TailorFS: An Adaptive file system to support dynamic I/O requirements of HPC workloads

Hariharan Devarajan, Kathryn Mohror
Lawrence Livermore National Laboratory
hariharandev1@llnl.gov, kathryn@llnl.gov

***Abstract*—High-Performance Computing (HPC) systems typically provide a global storage system to support large-scale workload's I/O requirements. However, these workloads have diversified from traditional simulation to include big data analytics and AI workloads. HPC systems include hardware accelerators and storage software with configurations to support this workload diversification. However, selecting the correct combination of these configurations for each workload is a complex task even for I/O experts. To address this problem, we designed TailorFS, a software abstraction using FSView that transparently selects the appropriate file system characteristics including software abstractions, hardware accelerators, and optimizations to accelerate I/O for a given workload. Given a workload's behavior, TailorFS chooses an appropriate software existing in the system and a corresponding configuration to dynamically optimize the workload. With TailorFS, users can utilize any interface to perform I/O on the file system and get the correct software selected for them based on the described behavior. Finally, TailorFS can accelerate I/O for different workload types by up to 46× by utilizing workload characteristics. In conclusion, TailorFS can accelerate complex HPC workflows by up to 6.5× better I/O performance on the Lassen supercomputer by using dynamically created workload-aware FSViews.**

***Index Terms*—tailorfs, fsview, dynamic optimization, workload-aware, I/O intent-driven**

I. Introduction

Modern HPC workloads that transfer large amounts of data [1] are known as data-intensive workloads. These workloads consist of single applications [2], such as simulation, analysis, and artificial intelligence (AI) applications, and multiple applications that coordinate within a workflow [3]. HPC systems are equipped with new hardware (e.g., hierarchical storage [4]–[6]) and software solutions (e.g., middleware I/O libraries [7]–[9]) to tackle workload diversity. These solutions have demonstrated their effectiveness in improving workloads' I/O performance.

Recent studies have proposed several hardware and software solutions to tackle the diversity of workloads. Each solution caters to a specific class of workloads often characterized by their I/O behavior. For example, hierarchical storage accelerators such as node-local and shared burst buffers along with buffering solutions [7]–[9] accelerate traditional simulation applications by accepting write data into faster tiers and asynchronously moving them to slower but larger storage devices. Another example, HPC-Cloud integration solutions [10]–[13] enable efficient access to scientific data for high-performance data analytics kernels such as k-means clustering and linear regression algorithms to increase their I/O performance. Additionally, data management solutions [14]–[16] provide efficient data ingestion of scientific data to increase I/O performance for AI frameworks such as TensorFlow, PyTorch, and LBANN. These approaches demonstrate the benefit of specializing in software and hardware for specific use cases that can significantly improve the I/O performance of many workloads.

Even with existing approaches for improving performance, it is a challenge for users to adopt these approaches into their workflows for three key reasons. First, in terms of hardware support, HPC systems provide hierarchical storage devices that can improve performance for specific use cases. However, different devices should be used in different cases, such as SSD for random access or shared memory for intra-node process communication. Second, researchers provide software abstractions [7]–[9] and configurations to simplify the access to hierarchical storage. However, each solution is best for different workloads, such as UnifyFS [7] is best for collective I/O buffering, whereas Hermes [8] provides independent data placement strategies to efficiently support file-per-process workloads. This makes the choice of which software to use and with what configuration a hard problem to solve for users. Third, even if we assist application users with specific software and configurations manually, we see most workloads consist of multiple applications coordinated in a workflow. The presence of workflows results in different I/O requirements for different parts of the workload and makes the manual application of these software solutions for this dynamic environment impractical. Therefore we need an automated way to present a file system abstraction for the user, which can change its hardware and software characteristics dynamically and improve I/O performance for this complex and diverse environment.

In this work, we develop TailorFS: a middleware library that dynamically selects the best software and its configurations available on the HPC system to optimize different parts of the workload. The fundamental idea behind TailorFS is a software abstraction called *FSViews* that selects existing system software that matches the workload's I/O requirements. For example, the FSView for processes in workload that perform collective writes to a file uses UnifyFS to buffer collective writes efficiently into node-local burst buffers. Alternatively, the FSview for a set of processes in the same workload that performs random, independent reads uses ZFS over node-local NVMe with the POSIX interface. TailorFS simply determines the best way to use the I/O software stack and hardware

already on the system for the current workload's I/O requirements. Users can utilize any interface to interact with the file system, and TailorFS transparently utilizes the best option available from the existing software stack to accelerate I/O. We run a microbenchmark with the available hardware and software solutions on the Lassen cluster at LLNL to determine the best software for different workloads' I/O requirements. These results are used to drive the TailorFS engine that chooses the best software and configurations for the 1000 Genomes workflow on the Lassen machine and optimizes the workflow by up to 6.5x. TailorFS's benchmarking module can be executed on a new supercomputer once to allow users to fully utilize the supercomputer without needing explicit changes to their workloads. The contributions of this work are:

1) *Design and implementation of TailorFS* for creating dynamic FSViews with workload-aware features to handle changing I/O requirements within an HPC workload.
2) *Comprehensive study of I/O behavior* by benchmarking a collection of different workloads aimed at understanding the impact of solutions on overall I/O performance.
3) *Evaluate the benefit of TailorFS* on large-scale workflows on the Lassen supercomputer.

II. Background

In the last decade, HPC data-intensive workloads have significantly diversified with applications from the cloud environments [17]. The primary examples of these workloads are high-performance data analytics and AI that support traditional simulations to reduce the modeling cost within the simulation workloads [18]. This diversification of workloads in HPC environments has resulted in I/O requirement diversification in these workloads.

Modern high-performance data analytics workloads perform embarrassingly parallel algorithms to access and perform transformations on the data [19]. We describe the I/O requirements for some of the popular workloads in HPC from analytics, artificial intelligence, and multi-stage workflows using BD-CATS [20], k-means Clustering [21], and Linear Regression [22] for analytical workloads. The common I/O characteristic of these workloads is that they iteratively traverse over the data to converge to a solution for their respective analytical algorithm. A key difference between these workloads' I/O characteristics is that the data they consume are accessed with different I/O features. BD-CATS operates on a shared HDF5 file using MPI-IO semantics, k-means reads points from files often divided by the number of tasks in the algorithm, and Linear Regression models often consume large CSV data of floating-point values.

For background on AI workloads, we describe CosmoFlow using LBANN [23], Neutrino and Cosmic Tagger with UNet (Cosmic Tagger) using PyTorch [24], and Distributed Flood Filling Networks (DFFN) using TensorFlow [25]. All these workloads consume large HDF5 datasets, but the nature of the dataset differs between these workloads. In terms of I/O characteristics, CosmoFlow consumes multiple HDF5 files using MPI-IO semantics (shared-file access), where each file represents a sample of size 32 MB. In contrast, Cosmic Tagger and DFFN consume large HDF5 files using h5py APIs and stripe the file in memory. Another difference in these workloads is the nature of the frameworks used. For instance, CosmoFlow has been implemented in multiple frameworks LBANN (most recent), TensorFlow, and PyTorch. Each AI framework has its I/O optimizations and configurations, which will impact the way the storage system performs prefetching for each of these instances of the workload. The I/O characteristic of the type of AI framework used from these workloads dictates different I/O requirements from the workload for the storage system in configuring prefetching schemes for the dataset.

