Tarazu: An Adaptive End-to-end I/O Load-balancing Framework for Large-scale Parallel File Systems

ARNAB K. PAUL, BITS Pilani, KK Birla Goa Campus, Zuarinagar, India
SARAH NEUWIRTH, Johannes Gutenberg University Mainz, Mainz, Germany
BHARTI WADHWA, IBM Research, Yorktown Heights, USA
FEIYI WANG and SARP ORAL, Oak Ridge National Laboratory, Oak Ridge, USA
ALI R. BUTT, Virginia Tech, Blacksburg, USA

The imbalanced I/O load on large parallel file systems affects the parallel I/O performance of high-performance computing (HPC) applications. One of the main reasons for I/O imbalances is the lack of a global view of system-wide resource consumption. While approaches to address the problem already exist, the diversity of HPC workloads combined with different file striping patterns prevents widespread adoption of these approaches. In addition, load-balancing techniques should be transparent to client applications. To address these issues, we propose `Tarazu`, an end-to-end control plane where clients transparently and adaptively write to a set of selected I/O servers to achieve balanced data placement. Our control plane leverages real-time load statistics for global data placement on distributed storage servers, while our design model employs trace-based optimization techniques to minimize latency for I/O load requests between clients and servers and to handle multiple striping patterns in files. We evaluate our proposed system on an experimental cluster for two common use cases: the synthetic I/O benchmark IOR and the scientific application I/O kernel HACC-I/O. We also use a discrete-time simulator with real HPC application traces from emerging workloads running on the Summit supercomputer to validate the effectiveness and scalability of `Tarazu` in large-scale storage environments. The results show improvements in load balancing and read performance of up to 33% and 43%, respectively, compared to the state-of-the-art.

CCS Concepts: • **General and reference** → **Performance**; • **Computer systems organization** → **Grid computing**; **Secondary storage organization;**

A. K. Paul and S. Neuwirth contributed equally to this article.

This work has been sponsored in part by the National Science Foundation under grants CCF-1919113, CNS-1405697, CNS-1615411, CNS-1565314/1838271 OAC-1835890, CSR-2312785, CSR-2106634/2312785, and CCF-1919113/1919075. This research also used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725. We also acknowledge the support of EUPEX, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under GA No 101033975. The JU receives support from the European Union's Horizon 2020 research and innovation programme, France, Germany, Italy, Greece, United Kingdom, Czech Republic, and Croatia. This work is also sponsored by the grants in BITS Pilani - BBF/BITS(G)/FY2022-23/BCPS-123, GOA/ACG/2022-2023/Oct/11, and BPGC/RIG/2021-22/06-2022/02.

Authors' addresses: A. K. Paul, BITS Pilani, Birla Institute of Technology and Science, Pilani - Goa Campus, Zuarinagar, Goa - 403726, India; e-mail: arnabp@goa.bits-pilani.ac.in; S. Neuwirth, Johannes Gutenberg University Mainz - Zentrum fuer Datenverarbeitung, 55099 Mainz, Germany; e-mail: neuwirth@uni-mainz.de; B. Wadhwa, IBM Research - Yorktown Heights, NY 10598, United States; e-mail: wadhwa@ibm.com; F. Wang and S. Oral, Oak Ridge National Laboratory - 1 Bethel Valley Rd, Oak Ridge, TN 37830, United States; e-mails: fwang2@ornl.gov, oralhs@ornl.gov; A. R. Butt, Virginia Tech - Blacksburg, VA 24061, United States; e-mail: butta@vt.edu.

ACM 1553-3077/2024/04-ART11
https://doi.org/10.1145/3641885

Additional Key Words and Phrases: Parallel file system, progressive file layout, lustre, time series modeling

ACM Reference Format:
Arnab K. Paul, Sarah Neuwirth, Bharti Wadhwa, Feiyi Wang, Sarp Oral, and Ali R. Butt. 2024. Tarazu: An Adaptive End-to-end I/O Load-balancing Framework for Large-scale Parallel File Systems. *ACM Trans. Storage* 20, 2, Article 11 (April 2024), 42 pages. https://doi.org/10.1145/3641885

1 INTRODUCTION

Unbalanced load distribution and poor resource allocation schemes have been identified as major contributors to performance penalties in many HPC storage systems, including Lustre [10], one of the most widely used parallel file systems for scientific computing. Several recent works address the load-balancing issue. Server-side approaches [18, 67] aim to allocate resources for all concurrently running applications simultaneously. This includes the standard approach used in Lustre's request ordering system, the **Network Resource Scheduler (NRS)** [74]. Other techniques attempt to minimize resource contention on a per-application basis, i.e., client-side approaches [58, 91]. While client-side and server-side approaches work well in isolation for some applications, the diversity of HPC I/O workloads unfortunately leads to situations where both isolated approaches lose performance.

Another commonly seen but neglected aspect when managing large-scale parallel file systems with a diverse workload and different file I/O sizes is the use of poor file striping patterns [51]. In general, file striping enables parallel file I/O and ideally provides a high application I/O throughput [36]. However, often a much lower stripe count is used than recommended for large files, resulting in a poor resource allocation scheme. This ultimately can lead to an imbalanced utilization of storage components, or worst case, can cause some of the storage targets to completely fill up. These suboptimal file layouts are not necessarily the result of conscious choices by the users, but simply the result of inherited file layouts that are configured as the system-wide default. These observations coupled with the constantly increasing size of HPC systems result in an intensified I/O subsystem complexity and a decreased system reliability (i.e., mean time to failure). Therefore, sub-optimal placement of file stripes can indirectly lead to lower I/O bandwidth. Load imbalance on the storage servers and targets also has a direct effect on I/O bandwidth utilization, as a higher load on a storage server can lead to I/O and network congestion for all the I/O requests forwarded to that server. There have been client-side approaches (for example, Reference [19]) to tackle jitter-free I/O, but there has been no work that tackles the end-to-end problem including clients and storage servers to achieve better I/O bandwidth utilization. As a result, since there is no "one-file-layout-for-all," and a load-balanced set of servers and targets can lessen I/O congestion issues, a configurable and smart load-balancing framework is needed that can adapt to different file layouts, facilitate scientific code development for users, and make efficient use of extreme-scale parallel I/O and storage resources.

Previous works such as iez [87] and AIOT [103] combine the application-centric strengths of client-side approaches with the system-centric strengths of server-side approaches. For example, iez provides an application-agnostic global view of all resources to the **Metadata Server (MDS)**. This includes the current statistics of **Object Storage Servers (OSSs)** and the set of **Object Storage Targets (OSTs)** where data resides. It coordinates the I/O requests from all concurrently running applications simultaneously to optimize the I/O placement strategy on a per-client basis. However, iez and AIOT have two major drawbacks. First, the algorithm for predicting application I/O request patterns run in a centralized fashion, which limits the scalability of load-balancing frameworks. Second, both frameworks are not able to efficiently adapt load balancing to the different file layout requirements of different file sizes when running different HPC workloads simultaneously.

Our work focuses on three main areas. First, we introduce Tarazu, an end-to-end control plane, which provides an intelligent and adaptive placement algorithm that is able to tune to varying file layouts for different I/O request sizes of concurrently running applications, but also consider the current load on the file system when placing the file data. By utilizing the **Progressive File Layout (PFL)** [26, 51], a recent Lustre feature that offers significant new flexibility in optimizing file layouts for various I/O patterns and sizes, Tarazu is able to adjust the file layout, depending on the I/O size, by adapting the striping pattern as the file size increases. Tarazu combined with PFL enables efficient utilization of the parallel storage resources for varying I/O sizes and workloads. Second, we eliminate the major drawback of the centralized prediction algorithm that limits scalability by moving the prediction model to the clients instead of the MDS and forwarding all predicted file requests to the MDS for optimal file placement in a scalable manner. Third, we design, implement, and validate a discrete-event simulator to enable scalability tests for large-scale parallel storage systems and to verify the effectiveness of our proposed work.

In a nutshell, Tarazu collects real-time information from clients and servers about applications' storage requirements, as well as the load on storage servers, and maps I/O requests to OSTs and OSSs in a balanced manner to ensure efficient use of all storage system components. The system considers per-client job requests in an adaptive per-client prediction model to synchronize holistic job placement and resource allocation. Our data placement strategy supports two widely used classes of file sharing modes, i.e., **File-Per-Process (FPP)** and **Single-Shared-File (SSF)**, for both PFL and non-PFL layouts. In addition, Tarazu supports popular I/O interfaces such as POSIX I/O [88], MPI-IO [81], and HDF5 [25]. In summary, we make the following contributions:

(1) We design and implement a prediction algorithm and placement library for the end-to-end control plane Tarazu, which incorporates both server- and client-side functionality to optimize the I/O placement and resource allocation. Tarazu is able to handle multiple file striping patterns concurrently and in a scalable fashion.
(2) We evaluate the effectiveness of Tarazu on a cluster setup with two common use cases: the synthetic I/O benchmark IOR [40] and the scientific application kernel HACC-I/O [28]. The results show that Tarazu improves the load balance by up to 33% and read performance by up to 43% compared to the default load balancing adopted in Lustre.
(3) We demonstrate the effectiveness and scalability of Tarazu on large-scale storage deployments by evaluating different Lustre setups with a discrete-time simulator based on the Tarazu system design and use three real-world scientific HPC application traces collected on the Summit supercomputer [41] to model our clients and a mixed system workload.

2 BACKGROUND

Recent work [5, 18, 66, 69, 71, 91] has shown that unbalanced I/O load in HPC systems can lead to serious resource contention and degradation of overall I/O performance. The parallel I/O system and the complex path of an application's I/O request—consisting of myriad components such as I/O libraries, network resources, and back-end memory—are inherently complex. Today's HPC implementations lack a centralized, system-wide I/O coordination and control mechanism to address the overall problem of resource contention. As a result, existing parallel file and storage systems can only optimize some parts of the I/O path, but not the entire end-to-end path. In the following, we provide a brief overview of the Lustre file system and its file allocation policies. Afterwards, we discuss the relation between parallel I/O and file requests. The section concludes with a discussion about the progressive file layout and its benefits for emerging HPC workloads.

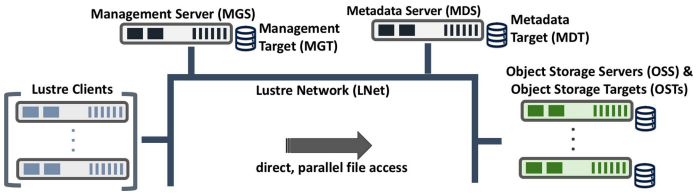


Fig. 1. Overview of the Lustre architecture.

2.1 Introduction to Lustre

We have implemented Tarazu atop Lustre, one of the most widely deployed parallel file systems in the world's top supercomputing systems [82]. For example, the world's first exascale supercomputer Frontier relies on a center-wide Lustre file system named Orion to provide three storage tiers: (1) a 480× NVMe flash drive metadata tier, (2) a 5,400× NVMe SSD performance tier with 11.5 PB of capacity, and (3) a 47,700× HDD capacity tier with 679 PB of capacity. Lustre is a file system that scales to meet the requirements of applications running on a range of systems from small-scale HPC environments up to the very largest supercomputers and has been created using object-based storage building blocks to maximize scalability.

2.1.1 Lustre Architecture. Lustre is a scalable storage platform that is based on distributed object-based storage. Figure 1 shows a high-level overview of the Lustre architecture and its key building blocks. Lustre clients provide a POSIX-compliant interface between applications and the storage servers. The application data is managed by two types of servers, ***Metadata Server* (MDS)** and ***Object Storage Server* (OSS)**. MDS manages all namespace operations and stores the namespace metadata on one or more storage targets called ***Metadata Targets* (MDTs)**. The bulk storage of contents of application data files is provided by OSSs. Each OSS typically manages between two and eight ***Object Storage Targets* (OSTs)**, although more are possible, and stores the data on one or more OSTs. OSTs are stored on direct-attached storage. Each data file is typically striped across multiple OSTs; the stripe count can be specified by the user. The distributed components are connected via the high-speed data network protocol LNet [55], which supports different network technologies, such as Ethernet and InfiniBand [72]. LNet is designed to support full RDMA throughput and zero copy communications when supported by the underlying network technology.

One of Lustre's key performance features is file striping using RAID 0, which is the process of dividing a body of data into blocks and spreading the data blocks across multiple storage devices in a redundant array of independent disks group. Striping allows segments or chunks of data in a file to be stored on different OSTs, as shown in Figure 2. The RAID 0 pattern stripes the data across a certain number of objects. The number of objects in a single file is called the *stripe count*. Each object contains chunks of data from the file, and chunks are written to the file in a circular round-robin manner. When the chunk of data being written to a particular object exceeds the configured *stripe size*, the next chunk of data in the file is stored on the next object. In Figure 2, the stripe size for file C is larger than the stripe size for file A, so more data can be stored in a single stripe for file C. File striping offers two main benefits:

(1) The *ability to store large files* by placing chunks of a file on multiple OSTs, i.e., a file's size is not limited to the space available on a single OST.
(2) An *increase in bandwidth,* because multiple processes can simultaneously access the same file, i.e., a file's I/O bandwidth is not limited to a single OST.

2.1.2 Managing Free Spaces. To provide optimized I/O performance, the MDT assigns file stripes to OSTs based on location (which OSS) and size considerations (free space) to optimize

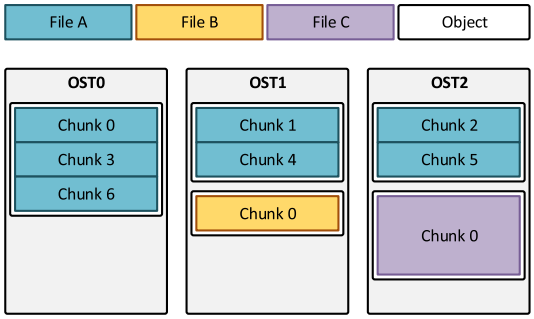
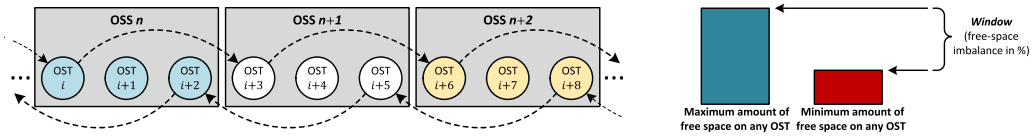


Fig. 2. Example file striping on a Lustre file system.



(a) Round-Robin Allocator. **(b)** Target Window.

Fig. 3. Lustre provides two stripe allocation methods for managing free spaces: round-robin and weighted. By default, the allocation method is determined by the amount of free-space imbalance on the OSTs.

file system performance. Less-used OSTs are preferentially selected for stripes, and stripes are preferentially spread out between OSSs to better utilize network bandwidth. The default OST load-balancing approach uses ***Lustre's standard allocation* (LSA)** policy to distribute I/O load over OSTs. Lustre comes with two stripe allocation methods: the round-robin and the weighted allocator. Depending on the *free-space imbalance* on the OSTs, Lustre transparently switches between the faster round-robin allocator, which maximizes network balancing, and the weighted allocator, which fills less-used OSTs faster by using a weighted random algorithm.

