No-Op performs at least as well as blk-switch since T-Apps and L-Apps are mapped to separate queues. This style of workload is similar to HPC



(a) PFS performance



(b) Labios Performance



(c) FileBench Performance

Fig. 9. LabStor under different storage systems

storage nodes, where the node is dedicated to processing I/O requests from a single application (the parallel filesystem). However, after colocating, the performance and QoS of the L-Apps degrade due to head-of-line blocking. This style of workload is more similar to that of a multi-tenant environment (e.g., Cloud or HPC compute nodes), where many conflicting applications may be running. However, by avoiding context switching and upper layers of the I/O stack, LabStor decreases latency by 20% over the kernel version of blk-switch. In addition, LabStor's No-Op reduces latency by an additional 5% when the apps are isolated. Overall, low-level policy decisions can be developed and customized in LabStor with minimal overhead, and the choice of policy can have noticeable performance impacts depending on the workload.

C. *End-to-end Performance with Real Workloads*

In these evaluations, we demonstrate the end-to-end performance of different LabStor configurations for various real workloads.

PFS: In this evaluation, we demonstrate the benefit of deploying a PFS over customized LabStacks. We use OrangeFS with the metadata server deployed separately from the data servers and with a stripe size of 64KB. The metadata server runs on a node with NVMe SSDs. We run two workloads: VPIC [76] and BD-CATS [77]. VPIC is a particle simulation code where each process produces particle data and writes them at each time step. VPIC writes 8 million particles where each particle is a vector of 8 floating point values. We run VPIC with 640 processes in 16 time steps, totaling 165GB of data. BD-CATS reads the data generated by VPIC to perform a parallel clustering algorithm. BD-CATS also uses 640 processes.

From Figure 9(a), it is shown that a PFS gains performance improvements of 6-12% when deployed over a customized I/O stack. These benefits primarily come from the increased throughput of the metadata server, which manages the location of stripes. Overall, roughly 100 million metadata operations are made to the metadata server in both workloads, amounting to roughly 4-6 seconds of execution time. The remaining time is spent in I/O and network. The improved metadata performance stems from using kernel-bypass I/O (LabFS-All) and by minimizing permissions checking overheads (LabFS-Min). When the data storage servers are backed by HDDs, the benefit of improved metadata performance is overshadowed by the cost of I/O. However, as the overhead of I/O decreases by switching to SSDs and NVMes, performance benefits are more noticeable.

Distributed Object Store: In this evaluation, we demonstrate the benefit of customizing I/O interfaces on a state-of-the-art distributed I/O system, LABIOS [60], which bridges the gap between Cloud and HPC workloads through the use of a new data representation – the label. In this test, we measured the I/O bandwidth and throughput of LABIOS Workers, which are responsible for storing and retrieving data. We perform a workload which triggers LABIOS to generate 8KB I/Os. Typically, LABIOS stores labels by translating them to a UNIX file which is written on the disk by POSIX I/O. Each label write triggers a sequence of POSIX calls (fopen()-fseek()-fwrite()-fclose()). This behavior is common among distributed NoSQL DBs, Document Stores, and Key-Value Stores. We compare the performance of using various backends including LabKVS for the workload. We ran this experiment over several types of drives but we present only the NVMe results since it eliminates the differences of the medium and highlights the impact of the stacks used. We avoided presenting the results HDDs since no performance was gained due to seek penalties.

From Figure 9(b), it can be seen that filesystem performance degrades by at least 12% when compared to LabKVS on NVMe and PMEM. This is because filesystems impose the POSIX abstraction over storage, which does not translate well to a key-value store workload, requiring many syscalls in order to place data. As opposed to the typical open-modify-close procedure required by filesystems, LabKVS simply performs put/get, which reduces the number of syscalls from 4 down to 1. In addition, by relaxing the access control guarantees of LabKVS, up to an additional 16% performance is gained. Overall, simply relying on the age-old POSIX API limits the expressiveness of I/O systems, requiring them to comply with awkward file translations and increased software overhead.