For background on workflow I/O requirements, we describe large-scale workflows such as Multiscale Machine-Learned Modeling Infrastructure [26] (MuMMI), Montage [27], and 1000 Genome [28] workflows. These workflows contain multi-step applications with a collection of simulation, analysis, and AI stages. These workflows contain several unique I/O characteristics. First, the behavior of I/O changes as the workflow goes through different stages. For example, during the simulation stage of MuMMI, the application produces a write-heavy workload of generating checkpoints. Whereas during the AI stage, the workload iteratively reads the simulation data to train the AI model over several epochs. Second, as the workflow progresses, the I/O behavior changes for the same file. Specifically in Montage, the intermediate files are written by a group of processes that are only read by other stages. Therefore, the I/O characteristic changed from write-only for one application to read-only for another on the same collection of files. Finally, different files within a workflow can exhibit different I/O characteristics. For instance, the 1000 Genomes workflow accesses millions of files, some of which are accessed in producer/consumer access patterns, some are only produced as checkpoints, some are generated, some are only needed temporarily, and others are accessed independently by workflow processes.

The I/O requirement for the storage system varies greatly for each of these diverse characteristics. For instance, to support efficient producer/consumer between stages of the workflow, the storage system needs to provide efficient data-sharing optimizations such as RDMA and NVMe-over-fabric that enable access to remote compute node storage. Additionally, to manage node-local temporary data efficiently, the storage system should temporarily cache and, once no longer needed, discard the data from the local shared memory of the compute nodes.

III. Motivation

The diversity and dynamicity of I/O requirements among HPC workflows make storage infrastructures with a one-size-fits-all approach highly inefficient. This has led to sub-par I/O performance for workloads that get less than 1% of the peak bandwidth from underlying storage infrastructure [29]. Diverse HPC workloads would have a different storage requirements for the storage system (Figure 1). For instance, simulation workloads with checkpoint and restart features require a

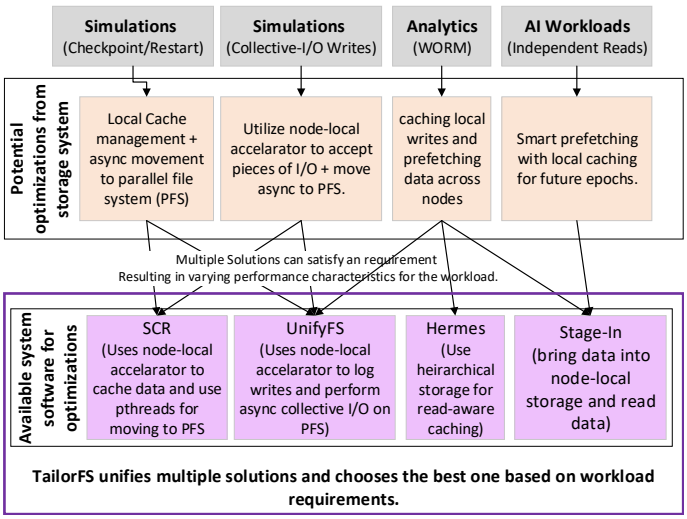


Fig. 1. Diverse HPC workloads have different requirements from storage systems that multiple storage solutions can serve. Each selection would result in different performance characteristics. TailorFS unifies these solutions as file system features to alleviate this burden from the users.

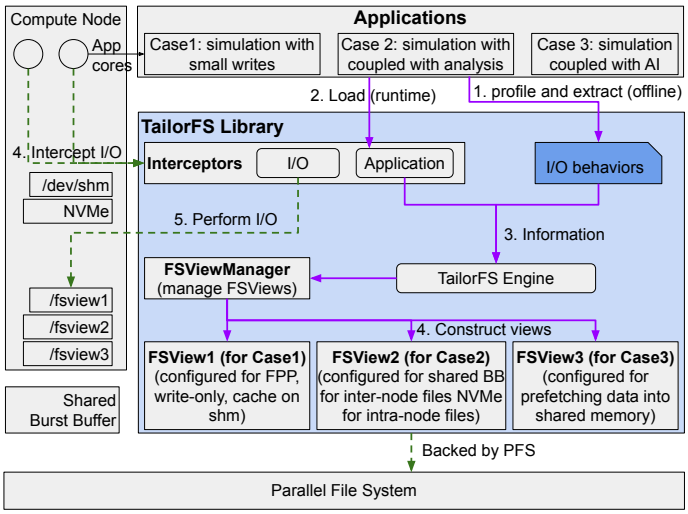


Fig. 2. From a high level, TailorFS is a middleware library that utilizes extracted I/O behaviors from HPC workloads to map it to a diverse set of I/O software and hardware available in the system. This mapping happens at the granularity of each file stored and accessed from the file system. TailorFS provides a virtual file system abstraction called FSView that provides specialized storage features for each part of the workload.

storage system to accept the data from the workload quickly and then reliably move to parallel file systems (PFS). Another example is AI workloads that require smart prefetching to make samples available for training with no perceivable delay. These requirements can be served by many storage middleware solutions within an HPC system. Each storage solution has its own file system features, such as granularity of I/O, persistency guarantees, and requirement of resources for its operation. For instance, we can utilize both Hermes and UnifyFS to accelerate WORM access patterns. However, UnifyFS has a more isolated data movement behavior over node-local storage, whereas Hermes provides a locality-aware prefetching for workloads. Both these solutions would work for this workload, but an emphasis on read-optimizations would result in higher overall performance. Most HPC users are not aware of the internal workings of these solutions and tend to utilize the easiest option available. Based on these observations, TailorFS aims to unify multiple solutions in the software stack and choose the best software and its configurations dynamically based on workload characteristics.

IV. TailorFS

TailorFS is a middleware library between the workloads and the HPC system that provides a tunable file system abstraction with efficient storage features based on the I/O requirements of various workloads. TailorFS's tunable file system abstraction allows users to transparently use the correct solution from the software stack for their workload behavior. TailorFS includes four features based on our observations of different I/O patterns. First, the *I/O interface* with which the workload should interact with the file system should be suitable for the data access performed by the workload. Second, *middleware I/O libraries* available in the system to accelerate certain types of I/O patterns should be correctly configured to optimize workload-specific data access. Third, *storage accelarators* available in the system, such as node-local or shared burst buffers, should be used to reduce I/O latency and increase I/O bandwidth in workloads. Finally, *workload-aware I/O optimizations* to dynamically provide specific file system optimizations for specific parts of the workloads, such as buffering and prefetching, are essential to maximize I/O performance for workloads. To design TailorFS, we prioritize the following core objectives.

1) *Dynamic:* TailorFS dynamically constructs an abstract file system with workload-specific storage features.
2) *Granular:* TailorFS adjusts file system features for different granularities such as workflows, applications, and files.
3) *Extensible:* TailorFS is extensible so that new middleware libraries or file systems can be supported.
4) *Lightweight:* TailorFS library maintains a low memory footprint and a low overhead on I/O performance.

TailorFS is a middleware library that attaches to the workload's process (Figure 2). The three main components of TailorFS are Interceptors, FSViewManager, and TailorFS Engine. The flow of operations consists of a control (purple line and solid) and data paths (green line and dotted). The control path is responsible for building abstract file systems for the workload whereas the data path involves the workload's I/O path within the HPC system. To understand the requirements of HPC workloads, researchers utilize application profilers with analysis tools [30] to extract I/O behaviors from different parts of the application (Figure 2 arrow 1.). This extraction is performed offline, and the extracted I/O behaviors are given to TailorFS as input for building workload-aware file systems. During the execution of the workload, the TailorFS library is preloaded along with the workload as part of the batch scheduler's job submission script (Figure 2 arrow 2.). When the TailorFS library is loaded, the application information (from system runtime) along with its I/O behavior is loaded and passed to the TailorFS Engine (Figure 2 arrow 3.).