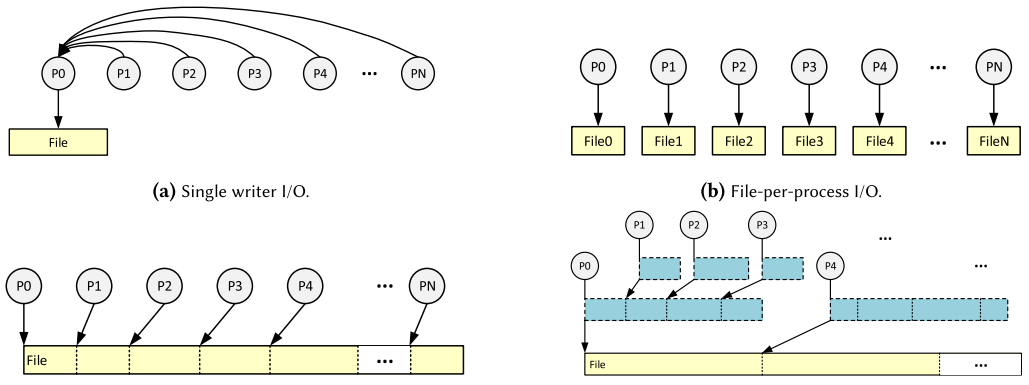
The *round-robin allocator* alternates stripes between OSTs on different OSSs, so the OST used for stripe 0 of each file is evenly distributed among OSTs, regardless of the stripe count, as depicted in Figure 3(a). Note that the list of OSTs for a file is not necessarily sequential with regards to the OST index. In contrast, the *weighted allocator* uses a weighted random mechanism to select OSTs. OSTs that are the least full have a higher probability of being allocated (in an attempt to bring the storage system back into balance), but there is still some chance that full OSTs could be selected. The *target window*, as shown in Figure 3(b), specifies the allowed free-space imbalance and defines when to switch between the two strategies. Let us assume that *max* is the maximum amount of free space on any OST in the file system, *min* is the minimum amount of free space on any OST, and *Window* defines the quality of service threshold of allowed free-space balance:

$$(max - min) \leq \frac{Window}{100} * max. \tag{1}$$

If Equation (1) is true, then the OSTs are considered balanced and the round-robin allocator is used. This means that all the OST usages are within a small window of each other (which by default is set in between 17% and 20%). The weighted allocator is used when any two OSTs are imbalanced.

2.2 Introduction to Parallel I/O and File Requests

In the context of HPC systems, parallel I/O [9, 73] describes the ability to perform multiple input and output operations at the same time, for instance, simultaneous outputs to storage and display



(a) Single writer I/O.

(b) File-per-process I/O.

(c) Single shared file I/O: independent.

(d) Single shared file I/O: collective buffering.

Fig. 4. Overview of file-sharing strategies [55].

devices. It is a fundamental feature of large-scale HPC environments. Parallel file systems distribute the workload over multiple I/O paths and components to satisfy the I/O requirements in terms of performance, capacity, and scalability. Scientific I/O is performed by large-scale applications from different scientific domains. HPC applications frequently issue I/O operations to access (i.e., read or write) TBs of data; some applications produce a few hundred TBs or even PBs of data. Typically, domain scientists think about their data in terms of their science problems, e.g., molecules, atoms, grid cells, and particles. Ultimately, physical disks store bytes of data, which makes such workloads difficult to handle for the storage system. Most HPC storage systems employ a parallel file system such as Lustre or GPFS to hide the complex nature of the underlying storage infrastructure, e.g., **solid state drives (SSDs)**, spinning disks, and RAID arrays, and provide a single address space for reading and writing to files. The I/O behavior of an application depends on multiple factors such as type of I/O operation (i.e., read or write), file-sharing strategy (single-shared-file versus file-per-process), I/O intensity, and current system load when the application is executed.

There are three common file-sharing strategies used by applications to interact with the parallel file system, which are described in the following: In *Single Writer I/O*, also known as sequential I/O, one process aggregates data from all other processes and then performs I/O operations to one or more files. The ratio of writers to running processes is 1 to N, as depicted in Figure 4(a). This pattern is very simple and can provide a good performance for very small I/O sizes but does not scale for large-scale application runs, since it is limited by a single I/O process. In *File-Per-Process I/O*, each process performs I/O operations on individual files, as shown in Figure 4(b). If an application runs with N processes, then N or more files are created and accessed (N:M ratio with $N \leq M$). Up to a certain point, this pattern can perform very well, but is limited by each individual process that performs I/O. It is the simplest implementation of parallel I/O enabling the possibility to take advantage of an underlying parallel file system. However, it can quickly accumulate many files. Parallel file systems often perform well with this strategy up to several thousands of files, but synchronizing metadata for a large collection of files introduces a potential bottleneck. Also, an increasing number of simultaneous disk accesses creates contention on file system resources. Finally, the *Single-Shared-File* pattern allows many processes to share a common file handle but write to exclusive regions of a file. Figures 4(c) and 4(d) show the independent and collective buffering variants of this strategy. In the independent variant, all processes of an application write to the same file, while in the collective buffering variant, the performance of shared file access is improved by offloading some of the coordination work from the file system to the application. The data layout

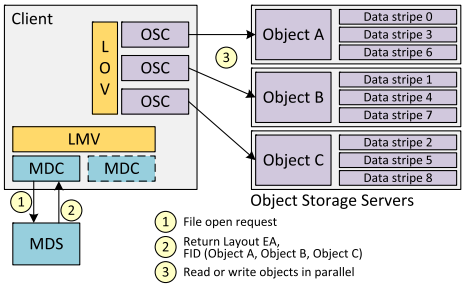


Fig. 5. Lustre client requesting file data [55].

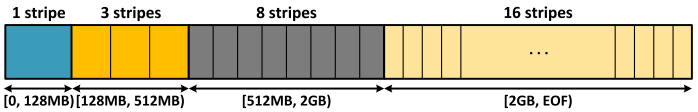


Fig. 6. An example PFL file layout with four non-overlapping extents.

within the file is very important to reduce concurrent accesses to the same region. Contesting processes can introduce a significant overhead, since the file system uses a *lock manager* to serialize the access and guarantee file consistency. The advantage of the single shared file I/O pattern lies in the data management and portability, e.g., when using a high-level I/O library such as HDF5.

Ultimately, I/O operations are translated into file requests accessing the parallel shared file system. Figure 5 uses Lustre as an example to show how an I/O operation is turned into a file request by the Lustre client running on a compute node. First, the Lustre client sends a **remote procedure call (RPC)** to the MDS via the ***logical metadata volume* (LMV)** and ***metadata client* (MDC)** to request a file lock (1). This can either be a read lock with look-up intent or a write lock with create intent. When the MDS has processed the request, it returns a file lock and all available metadata and file layout attributes to the Lustre client (2). If the file does not exist yet (i.e., file create request is performed), then the MDS will also allocate OST objects via the ***logical object volume* (LOV)** and ***object storage client* (OSC)** for the file based on the requested striping layout and current allocator policy when the file is opened for the first time. With the help of the file layout information, the client is able to access the file directly on the OSTs (3).

2.3 Progressive File Layout and Emerging Hybrid HPC Workloads

Striping enables users to obtain a high parallel I/O performance [36]. Files are divided into stripes, which are stored across multiple OSTs. This mechanism enables parallel read and write accesses to files and therefore parallel I/O. The ***Progressive File Layout* (PFL)** [51] is a recent Lustre feature where a file can have different striping patterns for different regions of the file to balance the space and bandwidth usage against the stripe count. Using PFL, a file can have several non-overlapping extents, with each extent having different striping parameters. This can provide lower overhead for small files that require only a single stripe, higher bandwidth for larger files, and wide distribution of storage usage for a very large file.

An example PFL configuration with four sub-layouts is shown in Figure 6. The first extent has a single stripe up to 128 MB, the second extent will have three stripes up to 512 MB, the third extent will have eight stripes, and the last component goes to the end of file and will have up to 16 stripes. The PFL feature is implemented using composite file layouts. The number of sub-layouts in each file and the number of stripes in each sub-layout can be specified either as a system-wide

default or by the user using the `lfs setstripe` command. For the configuration of PFL, the Lustre operations manual [61] recommends to keep a linear balance between the total number of stripes and the expected file size. Also, the layout should stop growing when the total number of stripes equals or exceeds the number of available OSTs.

HPC applications are evolving to include not only traditional scale-up modeling and simulation bulk-synchronous workloads but also scale-out workloads [57] such as advanced data analytics and machine learning [97, 100], deep learning [14], and data-intensive workflows [16, 17, 24]—challenging the long- and widely held belief that HPC workloads are write-intensive, as shown by a recent I/O behavior analysis [65]. In contrast to the traditional well-structured HPC I/O patterns (for example, checkpoint/restart, multi-dimensional I/O access), emerging workflows will often utilize non-sequential, metadata-intensive, and small-transaction reads and writes, and invoke file read requests to the HPC parallel file systems [24]. PFL has been designed to cope with the changing landscape of I/O workloads so applications observe a reasonable performance for a variety of file I/O patterns without the need to explicitly understand the underlying I/O model.

3 RELATED WORK

In this work, we seek to design an end-to-end I/O load-balancing control plane for large-scale parallel file systems. Therefore, two research areas are of particular interest for this work: *end-to-end I/O monitoring* and *resource load balancing*.

3.1 End-to-end I/O Monitoring

Existing work in end-to-end I/O monitoring has focused mainly on I/O tracing and profiling tools, which can be divided into two main categories: application-oriented tools and back-end-oriented tools. Recent research work also focuses on the end-to-end I/O path analysis, thus introducing end-to-end I/O monitoring tools as a third category.

Application-oriented tools focus on collecting detailed information about particular application runs to tune applications for increased scientific productivity or to gain insight into trends in large-scale computing systems. These tools include, for example, Darshan [12], IPM [85], and RIOT [94], all of which are designed to capture an accurate picture of application I/O behavior, including key characteristics such as access patterns within files, in the context of the parallel I/O stack on compute nodes with a minimal overhead. Patel et al. [64], for example, used Darshan to perform an in-depth characterization of access, reuse, and sharing characteristics of I/O-intensive files. Wu et al. introduced a scalable tracing and replay methodology for MPI and I/O event tracing called ScalaTrace [53, 95, 96]. Another popular tool is Recorder [49], a multi-level I/O tracing tool that captures HDF5, MPI-I/O, and POSIX I/O calls, which requires no modification or recompilation of the application. It has been extended to also support tracing of most metadata POSIX calls [89].

Back-end-oriented tools focus on collecting I/O performance data on the system-level in the form of summary statistics. Example tools include LIOProf [99], LustreDU [45, 62], and LMT [27]. Apollo [75] is a real-time storage resource monitoring tool, which relies on publisher-subscriber semantics for low latency and low overhead. Its target is to provide a current view of the system to aid middleware services in making more optimal decisions. Finally, Paul et al. [66] analyzed application-agnostic file system statistics gathered on compute nodes as well as metadata and object storage file system servers.

Finally, *end-to-end I/O monitoring tools* try to provide holistic insight from an application and system perspective, including factors such as the network, I/O, resource allocation, and system software stack. An initial attempt was to utilize static instrumentation to trace parallel I/O calls. For example, SIOX [93] and IOPin [35] extended the application-level I/O instrumentation introduced by Darshan to other system levels to characterize I/O workloads across the parallel I/O stack.

However, their overhead impedes their use in large-scale HPC production environments [78]. In recent years, end-to-end frameworks have become increasingly popular. TOKIO [39], for example, relies on the combination of front-end tools (Darshan, Recorder) and back-end tools (LMT). UMAMI [48] combines on-demand, modular synthesis of I/O characterization data into a unified monitoring and metrics interface, which provides cross-layer I/O performance analysis and visualization. GUIDE [86], however, is a framework used to collect, federate, and analyze center-wide and multi-source log data from the **Oak Ridge Leadership Computing Facility (OLCF)**. Finally, the **MAWA-HPC (Modular and Automated Workload Analysis for HPC Systems)** [108, 109] project aims to develop a generic workflow and tooling suite that can be transparently applied to applications and workloads from different science domains. Through its modular design, the workflow is able to support various community tools, which increases its compatibility with different applications. Similar to UMAMI, MAWA-HPC provides cross-layer performance analysis and visualization. Beacon [101, 102] complements previous work by providing a real-time end-to-end I/O monitoring framework. It can be used to analyze performance and resource utilization, but also for automatic anomaly detection and continuous per-application I/O pattern profiling. Beacon is currently deployed on the TaihuLight system.

When designing an end-to-end I/O control plane, key aspects such as low latency, low overhead, and an application-agnostic global view of resources play an important role in the overall system design. Therefore, this work combines different approaches from the discussed related works to provide transparent coordination of I/O requests and intelligent and adaptive placement of file I/O. For example, `Tarazu` relies on the mechanisms of the lightweight tracing tool Recorder and employs publisher-subscriber semantics to collect application-agnostic file system statistics.

3.2 Resource Load Balancing

Given typical non-uniform data allocation patterns across storage resources, the striping of application data across multiple OSTs often leads to load imbalance. The main limitation is that LSA only aims to balance the load on OSTs, without any consideration about other components, such as MDS and OSSs. Previous work on HPC I/O behavior [55, 67, 87] has shown that LSA can take a long time to balance a system, since it is unable to capture the complex behavior of modern HPC applications. Consequently, the default policy falls short of providing the desired I/O balanced storage system.

Load balancing and resource contention mitigation in large-scale parallel file system deployments are extensively studied research topics [4]. One approach is to address the problem from the client-side on a per-application basis [56, 58, 91]. For example, the I/O calls can be intercepted on the client-side during runtime and the OST assignments can be managed accordingly to mitigate resource contention [29, 44, 83, 107]. One example is TAPP-I/O [58], which transparently intercepts metadata operations, supports both statically and dynamically linked applications, and provides a heuristic-based placement strategy for FPP and SSF. However, the main limitation of these approaches is that they do not consider the requirements of other applications running concurrently on the system due to lack of a global system view and therefore only tune the I/O of individual applications.

Another approach is to have a global view of storage servers and server-side statistics and consider the load balance and job interference across all applications instead of a per-job basis. Here, the load-balancing problem is handled from the server-side perspective [18, 67, 80, 105]. The main limitations of such approaches are that they require the modification of application source code and do not consider different file I/O (SSF or FPP) and striping layouts.

Recent work [2, 6–8] has introduced auto-tuning approaches for specific high-level I/O libraries such as MPI-IO and HDF5 to learn and predict the I/O behavior of HPC applications to improve

the parallel read and write performance. Another alternative is presented by the ***Optimal Overloaded IO Protection System*** **(OOOPS)** [32], which detects and throttles I/O-intensive workloads to reduce excessive pressure on the metadata servers and service nodes. Ji et al. [33] introduced an application-adaptive ***dynamic forwarding resource allocation*** **(DFRA)**, which, based on monitoring data from the real-time I/O monitoring system Beacon [101], determines whether an application should be granted more forwarding resources or given dedicated forwarding nodes. Hence, DFRA attempts to mitigate the load imbalance at the forwarding layer and can be considered complementary to this work. In 2022, Yang et al. [103] presented an end-to-end and ***adaptive I/O optimization tool*** **(AIOT)**, which is also based on the Beacon framework. AIOT tunes system parameters across multiple layers of the storage system by using the automated identified application I/O behaviors and the instant status of the workload of storage system. The main drawback of AIOT is its centralized design for predicting and tuning I/O behavior.

The aforementioned approaches improve the parallel I/O performance for individual applications by effectively reducing the resource contention and improving the load balance but fail to exploit the opportunity of an interference-aware, end-to-end I/O path optimization. They also fail to achieve effective resource utilization (e.g., bandwidth) and performance improvements by adapting the load balance to different I/O sizes. In contrast, `Tarazu` provides an end-to-end load-balancing solution for concurrent applications for both PFL and non-PFL layouts, which globally coordinates between clients and servers of parallel file systems in a scalable and decentralized manner.

4 MOTIVATION: LOAD IMBALANCE IN DEFAULT LUSTRE SETUPS

In the following, we use two well-known HPC benchmarks to highlight the load imbalance in a default Lustre deployment and to motivate the need for a framework such as `Tarazu`.

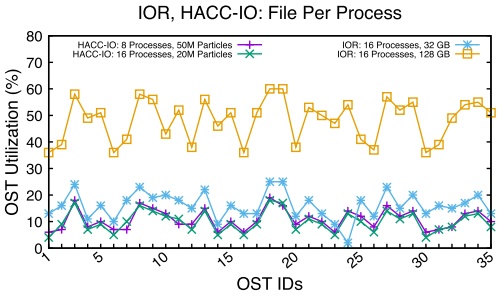
4.1 Use Cases and Benchmarks

The ***Hardware Accelerated Cosmology Code*** **(HACC)** [28] application uses N-body techniques to simulate the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. HACC-I/O mimics the I/O patterns and evaluates the performance of the HACC simulation code. It can be used with the MPI-I/O and POSIX I/O interfaces and differentiates between FPP and SSF file-sharing modes.