Cloud Workloads: Filebench contains four workloads: varmail, webserver, webproxy, and fileserver. We use the default [72] configurations of filebench for the workloads and run over NVMe and PMEM. The results of the NVMe evaluation can be seen on Figure 9. The PMEM experiments return identical trends as those executed over NVMe. In most cases, LabStacks containing LabFS perform markedly better than the alternatives (up to 2.5x throughput) by reducing context switching and the I/O path length. The main exception is fileserver, which performs many large I/Os and is thus dominated by I/O time.

V. Conclusion

In this work, we address the limitations of developing I/O stacks within the confines of monolithic kernels, such as Linux. We present LabStor: a modular, extensible, and high-performance I/O platform. We showcase how modularity can be leveraged to provide expressive, customizable and high-velocity I/O stacks. We leverage this modularity to build composable I/O stacks that provide users with the ability to upgrade and manage their I/O stacks in an easy and efficient manner. Finally, we showcase how utilizing both in-kernel and userspace drivers can help tie these ideas together into a functional runtime environment. Experimental results demonstrate that customized LabStor I/O stacks can yield performance gains of up to 60% over alternatives under various workloads and storage devices.

VI. Acknowledgment

This research is fully supported by the NSF under Grants OCI-1835764, CSR-1814872, and CSSI-2104013. Results in this paper were obtained using the Chameleon testbed supported by the NSF.

References

[1] L. Logan, J. Lofstead, S. Levy, P. Widener, X.-H. Sun, and A. Kougkas, "pmemcpy: a simple, lightweight, and portable i/o library for storing data in persistent memory," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 664–670.

[2] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding lustre filesystem internals," *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, vol. 120, 2009.

[3] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 353–369. [Online]. Available: https://doi.org/10.1145/3341301.3359656

[4] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 323–338.

[5] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "{NVMeDirect}: A user-space {I/O} framework for application-specific optimization on {NVMe}{SSDs}," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[6] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The xtreemfs architecture—a case for object-based file systems in grids," *Concurrency and computation: Practice and experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[7] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system." in *FAST*, vol. 8, 2008, pp. 1–17.

[8] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for {Non-Volatile} main memories and disks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 207–219.

[9] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true {Direct-Access} file system with {DevFS}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 241–256.

[10] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[11] S. Gugnani and X. Lu, "Dstore: A fast, tailless, and quiescent-free object store for pmem," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 31–43.

[12] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 603–616.

[13] S. Miller, K. Zhang, M. Chen, R. Jennings, A. Chen, D. Zhuo, and T. Anderson, "High velocity kernel file systems with bento," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 65–79. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/miller

[14] Q. Huai, W. Hsu, J. Lu, H. Liang, H. Xu, and W. Chen, "{XFUSE}: An infrastructure for running filesystem services in user space," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 863–875.

[15] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Scale and performance in a filesystem semi-microkernel," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 819–835. [Online]. Available: https://doi.org/10.1145/3477132.3483581

[16] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, "Direct-fuse: Removing the middleman for high-performance fuse file system support," in *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, 2018, pp. 1–8.

[17] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble *et al.*, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 399–413.

[18] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, "I'm not dead yet! the role of the operating system in a kernel-bypass era," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–80. [Online]. Available: https://doi.org/10.1145/3317550.3321422

[19] D. Ji, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan, "Microtee: designing tee os based on the microkernel architecture," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2019, pp. 26–33.

[20] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal, "Rearchitecting linux storage stack for µs latency and high throughput," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 113–128. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/hwang

[21] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "{Multi-Queue} fair queuing," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 301–314.

[22] M. Yi, M. Lee, and Y. I. Eom, "Cffq: I/o scheduler for providing fairness and high performance in ssd devices," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, 2017, pp. 1–6.

[23] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving read performance by isolating multiple queues in nvme ssds," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, 2017, pp. 1–6.

[24] J. Woo, M. Ahn, G. Lee, and J. Jeong, "{D2FQ}:{Device-Direct} fair queueing for {NVMe}{SSDs}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 403–415.

[25] P. Mishra and A. K. Somani, "Host managed contention avoidance storage solutions for big data," *Journal of Big Data*, vol. 4, no. 1, pp. 1–42, 2017.

[26] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[27] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "{MDev-NVMe}: A {NVMe} storage virtualization solution with mediated {Pass-Through}," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 665–676.