The TailorFS Engine is the central component of the TailorFS library which maps the I/O behavior to corresponding file system characteristics called an FSView. This mapping is passed to the FSViewManager, which is responsible for creating and destroying FSViews based on the requirement of the workload. An FSView created by the FSViewManager, is a file system abstraction that contains the I/O interface, software libraries, storage accelerator, and I/O optimizations that should be used for specific files in the workload. An FSView is shared between files, processes, or applications that have the same I/O requirement for the storage infrastructure. For instance, for simulation applications with small writes, the FSViewManager would create a FSView that would utilize shared memory to cache small writes and asynchronously flush them to PFS (Figure 2 Case 1). Whereas for an AI workload, the FSView would be configured to do aggressive prefetching into node-local shared memory for reducing the cost of read operations from PFS (Figure 2 Case 3). Once the FSViews are created, the workload performs I/O operations that are intercepted by TailorFS and are redirected to specific FSView instances that optimize the data path for the workloads. Building different FSViews dynamically is the key technology to achieve tunable file system abstractions so that the workload can achieve high I/O performance under dynamic and diverse environments.

A. FSViews

An FSView is an abstract file system that provides specialized storage features to match the workload's I/O requirements. Each FSView contains one or more I/O Entities in the system. An I/O Entity in the system is a file, a folder, or a file segment. Each I/O Entity has a unique identifier consisting of a path, a starting offset, ending offset, and file system-specific attributes such as permissions and timestamps (as seen in `fstat`). Every I/O Entity has a physical existence within the underlying file system (such as shared memory, XFS on node-local SSD, Datawarp on shared BB, or PFS), and therefore, we reuse the metadata management of the underlying file system. This re-utilization of the underlying file system enables us to share I/O Entities across different FSViews within TailorFS without any side effects such as the coherency of metadata. I/O Entities support all file system functions, such as read, write, and metadata operations.

The storage features supported by an FSView are I/O interface, I/O libraries, storage accelerator, and I/O optimizations to be provided for the workload. FSViews do not change the data but impact "where" and "how" the I/O occurs; therefore, the data is accessible between different FSViews. FSView provides special features catering to a specialized use case such as read-only, write-only, or read-after-write. Additionally, we expect a many-to-one mapping between files and FSViews in the system because, generally, in an application, multiple files with the same characteristics would require the same FSView features. Therefore, we expect the number of FSView in the tens scale, not more. Additionally, if required, an FSView is backed by a global PFS such as GPFS/Lustre to enable inter-node data access.

The concept of FSViews might seem similar to Database Views, but they are different in these two fundamental aspects. First, Database Views provide a simple interface for the user to interact with complex database tables, but the underlying queries are executed over the actual tables in the database. In contrast, an FSView changes the user's view of the data and how it is stored and retrieved from the file system. Second, Database Views cannot be updated (or written to), whereas in FSViews as the physical entities are the same (i.e., a file) we can write files to an FSView and then read them with different I/O features from a different FSView. That is, the write granularity need not match the read granularity for the workload.

B. FSViewManager

FSViewManager is the central component of the TailorFS library that is responsible for managing FSViews. During the library's initialization the FSViewManager iterates through different I/O Entities in the I/O behavior and invokes the TailorFS Engine to get the features of each I/O Entity's FSView. Every type of FSView has a unique identifier in the system, which is constructed based on the features of the FSView. The unique identifier enables different files that require the same file system features to share the same FSView. The FSViewManager defines the features of the FSView as a logical abstraction over existing storage infrastructures, such as I/O interface, I/O libraries, storage accelerator, and I/O optimizations to be used within the FSView. This design choice makes the creation and destruction of FSViews lightweight with no data coherency issue on the underlying file system. The costly part during creation or deletion is the flush or prefetching optimizations within the FSView. The flush optimization is periodically performed asynchronously during specific file system events such as file close or library finalization. In comparison, the prefetching optimization is performed asynchronously during library initialization itself. This allows the FSView to move the data in the background when the application is doing computation.

C. TailorFS Engine

TailorFS Engine aims to map I/O Entities to FSViews with specific features. To achieve this goal, we studied the I/O performance of a comprehensive benchmark using different I/O interfaces, I/O libraries, storage devices, and I/O optimizations. Then, we developed a simple rule engine using a decision tree that we use to configure FSViews. The decision tree learns the best combination of features for given workload characteristics. For the comprehensive benchmarking, we describe the system hardware, software used, test configurations, and interpret the results.

1) Hardware: We ran the comprehensive benchmark on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [31]. As a 23-petaflop IBM Power9 system, it is the 30th supercomputer in the world (as of June 2022), based on the Top500 list. Lassen has 795 nodes connected with a Mellanox 100 GB/s enhanced data rate (EDR) InfiniBand network and a 24 PiB IBM Spectrum Scale file system (also known as GPFS). Individual Lassen nodes

TABLE I
BENCHMARKING RESULTS FOR VARIOUS TEST CONFIGURATIONS

SNo	Access Pattern	Process Data Sharing	Transfer Size	32 nodes				64 nodes				128 nodes			
				Write Only	Read Only	RAW	WORM	Write Only	Read Only	RAW	WORM	Write Only	Read Only	RAW	WORM
1	seq	single proc	4KB	STDIO shm (5.7x)	STDIO shm (4.9x)	STDIO shm (4.1x)	MPIIO bb (6.1x)	STDIO shm (5.6x)	STDIO shm (4.9x)	STDIO shm (4.1x)	MPIIO bb (10x)	STDIO shm (5.5x)	STDIO shm (4.7x)	STDIO shm (4.1x)	MPIIO bb (10.8x)
2			64KB	POSIX shm (4.5x)	POSIX shm (3.8x)	POSIX shm (2.6x)	POSIX shm (2.4x)	POSIX shm (4.5x)	POSIX/ STDIO shm (10.9x)	MPIIO/ POSIX shm (4.3x)	POSIX pfs (2.7x)	POSIX shm (4.5x)	POSIX shm (12.2x)	MPIIO shm (5.7x)	MPIIO shm (3.9x)
3			1MB	MPIIO/ POSIX shm (10.1x)	MPIIO/ POSIX shm (14.5x)	MPIIO shm (9.4x)	STDIO shm (8.2x)	MPIIO shm (9.2x)	POSIX/ STDIO shm (31x)	MPIIO shm (11x)	STDIO shm (12.4x)	MPIIO shm (7.5x)	POSIX shm (75x)	MPIIO shm (15x)	STDIO shm (313x)
4		intra node	4KB			STDIO shm (3.3x)	POSIX bb (6.4x)			STDIO shm (3.3x)	POSIX bb (6.4x)			STDIO shm (3.2x)	POSIX bb (5.8x)
5			64KB			POSIX shm (4x)	POSIX shm (2.2x)			POSIX shm (6.4x)	POSIX shm (2.9x)			POSIX shm (11.2x)	POSIX shm (14.9x)
6			1MB			STDIO shm (7.3x)	STDIO shm (4.5x)			POSIX/ STDIO shm (7.7x)	POSIX shm (6.9x)			POSIX shm (6.2x)	POSIX shm (7.2x)
7		inter node	4KB			STDIO pfs (1.56x)	STDIO pfs (1.28x)			STDIO pfs (1.63x)	STDIO pfs (1.3x)			STDIO pfs (1.6x)	STDIO pfs (1.3x)
8			64KB			POSIX pfs (1.3x)	MPIIO pfs (1.4x)			POSIX pfs (1.2x)	STDIO pfs (1.3x)			POSIX pfs (1.2x)	POSIX pfs (1.1x)
9			1MB			STDIO pfs (1.1x)	POSIX pfs (1.1x)			MPIIO pfs (1.1x)	STDIO pfs (2.2x)			MPIIO pfs (1.1x)	MPIIO pfs (2.1x)
10	random	single proc	4KB		STDIO shm (4.9x)	STDIO shm (4.2x)	MPIIO bb (7.9x)		STDIO shm (4.9x)	STDIO shm (4x)	MPIIO bb (5.3x)		STDIO shm (4.9x)	STDIO shm (4x)	MPIIO bb (12.4x)
11			64KB		POSIX/ STDIO shm (4.1x)	POSIX shm (3.5x)	POSIX shm (2.4x)		POSIX/ STDIO shm (6.2x)	POSIX/ MPIIO shm (3.2x)	MPIIO shm (2.5x)		POSIX shm (43.6x)	MPIIO shm (6.4x)	MPIIO shm (3.1x)
12			1MB		MPIIO shm (44.2x))	MPIIO shm (20.2x)	MPIIO shm (12.7x)		MPIIO shm (99.9x)	MPIIO shm (20.9x)	MPIIO shm (2533x)		POSIX shm (118x)	MPIIO shm (35.4x)	POSIX shm (23.1x)
13		intra node	4KB			STDIO shm (3.3x)	POSIX bb (8.1x)			STDIO shm (3.2x)	POSIX bb (5.9x)			STDIO shm (3.1x)	POSIX bb (5.9x)
14			64KB			POSIX shm (4.8x)	POSIX shm (2.2x)			POSIX shm (6.2x)	POSIX shm (5.6x)			POSIX shm (10x)	POSIX shm (6.3x)
15			1MB			POSIX/ STDIO shm (6x)	STDIO shm (6.5x)			POSIX shm (13.3x)	POSIX shm (1992x)			POSIX/ MPIIO shm (5.9x)	POSIX/ STDIO shm (7.6x)
16		inter node	4KB			STDIO pfs (1.5x)	STDIO pfs (1.5x)			STDIO pfs (1.7x)	STDIO pfs (1.5x)			STDIO pfs (1.5x)	STDIO pfs (1.5x)
17			64KB			POSIX pfs (1.3x)	STDIO pfs (1.4x)			POSIX pfs (1.4x)	STDIO pfs (1.1x)			POSIX pfs (1.3x)	POSIX pfs (1.4x)
18			1MB			POSIX/ STDIO pfs (1.7x)	STDIO pfs (1.9x)			STDIO pfs (2.4x)	MPIIO pfs (164x)			STDIO pfs (4.2x)	STDIO pfs (2.4x)