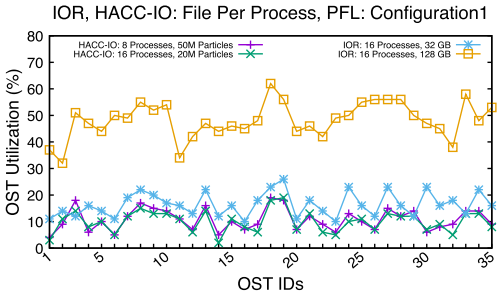
The ***InterleavedOrRandom*** **(IOR)** [40] benchmark provides a flexible way to measure the parallel file system's I/O performance under different read/write sizes, concurrencies, file formats, and file layout strategies. It measures the performance for different configurations including I/O interfaces ranging from traditional POSIX I/O to advanced parallel I/O interfaces like MPI-IO and differentiates parallel I/O strategies between file-per-process and single-shared-file approaches.

4.2 Observed Load Imbalance

To highlight the load imbalance in a default Lustre setup, we use **Lustre's standard allocation (LSA)** strategy to distribute the I/O load on the OSTs. We deployed a testbed consisting of a 10-node cluster, with 1 MDS, 7 OSSs, and 2 Clients. Each OSS manages 5 OSTs with a capacity of 10 GB each. Hence, the cluster has 35 OSTs in total with a capacity of 350 GB. IOR was run with 16 processes for two different configurations, resulting in 32 GB and 128 GB of data to be stored on the OSTs in the FPP access mode. In addition, HACC-I/O was run for 8 and 16 processes with 50 *million* and 20 *million* particles generating 14.3 *GB* and 11.7 *GB* data, respectively. All experiments were run for both the PFL and non-PFL setup. For the PFL setup, we used the same configuration as shown in Figure 6, referred to as *Configuration 1*. For the non-PFL setup, the stripe count was set to eight.



(a) Non-PFL file layout (stripe count = 8).



(b) PFL file layout (Configuration 1).

Fig. 7. OST utilization with the default OST allocation policy in Lustre (LSA) for IOR and HACC-I/O for non-PFL and PFL file layouts.

Figure 7(a) shows the OST utilization for different runs of IOR and HACC-IO in the FPP mode with the non-PFL file striping configuration. In a balanced load setting, these graphs would be straight lines, but in the studied scenario, the load is observed to be imbalanced with some OSTs getting a higher I/O load than others. A similar pattern can be seen in Figure 7(b) for IOR and HACC-I/O in the FPP mode using the PFL configuration. These results show that a default Lustre deployment, which relies on LSA to allocate OSTs for each job, can suffer from a significant load imbalance at the server-level. The load imbalance persists at different scales and different striping layouts (PFL and non-PFL) and thus can lead to imbalanced resource usage and contention.

It should be noted that the OST utilization for non-PFL and PFL files looks similar. The reason for this is how Lustre internally maps data blocks onto the stripe objects on the OSTs. By default, when the free space across OSTs differs by less than 20%, round-robin is used to distribute the file I/O across multiple OSTs. For example for HACC-I/O with FPP, 8 processes and 50 *million* particles, the non-PFL layout writes 8 stripes per file with 219 MB per stripe, while PFL divides the files in 12 stripes per file with 4 128 MB stripes and 7 192 MB stripes (the last stripe of the third extent remains unallocated, since one file is only 1.830.4 MB in size).

Regarding the observed load imbalance, the following points should be noted:

— The load imbalance on the OSTs occurs during the *file creation phase*. Each file creation request contains two parameters—the number of stripes and the file size associated with the file. Therefore, our load-balancing algorithm needs to optimally place the files during the file creation phase.

— The jagged line plot of the OST utilization in Figures 7(a) and 7(b) indicates that the load on the OSTs is not balanced during an application run. A balanced set of OSTs exhibits a straight line for OST utilization. This load imbalance on the OSTs results in an imbalance in the read and write requests coming to OSSs, which leads to I/O congestion and thus lower overall I/O bandwidth for the application. Therefore, our load-balancing algorithm should be able to balance load on both OSTs and OSSs to improve the overall application bandwidth.

— The goal of a parallel file system with load balancing should be to keep the total load of the storage targets within reasonable limits and to use all OSTs and OSSs in a similar manner so not only a certain set of OSSs and OSTs fulfills the majority of the I/O requirements. Therefore, the percentage of OST utilization should not be very close to 100%, because the storage targets that reach 100% utilization will operate slower and cause I/O bottlenecks.

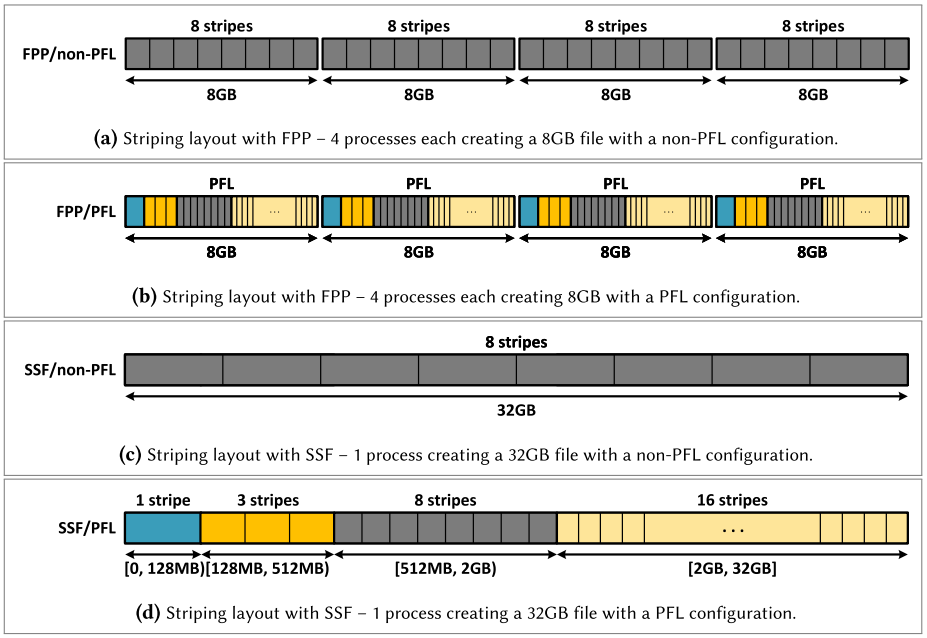


Fig. 8. Comparison between FPP and SSF with non-PFL and PFL striping layouts.

4.3 Parallel File Access with Varying Striping Layouts

Before we discuss the Tarazu software architecture in detail, we explain how the layouts of the non-PFL and PFL files differ for FPP and SSF. We focus on the example configurations shown in Figure 8. The PFL layout used is *Configuration 1*, which has already been introduced in Section 2.3.

In the FPP mode example, four processes each write 8 GB files to the parallel file system. In the non-PFL layout, each 8 GB file is split into a predefined number of equal-sized stripes (in this example, the stripe count is 8), while in PFL layout, each 8 GB file is partitioned according to the defined PFL layout for these files or directories (here *Configuration 1*). In SSF mode, a single process creates a single file while all other processes perform I/O operations on the file. This file is divided into a predetermined number of stripes according to a non-PFL or a PFL layout.

It is evident that the different stripes can be of different sizes, depending on the file-sharing mode and striping layout. Also, for non-PFL files, individual stripes can be very large, which can cause load imbalance. In addition, data segments are stored in a RAID 0 pattern during striping, as already explained in Section 2.1. This can pose a significant problem especially when several processes want to write to the same file. Lustre provides file locking on a server-basis, which can lead to contention for concurrent file operations, especially when accessing segments of files in the RAID 0 pattern (i.e., in a circular round-robin manner). The challenge is to select OSTs to place different sized stripes for concurrent workloads with different striping layouts such that all OSTs have load balancing and less resource contention, improving I/O for different workload characteristics.

5 TARAZU SYSTEM DESIGN

In the following, we will introduce our design philosophy, provide an overview of the software architecture, and discuss every software component individually. We have implemented Tarazu for the widely used parallel file system Lustre. Please note that our design can be extended for

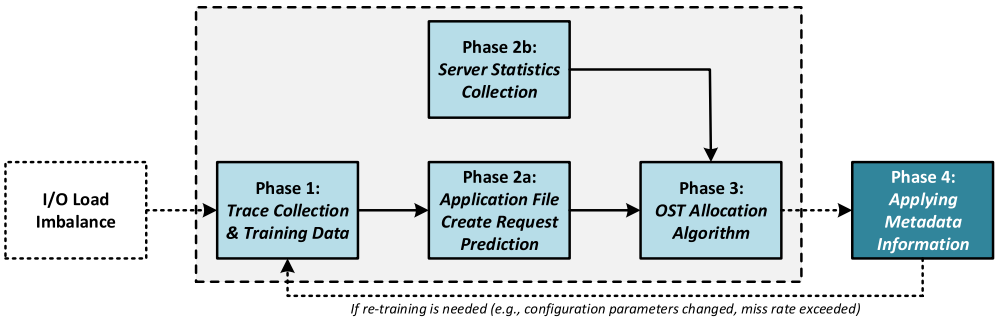


Fig. 9. Iterative end-to-end control flow in Tarazu for I/O load balancing.

use in other HPC parallel file and distributed storage systems that employ a similar hierarchical structure.

5.1 Design Philosophy and Contributions

The design of Tarazu adopts the following philosophy and makes the following contributions:

- ***End-to-end Control Plane:*** One of the most important design features in Tarazu is the implementation of an end-to-end control plane that considers the application behavior and combines that information with the current load on the storage servers. Therefore, an informed and adaptive decision can be made about file strip placement by having a holistic view of the current application and server load.
- ***Application-agnostic Global View of Resources:*** Storage servers act indifferently to the applications sending I/O requests, that is, servers do not have application-level information. Therefore, the placement algorithm in Tarazu should consider a global view of all the resources (server and client) before deciding the placement of the file stripes.
- ***Automatic Coordination of I/O Requests from Concurrent Workloads:*** All applications perform I/O on the same shared file system. Therefore, in Tarazu, instead of a per-application file placement algorithm, a holistic solution for all applications is needed, which requires the coordination of I/O requests from all concurrent applications.
- ***Intelligent and Adaptive* Placement Algorithm:** The placement algorithm should be intelligent enough to make accurate predictions about future file requests by tracking the I/O behavior of the application. Since this depends on the behavior of individual clients, Tarazu's prediction model should run on the clients, dynamically improve the prediction accuracy, and forward the current and predicted I/O requests to the central placement algorithm.
- ***Transparent Placement of Application Files:*** Applications should have no knowledge of how the entire placement algorithm works. Therefore, one of the most important design decisions in Tarazu is the transparent approach to file stripe placement. This ensures that the application code does not need to be modified or recompiled.

5.2 End-to-end Control Flow

Figure 9 shows the high-level control flow of Tarazu. To mitigate I/O load imbalance in the parallel file system, Tarazu first collects file I/O traces during an application run and trains the prediction model on the collected historical traces for an application (*Phase 1*).

This trained model is then used to predict file creation requests (*Phase 2a*). As discussed previously, file creation requests lead to load imbalance in OSTs. Therefore, before the actual application run, our prediction model will predict the file size for all file creation requests in an application.

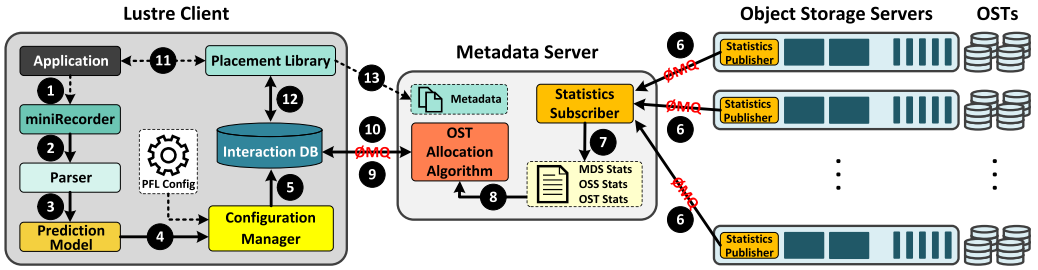


Fig. 10. Overview of the Tarazu software architecture.

The number of file stripes is also collected from the configuration file. This set of predicted file create requests (file size and the number of stripes) from the client is sent to the OST allocation algorithm running on the MDS that has a global view of the system.

The MDS collects real-time statistics on OSS and OST resource usage (*Phase 2b*) asynchronously and in parallel with Phase 2a. Based on the set of file creation requests sent by all clients and the server statistics, the OST allocation algorithm running on the MDS maps each file to a set of OSTs (*Phase 3*) such that there is OST and OSS-level load balance in the system. The file creation requests and the corresponding set of load-balanced OSTs are given back to the respective clients.

When the actual file creation requests come from the application, the mapped, load-balanced set of OSTs is allotted for the corresponding file creation request and the metadata information is applied to the actual request (*Phase 4*). This helps reduce the latency of file creation requests and achieve a scalable load-balanced design.

Phases 1 through 3, which involve retraining the model based on historical traces and making a prediction, are only required if there are many actual file creation requests that are not included in the predicted set of requests, resulting in a higher miss rate, or if the application striping pattern and file-sharing strategy change. This separation of Phases 1–3 and Phase 4 helps design a transparent load-balancing framework that does not require changing the application source code. At the same time, it enables seamless load balancing by reducing the overall latency of file creation requests, resulting in better application I/O throughput. This control flow also results in the prediction model not interfering with the actual application flow.

5.3 System Overview

Figure 10 shows an overview of the Tarazu software architecture. It shows an end-to-end control plane for managing I/O. The design relies on components on both the client and server side.

When applications are initially run, `miniRecorder` (1), a custom tracing tool, is used on the client side to collect characteristic information about the workload's I/O accesses, such as the number of bytes written, the file name, the number of stripes, and the MPI rank and communicator for each file. Tarazu thus identifies the expected I/O behavior of an application, which normally does not change between multiple runs of an application. The collected traces are passed to the `parser` (2), which then uses the information to drive the `prediction model` (3). We use ARIMA time series modeling [11] for our predictions. The time series prediction output provides estimates of the application's future especially, file creation requests, which are passed to the `configuration manager` (4). The configuration manager, in turn, is responsible for determining whether an application's layout uses PFL. If PFL is used, then `PFL Config` stores the required PFL configurations used by the configuration manager. Tarazu provides a default configuration for PFL, which can be customized by the user to be system-specific if needed. The output of the `configuration manager` (5) (file creation requests and file configuration—number of stripes) is then stored in

an `interaction database` for later use. We refer to the database as an "interaction database," because it represents a point of interaction between our server-side and client-side software components.

On the server side (i.e. on the OSS), we collect various statistics such as CPU and memory usage information, the associated OST capacity (*kbytestotal*), and the number of bytes available on the OSTs (*kbytesavail*). These statistics are collected from the OSSs via `statistics publisher` modules (6), i.e., we rely on a publisher-subscriber model using the asynchronous messaging library ZeroMQ [31]. These statistics are parsed by the `statistics subscriber` on the MDS, which generates a file (7) containing updated statistics for the MDS, OSS, and associated OSTs. This file is then forwarded to the `OST allocation algorithm` (8). In addition, the OST allocation algorithm receives as input the predicted set of file creation requests and the number of stripes for each file (9) from the client-side interaction database from all the clients via ZeroMQ message queues. The output is a list of OSTs to be assigned for each request, resulting in a balanced distribution among the participating OSSs and OSTs. The allocated OSTs are stored in the interaction database (10) along with the predicted file creation requests. Finally, the `placement library` (`Tarazu-PL`), intercepts the actual file creation requests (11) from the applications, consults the interaction database (12), and maps the application requests to appropriate resources by creating the metadata on the MDS for a given file (13).