[28] B. Peng, J. Yao, Y. Dong, and H. Guan, "Mdev-nvme: Mediated pass-through nvme virtualization solution with adaptive polling," *IEEE Transactions on Computers*, 2020.

[29] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, nov 2015. [Online]. Available: https://doi.org/10.1145/2812806

[30] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true Direct-Access file system with DevFS," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 241–256. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/kannan

[31] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 478–493. [Online]. Available: https://doi.org/10.1145/3341301.3359637

[32] "Direct access for files," 2014. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[33] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 154–161.

[34] M. Bjørling, J. Gonzalez, and P. Bonnet, "{LightNVM}: The linux {Open-Channel}{SSD} subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 359–374.

[35] N. Moti, F. Schimmelpfennig, R. Salkhordeh, D. Klopp, T. Cortes, U. Rückert, and A. Brinkmann, "Simurgh: A fully decentralized and secure nvmm user space file system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476180

[36] Y. Chen, Y. Lu, B. Zhu, and J. Shu, "Kernel/user-level collaborative persistent memory file system with efficiency and protection," *arXiv preprint arXiv:1908.10740*, 2019.

[37] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an {RDMA-enabled} distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[38] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with {Kernel-Userspace} collaboration," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 81–95.

[39] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 194–209.

[40] T. Vinçon, A. Bernhardt, I. Petrov, L. Weber, and A. Koch, "nkv: near-data processing with kv-stores on native computational storage," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020, pp. 1–11.

[41] B. Zhang and D. H. Du, "Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction," *ACM Transactions on Storage (TOS)*, vol. 17, no. 3, pp. 1–26, 2021.

[42] T. Bisson, K. Chen, C. Choi, V. Balakrishnan, and Y.-s. Kee, "Crail-kv: A high-performance distributed key-value store leveraging native kv-ssds over nvme-of," in *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2018, pp. 1–8.

[43] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 460–477.

[44] W. Wang and S. Diestelhorst, "Quantify the performance overheads of pmdk," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 50–52.

[45] L. Zhang and S. Swanson, "Pangolin: A {Fault-Tolerant} persistent memory programming library," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 897–912.

[46] Y. Ren, C. Min, and S. Kannan, "{CrossFS}: A cross-layered {Direct-Access} file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 137–154.

[47] T. L. Kernel, "Fuse." [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[48] J. Lofstead and A. Younge, "Gestalt computing: Hybrid traditional hpc and cloud hardware and software support," *CLOUD COMPUTING 2022*, p. 63, 2022.

[49] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 169–182.

[50] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[51] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[52] Top500, "Operating system family / linux," November 2021. [Online]. Available: https://www.top500.org/statistics/details/osfam/1/

[53] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, "Efficient scheduling policies for {Microsecond-Scale} tasks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1–18.

[54] J. Liu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Kannan, "File systems as processes," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotstorage19/presentation/liu

[55] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 195–211. [Online]. Available: https://doi.org/10.1145/3477132.3483569

[56] N. Moti, F. Schimmelpfennig, R. Salkhordeh, D. Klopp, T. Cortes, U. Rückert, and A. Brinkmann, "Simurgh: a fully decentralized and secure nvmm user space file system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[57] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "Mlpack: A scalable c++ machine learning library," *Journal of Machine Learning Research*, vol. 14, no. Mar, pp. 801–805, 2013.

[58] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[59] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "Hcompress: Hierarchical data compression for multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 557–566.

[60] A. Kougkas, H. Devarajan, J. Lofstead, and X.-H. Sun, "Labios: A distributed label-based i/o system," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 13–24.

[61] A. Kougkas, H. Devarajan, and X.-H. Sun, "Iris: I/o redirection via integrated storage," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 33–42.

[62] ——, "Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 219–230.

[63] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "Dlio: A data-centric benchmark for scientific deep learning applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 81–91.

[64] J. Cernuda, H. Devarajan, L. Logan, K. Bateman, N. Rajesh, J. Ye, A. Kougkas, and X.-H. Sun, "Hflow: A dynamic and elastic multi-layered i/o forwarder," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 114–124.

[65] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic i/o scheduler for shared non-volatile burst buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 290–301.

[66] L. Soares and M. Stumm, "{FlexSC}: Flexible system call scheduling with {Exception-Less} system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[67] W. J. Stewart, *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009.