consist of two IBM POWER9 CPUs (IBM AC922 servers) with 256GB of system memory and four Nvidia Volta V100 GPUs with 64GB of HBM2 memory.

2) Software: We developed a micro-benchmark to simulate an HPC workload with the following four characteristics. First, *workload's nature* defines the characteristics of I/O of the workload: write-only, read-only, read-after-write (RAW), and write-once-read-many (WORM). Second, *file access pattern* specifies whether the file is accessed sequentially or randomly. Third, *process-data sharing* defines the collection of processes that share a file, such as decoupled access across nodes or collective access through MPI processes (MPI-IO) and UnifyFS [32]. Fourth, *transfer size* defines the I/O request size. We ran the benchmark using different storage targets: shared memory (shm), burst buffer (bb), and parallel file system (PFS). In each case, we flush the final data (if updated) or prefetch the data (if read) to or from the PFS from or to a storage accelerator. To understand the application performance, we separately measure metadata operations such as open and close, data operations such as read and write, and data movement operations such as flush and prefetch. Finally, to understand the impact of scalability of I/O operations, we test our benchmark on scales from 4 nodes to 128 nodes. We perform each test ten times and use the average to reduce I/O variability.

3) Results: We present a subset of the results of our study [33] in Table I. For these specific results, we highlight the findings across the complete dataset collected during benchmarking. This benchmarking across the available storage infrastructure was performed once for the Lassen machine, and then users could transparently utilize the complex hardware and software choices within the infrastructure. In some instances, two solutions are equivalent; those solutions are marked with "/" in Table I.

Different workloads perform better with different I/O interfaces. For instance, for a transfer size of 1MB (Row 3) for 128 nodes, write-only workloads perform 7.5× faster with MPI-IO, read-only with POSIX is 75× faster, and RAW workload with STDIO is 15× faster (due to better write-back caching in STDIO). Also, different workloads perform better with different storage accelerators. For instance, write-only workloads with independent process access execute 6.3× faster with shared memory across the nine test cases. In contrast, frequent read operations with 4KB transfer size run 4.8× faster with shared memory.

The storage accelerator selection depends on the location of processes sharing the file. For instance, if the same process or processes on the same node read or write the file, then the workload should utilize shared memory (3.2×-11.2× faster). Otherwise, we should utilize shared file systems, such as parallel file systems, to share files among processes of different nodes (1.1×-4.2× faster). Note that we wrote data to shared memory for shared file access and flushed it asynchronously to the PFS, and we asynchronously prefetched the data to the other node's shared memory for reading. This case increases the access latency as the consuming process needs to wait for data to be available on the PFS before reading it. Different process placements do not greatly impact the choice of the I/O interface. The choice of the I/O interface depends on other factors, as described in other parts of the analysis.

Changing the access pattern for the workload does not

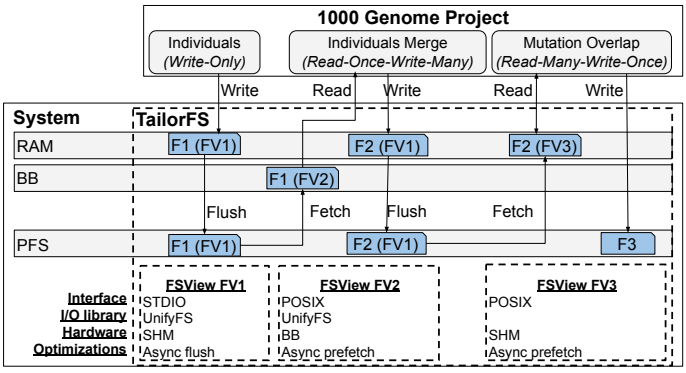


Fig. 3. In the Genome Project example, different applications have different access patterns in producer/consumer patterns. TailorFS's FSViews can enable different storage features for different applications on the same file dynamically. This leads to near-optimal I/O performance on HPC systems.

change the choice of storage accelerators between shared memory and node-local burst buffer. The performance and available capacity difference between these devices overall dictate the choice. Similarly, all these I/O interfaces used in this study treat data as bytes of data, and therefore they do not favor one pattern over another.

Different I/O interfaces optimize different transfer sizes. For instance, in 128 nodes, 4 KB transfer sizes run 3.1×-5.5× faster with the STDIO interface due to aggressive buffering of I/O requests, 64KB transfer sizes generally run 1.1×-75× faster with the POSIX interface, and large 1MB transfer sizes are 3.9×-35.4× faster with the MPIIO interface. Different transfer sizes have different effects on the performance of device caches. For instance, node-local burst buffer does not perform well with STDIO (4.2× slower), as the double buffering effect on large data reduces I/O performance.

Increasing the number of nodes for the workload increases the benefit of using node-local devices over parallel file systems. However, the best interface and storage accelerator selection does not change. The performance difference with different interfaces increases as we scale the workload. For instance, for random access and single process (Row 11), the difference between POSIX and STDIO interface on shared memory is only 3% for 64 nodes but is 12% for 128 nodes. Note, we do not see any case where UnifyFS outperforms MPI-IO for collective I/O cases. This is because the benefits of UnifyFS is seen at scales larger than 512 nodes.

Finally, for certain cases throughout this table, multiple interfaces can be used over the same storage accelerator with minimal performance difference. For instance, in Row 3, we can use MPI-IO and POSIX over shared memory for write-only or read-only workloads. In this case, the difference in performance between these interfaces is less than five percent. As we increase the scale, the difference in performance between interfaces increases more. In the cases of multiple options, the engine selects the interface that is closest to the workload's original interface to reduce the cost of translation between different I/O interfaces.