Mapping the system design of `Tarazu` (see Figure 10) to that of the control flow (see Figure 9), miniRecorder, and Parser form Phase 1. Prediction Model and Configuration Manager form Phase 2a. Phase 2b consists of Statistics Publisher and Statistics Subscriber. Phase 3 has the OST Allocation Algorithm. Phase 4 is formed by the Placement Library. The interaction database is filled with the predicted set of file creation requests and the striping configuration in Phase 2a. The OST allocation algorithm in Phase 3 uses the interaction database to map OSTs to file creation requests and store those mappings in the database. The Placement library in Phase 4 refers to the interaction database for the OST mapping list when the actual file creation request arrives. As the prediction model runs during the application start time (to get the predicted set of file creation requests based on historical runs of the application) and only in cases where re-training is required (when the application's file striping pattern or I/O sharing strategy changes), there is no interference of the trace collector and the prediction model with the normal execution of the application. The interaction database resides on every client. Each row in the database corresponds to one file creation request. Therefore, the size of the database is directly related to number of files an application creates. The database will be filled up by the OST allocation algorithm at the start of the application. Therefore, the database is queried by the Placement library only when an actual file creation request arrives. This querying latency is lower than the network latency that the application would normally need to face when a file creation request arrives, as there needs to be interaction between client and the MDS. Therefore, our design reduces the I/O latency of an application during the file creation request. Our design of `Tarazu` is hardware-agnostic, as the design components do not utilize a lot of memory and compute resources. Therefore, `Tarazu` has a minimal impact on the actual I/O activity of applications running in a cluster. Moreover, the cluster setup for our experiments is done on commodity hardware. With real-world large-scale HPC clusters, `Tarazu` will perform even better. Next, we explain each phase of the control flow with respect to the design components.

5.4 Phase 1: Trace Collection and Training Data

We implement a simple, lightweight I/O tracing library, `miniRecorder`, based on the Recorder [49] multi-level I/O tracing framework. The intercepted function calls are restricted to file creation and write calls. Figure 11 displays the dynamic interception of I/O metadata operations at runtime.

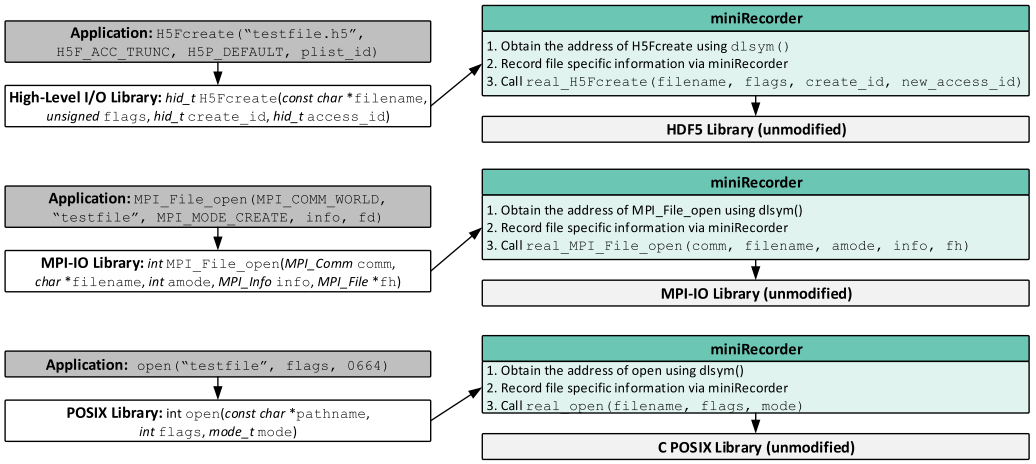


Fig. 11. Dynamic interception of I/O metadata operations at runtime.

The load imbalance on OSSs and OSTs is due to the write and create requests [49, 67], as this is when the actual stripes for the files are allocated on the OSTs, as described in Section 2.1. Therefore, the tracing tool only needs to be used to capture the I/O creation and write behavior of an application or workflow. Afterwards, miniRecorder is only run whenever the ARIMA-inspired prediction model performs poorly and the OST prediction algorithm needs to be re-trained (see Section 5.5.1). In the next step, the data collected by miniRecorder is converted into a readable (comma-separated) file in *.csv* format by a `parser` and then sent to the prediction model. This file serves as the starting point and training data for the ARIMA-based prediction algorithm.

5.5 Phase 2a: Application File Create Request Prediction

For each application, our prediction model foretells the file size for all file create and write requests performed by an application. We rely on predictions based on ARIMA time series modeling and a configuration manager responsible for determining the striping layout of a file to achieve this.

5.5.1 ***ARIMA-Inspired Prediction Algorithm (AIPA).*** Recent work [67] suggests that I/O patterns of HPC applications are predictable. This observation is also confirmed by multiple HPC practitioners. Therefore, the `miniRecorder` module in `Tarazu` collects three important features of HPC I/O at the client side—number of bytes written, MPI rank, and the stripe count. These parameters are used to train a machine learning model to predict client I/O requests—most importantly, the file creation requests for future runs. The machine learning model is run on the client side to avoid overloading the metadata server with predictive modeling.

Previous work [2, 6, 8] has presented auto-tuning approaches for MPI-IO and Lustre to learn and predict the I/O parameters to improve read and write performance of HPC applications. Researchers have previously used ***AutoRegressive Integrated Moving Average* (ARIMA)** model [11], **Seasonal Integrated ARMA (SARIMA)**, and **Fractionally Integrated ARMA (ARFIMA)** to estimate CPU, RAM, and network usage for HPC workloads [37]. Formal grammar has also been used to predict I/O behaviors in HPC [20], which proves that HPC I/O is predictive. However, formal grammar is ineffective for predicting data in a time-series manner, effectively consuming low resources. Markov Chain Model [76] has also been used to exploit knowledge of spatial and temporal I/O requests [60, 67]. Recently, a regression-based I/O performance prediction scheme for HPC environments was also proposed [34]. Based on previous studies, the time

series nature of the traces provided by our tracking tool allows us two options for our prediction model—ARIMA and Markov chain. We initially find the I/O request prediction accuracy and resource consumption using both models. We observe that for IOR data, ARIMA has a 99.1% accuracy with 1.2% CPU overhead and 0.01% memory usage, while Markov chain model yielded an accuracy of 95.5% utilizing 4.5% CPU and 0.01% memory.

Prediction models should not interfere with the client-side I/O activities. Therefore, to ensure prediction on the client side in an online setting, we use the ARIMA model that provides better accuracy at lower resource consumption.

Two design aspects for the ARIMA-based prediction model are:

(1) The prediction model runs on the clients. A different design decision could have been running a global prediction model on the MDS, but that would limit scalability and also increase the file creation latency. The client-side model is responsible for predicting future file creation requests of the client application based on the current request. Both the current and the future set of requests are stored in the interaction database, which is fed to the OST allocation algorithm. The list of OSTs to guarantee load balance for all requests is stored in the interaction database. Once a new creation request arrives, greater than 99% accuracy of our ARIMA model ensures that the request will be found in the interaction database and there will be no overhead during the file creation process. The prediction for future requests is performed only when there are 10% misses in the interaction database. The time taken to predict is on the order of milliseconds. This ensures that there is minimal interference of the model with the application's runtime behavior.

(2) Re-training of the model is done only when the prediction needs to be done more than three times, which means that cumulatively 30% misses occur in the interaction database. This mostly happens in cases when, for a particular application, the scale of the run or the configuration parameters are changed by the user. The model is retrained on the client side based on the application's current set of requests as well the historical traces. The resource consumption for the re-training step is the same as reported above. The minimal CPU and memory usage of our model ensures that the application's normal runtime behavior is not affected during model retraining.

Our prediction model is implemented using the `statsmodels.tsa.arima_model` package in Python. Our results show a 98.3% accuracy in HACC-I/O data and 99.1% accuracy in IOR data.

5.5.2 Configuration Manager. The `configuration manager` determines if an application is performing I/O in PFL or non-PFL layout. As input, it takes the predicted set of requests, containing the stripe count, and write bytes, with the corresponding file name and MPI rank from the `prediction model`, and the file containing all file layouts for a client. If the file names or the application directory in the predicted requests set match those in the PFL configuration file, then the stripe size and stripe counts for the file are set based on the PFL configuration. A sample PFL configuration file for the example *Configuration 1*, written in `.ini` format, is shown in Figure 12. If there are no matches found for the file name or the directory in the PFL configuration, then the non-PFL layout is used. The output of the configuration manager is the set of all predicted requests combined with the corresponding stripe size and stripe count of the files. This entire set is sent to the `interaction database`.

An important design concern to calculate the stripe size for both layouts is the 64k-alignment constraint imposed by Lustre. This constraint states that the stripe size should be an even multiple of 64k or 65,536 bytes (**Alignment Parameter (AP)**). For file sizes that are not AP-aligned, we use Equations (2) and (3). The method ensures a 64k-aligned stripe size for all the stripes allocated on the *stripeCount* number of OSTs by allocating a slightly bigger file than is requested by the client.

```
;config file for PFL Configuration Layout 1
;lfs setstripe -E 128M -c 1 -E 512M -c 3 -E 2G -c 8 -E -1 -c 16 /mnt/lustre/ior
[Client]
Buckets = 4                #Type the number of buckets (here: 4)
[Bucket_1]
Extent = 128M              #-E 128M -c 1
Stripe = 1
[Bucket_2]
Extent = 512M              #-E 512M -c 3
Stripe = 3
[Bucket_3]
Extent = 2G                #-E 2G -c 8
Stripe = 8
[Bucket_4]
Extent = -1                #-E 8G -c 16
Stripe = 16
```

Fig. 12. Example PFL configuration file for *Configuration 1*.

Table 1. Interaction Database Snapshot for IOR in FPP Mode Using a PFL Layout

File Name	File Size	Extent ID	Extent Start	Extent End	Stripe Size	Stripe Count	MPI Rank
/mnt/lustre/ior/test.0	8,589,934,592	1	0	134,217,728	134,217,728	1	0
/mnt/lustre/ior/test.0	8,589,934,592	2	134,217,728	536,870,912	134,217,728	3	0
/mnt/lustre/ior/test.0	8,589,934,592	3	536,870,912	2,147,483,648	201,326,592	8	0
/mnt/lustre/ior/test.0	8,589,934,592	4	2,147,483,648	8,589,934,592	402,653,184	16	0
/mnt/lustre/ior/test.1	8,589,934,592	1	0	134,217,728	134,217,728	1	1
/mnt/lustre/ior/test.1	8,589,934,592	2	134,217,728	536,870,912	134,217,728	3	1
/mnt/lustre/ior/test.1	8,589,934,592	3	536,870,912	2,147,483,648	201,326,592	8	1
/mnt/lustre/ior/test.1	8,589,934,592	4	2,147,483,648	8,589,934,592	402,653,184	16	1

For example, a 766.175 MB file size will be allocated an 833 KB (0.1%) bigger file, which ensures allocating equal-sized stripes on all the OSTs, and hence contributing to a load-balanced setup. Further details on the equations can be found in Reference [87].

$$writeBytes = AP * 2 * N * stripeCount \tag{2}$$

with

$$N = \left\lceil \frac{writeBytes}{AP * 2 * stripeCount} \right\rceil. \tag{3}$$

5.5.3 Interaction Database. The `interaction database` is an SQL database that resides on the Lustre clients. It serves as the medium through which the MDS and clients interact with each another. First, the output set from the `configuration manager` is stored in the database. Different table formats are used to store PFL and non-PFL file layouts. Tables 1 and 2 show example snapshots of the interaction database for IOR in FPP mode with two processes each writing 8 GB in PFL and non-PFL layout, respectively. For the PFL layout, we use the example *Configuration 1*, as discussed in Section 2.3. As seen in Table 1, every file is associated with all the extents specified in the PFL configuration. For each file, we store the file name, the file size in bytes, the extent ID from the PFL configuration file, the corresponding start and end range for the extent, stripe size, stripe count, and the MPI rank. For the non-PFL layout, shown in Table 2, we store the file names, stripe size of the files, number of stripes associated with every file, and the MPI Rank.

As every client application has an individual database on the client node, the size of the database depends on the total number of files created by an application. This decentralized nature of the database helps in scalability by not needing to save the tremendous number of files created by all applications running in the HPC cluster in a single database. The database querying frequency

Table 2. Interaction Database Snapshot for IOR in FPP Mode Using a Non-PFL Layout

File Name	Stripe Size	Stripe Count	MPI Rank
/mnt/lustre/ior/test.0	1,073,741,824	8	0
/mnt/lustre/ior/test.1	1,073,741,824	8	1

Table 3. List of Statistics Collected from Relevant System Components

Component	Factors	Discussion

Metadata Server (MDS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		CPU and memory utilization reflect the system load.
		Load on the Lustre networking layer connected to MDS.

Object Storage Server (OSS)	`CPU%`
	`Memory%`
	`/proc/sys/lnet/stats`
		Reflects the system load of the management server.
		Load on the Lustre networking layer connected to OSS.

Object Storage Target (OST)	`obdfilter.*.stats`
	`obdfilter.*.job_stats`
	`obdfilter.*OST*.kbytesfree`
	`obdfilter.*OST*.brw_stats`
		Overall statistics per OST.
		Statistics per job per OST.
		Available disk space per OST.
		I/O read/write time and sizes per OST.

is also kept to a minimum, as the database needs to be accessed only for file creation requests. Subsequent file read and write accesses do not need the database.

5.6 Phase 2b: Server Statistics Collection

The statistics collection needs to be lightweight and scalable so it can handle up to thousands of OSSs in a seamless manner without affecting the file system activities. Therefore, ZeroMQ (ϕMQ) [31] is used as a message queue, which has been proven to be lightweight and efficient at large-scale. To ensure scalability, an asynchronous publish-subscribe model is used where the OSSs act as publishers and the MDS acts as a subscriber. Table 3 shows the list of system metrics collected.

Each OSS has a `statistics publisher` module responsible for sending system usage statistics from the OSS to the MDS. A configuration file is given to every OSS that contains a list of all OSTs along with the OSS ID. For each OSS, the total capacity and the available capacity are recorded from the *lustre/obdfilter* directory. We also collect the CPU, memory, and network usage of the OSS by reading the *proc/loadavg*, *proc/meminfo*, and *lnet* files. The statistics publisher module runs every 60 seconds on all OSSs so we can get the most current statistics about the MDS without overloading the OSSs. The lightweight nature of the statistics collection is tested where the results show that, on average, it has negligible CPU and only 0.1% memory usage on the OSSs.

The `statistics subscriber` on the MDS is responsible for

(1) collecting CPU, memory, and network utilization from the MDS,
(2) subscribing to statistics from the OSSs via ZeroMQ, and
(3) parsing and preparing all the collected statistics.

The CPU, memory, and network usage recorded on the MDS is important in determining when to run the OST allocation algorithm. To not interrupt normal MDS activity, the OST allocation algorithm runs only when the CPU and memory utilization fall below 70% and 50%, respectively. These utilization thresholds can be manually tuned. The collected statistics from the MDS and

ALGORITHM 1: Obtaining list of OSTs for each request.

```
Input: OSS statistics cpu & mem, OST statistics totalKbytes & kbytesAvail, Write Requests stripeSize
       & stripeCount
Output: OSTAllocationList
1  begin
2      for OSS oss in OSSList do
3          ossLoad = (cpuweight * cpu) + (memweight * mem)
4          for OST ost in OSTList do
5              ostCostToReach = OSSLoad
6              ostCost = (totalKbytes − kbytesAvail)/totalKbytes
7              maxStripeSize = max(stripeSize) from both PFL and non-PFL interaction databases
8              ostCapacity = kbytesAvail/maxStripeSize
9      flowGraph = buildGraph(Requests, OSS, OST)
10     OSTAllocationList = minCostMaxFlow(flowGraph)
11     return (OSTAllocationList)
12 Function buildGraph
       Input: Requests req, StripeCount sc, OSTCostToReach ossLoad, ostCost ostLoad, OSTCapacity
              ostCap
       Output: FlowGraph G
13     totalDemand = sum of stripeCount for all Requests
14     G.addNode('source', totalDemand)
15     G.addNode('sink', -totalDemand)
16     for request r in reg do
17         G.addEdge('source', r, cost = 0, capacity = sc)
18         for OST ost in ostList do
19             G.addEdge(r, ost, cost = ossLoad, capacity = 1)
20     for OST ost in ostList do
21         G.addEdge(ost, 'sink', cost = ostLoad, capacity = ostCap)
22     return (G)
```

the OSS are parsed and used as input to the OST allocation algorithm. Our results show that the statistics collection on the MDS has a CPU utilization of 0.1% and negligible memory usage.