[68] J. U. Farley and L. W. Ring, "A stochastic model of supermarket traffic flow," *Operations Research*, vol. 14, no. 4, pp. 555–567, 1966.

[69] L. Torvalds, "Linux kernel github," 2016, commit hash: 1f5bd336b9150560458b03460cbcfcfbcf8995b1. [Online]. Available: https://github.com/torvalds/linux

[70] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[71] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 71–85.

[72] R. McDougall and J. Mauro, "Filebench," *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. pdf (Cited on page 56.)*, 2005.

[73] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, "Automatic,{Application-Aware}{I/O} forwarding resource allocation," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 265–279.

[74] A. K. Paul, B. Wang, N. Rutman, C. Spitz, and A. R. Butt, "Efficient metadata indexing for hpc storage systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 162–171.

[75] S. Patil, K. Ren, and G. Gibson, "A case for scaling hpc metadata performance through de-specialization," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 30–35.

[76] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[77] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "Bd-cats: big data clustering at trillion particle scale," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

LabStor is a platform consisting of multiple libraries and binaries. Scripts for automating the compilation, deployment, and testing of this system are provided in GitHub[1]. A readme file is also provided which contains detailed steps on how to install and test LabStor.

System: We ran all of our experiments in Chameleon using the storage hierarchy node, equipped with NVMe (Intel P3700; 2TB), SSD (Intel SSDSC2BX01; 1.6TB) HDD (Seagate ST600MP0005; 600GB), and 512 GiB of RAM. It contains 2xIntel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz for a total of 24 cores and 48 threads.

Software: We use Ubuntu 20.04 with kernel 5.4.0-100-generic as the OS and kernel for all experiments. For our synthetic benchmarks we use FIO 3.28 and FxMark[2]. For the non-synthetic evaluations, we use LABIOS[3], specifically the worker code and FileBench-1.4.9.1. To setup emulation for PMEM devices, we apply minor modifications to the OS bootloader. Instructions to do this are provided in the GitHub. We use SPDK v21.10 in some experiments as an NVMe driver. All this software is provided in the LabStor repo.

Compiling LabStor: To compile LabStor, a single CMake script is provided in the head of the repo. The script compiles all LabStor code. Users can set the CMAKE_INSTALL_PREFIX to install all binaries, libraries, and headers into a separate directory; but this is optional. More detail on compiling/installing LabStor is provided in the GitHub.

LabStor Components: LabStor is comprised of several binaries and executables. After compilation, a client library, an executable for the Runtime, and a kernel module for the Kernel Ops Manager will be produced. In addition, there are various LabMods (e.g., GenericFS, LabFS, LRU, NoOp, SPDK, Kernel Drivers) which can be dynamically loaded into the Runtime using the provided mount commands.

LabStor Deployment/Testing: An automation script is provided (benchmark/test.py) to setup and run all tests used in LabStor. This script automatically deploys the kernel module, the LabStor Runtime, and the various LabMods required for each individual test. Details of the test script are provided in the GitHub. For experiments which perform I/O at any point, the script clears all system caches before running the test.

0.1 Evaluation Conduct

Following is a detailed explanation of how all evaluations where performed. Detailed technical details on how to performed each experiment are available in the provided script.

I/O Anatomy: We used LabStor to construct a LabStack containing LabFS, LRU caching, NoOp I/O scheduling, and Kernel Drivers. We collect time measurements from each of the modules on the stack during a read and write operation. Time is collected using a separate, dedicated timer thread which stores the current time in shared memory by continuously polling std::chrono.

[1]https://github.com/scs-lab/labstor
[2]https://github.com/sslab-gatech/fxmark. Commit 3f29552
[3]https://github.com/scs-lab/labios. Commit a03384a

Live Upgrade: We used LabStor to construct a LabStack containing only a dummy LabMod and send 100,000 messages to the LabMod using a custom driver program. We then upgrade the dummy LabMod while it is executing and measure overall execution time. The Runtime is configured with one worker.

Work Orchestrator: Dynamic CPU Allocation: We run a workload where each client thread randomly writes 1GB of data with 4KB request sizes and vary the number of clients (between 1 and 16) using a custom driver program. The LabStack tested uses NoOp scheduling with Kernel Driver LabMod over NVMe. We compare three Runtime work orchestrator configurations: 1 worker, 8 workers, and a dynamic number of workers.