D. Working Example

To demonstrate the working of the TailorFS library, we take an illustrative example of the 1000 Genomes workflow. We depict a portion of the workflow so that we can demonstrate the benefit of FSView abstraction for the workload (Figure 3). In this example, we show three applications (out of seven) that demonstrate different I/O characteristics and therefore have different I/O requirements for the underlying file system. The first application is *Individuals* which is a write-only application generating 100 million "F1" files with a small transfer size of <4KB. This application requires the file system to buffer the small write and then flush the complete files to the parallel file system. Therefore, TailorFS creates an `FSView` "FV1" that utilizes shared memory to buffer all incoming small writes and then asynchronously flush the files on close. The second application is *Individuals Merge* which reads the files generated by *Individuals* application once and generates a merged file "F2" with multiple read and write operations. In this case, as the *Individuals Merge* application reads the "F1" files once, the application requires the file system to use a large cache to keep these files for access to the application while keeping node-local shared memory available for fast writing of the merge file. Therefore, TailorFS creates `FSView` "FV2" that utilizes burst buffer (BB) for file "F1", where it asynchronously prefetches the file during application initialization and reads the data directly from the BB. For writing the merged file F2, TailorFS utilizes the `FSView FV1`, which again writes the merged files into shared memory and asynchronously flushes them into PFS. The third *Mutation Overlap* application reads the merged files multiple times and outputs the final data once. In this case, the application requires the file system to cache the merged files in the fastest layer for the lowest access latency, while the final output needs to be stored in the final resting file system, like PFS. In this case, TailorFS creates `FSView FV3`, which prefetches all merged files into the shared memory of the compute node where *Mutation Overlap* application is running. This makes the reading part of the workload (which dominates the I/O time) extremely optimized. As the final output needs to be stored on PFS, TailorFS does not optimize F3 and writes it directly to PFS. These examples demonstrate that the same files can be provided with different storage features based on the workload's I/O requirements.

E. Design Discussions

1) **Generality of the benchmarking:** The benchmarking module of TailorFS aims to capture the strengths of the hardware and software components within a storage infrastructure. This will allow users to transparently utilize the best solution without changing the user code. The benchmarking module would be executed once for the machine and then TailorFS can optimize user workloads.
2) **Handling I/O semantics:** I/O semantics, such as fault tolerance and consistency, are inherited from the software and hardware present in the storage infrastructure. TailorFS does not explicitly provide any new semantics or I/O interfaces. It bridges the gap between the existing workload's

I/O interfaces and the storage infrastructure provided by new HPC systems.

3) **Need for profiling data:** The profiling data is needed for fully transparent optimization using TailorFS. The profiling data can focus on specific characteristics that define the storage architecture of an HPC system. For example, on Lassen, workloads were characterized based on workload's nature (write-only), access pattern (sequential), process data sharing (file-per-process), transfer size distribution, and workload scale. This data can be easily extracted from I/O profiling tools such as Darshan [34] and Score-P [35].

F. Implementation

The TailorFS library [36] is written in C++ with I/O interceptors built using the Gotcha library [37]. To extract workload's I/O characteristics, we use Vani Tool Suite [30] that executes python-based analysis scripts over Recorder [38] traces. We chose Recorder to gather comprehensive traces of the workloads because Recorder is the only tracing tool that provides multilevel I/O traces along with CPU and GPU calls. We use multicomponent traces (I/O, CPU, and GPU) to perform complex data-dependency analysis on applications and workflows.

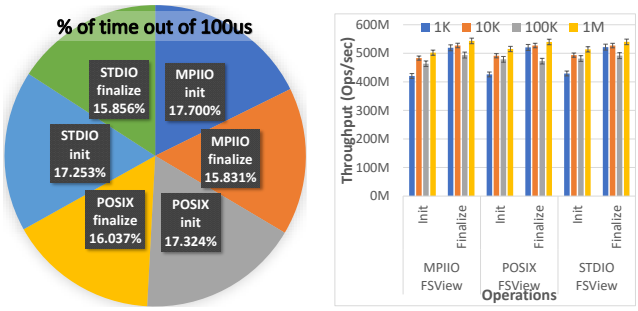
For intercepting I/O calls from the workflow in a low-cost and extensible manner, we utilize the Gotcha library. Within this implementation, not all applications intercept all I/O calls. Instead, we use the I/O behaviors extracted from recorder traces to only intercept the calls that are relevant to the application and optimize the files that the FSViewManager selected, which lowers the overhead of intercepting I/O calls. As in a workflow, we may be dealing with many interfaces across multiple applications, and performing interception for I/O calls that might come for other applications can increase the binding cost at application startup. Additionally, for the implementation of the TailorFS engine, we feed the matrix of the best solution (gathered through comprehensive benchmarking in Section IV-C) to a decision tree classifier that learns the pattern of features and solution and generalizes with high accuracy.

For FSViews, we have implemented the following features. For I/O interfaces, we have implemented POSIX, STDIO, MPI-IO, and UnifyFS API. For I/O libraries, we have implemented the use of the UnifyFS library for performing collective I/O for checkpoint files in HPC simulations. For storage accelerators, we have implemented support within the TailorFS library for shared memory, node-local burst buffer, and GPFS on the Lassen supercomputer. Finally, for I/O optimizations, we have implemented asynchronous flush and prefetching of files from and to storage accelerators. In the future, we will implement optimizations such as hierarchical prefetching [39], compression [5], and replication [40].

V. Evaluations

A. Methodology

To evaluate the benefit of building FSViews for large-scale HPC workloads, we performed comprehensive experiments to test the internal capabilities of the TailorFS library (with a particular focus on `FSViewManager`). Then, we evaluated the end-to-end performance of workloads using the TailorFS library using synthetic experiments that test different types of workloads with 256 nodes. Finally, to demonstrate the use of the TailorFS library on a large-scale HPC system, we evaluate the 1000-genome workflow with the Pegasus Workflow Manager (PWM) on the Lassen supercomputer. We ran these tests five times, and the variance in the data is noted in the figures.



(a) Anatomy (b) Performance

Fig. 4. Management of FSViews by FSViewManager incurs a overhead of less than 0.01% of the overall execution time (Figure a). The throughput of creation and destruction of FSViews is 498M ops/sec suggesting it will not bottleneck I/O operations.

1) Hardware: We ran the workloads on the Lassen supercomputer at Lawrence Livermore National Laboratory (LLNL) [31].

2) Workloads: We use micro-benchmarks, described within each section below, to showcase different performance metrics on the designed components and end-to-end performance of different workloads. Additionally, we run the 1000 Genomes scientific HPC workflows using PWM. The workflow that identifies mutational overlaps to find potential disease-related mutations in genome data.

3) Tools: We trace the workloads using the Recorder tool with an average overhead of 7%. Then, we use the Vani tool suite to automatically extract I/O behaviors from Pegasus workflows [41]. The tool is used with both workflows to extract I/O behavior for all files within the workflows and is stored in a JSON. This JSON file is loaded during *pegasus-mpi-cluster* job initialization within the workflow while loading the TailorFS library.

B. Performance of TailorFS Library

The TailorFS library creates and destroys FSViews within the system and redirects I/O to these FSViews. The redirection cost is $< 0.01\%$ of the operation. The former part consists of two routines: a) init: for creating FSViews and b) finalize: for destroying FSViews. To measure the overhead of creating and destroying FSViews, we measure the relative cost of operations within the system to measure the cost of routines within the library. Additionally, we measure the overall throughput of operations in the library to ensure it is not a bottleneck for HPC storage systems. In all these tests, we observe a memory footprint of 6.84 KB per node on the Lassen supercomputer with thousands of FSViews created by the TailorFS library.