5.7 Phase 3: OST Allocation Algorithm

Algorithm 1 shows the steps employed for allocating OSTs for a client write request. The inputs used for the `OST allocation algorithm` consist of the statistics collected from the OSSs and OSTs as well as the write requests from the PFL and non-PFL interaction databases residing on the clients. The information required is the stripe size and stripe count of every request. The requests from all client applications are collected and together sent as input to the allocation algorithm. This makes the algorithm non-sensitive towards any particular application and have a global view of all the applications and OSSs. The OST allocation algorithm employs a minimum-cost maximum-flow approach [3]. The flow graph that is used to solve the problem is shown in Figure 13.

The cost to reach an OST is the load of the OSS containing the OST. The cost of an OST is defined as the ratio of bytes already used in the OST to the total size of the OST. The allocation algorithm should be able to handle both PFL and non-PFL applications. The PFL requests have varied stripe sizes. Therefore, to have consistency in the allocation algorithm, we compute the maximum stripe

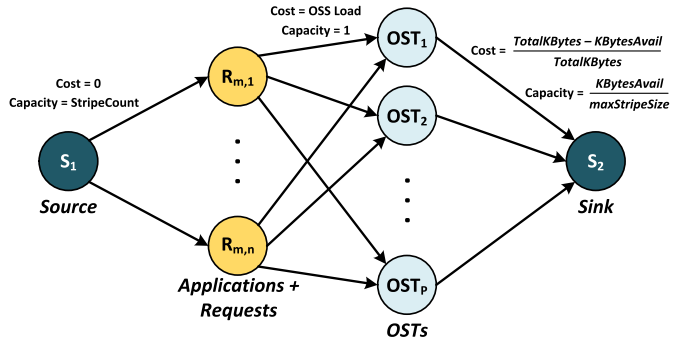


Fig. 13. Minimum-cost maximum-flow graph used in the OST allocation algorithm.

Table 4. Interaction Database Snapshot Showing the OST Allocation for IOR in FPP Mode Using a PFL Layout (Ext.: Extent, St.: Stripe)

File Name	File Size	Ext.ID	Ext.Start	Ext.End	St.Size	#St.	Rank	OST List
/mnt/lustre/ior/test.0	8,589,934,592	1	0	134,217,728	134,217,728	1	0	10
/mnt/lustre/ior/test.0	8,589,934,592	2	134,217,728	536,870,912	134,217,728	3	0	24 29 34
/mnt/lustre/ior/test.0	8,589,934,592	3	536,870,912	2,147,483,648	201,326,592	8	0	15 2 28 25 11 21 7 33
/mnt/lustre/ior/test.0	8,589,934,592	4	2,147,483,648	8,589,934,592	402,653,184	16	0	14 23 16 30 1 6 4 26 32 20 12 31 22 35 5 17
/mnt/lustre/ior/test.1	8,589,934,592	1	0	134,217,728	134,217,728	1	1	24
/mnt/lustre/ior/test.1	8,589,934,592	2	134,217,728	536,870,912	134,217,728	3	1	13 9 29
/mnt/lustre/ior/test.1	8,589,934,592	3	536,870,912	2,147,483,648	201,326,592	8	1	22 11 35 21 5 7 17 10
/mnt/lustre/ior/test.1	8,589,934,592	4	2,147,483,648	8,589,934,592	402,653,184	16	1	14 23 16 30 1 6 15 4 26 2 32 20 12 28 31 25

size across both PFL and non-PFL databases from all client applications. The capacity of an OST is defined as the number of stripes that can be handled by the OST. This is calculated by dividing the available space in the OST by the maximum stripe size.

To construct the flow graph shown in Figure 13, *source* and *sink* nodes need to be identified. The total demand for the *source* node is the total number of stripes requested by all application requests, and the total demand for the *sink* node is the negative amount of the total number of stripes requested. The Ford-Fulkerson algorithm [84] is used to solve the minimum-cost maximum-flow problem. This approach outputs a list of OSTs (*OSTAllocationList*), which will yield a balanced load over all OSSs and OSTs. For our implementation, we use the Python library `networkx`. Our tests show that the algorithm uses about 1.58% of CPU and 0.1% of memory on the MDS.

The MDS uses a scalable publisher-subscriber model via ZeroMQ [31] to interoperate with the interaction database, which helps in scaling `Tarazu` to a large number of clients [70]. We use the MySQL 8.0.12 Community Server Edition. Our results show that writing and retrieving data from the interaction database is very efficient, using <0.3% and <0.4% of CPU and memory, respectively.

The *OSTAllocationList* is then shared with the respective clients using our publisher-subscriber model via ZeroMQ. The complete set of requests is stored in the `interaction database`. Example entries for the database with the complete allocation for an IOR application in FPP mode with two processes each writing an 8 GB file in both PFL (*Configuration 1*) and non-PFL layouts are shown in Tables 4 and 5, respectively. We add a new column *OST List* in the database. The *OST List* is a space-separated load-balanced list of OSTs for every write request. This example is for a setup with 7 OSSs and 35 OSTs (5 OSTs associated with every OSS)—therefore, OST IDs range from 1 to 35. The placement library (`Tarazu-PL`) uses this information to place the requests, thus completing

Table 5. Interaction Database Snapshot Showing OST Allocation for IOR in FPP Mode Using a Non-PFL Layout

File Name	Stripe Size	Stripe Count	MPI Rank	OST List
/mnt/lustre/ior/test.0	1,073,741,824	8	0	30 20 5 1 22 35 14 23
/mnt/lustre/ior/test.1	1,073,741,824	8	1	20 19 1 9 29 2 33 18

ALGORITHM 2: File layout creation on the MDS.

```
Input: File Name file, Access mode flags
Output: Call to real metadata operation (e.g., open())
1  begin
2      if fileExits(file) == TRUE then                          // File exits; return.
3          return realMetadataOperation(file, flags)
4      flags = flags | O_LOV_DELAY_CREATE
5      result = queryInteractionDatabase(file)
6      if numMySQLrows(result) == 1 then                        // Non-PFL layout.
7          row = fetchMySQLrow(result)
8          layoutEA = allocLayoutEA(row.stripeCount, row.stripeSize, row → OSTs)
9          createLayoutEAonMDS(file, flags, 0644, layoutEA)
10     else if numMySQLrows(result) > 1 then                    // PFL layout.
11         while row = fetchMySQLrow(result) do
12             allocExtentPFL(layoutPFL, row.extentEnd, row.stripeCount, row.stripeSize,
                 row → OSTs)
13         createCompositeLayoutMDS(file, flags, 0644, layoutPFL)
14     return realMetadataOperation(file, flags)
```

the load-balanced allocation of resources. If for any run of the application Tarazu-PL is unable to find more than 50% of files in the interaction database, then miniRecorder, AIPA, and the OST Allocation Algorithm will be executed again to update the interaction database.

5.8 Phase 4: Applying Metadata Information

The Tarazu-PL placement library complements the prediction model by providing a lightweight, portable, and user-friendly mechanism to apply predicted file layout information (i.e., the metadata information of a file) to an application's file I/O without the need to modify the source code. Tarazu-PL relies on function interposition provided by the dynamic linker GNU to prioritize itself over standard system calls and the **profiling interface to MPI (PMPI)**. Hence, Tarazu-PL can be used by preloading it through LD_PRELOAD. This results in metadata operations being passed to Tarazu-PL, which supports both non-PFL and PFL file layouts for FPP and SSF as well as various I/O interfaces such as POSIX I/O, MPI-IO, and HDF5.

For each file creation request, Tarazu-PL queries the interaction database once via the MySQL C API with the file name passed by the original metadata operation, fetches the matching rows, and applies the predicted striping pattern to the file if the file does not already exist. To facilitate this, Lustre provides a user library called *llapi* that allows file striping patterns to be specified via a C API. Tarazu-PL mimics the behavior of *llapi* and communicates directly with Lustre's **Logical Object Volume (LOV)** client to create the file metadata on the MDS, similar to References [58, 87].

Algorithm 2 describes a simplified version of the placement library's mode of operation, which is run on the clients. If the result returned by the MySQL query contains only one row, then

the non-PFL layout is used by allocating a ***Layout Extended Attributes*** **(Layout EA)** with the predicted striping pattern on the MDS. The striping pattern is applied by initializing the layout EA with the stripe count, stripe size, and the list of OSTs retrieved from the interaction database. It should be noted that the configured striping pattern differs from the default RAID 0 pattern typically applied by Lustre. Instead of writing multiple data segments in a round-robin fashion as introduced in Section 2.1, `Tarazu` provides larger, contiguous stripes, therefore avoiding file locking contention when multiple processes are accessing the same file. If the query results in more than one row, then PFL is used. For both FPP and SSF, each row represents a non-overlapping range. `Tarazu-PL` iterates over all lines, assigns an array of sub-layouts (one for each file extent), and applies the predicted striping pattern to the file by storing it in a composite layout on the MDS. Composite layouts allow specifying different striping patterns for different extents in the same file.

6 SIMULATOR ENVIRONMENT AND WORKLOAD GENERATION

To enable scaling experiments, we implement a discrete-time simulator. In the following, we describe the design of the simulator based on Darshan-based workload generation and give a brief validation of the simulation results.

6.1 Overview of Existing Parallel File System Simulators

Before we discuss the design of the discrete-event simulator, we present an overview of previous work on file system simulation and argue why they cannot be directly applied to our research.

The *Lustre simulator* [104] was developed as an event-driven simulation platform to research scalability, analyze I/O behaviors, and design various algorithms at large scale. It simulates disks, the Linux I/O elevator, a file system with mballoc block allocation, a packet-level network, and three Lustre subsystems: client, MDS, and OSS. The main focus of this simulation tool is the evaluation of the **Network Request Scheduler (NRS)**. Since this simulator was developed in 2009, it is based on Lustre 1.8 and therefore is not compatible with our experiments.

Another open-source simulator developed in 2009 is ***IMPIOUS*** **(Imprecisely Modelling I/O is Usually Successful)** [52]. IMPIOUS is trace-driven and provides abstract models to capture the key characteristics of three parallel file systems; PVFS, PanFS, and Ceph. Depending on the simulated file system, the simulator can be configured to distinguish different characteristics such as data placement strategies, resource locking protocols, redundancy strategies, and client-side caching strategies. Due to its age and the lack of supporting Lustre as a file system, IMPIOUS cannot be used for our evaluation.

Liu et al. have introduced *PFSsim* [46, 47], which is also a trace-driven simulator designed for evaluating I/O scheduling algorithms in parallel file systems. It uses OMNeT++ for detailed network models and relies on DiskSim [38] to simulate disk operations. Since PFSsim only supports PVFS2 and mainly focuses on I/O scheduling algorithms, it cannot be used to evaluate `Tarazu` at scale.

In 2012, the parallel file system simulator *FileSim* [21] was introduced by Erazo et al., which is based on SimCore, a generic discrete-event simulation library. It provides pluggable models with different levels of modeling abstraction for different parallel file system components. In addition, FileSim supports trace-driven simulation, which can be used to validate parallel file system models by comparing against the behavior observed in the real systems. Even though the description of the simulator would fit our requirements needed to simulate `Tarazu` in large-scale deployments, it was not available anymore at the time of this writing.

The ***Hybrid Parallel I/O and Storage System Simulator*** **(HPIS3)** [23] was introduced in 2014 by Feng et al. It provides a co-design tool targeting the optimization of hybrid parallel I/O and

storage systems, where a set of SSDs and HDDs are deployed as storage nodes. HPIS3 is built on **Rensselaer Optimistic Simulation System (ROSS)** [13], a parallel simulation platform, and capable of simulating a variety of parallel storage systems with two distinct types of hybrid system design, namely, buffered-SSD and tiered-SSD storage systems. Hence, HPIS3 is targeting a different scenario than Tarazu and therefore is not applicable to our use case.

Other simulators relying on ROSS are CODES [15] and BigSim [106]. CODES provides a tool for I/O and storage system simulations. Its main target is the exploration and co-design of exascale storage systems for different I/O workloads. Initially, workloads could only be described via the CODES I/O language. In 2015, Snyder et al. [79] proposed an **I/O workload abstraction (IOWA)**. IOWA describes different techniques to generate workload for simulation frameworks depending on the use case, including workload generation from Recorder [49] and Darshan [12]. Since our simulation use case is mostly concerned with the bandwidth performance when reading or writing to the parallel file system, we only adopt the workload generation techniques proposed by IOWA.

6.2 Simulator Design

As discussed in the previous section, there are no simulators that can either simulate the different components in the Lustre file system or integrate various OST allocation algorithms to help evaluate Tarazu. Therefore, we have designed and implemented a discrete-time simulator based on the overall design of the Lustre file system, as shown in Figure 1. The simulator also allows implementations of different OST allocation algorithms (such as LSA and Tarazu). This helps in evaluating Tarazu at scale by comparing it to the default LSA allocation strategy in Lustre.

The simulator consists of four key components that are very similar to those of Lustre's OST, OSS, MDT, and MDS. These implement the various Lustre operations and allow us to collect data about the system behavior. The MDS is also equipped with multiple strategies for OST selection, such as round-robin, random, and the OST allocation algorithm designed for Tarazu. We have implemented a wrapper component that enables communication between our various simulator components. The wrapper is responsible for processing the input, managing the MDS, OST, and OSS communication and data exchange, and driving the simulation. All the network components in the simulator are modeled using the **Network Simulator (NS-3)** [59]. The application traces collected from the client side are modeled as clients in the simulator. In our simulations, all initial conditions are the same at the beginning of each OST allocation strategy.

The steps for building the simulator are shown in Algorithm 3. The application trace file generated from the Darshan traces, which is explained in the next section, serves as input to the simulator along with the configuration file for PFL, number of I/O routers, and LNet routers. First the time-series set of requests are generated from the application traces by reading the configuration file and calculating the request size and stripe count. The network topology of the simulator is built next.

The major components of our Lustre Simulator are as follows:

- **Application**: Parses the trace file and the PFL configuration. The trace file is converted into a time-series event list with request size and stripe count.
- **I/O Router**: These are simulated as network components placed between the client nodes (*Application*) and the storage nodes. It forms one of the most important components to simulate the network traffic from multiple applications in our simulator.
- **LNet Router**: Handles the OSS in the cluster. All packets coming to the I/O Router will be redirected through the LNet Router, which keeps track of the OSS where the packets are being sent and accordingly updates the load on the OSS.

ALGORITHM 3: Lustre Simulator.

```
Input: Application traceFile, Config File config, Number of IO Routers numIORtr, Number of LNet
       Routers numLNetRtr
1  begin
2      Read Application Traces
3      numRequests = len(traceFile)
4      stripeCount = read config
5      for Request req in numRequests do
6          events.append(reqSize, stripeCount, timeStamp)
7  begin
8      Build Network Topology
9      for LNetRtr in numLNetRtr do
10         Get numOSS and CPULoadPerOSS
11         for oss in numOSS do
12             Get numOSTs
13             for ost in numOST do
14                 Get totalDiskSpace and usedDiskSpace
15     Set DataRate and LinkDelay across I/O Routers, LNet Routers, OSS, and OST.
16 begin
17     Schedule Events
18     for event in numEvents do
19         stripeSize = reqSize/stripCount
20         begin
21             Get OSTs for Request
22             Run the placement algorithm (LSA or Tarazu) to get the OSTList
23         for OST in OSTList do
24             WriteToOST(stripeSize)
25             begin
26                 Get the OSS from the LNetRtr
27                 Update CPULoadPerOSS
28                 Update usedDiskSpace in OST
```

— **OSS**: This module handles the load distribution on the OSTs under each OSS. For each OSS, it keeps track of the list of active OSTs, CPU usage of the OSS—which is calculated by the combined usage of all the associated OSTs.
— **OST**: This module handles the final step of the packet transfer from the Application. It reduces the disk space on the OST in accordance to the packet size as well as contributes to the calculation of the OSS CPU usage.