Work Orchestration: Request Partitioning: We deploy two LabStacks: latency-sensitive (L) and compressor (C). The L-LabStack contains an LRU cache, No-Op I/O scheduler, and uses the Kernel Driver LabMod as a backend. The C-LabStack adds ZLIB compression. We run a metadata-intensive workload (L-App) which creates 5,000 files per-thread over the L-LabStack, and a large I/O workload (C-App) which writes 128GB of data per-thread with request sizes of 32MB through the C-LabStack. Both the number of L-App and C-App threads are fixed at 8. We vary the number of Runtime workers to be between 1 and 8.

Storage API performance: We compared the kernel's APIs (I/O uring, libaio, aio, posix) to LabStor's SPDK and Kernel Driver LabMods over HDDs, SSD, NVMe, and emulated PMEM using fio. We used a single thread and request sizes of 4KB and 128KB. When using the kernel's APIs, we perform direct I/O (O_DIRECT) to the device files, bypassing filesystem implementations.

I/O Schedulers: We integrate the No-Op and blk-switch I/O schedulers into LabStor and compare against their in-kernel counterparts. For this evaluation, we compiled the custom kernel required for blk-switch to work (kernel 5.4.43). LabStor was still executed in kernel 5.4.0 to avoid additional development. We deploy two fio instances: throughput-bound (T-App) and latency-bound (L-App). The T-App produces 64KB random writes per-thread with an I/O depth of 32. The L-App produces 4KB random writes per-thread (I/O depth 1). Both apps run for a period of 1 minute. Both the T-App and L-App have 8 threads, and the LabStor Runtime is configured to have 8 workers.

Metadata throughput: We compare LabStor against various filesystems for metadata performance using FxMark. We vary the number of client threads to be between 1 and 24. The LabStor Runtime is configured with 16 workers. The LabStacks are defined as follows:

(1) *Centralized+Permissions*: Permissions, LabFS. LabStack executes asynchronously.
(2) *Centralized*: Removes the Permissions Module.
(3) *Minimal*: Removes the Permissions Module. LabStack executes synchronously.

Labios: We ran the Labios Worker program, which uses open/lseek/write/close to perform I/O. We use a single thread to write 8KB IOs to NVMe (randomly) and emulated PMEM. The LabStor Runtime is also configured with a single thread. We compare three LabStacks against EXT4/XFS/F2FS. The LabStacks are defined as follows:

(1) *Centralized+Permissions*: Permissions, LabKVS, NoOp I/O scheduling, Kernel Drivers. LabStack executes asynchronously.
(2) *Centralized*: Removes the Permissions Module.
(3) *Minimal*: Removes the Permissions Module. LabStack executes synchronously.

Filebench: For the filebench workload, we ran varmail, webserver, webproxy, and fileserver using the default configurations over NVMe and emulated PMEM. The Runtime is configured with 8 workers. We compared EXT4, XFS, and F2FS against three LabStacks. The LabStacks are defined as follows:

(1) *Centralized+Permissions*: Permissions, LabFS, LRU Caching, NoOp I/O scheduling, Kernel Drivers. LabStack executes asynchronously.
(2) *Centralized*: Removes the Permissions Module.
(3) *Minimal*: Removes the Permissions Module. LabStack executes synchronously.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6941987`
Artifact name: LabStor GitHub

Reproduction of the artifact without container: We cannot containerize or virtualize the LabStor experiments, as we are including optimizations which require modifications to the host machine. However, LabStor has only three dependencies: kernel 5.4.0, Yaml-CPP (commit db6deed), and Boost 1.74. The only limiting dependency is the required kernel version, which is kernel 5.4.0. Ideally, a baremetal machine which can have any OS or kernel version installed (e.g., Chameleon Cloud) would be used. Besides this, the LabStor GitHub has automation scripts for installing necessary dependencies (including kernel versions) and running experiments. A single script is used for the installation of all dependencies (install.sh in the root of the GitHub). A single script is used for the execution of the test cases (benchmark/test.py). The repo has more details about the exact execution of the scripts and installation processes.