1) Anatomy of managing FSViews: To ensure that creation and destruction of FSViews are lightweight (as explained in

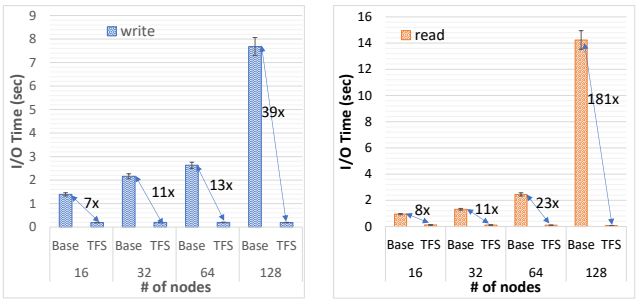
Subsection IV-B), we create a micro-benchmark that runs on 32 compute nodes with 40 processes on each node. Each process creates and destroys a million FSViews with different I/O interfaces such as STDIO, POSIX, and MPI-IO. We calculate the relative cost of each operation (shown in Figure 4(a)). We observe that the creation and destruction of FSViews for all our implemented interfaces is $\approx 15\mu$s. The initialization cost is $\approx 2\mu$s as it creates a background thread for prefetching operation. In all of our tests, this overhead is less than 0.01% of the overall execution time which demonstrates that TailorFS's FSViewManager can efficiently deploy different FSViews for the workload with negligible overhead.

2) Performance of managing FSViews: To ensure that the creation and destruction of FSViews by the TailorFS library does not bottleneck modern storage systems, we create a micro-benchmark that measures the throughput of operations. We run our micro-benchmark on 32 nodes, with each process producing 1K, 10K, 100K, and 1M operations, and we calculate the system's overall parallel throughput. The results are shown in Figure 4(b). The x-axis depicts different FSViews with their operations. The y-axis shows the throughput in operations per second. We see that all operations scale well as we increase the number of operations because FSViews are created and destroyed locally to the process. We also observe that we can achieve a throughput of 474M ops/sec and 518M ops/sec average for initialization (init in the figure) and finalization (finalize in the figure) operations within any FSView. This throughput suggests that the TailorFS library will not become a bottleneck for storage systems and user applications, as I/O operations achieve throughput in thousands ops/sec for node-local burst buffers (the fastest storage accelerators).

C. Benefit of FSViews for workloads

HPC workloads have different requirements for the I/O system based on workload type, access pattern, process file sharing, and scale. To understand the effects of using a customized FSView for each of these cases, our micro-benchmark isolates different cases of I/O behavior and measures the benefit of TailorFS on these workloads. We tested 3000 test cases, but we chose a subset of those results to summarize the findings of these tests. Specifically, we include all the workload types, such as write-only, read-only, read-after-write (RAW), and write-once-read-many (WORM) workloads.

Among our different transfer size cases, we present a 1MB transfer size in these figures. The benefit of TailorFS for smaller transfer sizes i.e., 4KB and 64KB are orders of magnitude higher than running directly on GPFS. Also, for RAW and WORM patterns, we tested three process file-sharing grouping such as file-per-process (FPP), intra-node file sharing, and inter-node file sharing. Again we chose inter-node file sharing to demonstrate some findings that are not present in write-only and read-only workloads. Finally, among the access pattern sequential access pattern performs better on GPFS than random, therefore, we selected sequential access as our chosen cases to demonstrate the benefit of TailorFS. Note, if we choose other cases such as random access pattern, smaller



(a) Write Only FPP (b) Read Only FPP

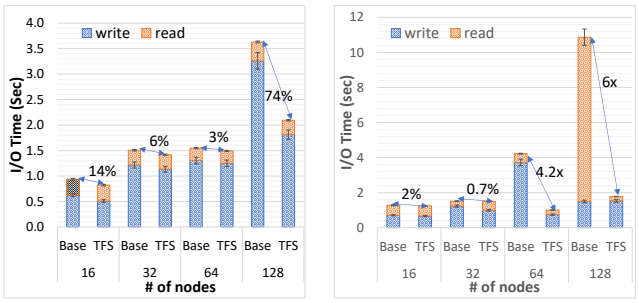
Fig. 5. Benefit of FSViews for 1MB requests with sequential accesses with file-per-process pattern

transfer size, or intra-node process grouping, the benefits of TailorFS are 48× more at an average due to the benefit of utilizing storage accelerators.

For each case, the baseline ("Base") workload uses the POSIX interface over GPFS and TailorFS ("TFS"). The workload preloads the TailorFS library. On each node, we execute 40 processes for the workload and each process performs 1024 I/O operations of each type. The x-axis shows the weak scaling of the workload from 16 nodes to 128 nodes along with the solutions tested and the y-axis shows the I/O time for performing read or write operations by the workload in seconds.

1) Write Only Workload: In this test case, we run a write-only workload emulating scientific simulation [34] workloads where every process sequentially writes checkpoints independently of its process memory (Figure 5(a)). As we increase the number of nodes (weak scaling), the cost of I/O increases for the baseline (Base in the Figure). For TailorFS, the FSView is configured to use MPI-IO interface with independent I/O calls over shared memory with asynchronous flushing. This increases the performance of write operations by up to 39× in the largest scale. Unlike this test where workload performs 1 MB request, if the workload performed 4 KB transfer size, we observed a maximum speed up of 58× for 128 node case. This is because utilizing shared memory for small transfer size accesses can provide more speed up over GPFS than for large transfer size accesses.

2) Read Only Workload: In this test case, we run a read-only workload emulating scientific AI workloads [25] where every process sequentially reads image data serialized into TFRecords independently (Figure 5(b)). As we increase the number of nodes (weak scaling), the cost of I/O increases for the baseline (Base in the Figure). For TailorFS, the FSView is configured to use MPI-IO interface for 16 and 32 nodes and POSIX for 64 and 128 nodes. Additionally, the FSViews perform data prefetching on shared memory in all the cases. Once the data is prefetched into the shared memory, the read operations within TailorFS are extremely fast. Even the prefetching is efficient as we can utilize more nodes to prefetch for larger scales. Also, moving a file is faster than performing small I/O operations on a file. This characteristic of the FSView within TailorFS results in a speedup of 8-181× for read-only workloads. This speedup further increases for random accesses to up to 320× for 128 nodes. This is because



(a) RAW Inter-node PC (b) WORM Inter-node PC

Fig. 6. Benefit of FSViews for 1MB requests with inter-node data sharing

GPFS is more susceptible to random accesses than node-local shared memory. Also, similar to the write-only case doing I/O with a smaller granularity of 4KB reduces PFS performance resulting in more than 530× speedup in the best case.

3) Read-After-Write Workload: In this test case, we run a read-after-write workload emulating checkpoint-restart workloads in scientific simulations where a producer process sequentially writes checkpoints and then a consumer process, on a different node, performs a restart operation by reading the checkpoint back (Figure 6(a)). As we increase the number of nodes (weak scaling), the cost of I/O increases for the baseline (Base in the Figure). For TailorFS, the FSView is configured to use the GPFS directly using STDIO interface. Coincidentally, this interface and storage performance for a smaller scale matches that of POSIX over GPFS. As shown in internal evaluations, the small overhead of TailorFS, even in these cases, ensures TailorFS is equal to or better than the baseline in all cases. However, for 128 node cases, you can over 74% better I/O performance as the FSView uses MPI-IO. This test case demonstrates the power of dynamically creating workload-aware FSViews so that users don't have to change code based on the scale of the workload. For the test case where the producer and consumer processes coincide or are located in the same node, TailorFS can achieve up to 44× and 21× speedup respectively due to the utilization of node-local storage accelarators.

4) Write-Once-Read-Many Workload: In this test case, we run a write-once-read-many workload emulating HPC data analytics workloads where the map stage (consumer processes) repeatedly read file many times and finally the reduce stage (or producer) on a different node writes the final results of the analytics to the parallel file system (Figure 6(a)). As we increase the number of nodes (weak scaling), the cost of I/O increases for the baseline (Base in the Figure). For TailorFS, the FSView is configured to use POSIX for 16 and 32 nodes and the STDIO interface for 64 and 128 nodes over GPFS. For smaller scales, the performance of the STDIO interface over GPFS is similar to POSIX therefore to reduce overhead TailorFS uses the POSIX interface itself rather than changing the interface. This creates a similar performance for both scales (with a variability of 7%). However, for larger scales, using STDIO can greatly benefit the workload due to buffering performed by the STDIO interface and therefore, TailorFS achieves a speed up of up to 6x for 128 nodes scale. Again this reinforces the benefit of dynamically building FSViews for workload-specific I/O requirements.