This network topology helps in simulating the application requests. As discussed before, the time series set of requests is stored as an event list. At the simulated time instance, the event is taken from the event list, and the stripe size is calculated by dividing the write bytes by the number of stripes. The stripe size and the number of stripes help in getting the OST list by running the appropriate OST allocation algorithm (LSA or Tarazu). Once the OST list is received, the writing to each corresponding OST is simulated by the I/O Router followed by the LNetRouter, the OSS,

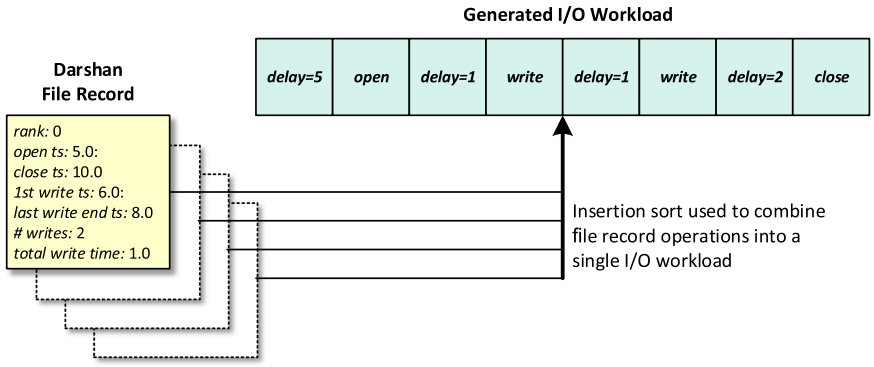


Fig. 14. Transforming Darshan file records into an I/O workload derived from Reference [79].

and finally the OST. Accordingly, the disk usage in OST and the CPU usage in the OSS is updated based on the stripe size.

6.3 Darshan-based Workload Generation

Darshan logs [12] are used to generate I/O traces for real-world HPC workloads. The Darshan I/O characterization tool maintains details of every file that is opened by the application. The I/O interfaces recorded are POSIX, MPI-IO, STDIO, HDF5, and PnetCDF used to access the file. For the purpose of this article, we only focus on the POSIX and MPI-IO interfaces. The major counters collected by Darshan for every file at the POSIX interface are:

— Timestamps of the first file open/read/write/close operation - `POSIX_F_*_START_TIMESTAMP`
— Timestamps of the last file open/read/write/close operation - `POSIX_F_*_END_TIMESTAMP`
— Cumulative time spent on reading from a file - `POSIX_F_READ_TIME`
— Cumulative time spent on writing to a file - `POSIX_F_WRITE_TIME`
— Total number of bytes that were read from a file - `POSIX_BYTES_READ`
— Total number of bytes written to a file - `POSIX_BYTES_WRITTEN`
— Rank that accessed a file - `rank`

The process of converting the file-wise Darshan records into a time-series I/O trace is discussed below. The ranks accessing the file indicate whether the application was run in file-per-process mode or single-shared-file mode. Each file is arranged based on increasing order of first open timestamp. The I/O idle time for every file is calculated by subtracting the cumulative time spent on reading and writing from the duration between the file open start and file close end timestamps. This idle time (or delay) along with the cumulative bytes read or written are uniformly distributed for every file within the file open start timestamp and file close end timestamp. Once the distribution of every file's I/O activity is done, insertion sort is used to sort and combine the I/O activity of the application in a time-series manner. This process of regenerating I/O workload of an application from its Darshan record is in sync with a technique proposed by Snyder et al. [79]. Figure 14 shows the general process of transforming Darshan logs into comprehensive I/O workloads. In this work, we use this technique to build the trace files of three real-world workloads discussed in Section 7.1.3 that are fed as inputs to the Lustre simulator discussed in the previous section.

6.4 Validation of the Lustre Simulator

To validate the ns-3-based Lustre simulator and the Darshan-based workload trace generation, we use the Darshan logs of IOR [40] that ran on Summit [41], the world's fourth-largest

Table 6. Comparison of File System Bandwidth by Running IOR Doing POSIX I/O on the Titan Supercomputer and the Lustre Simulator

File Size	#Nodes	Bandwidth (GB/s) Titan Supercomputer	Bandwidth (GB/s) Lustre Simulator
128 MB	32	7.9	6.2
128 MB	256	38.4	33.5
128 MB	512	54.3	48.7
128 MB	1,024	72.1	69.9
128 MB	2,048	90.8	88.1
128 MB	4,096	111.4	105.6
512 MB	1,024	76.5	75.8
512 MB	2,048	111.6	110.5
512 MB	4,096	132.5	130.4

supercomputer, according to the latest Top-500 list [82]. As the first step, time-series-based IOR traces are generated using the Darshan logs. These traces are then fed into the Lustre simulator. Since Summit is based on IBM Spectrum Scale, we use the real bandwidth results of IOR as well the Lustre setup from Titan [22], a decommissioned supercomputer housed at Oak Ridge National Laboratory that had Lustre file system as the storage backend.

The results from the IOR runs on the Lustre simulator and Titan are shown in Table 6. As seen from the results, for both 128 MB and 512 MB file sizes, and a large number of nodes, the Lustre simulator is able to provide approximately similar file system bandwidth. This enables us to use the Darshan-based workload trace generation approach along with the Lustre simulator for the scalability evaluation of Tarazu.

In addition, we also validate the correctness of the simulator by using the same cluster system setup for Lustre as described in Section 7.1.1 (35 OSTs, 7 OSSs) and executing the traces of HACC-I/O (8 processes, 50 million particles) under PFL Configuration 2 and IOR (8 processes, 64 GB) under PFL Configuration 1 simultaneously. The simulator provides a similar OST utilization percentage for both LSA and Tarazu as discussed in Section 7.2.4 with a read and write throughput of 428 MB/s and 529 MB/s, respectively, for LSA and 612 MB/s and 490 MB/s for Tarazu, respectively.

7 EVALUATION

To the best of the authors' knowledge, Tarazu is the first work to consider a global view of all system resources when deciding data placement for application I/O requests. Existing approaches (e.g., References [91] and [58]) balance load among I/O servers on a per-application basis on the client side and do not consider the global view, i.e., the requirements of other applications or the OSS and OST utilization. Moreover, such client-side techniques cannot handle multiple I/O requests from different applications simultaneously and are therefore not comparable to Tarazu. For these reasons, we decided to use the standard LSA approach as the basis for our performance evaluation.

7.1 Test Environment

7.1.1 Cluster System Setup. We evaluate Tarazu using a Lustre testbed, which consists of 10 nodes with 1 MDS, 7 OSSs, and 2 client nodes. All of the nodes run CentOS 7 atop a machine with 8 cores, 3.2 GHz AMD FX-8320E processor, and 16 GB main memory. Each OSS has 5 OSTs, each supporting 10 GB of attached storage, resulting in a 350 GB Lustre capacity. Our experiments are

Table 7. HPC Workloads Used for the Scalability Study

Application	*genomicPrediction*	*E3SM [98]*	*cosmoFlow [50]*
Domain Science	Bioinformatics	Earth Science	Astrophysics
#Compute Nodes	128	256	64
Total Read (GB)	110.6	100.8	250.6
Total Write (GB)	80.2	150.3	52.4

based on the HACC I/O kernel [28] and the IOR benchmark [40], as both can be run in FPP and SSF sharing modes. In all experiments, we use MPI-IO as the parallel I/O interface. All tests were performed at least three times, and the results represent the measured average values.

7.1.2 Performance Measurements and Metrics. We analyze the following performance metrics: effective read and write bandwidth, load balance, and resource utilization. To capture the degree of load balancing across all OSTs for a given test run, we define the metric *OST Cost* as the ratio of the maximum utilization of any OST to the average utilization of all OSTs, as shown in Equation (4). An ideal load balanced system has the *OST Cost* of 1.

$$\textbf{\textit{OST Cost}} = \frac{\text{Maximum OST Utilization}}{\text{Average OST Utilization}} \tag{4}$$

The *OST Utilization* of an OST is the storage used by the client application on the OST relative to the total storage available on the OST.

7.1.3 Large-scale HPC Workloads. We generate the traces of three real-world HPC workloads from different domain sciences using the process discussed in Section 6.3. The details of the workloads that ran on Summit [41]—the world's second-largest supercomputer based on the latest Top500 list [82]—are outlined in Table 7.

The *genomicPrediction* code uses the DeepGP package [110], which implements **multilayer perceptron (MLP)** networks, **convolutional neural network (CNN)**, ridge regression, and lasso regression for genomic prediction. Therefore, this workload implements deep learning models for predicting complex traits that fall in the category of emerging HPC workloads. For this reason, we see comparatively higher reads than writes in this workload.

The ***Energy Exascale Earth System Model (E3SM)*** [98] workload is part of an ongoing, state-of-the-science earth system modeling, simulation, and prediction project that optimizes the use of **Department of Energy (DOE)** laboratory resources to meet the science needs of the nation and the mission needs of DOE. A major motivation for the E3SM project is the paradigm shift in computing architectures and their related programming models as capability moves into the exascale era. The E3SM model simulates the fully coupled climate system at high-resolution (15–25 km) and will include coupling with energy systems; it has a unique capability for variable resolution modeling using unstructured grids in all its earth system component models. Therefore, this workload represents the exascale real-world HPC workload, which is simulation workload, and thus produces much higher writes than reads.

The *cosmoFlow* [50] workload aims to process large 3D cosmology datasets on modern HPC platforms. It adapts the deep learning network to a scalable architecture for a large problem size of voxels and predicts three cosmological parameters. The workload uses efficient primitives in MKL-DNN for 3D convolutional neural networks, which are used in an optimized TensorFlow [1] framework for CPU architectures. Thus, this workload represents the extreme use case of emerging machine learning workloads on HPC systems, which is shown by the large difference in the amount of data read and written.

Table 8. PFL Striping Layouts Used for the Evaluation

PFL Configuration 1		PFL Configuration 2	
Extent Range	***Stripe Count***	***Extent Range***	***Stripe Count***
[0, 128 MB)	1	[0, 128 MB)	1
[128 MB, 512 MB)	3	[128 MB, 2 GB)	12
[512 MB, 2 GB)	8	[2 GB, EOF)	32
[2 GB, EOF)	16		

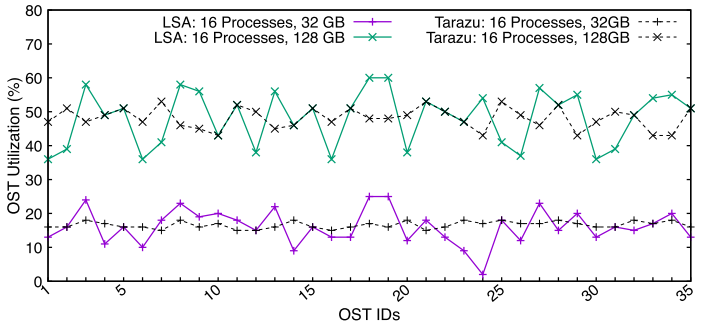


Fig. 15. OST utilization for IOR for FPP and non-PFL layout (stripe count = 8).

These traces of these three workloads are fed as input (representing data usage by client nodes) to the simulator described in Section 6.2. The evaluation is described in Section 7.5.

7.1.4 File Striping Layouts. For the non-PFL setup, the stripe count is set to 8 and the stripe size is calculated as described in Section 5.5.2. To evaluate `Tarazu` with PFL, we use two different PFL configurations, as shown in Table 8. *Configuration 1* was introduced in Section 2.3 and reflects a standard choice for a PFL file layout as described in the Lustre operations manual [61]. *Configuration 2* is a three-component PFL layout, which exceeds the number of available OSTs in our cluster system. Hence, according to the Lustre manual, *Configuration 2* should result in worse I/O performance than *Configuration 1* for our cluster of only 35 OSTs, while providing comparable results as Configuration 1 on our large-scale simulator.

7.2 OST Utilization

7.2.1 Load Balance for IOR in FPP Mode. Figure 15 shows the comparison of load under `Tarazu` and the default LSA allocation policy on 35 OSTs in our cluster setup in terms of *OST Utilization* for the IOR benchmark with different data sizes. We see that `Tarazu` balances the load on all OSTs in a near-optimal manner. For example, for 16 processes and 32 GB of data in total, the maximum load observed with LSA is on OST-18 with an OST utilization of 25%, while the average utilization is 16.06%, resulting in an *OST Cost* of 1.56. In contrast, the maximum load observed under `Tarazu` is 18% with the corresponding *OST Cost* of 1.08, i.e., we observe a near-optimal resource utilization. The almost horizontal line for the *OST Utilization* for `Tarazu` emphasizes its effectiveness. Overall, `Tarazu` is able to reduce the *OST Cost* by 30.8% compared to the default approach. We observe a similar trend for IOR with 16 processes and 128 GB of data in total. In this case, the *OST Cost* is 1.25 and 1.1 under LSA and `Tarazu`, respectively, which means that `Tarazu` provides a 12% better *OST Cost* than the default LSA policy.

As can be seen in Figures 16 and 17, `Tarazu` is able to balance the load near-optimal for IOR using PFL Configuration 1 and 2 as well. For 16 processes writing 32 GB data in total, we obtain

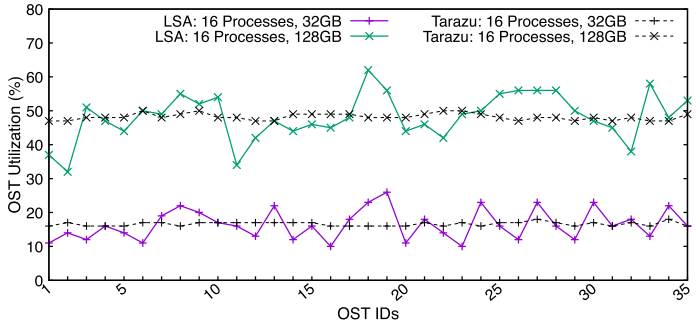


Fig. 16. OST utilization for IOR in FPP mode and PFL Configuration 1.

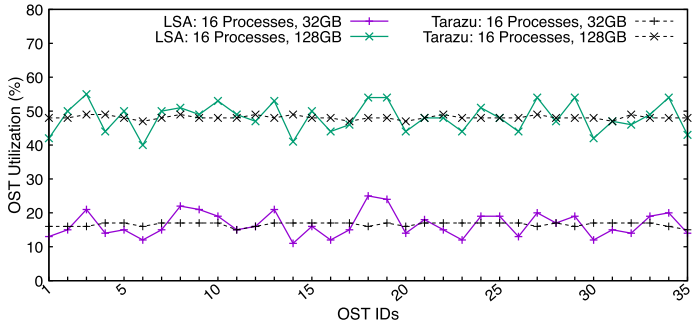


Fig. 17. OST utilization for IOR in FPP mode and PFL Configuration 2.

an OST cost of 1.58 and 1.08 under LSA and Tarazu, respectively, for PFL Configuration 1, and an *OST cost* of 1.5 and 1.02 under LSA and Tarazu, respectively, for PFL Configuration 2, thereby providing a 31.6% and 32% improvement in load balance. Similar results can be seen for IOR with 16 processes and 128 GB of total file data.

It is important to note that for large files, such as the second IOR experiment with 8 GB non-PFL files per process, the data distribution with Tarazu is less optimal than with small non-PFL files or PFL files in general. This is because typically the stripe count is set to a fixed default value by the system administrator, in this case eight, and the complete file is divided in the specified number of stripes with relatively large stripes. Hence, PFL is much better suited for real-world HPC applications and therefore significantly contributes to the effectiveness of Tarazu.

7.2.2 Load Balance for HACC-I/O in FPP Mode. For HACC-I/O, we evaluate Tarazu with 8 processes with 50 *million* particle data and 16 processes with 20 *million* particle data. Each process creates one file that is stored on the Lustre OSTs with the default non-PFL layout (i.e., stripe count of 8) and using the two PFL configurations. The total data generated is approximately 14.3 *GB* and 11.7 *GB* for 8 and 16 processes, respectively.