D. Complex multistage workflow performance

We evaluate the concept of FSViews in TailorFS on the 1000 Genome workflow. The workflow is executed with PWM and GPFS as the baseline. We trace the workflows and extract I/O behavior information in JSON format for the optimized version. This JSON file is loaded into the executables using LD_PRELOAD during TailorFS library initialization. In these tests, we measure the cost of I/O on each file, categorized into read-only, FPP, and producer/consumer files. Additionally, we include "other" to signify cases that do not fall under these optimizations. The behavior of 1000 Genomes workflow is a good representation of complex applications such as CM1, HACC, VPIC, BDCATS, and Montage, where different files are accessed with different requirements for the file system.

The 1000 Genomes project provides a reference for human variation, using the reconstructed genomes of 2,504 individuals across 26 different populations to energize these approaches. This workflow identifies mutational overlaps using data from the 1000 Genomes project to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. The workflow fetches, parses, and analyzes the project data, cross-matching the extracted data (which person has which mutations) with the mutation's sift score (how bad they the mutations are). It then performs a few analyses, including plotting. We run the 1000-genome workflow with Pegasus over GPFS.

For 1000 Genome: The Pegasus Workflow Manager (PWM) has the process-task dependency that provides a DAG, the number of tasks (1736 processes), tasks per node (40), data configuration (shared file system), and scratch dir location (/p/gpfs1). It has six applications coordinating the workflow. The process distribution between different applications: individuals-1632, frequency-62, mutation overlap-56, and others. Additionally, the distribution of I/O between applications: individuals perform 70TB and frequency perform 500GB of read/write operations. In this workflow, the read operations dominate the I/O (72TB of reads vs. 100GB of writes). Finally, the distribution of patterns within the workflow is 34% of read-only files, 34% of independent reads, and 32% shared access on files.

We see improvement in performance in both cases because the amount of work is the same as we increase the number of nodes in both baseline and the TailorFS (Figure 7). For read-only files, some files with ≈4 KB transfer size use an FSView with STDIO interface and other files with files ≈64 KB transfer size use a POSIX interface with prefetching on shared memory. For Independent read access files with ≈1 MB transfer size, FSView uses MPI-IO with burst buffer configuration, whereas for Independent write access files with ≈4 KB transfer size, use STDIO interface over shared memory. For producer-consumer files shared between different nodes (Shared files in the figure), for writes, FSView utilizes MPI-IO, whereas, for reads, FSView utilizes POSIX

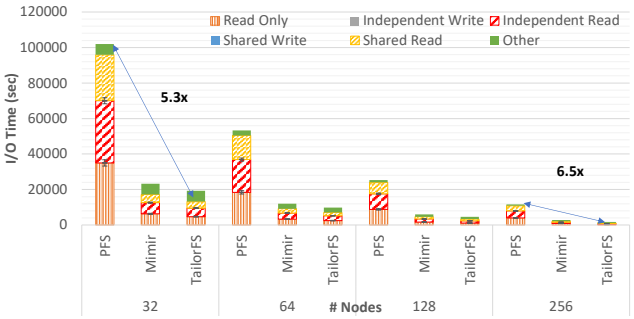


Fig. 7. Genome workflow is optimized using the TailorFS library by 5.2×-6.5× overall with 4.7×-6.2× for shared file accesses, 5.8×-9.8× for independent file accesses, and 8×-12× for read-only files.

over GPFS. Finally, for producer-consumer files shared on the same node (Shared files in the figure), for writes, FSView utilizes STDIO, whereas, for reads, FSView utilizes POSIX over shared memory. Specifically, the files with shared file accesses are optimized by 4.7×-6.2×, independent file accesses are optimized by 5.8×-9.8×, and read-only files are optimized by 8×-12×. Additionally, the dynamic selection of FSViews for every group of files results in workload-aware optimization for each file and application within the workflow. Specifically, the 1000 Genome workflow is optimized by 5.3-6.5× as compared with GPFS baseline.

VI. Related Work

A. Composable storage services

To handle the diversity of I/O requirements in HPC workflows, scientists have proposed data-oriented microservices to build custom storage infrastructures by composing a set of services for data management and exchange. The state-of-the-art composable storage services are BESPOKV, Faodel, Malacology, and Mochi. A common goal between these software solutions is to provide a highly targeted storage semantic to the HPC workflow. BESPOKV [42] is a framework targeted towards defining highly-specialized key-value stores by featuring capabilities such as sharding, replication, different consistency models, and fault tolerance. Faodel [43] is a framework that composes of a collection of data management services such as key-blob abstraction through Kelpie, memory management through Lunasa, and communication management using Opbox. Finally, the Mochi framework [44] enables the composition of specialized distributed data services from a collection of connectable modules and subservices. Rather than forcing all applications to use a one-size-fits-all data staging and I/O software configuration, Mochi allows each application to use a data service specialized to its needs and access patterns. The goal of building workload-aware storage infrastructure demonstrated by these solutions resonates with TailorFS. However, one key difference is that instead of building many composable storage services, we provide a singular dynamic FS View abstraction that contains specific features for the workload. In practice, TailorFS can be a component in these storage service infrastructures to provide tunable file system features for HPC workloads.

B. Temporary file systems

Scientists have proposed several temporary file systems to share data between consumers and producers for HPC workflows. State-of-the-art examples of these technologies are GekkoFS, UnifyFS [32], Hermes [8], and Proactive Data Containers [45] (PDC). GekkoFS [46] implements a temporary and highly scalable file system providing relaxed POSIX semantics tailored to the majority of HPC applications. This type of specialization allows applications using the existing POSIX interface (under specific constraints) to see dramatic performance improvements as compared with file systems supporting the complete specification. FSView provides a file system view with specific features/configurations. We can use any of the above-mentioned temporary file systems to achieve this view. This work aims to provide an application-level abstraction to use file systems with different features dynamically. To demonstrate our compatibility with existing Temporary file systems, we integrated UnifyFS for accelerating workloads that utilize collective I/O or perform frequent checkpoint-restart.

VII. Conclusion

In this work, we presented TailorFS: a middleware library that provides FSViews as a file system abstraction to dynamically optimize I/O for different parts of the HPC workload. We have three main observations in this work. First, FSViews need to encapsulate hardware, software, and I/O optimization as features to support diverse I/O requirements. Second, FSViewManager requires workload's I/O characteristics to efficiently manage the FSView's lifecycle. With this information, the FSViewManager can achieve a high 518M Ops/s throughput for creating and destroying FSViews, ensuring that FSView management is not the bottleneck in the distributed system. Third, the presence of FSViews with workload-aware features enables optimization of I/O by up to 46× in certain cases while incurring a low overhead of 0.01% in FSView management and a memory footprint of 7KB per node. In conclusion, we demonstrate the benefit of using FSView abstraction through the TailorFS library for 1000 Genomes workflow using the Pegasus workflow manager, resulting in up to 6.5× better I/O performance.

In the future, we plan to research on mutating FSViews through in-situ operations such as data filtering and compression. Additionally, we will explore the impact and benefit of synchronizing FSViews across multiple global storage systems, such as PFS and S3, as seen in modern supercomputers. Finally, we will explore different machine learning models to map between I/O characteristics and FSView features.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under the DOE Early Career Research Program (LLNL-CONF-854771).

REFERENCES

[1] M. Mercier, D. Glesser, Y. Georgiou, and O. Richard, "Big data and HPC collocation: Using HPC idle resources for Big Data analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 347–352.