Similar to IOR, we observe a significant improvement in load balancing for HACC-I/O, as shown in Figure 18, when compared to the default LSA policy. Note that C1 denotes PFL Configuration 1 and C2 Configuration 2, while N refers to non-PFL in Figure 18. The *OST Cost* for 16 processes with LSA and Tarazu for non-PFL layout is 1.8 and 1.2, respectively, where Tarazu reduces the *OST Cost* by 33%. A similar behavior is observed for PFL, where the *OST Cost* is significantly lower than for the LSA policy and is close to 1.00 for Tarazu, i.e., close to optimal.

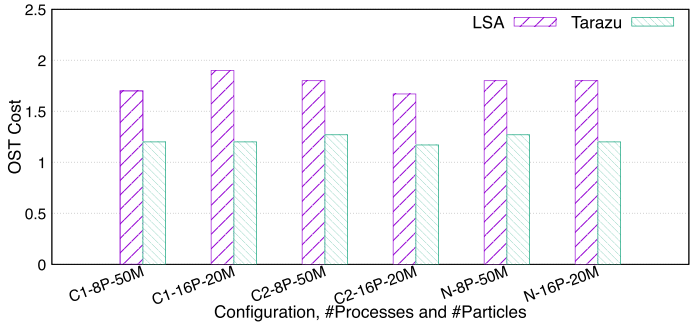


Fig. 18. OST cost for HACC-I/O in FPP mode. On the x-axis, C1 represents experiments with PFL Configuration 1, C2 represents experiments with PFL Configuration 2, and N represents non-PFL results.

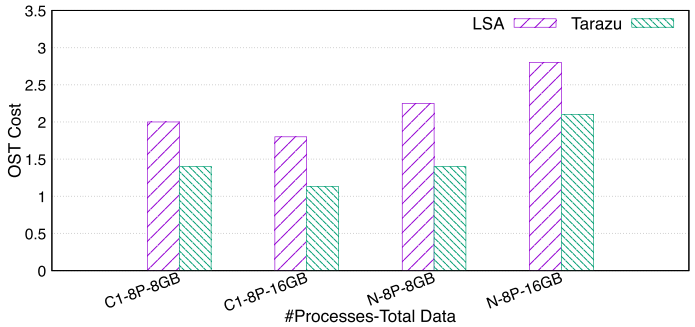


Fig. 19. OST Cost for IOR in SSF mode. On the x-axis, C1 represents experiments with PFL Configuration 1 and N represents non-PFL results.

7.2.3 Load Balance for Single Shared Files. In SSF mode, all processes write into and read from a single shared file. We run IOR in SSF mode for both non-PFL and PFL Configuration 1. We run IOR with 8 processes generating 8 *GB* and 16 *GB* files. The results are shown in Figure 19. We observe that Tarazu is able to reduce the *OST cost* for all scenarios when compared to the default LSA approach. The *OST cost* can be reduced by 38% and 37.8% in non-PFL and PFL layouts, respectively. Also, as discussed in the previous section, we observe that Tarazu provides better I/O performance for SSF when compared to LSA. When comparing PFL versus non-PFL files, overall the *OST cost* is lower when using PFL for single shared files.

7.2.4 Load Balance for Concurrent Application Runs. We also evaluate Tarazu with concurrent application runs by simultaneously running IOR and HACC-I/O with different job configurations from two different client nodes. Each client runs 8 MPI processes. Per process, one file is created and stored on the OSTs with PFL Configuration 1 for IOR and PFL Configuration 2 for HACC-I/O. The total number of particle data stored for each file for HACC-I/O is 50 *million*, and the total data size for IOR is 64 *GB*. As with the single application tests, we observe a significant improvement in load balancing for concurrent applications compared to LSA. Figure 20 shows the comparison of load under both approaches on our tesbed. We see that Tarazu balances the load over all OSTs. The maximum load observed with LSA is on OST9 with a utilization of 51%, while the average utilization is 35%, resulting in an *OST Cost* of 1.46. In contrast, under Tarazu, the *OST Cost* observed is 1.08, thereby improving the load balance by 26%. Moreover, the CPU utilization and memory usage on the MDS while using Tarazu for load balancing in concurrent application

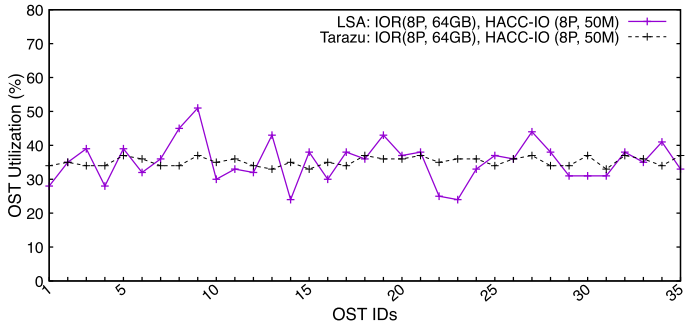


Fig. 20. OST utilization for a simultaneous execution of IOR and HACC-I/O in FPP mode with PFL Configuration 1 and PFL Configuration 2, respectively.

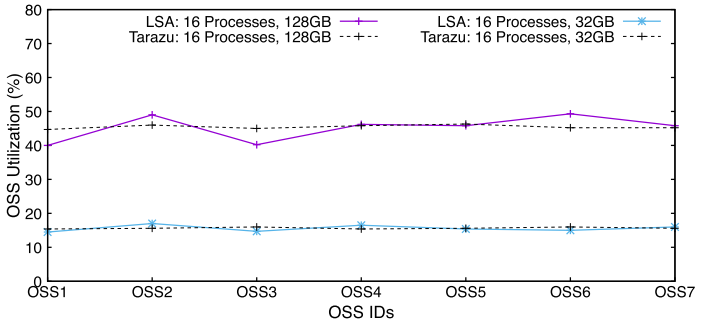


Fig. 21. OSS utilization for IOR in FPP mode for PFL Configuration 1.

runs is observed to be about 1.55% and 0.12%, respectively. We observe similar improvements in load balance for concurrent application in the non-PFL striping pattern as well.

7.3 OSS Utilization

We want to achieve an end-to-end load balance in the file system. Therefore, `Tarazu` needs to balance the load on both OSSs and OSTs for an overall load-balanced setup. In our next experiment, we measure the utilization of each OSS under the default Lustre allocation policy and `Tarazu`. To this end, we aggregate the load (storage utilization) on each OST and calculate the ratio of storage being used with respect to the total storage in each OSS. In a balanced scenario, each OSS should be utilized equally by hosting an equal share of application data.

Figure 21 shows the comparison of *OSS Utilization* of all seven OSSs of our testbed under `Tarazu` in comparison to the default approach. For the sake of brevity, we present the results of IOR with 16 processes in FPP mode, storing a total application data of 32 *GB* and 128 *GB* under PFL Configuration 1. We observe that with LSA, the OSSs are slightly imbalanced, while `Tarazu` provides a near-optimal load balance across the available OSSs.

7.4 I/O Performance

Next, we compare the effective read and write performance for HACC-I/O and IOR. We measure the *I/O rate* for storing the data to and reading the data from the OSTs. In Figures 22(a)–22(c), the read performance results are shown for the FPP and SSF sharing modes with both PFL and non-PFL striping layouts. We see an improvement of up to 43% in read performance for `Tarazu` compared

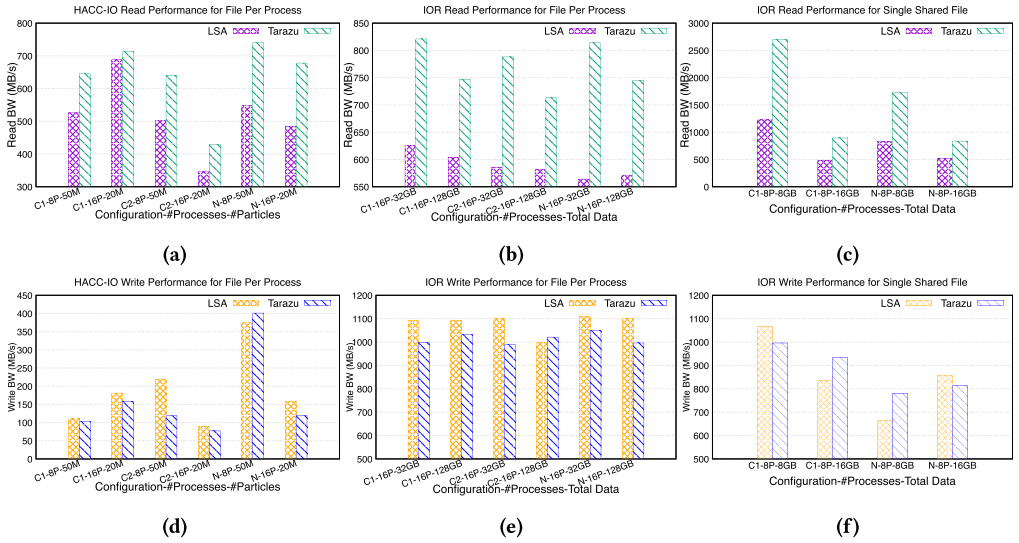


Fig. 22. Read and write performance of IOR and HACC-I/O for FPP and SSF with PFL and non-PFL layouts. On the x-axis, C1 represents experiments with PFL Configuration 1, C2 represents experiments with PFL Configuration 2, and N represents non-PFL results.

to the standard Lustre allocation policy. This improvement is achieved by evenly loading the OSTs and OSSs, which mitigates resource contention and thus improves parallel data access.

It should be noted that the read performance improvements for FPP with PFL Configuration 2 (denoted as C2 in the graphs) are marginally lower than with Configuration 1. These results are consistent with our earlier assumption that I/O performance degrades when the total number of stripes in a PFL file exceeds the number of available OSTs. A similar trend can be observed for experiments in the SSF mode, which is why we did not pursue any further experiments with PFL Configuration 2 on our 10-node testbed. In addition, it should be noted that for small systems such as our Lustre testbed, there is no significant benefit from using PFL, because the system and workload are too small to benefit from such an advanced feature.

The write performance results for HACC-I/O and IOR are shown in Figures 22(d)–22(f). Please note that the displayed results are small-scale experiments run on a rather small testbed with only seven OSSs and two client nodes. As the number of writing processes increases, the number of competing processes on the small testbed system increases. This results in increased file locking contention when accessing MDS and OSSs. Hence, the write performance with `Tarazu` is impacted by up to 10%, especially with PFL Configuration 2. When using PFL Configuration 1 and non-PFL settings, the write bandwidth is on par with or marginally lower compared to the default LSA policy. Since the main target of our end-to-end I/O control plane are data-intensive workloads with a larger amount of I/O requests and varying I/O request sizes (i.e., read/write data sizes) on HPC production environments, the observed overhead during file creation calls and allocation of appropriate OSTs for applications with limited I/O needs to be further investigated but can be neglected in most real-world scenarios. As can be seen in Section 7.5, the overhead is even more negligible or non-existent for concurrent applications on large-scale parallel storage deployments. Overall, we believe that the significant improvements in resource utilization and read performance outweigh the overhead observed when writing data to the parallel file system.

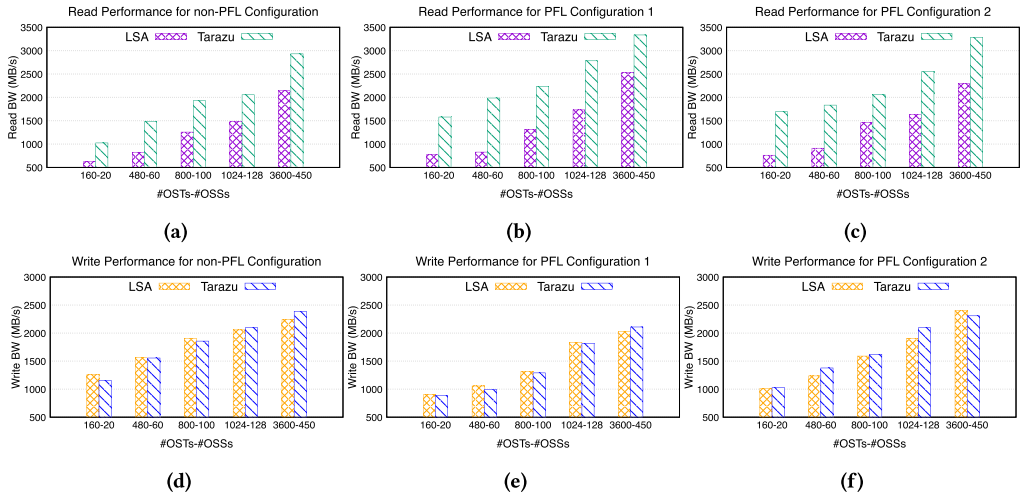


Fig. 23. Read and write performance of concurrent real-world application runs for different system scales.

7.5 Scalability Study

To showcase the scalability of our proposed framework, we evaluate `Tarazu` with the discrete-time simulator introduced in Section 6.2 using three real-world HPC application traces described in Section 7.1.3. We evaluate the scalability of `Tarazu` for running multiple HPC workloads simultaneously on a large number of compute nodes reading and writing data on a large number of OSTs, corresponding to the scale of a real HPC cluster. For the scalability study, we use three real-world HPC application traces, namely, *genomicPrediction*, *E3SM*, and *cosmoFlow* (see Section 7.1.3 for additional details), concurrently on the simulator and scale the number of OSTs from 160 to 3,600. In all configurations, the capacity of each OST is 10 GB and there are eight OSTs connected to each OSS, achieving a maximum file system capacity of 36 TB. While the capacity of the simulated file systems may appear to be rather small, the interesting part for this work is to scale the number of OSTs. The more OSSs and OSTs available, the higher the achievable aggregate I/O bandwidth. At the same time, the potential for load imbalance increases significantly. As an example, consider the storage system of the first exascale system Frontier [43]. It consists of 450 OSS nodes each equipped with 2× HDD OSTs and 1× NVMe OST. The configuration of our simulations is therefore representative for current exascale systems, at least from the OST perspective. The selected workloads provide a representative mix of emerging deep and machine learning and traditional simulation workloads.

Figures 23(a)–23(c) show the read performance results of the concurrent application runs using the non-PFL, PFL Configuration 1, and PFL Configuration 2 file layouts. For all file striping layouts, `Tarazu` outperforms LSA, and the overall performance with PFL exceeds that of non-PFL runs. The selected real-world application traces portray a mixed I/O workload with different file sizes. PFL was specifically designed to cope with such mixed workloads, which is reflected in the improved performance results. For example, for the configuration with 3,600 OSTs and 450 OSSs, `Tarazu` with PFL Configuration 1 provides a read bandwidth of 3.34 GB/s, which corresponds to a performance increase of 36.3%, while the read bandwidth with LSA is only 2.1 GB/s. As expected, both PFL configurations perform equally well on large-scale storage systems.

The simulator confirms that `Tarazu` scales well for running concurrent applications on medium and large Lustre installations. In contrast to the write performance results on our small Lustre testbed, the write bandwidth with `Tarazu` is equal to or better than LSA, as shown in

Figures 23(d)–23(f). This validates that Tarazu poses no bottleneck or significant overhead for real-world deployments. Instead, Tarazu provides a near-optimal load balance and a significantly improved read performance, which is critical for the post-processing of large-scale application data.

8 DISCUSSION AND FUTURE WORK

8.1 Emerging Workloads

High performance computing (HPC) workloads are no longer restricted to traditional checkpoint/restart applications. The rise in popularity and functionality of machine learning and deep learning approaches in various science domains, such as biology, earth science, and physics, has led to the read-intensive nature of applications [68]. Tarazu helps in improving the read throughput of such applications, as shown in the results of cosmoFlow with Tarazu. Moreover, applications will increasingly make use of the PFL feature of Lustre, which allows a higher flexibility during file creation. As part of our future work, we aim to integrate Tarazu with Lustre and test its efficacy on the world's first exascale supercomputer Frontier [42].

8.2 Initial Training through Statistical Analysis and Darshan Logs

Currently, Tarazu uses miniRecorder on the client side to record the I/O characteristics of applications. This information is then parsed, processed, and passed to the prediction model. To further automate Tarazu's workflow, we plan to support logs from popular I/O characterization and tracing tools like Darshan as input to our prediction model. This will further facilitate the use and integration of Tarazu into production systems, as many HPC centers already use tools such as Darshan by default for system analysis. To further fine-tune the prediction model and placement algorithm, modular and automated workload analysis frameworks such as MAWA-HPC [109] could be used as an additional input on the clients.