[2] H. Luu, B. Behzad, R. Aydt, and M. Winslett, "A multi-level approach for understanding I/O activity in HPC applications," in *2013 IEEE International Conference on Cluster Computing*, Sep. 2013, pp. 1–5.

[3] F. Chowdhury, Y. Zhu, F. Di Natale, A. Moody, E. Gonsiorowski, K. Mohror, and W. Yu, "Emulating I/O Behavior in Scientific Workflows on High Performance Computing Systems," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop*, Nov. 2020, pp. 34–39.

[4] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright, "Architecture and Design of Cray DataWarp," 2016. [Online]. Available: https://shorturl.at/5pQ2Q

[5] 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)*, May 2020, pp. 557–566, iSSN: 1530-2075.

[6] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: Association for Computing Machinery, Jun. 2009, pp. 39–48. [Online]. Available: https://doi.org/10.1145/1551609.1551618

[7] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, and C. P. Steffen, "UnifyFS: A Distributed Burst Buffer File System - 0.1.0," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. UnifyFS, Oct. 2017. [Online]. Available: https://www.osti.gov/biblio/1408515

[8] A. Kougkas, H. Devarajan, and X.-H. Sun, "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*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 219–230.

[9] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 134–144, iSSN: 2168-9253.

[10] A. Kougkas, H. Devarajan, and X.-H. Sun, "IRIS | Proceedings of the 2018 International Conference on Supercomputing." ACM, pp. 33–42. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3205289.3205322

[11] K. Feng, X.-H. Sun, X. Yang, and S. Zhou, "SciDP: Support HPC and Big Data Applications via Integrated Scientific Data Processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 114–123, iSSN: 2168-9253.

[12] K. Feng, "A Multi-Level Data Integration Approach for the Convergence of HPC and Big Data Systems - ProQuest," 2020.

[13] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. Weil, "Ceph as a scalable alternative to the hadoop distributed file system," *;login:: the magazine of USENIX & SAGE*, vol. 35, no. 4 (AGO), pp. 38–49, 2010, publisher: USENIX Association Section: ;login:: the magazine of USENIX & SAGE. [Online]. Available: https://dialnet.unirioja.es/servlet/articulo?codigo=4956945

[14] H. Devarajan, A. Kougkas, H. Zheng, V. Vishwanath, and X.-H. Sun, "Stimulus: Accelerate Data Management for Scientific AI applications in HPC," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022, pp. 109–118.

[15] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "LBANN: livermore big artificial neural network HPC toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ser. MLHPC '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 1–6. [Online]. Available: https://doi.org/10.1145/2834892.2834897

[16] W. Yu, K. Mohror, and A. Moody, "User-level File Systems Specialized for HPC Workloads Year-End Report," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. LLNL-SR-764802, Dec. 2018. [Online]. Available: https://www.osti.gov/biblio/1544466

[17] D. Milojicic, P. Faraboschi, N. Dube, and D. Roweth, "Future of HPC: Diversifying Heterogeneity," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, Feb. 2021, pp. 276–281.

[18] 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)*, May 2021, pp. 81–91.

[19] J. Kwon, N. L. Kim, M. Kang, and J. WonKim, "Design and Prototyping of Container-Enabled Cluster for High Performance Data Analytics," in *2019 International Conference on Information Networking (ICOIN)*, Jan. 2019, pp. 436–438, iSSN: 1976-7684.

[20] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "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*, Nov. 2015, pp. 1–12.

[21] A. S. Shirkhorshidi, S. Aghabozorgi, T. Y. Wah, and T. Herawan, "Big Data Clustering: A Review," in *Computational Science and Its Applications – ICCSA 2014*, ser. Lecture Notes in Computer Science, B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, Eds. Cham: Springer International Publishing, 2014, pp. 707–720.

[22] P. Xenopoulos, J. Daniel, M. Matheson, and S. Sukumar, "Big data analytics on HPC architectures: Performance and cost," in *2016 IEEE International Conference on Big Data*, Dec. 2016, pp. 2286–2295.

[23] S. A. Jacobs, B. Van Essen, D. Hysom, J.-S. Yeom, T. Moon, R. Anirudh, J. J. Thiagaranjan, S. Liu, P.-T. Bremer, J. Gaffney, T. Benson, P. Robinson, L. Peterson, and B. Spears, "Parallelizing Training of Deep Generative Models on Massive Scientific Datasets," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2019, pp. 1–10.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," 2016, pp. 265–283.

[26] J. Y. Moon, F. Di Natale, H. I. Ingolfsson, H. Bhatia, and J. R. Chavez, "Multiscale Machine-Learned Modeling Infrastructure RAS," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. MuMMI RAS, Sep. 2021.

[27] E. Deelman, G. Mehta, G. Singh, M.-H. Su, and K. Vahi, "Pegasus: Mapping Large-Scale Workflows to Distributed Resources," in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer, 2007, pp. 376–394.

[28] N. Siva, "1000 genomes project," *Nature Biotechnology*, vol. 26, no. 3, pp. 256–257, Mar. 2008, publisher: Nature Publishing Group. [Online]. Available: https://github.com/pegasus-isi/1000genome-workflow

[29] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 33–44. [Online]. Available: https://doi.org/10.1145/2749246.2749269

[30] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads," in *The 2022 IEEE International Conference on Cluster Computing*. Heidelberg, Germany: IEEE, Sep. 2022.

[31] "Lassen High Performance Computing." [Online]. Available: https://hpc.llnl.gov/hardware/platforms/lassen

[32] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 290–300, iSSN: 1530-2075.

[33] H. Devarajan, "TailorFS: Comprehensive benchmarking results." Sep. 2024. [Online]. Available: https://github.com/hariharan-devarajan/tailorfs/blob/master/scripts/paper_motivation.csv

[34] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O Characterization with Darshan," in *2016 5th*

Workshop on Extreme-Scale Programming Tools (ESPT), Nov. 2016, pp. 9–17.

[35] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer, 2012, pp. 79–91.

[36] H. Devarajan, "TailorFS: An Adaptive file system to support dynamic I/O requirements of HPC workloads," May 2024, original-date: 2022-06-17T05:36:12Z. [Online]. Available: https://github.com/hariharan-devarajan/tailorfs

[37] D. Poliakoff and M. Legendre, "GOTCHA," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. GOTCHA; 005263WKSTN00, Mar. 2017. [Online]. Available: https://www.osti.gov/biblio/1356296

[38] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[39] H. Devarajan, A. Kougkas, and X.-H. Sun, "HFetch: Hierarchical Data Prefetching for Scientific Workflows in Multi-Tiered Storage Environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020, pp. 62–72, iSSN: 1530-2075.

[40] ——, "HReplica: A Dynamic Data Replication Engine with Adaptive Compression for Multi-Tiered Storage," in *2020 IEEE International Conference on Big Data (Big Data)*, Dec. 2020, pp. 256–265.

[41] "workload-aware storage Cluster'22." [Online]. Available: https://www.overleaf.com/project/624486b499b0b2ba699e5a17

[42] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. Butt, "BESPOKV: Application Tailored Scale-Out Key-Value Stores," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 14–29.

[43] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, "Faodel: Data Management for Next-Generation Application Workflows," in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, ser. ScienceCloud'18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3217880.3217888

[44] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, and Q. Zheng, "Mochi: Composing Data Services for High-Performance Computing Environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, Jan. 2020. [Online]. Available: https://doi.org/10.1007/s11390-020-9802-0

[45] S. Byna, Q. Koziol, V. Vishwanath, J. Soumagne, H. Tang, J. Mu, B. Dong, R. A. Warren, F. Tessier, T. Wang, and J. Liu, "Proactive Data Containers (PDC): An Object-centric Data Store for Large-scale Computing Systems," vol. 2018, pp. IN34B–09, Dec. 2018.

[46] 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)*, Sep. 2021, pp. 114–124, iSSN: 2168-9253.