8.3 Rebalancing of Existing Application Datasets

One of the main limitations of Tarazu is that the load-balancing mechanism is only applied to newly generated data, and only when the placement library is explicitly used by the user. However, since read-intensive workloads in particular can benefit greatly from Tarazu, the integration of a rebalancing mechanism running in the background for already existing data and training sets is quite reasonable. This can be achieved, for example, by a server-side daemon that redistributes already existing data when the server load is low.

8.4 Metadata Load Balancing

One of the main contributions of Tarazu is the creation of a global view of all system resources along the I/O path, which is then used to guide data placement decisions and achieve load balancing by taking global statistics into account. Future systems will continue to scale, and we expect backend I/O servers and metadata servers to scale accordingly. Centralized metadata servers will therefore no longer be viable, as they serialize parallel file accesses, which can become a significant performance bottleneck. With the introduction of Lustre's **Distributed Name Space (DNE)** feature, the metadata load for a single file system can be distributed across multiple metadata servers. DNE is realized through a mechanism called *striped directories*. Here, the structure of a directory is split into shards that are distributed across multiple MDTs. However, the user must specify the layout for a specific directory for each application run or workflow.

In the future, we plan to extend the prediction model to not only facilitate sophisticated file striping layouts for individual files, but also provide transparent metadata load balancing to better support data-intensive workloads.

8.5 Integration of Tarazu into Other Hierarchical HPC File Systems

The latest implementation of Tarazu relies on Lustre's monitoring capabilities. Similar monitoring functionalities can be found in other multi-tier file systems such as BeeGFS [30], IBM GPFS (Spectrum Scale) [77], and Ceph [63, 90, 92]. Although our particular prototype is based on Lustre, the metrics and architectural component assumptions we have chosen can largely be generalized across the spectrum of large-scale parallel file systems, such as decoupling of MDSs and data servers (OSDs), data striping, and striping width and length. For example, Ceph's built-in *ceph mon dump* and GPFS's *mmpmon* provide even more detailed data collection capabilities and are compatible with our proposed algorithms.

The main challenge for supporting file systems such as BeeGFS and Ceph will be the integration of the stripe placement mechanisms. Here, special API extensions will be necessary to apply the striping patterns and advanced file layouts. For example, we are currently working on an extension to the Ceph user API that will adapt striping functionality from the llapi library to Ceph.

8.6 System-specific Autotuning of PFL Configuration

PFL simplifies the use of Lustre so users can expect reasonable performance for a variety of file I/O patterns without having to explicitly understand the parallel I/O model. Specifically, users do not necessarily need to know the size or concurrency of the output files before they are created and explicitly specify an optimal layout for each file to achieve good performance. Therefore, the integration of features like PFL is an essential step to support future HPC workloads. For PFL, it is recommended that small files have a lower stripe count (to reduce overhead), and as the file size increases, the stripe count should also be increased (to spread the storage footprint and increase bandwidth). In addition, the layout should only be expanded until the total number of strips reaches or exceeds the number of OSTs. At this point, it is beneficial to add a final layout expansion to EOF that spans all available OSTs to maximize the bandwidth at the end of the file (if it continues to grow significantly in size).

Currently, Tarazu only supports the use of a static PFL configuration. Discussions with system administrators of current HPC systems have confirmed that today's Lustre deployments often use system-wide static PFL configurations. This creates a tradeoff between I/O bandwidth performance and usability, as few users know which striping settings are beneficial for their workload. To exploit the full performance potential, we plan to develop a PFL tuner that can automatically determine the best PFL configuration, depending on the underlying parallel storage system and the application I/O pattern. Here, we will also consider the new ***Self-Extending Layout*** **(SEL)** [61] feature, which is an extension to PFL. SEL allows the MDS to change the defined PFL configuration dynamically.

8.7 Further Improving the Scalability of Tarazu

For simplicity reasons, Tarazu currently relies on MySQL databases for the implementation of the interaction database. However, relational databases do not perform well when a wide variety of information needs to be queried, because relational databases such as MySQL store data in an organized manner. Document-oriented databases like MongoDB, a type of NoSQL database, can store huge amounts of data and also have very powerful query and indexing capabilities. Future work will address the performance bottleneck in our design by evaluating different database technologies for the interaction database. This will minimize the incurred overhead when intercepting the file creation requests and therefore improve the I/O latency for file creation and write requests.

9 CONCLUSION

This article proposes Tarazu, an end-to-end control plane for optimizing parallel and distributed HPC I/O systems such as Lustre by providing efficient load balancing among storage servers.

Tarazu offers a global view of the system, enables coordination between clients and servers, and manages performance degradation due to resource contention by accounting for operations on both clients and servers. Our implementation facilitates near-optimal load balancing across OSTs and OSSs in Lustre and is able to handle concurrent execution of workloads with different file I/O sizes by supporting both PFL and non-PFL file layouts in a scalable and distributed manner.

We evaluate Tarazu on a real Lustre testbed with the benchmarks IOR and HACC-I/O for various file layouts and with different SSF and FPP sharing strategies. Compared to the default Lustre LSA policy, Tarazu provides up to 33% improvement in balancing the load. Moreover, we also observe an I/O performance improvement of up to 43% for reads without affecting the performance for writes. In addition, we design a time-discrete simulator based on ns-3 and use three real-world HPC application traces from the Summit supercomputer to demonstrate the effectiveness and scalability of the transparent design of Tarazu on large-scale storage systems. The selected workloads show that Tarazu is equally suitable for traditional simulation workloads and emerging HPC workloads.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 265–283. Retrieved from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Megha Agarwal, Divyansh Singhvi, Preeti Malakar, and Suren Byna. 2019. Active learning-based automatic tuning and prediction of parallel I/O performance. In *IEEE/ACM 4th International Parallel Data Systems Workshop (PDSW'19)*. IEEE, 20–29. DOI: https://doi.org/10.1109/PDSW49588.2019.00007

[3] Ravindra K. Ahuja. 2017. *Network Flows: Theory, Algorithms, and Applications*. Pearson Education, Chennai, India.

[4] Ali Anwar. 2018. *Towards Efficient and Flexible Object Storage Using Resource and Functional Partitioning*. Ph. D. Dissertation. Virginia Tech.

[5] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. 2016. MOS: Workload-aware elasticity for cloud object stores. In *25th ACM International Symposium on High-performance Parallel and Distributed Computing (HPDC'16)*. ACM, New York, NY, 177–188. DOI: https://doi.org/10.1145/2907294.2907304

[6] Ayse Bagbaba. 2020. Improving collective I/O performance with machine learning supported auto-tuning. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'20)*. IEEE, 814–821. DOI: https://doi.org/10.1109/IPDPSW50202.2020.00138

[7] Ayşe Bağbaba and Xuan Wang. 2021. Improving the MPI-IO performance of applications with genetic algorithm based auto-tuning. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'21)*. IEEE, 798–805. DOI: https://doi.org/10.1109/IPDPSW52791.2021.00118

[8] Babak Behzad, Surendra Byna, and Marc Snir. 2019. Optimizing I/O performance of HPC applications with autotuning. *ACM Trans. Parallel Comput.* 5, 4 (2019), 27 pages. DOI: https://doi.org/10.1145/3309205

[9] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. 2023. I/O access patterns in HPC applications: A 360-degree survey. *ACM Comput. Surv.* 56, 2 (2023), 41 pages. DOI: https://doi.org/10.1145/3611007

[10] P. J. Braam. 2004. *The Lustre Storage Architecture (Tech. Rep.)*. Technical Report. Retrieved from http://wiki.lustre.org/

[11] Peter J. Brockwell and Richard A. Davis. 2016. *Introduction to Time Series and Forecasting* (3rd ed.). Springer International Publishing, Cham, Switzerland.

[12] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 characterization of petascale I/O workloads. In *IEEE International Conference on Cluster Computing and Workshops*. IEEE, 12 pages. DOI: https://doi.org/10.1109/CLUSTR.2009.5289150

[13] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.* 9, 3 (1999), 224–253. DOI: https://doi.org/10.1145/347823.347828

[14] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/O characterization and performance evaluation of BeeGFS for deep learning. In *48th International Conference on Parallel Processing (ICPP'19)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3337821.3337902

[15] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross. 2011. CODES: Enabling Co-design of multi-layer exascale storage architectures. In *Workshop on Emerging Supercomputing Technologies*.

[16] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Tainã Coleman, Dan Laney, Dong Ahn, Shantenu Jha, Dorran Howell, Stian Soiland-Reyes, Ilkay Altintas, Douglas Thain, Rosa Filgueira, Yadu N. Babuji, Rosa M. Badia, Bartosz Balis, Silvina Caíno-Lores, Scott Callaghan, Frederik Coppens, Michael R. Crusoe, Kaushik De, Frank Di Natale, Tu Mai Anh Do, Bjoern Enders, Thomas Fahringer, Anne Fouilloux, Grigori Fursin, Alban Gaignard, Alex Ganose, Daniel Garijo, Sandra Gesing, Carole A. Goble, Adil Hasan, Sebastiaan Huber, Daniel S. Katz, Ulf Leser, Douglas Lowe, Bertram Ludäscher, Ketan Maheshwari, Maciej Malawski, Rajiv Mayani, Kshitij Mehta, André Merzky, Todd S. Munson, Jonathan Ozik, Loïc Pottier, Sashko Ristov, Mehdi Roozmeh, Renan Souza, Frédéric Suter, Benjamín Tovar, Matteo Turilli, Karan Vahi, Alvaro Vidal-Torreira, Wendy R. Whitcup, Michael Wilde, Alan Williams, Matthew Wolf, and Justin M. Wozniak. 2021. Workflows community summit: Advancing the state-of-the-art of scientific workflows management systems research and development. *CoRR* abs/2106.05177 (2021)

[17] C. S. Daley, D. Ghoshal, G. K. Lockwood, S. Dosanjh, L. Ramakrishnan, and N. J. Wright. 2020. Performance characterization of scientific workflows for the optimal use of Burst Buffers. *Fut. Gen. Comput. Syst.* 110 (2020), 468–480. DOI: https://doi.org/10.1016/j.future.2017.12.022

[18] Bin Dong, Xiuqiao Li, Qimeng Wu, Limin Xiao, and Li Ruan. 2012. A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers. *J. Parallel Distrib. Comput.* 72, 10 (2012), 1254–1268. DOI: https://doi.org/10.1016/j.jpdc.2012.05.006

[19] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. 2012. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O. In *IEEE International Conference on Cluster Computing*. IEEE, 155–163. DOI: https://doi.org/10.1109/CLUSTER.2012.26

[20] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. 2015. Using formal grammars to predict I/O behaviors in HPC: The Omnisc'IO approach. *IEEE Trans. Parallel Distrib. Syst.* 27, 8 (2015), 2435–2449. DOI: https://doi.org/10.1109/TPDS.2015.2485980

[21] Miguel A. Erazo, Ting Li, Jason Liu, and Stephan Eidenbenz. 2012. Toward comprehensive and accurate simulation performance prediction of parallel file systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*. IEEE, 1–12. DOI: https://doi.org/10.1109/DSN.2012.6263930

[22] Oak Ridge Leadership Facility. 2022. *Titan Supercomputer*. Oak Ridge National Laboratory. Retrieved from https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/

[23] Bo Feng, Ning Liu, Shuibing He, and Xian-He Sun. 2014. HPIS3: Towards a high-performance simulator for hybrid parallel I/O and storage systems. In *9th Parallel Data Storage Workshop*. IEEE, 37–42. DOI: https://doi.org/10.1109/PDSW.2014.12

[24] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. 2017. A characterization of workflow management systems for extreme-scale applications. *Fut. Gen. Comput. Syst.* 75 (2017), 228–238. DOI: https://doi.org/10.1016/j.future.2017.02.026

[25] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *EDBT/ICDT'11 Workshop on Array Databases*. ACM, New York, NY.

[26] John Fragalla, Bill Loewe, and Torben Kling Petersen. 2020. New Lustre features to improve Lustre metadata and small-file performance. *Concurr. Computat.: Pract. Exper.* 32, 20 (2020), 6 pages. DOI: https://doi.org/10.1002/cpe.5649

[27] Jim Garlick. 2010. Lustre Monitoring Tool (LMT). https://github.com/LLNL/lmt

[28] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. 2013. HACC: Extreme scaling and performance across diverse architectures. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2503210.2504566

[29] Shuibing He, Xian-He Sun, and Adnan Haider. 2015. HAS: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE. DOI: https://doi.org/10.1109/IPDPS.2015.23

[30] Jan Heichler. 2014. An Introduction to BeeGFS v1.1. https://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf. Accessed February 16, 2024.

[31] Pieter Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc.

[32] L. Huang and S. Liu. 2020. OOOPS: An innovative tool for IO workload management on supercomputers. In *IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS'20)*. IEEE. DOI: https://doi.org/10.1109/ICPADS51040.2020.00069

[33] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, Xiyang Wang, Nosayba El-Sayed, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. Automatic, application-aware I/O forwarding resource allocation. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 265–279. Retrieved from https://www.usenix.org/conference/fast19/presentation/ji

[34] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, Yongseok Son, and Hyeonsang Eom. 2020. Towards HPC I/O performance prediction through large-scale log analysis. In *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, New York, NY, 77–88. DOI: https://doi.org/10.1145/3369583.3392678

[35] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. 2012. IOPin: Runtime profiling of parallel I/O in HPC systems. In *SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 18–23. DOI: https://doi.org/10.1109/SC.Companion.2012.14

[36] Donghun Koo, Jik-Soo Kim, Soonwook Hwang, Hyeonsang Eom, and Jaehwan Lee. 2016. Utilizing progressive file layout leveraging SSDs in HPC cloud environments. In *IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W'16)*. IEEE, 90–95. DOI: https://doi.org/10.1109/FAS-W.2016.30

[37] Anoop S. Kumar and Somnath Mazumdar. 2016. Forecasting HPC workload using ARMA models and SSA. In *International Conference on Information Technology (ICIT'16)*. IEEE, 294–297. DOI: https://doi.org/10.1109/ICIT.2016.065

[38] Parallel Data Lab. 2024. *The DiskSim Simulation Environment (V4.0)*. Carnegie Mellon University. Retrieved from: https://www.pdl.cmu.edu/DiskSim/

[39] Lawrence Berkeley National Laboratory and Argonne National Laboratory. 2022. TOKIO: Total Knowledge of I/O. Retrieved from https://www.nersc.gov/research-and-development/storage-and-i-o-technologies/tokio/

[40] Lawrence Livermore National Laboratory. 2021. IOR Benchmark Summary. Retrieved from https://asc.llnl.gov/sequoia/benchmarks/IORsummaryv1.0.pdf

[41] Oak Ridge National Laboratory. 2021. Summit Supercomputer. Retrieved from https://www.olcf.ornl.gov/summit/

[42] Oak Ridge National Laboratory. 2022. Frontier Supercomputer. Retrieved from https://www.olcf.ornl.gov/frontier/

[43] Dustin Leverman. 2022. Oak Ridge National Laboratory Storage Ecosystem. In *Platform for Advanced Scientific Computing Conference (PASC'22)*. Retrieved from https://linklings.s3.amazonaws.com/organizations/pasc/pasc22/submissions/stype117/PYFgV-msa274s1.pdf

[44] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. 2014. Enabling dynamic file I/O path selection at runtime for parallel file system. *J. Supercomput.* 68, 2 (2014), 996–1021. DOI: https://doi.org/10.1007/s11227-013-1077-6

[45] Seung-Hwan Lim, Hyogi Sim, Raghul Gunasekaran, and Sudharshan S. Vazhkudai. 2017. Scientific user behavior and data-sharing trends in a petascale file system. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. ACM, New York, NY, 1–12. DOI: https://doi.org/10.1145/3126908.3126924

[46] Yonggang Liu, Renato Figueiredo, Dulcardo Clavijo, Yiqi Xu, and Ming Zhao. 2011. Towards simulation of parallel file system scheduling algorithms with PFSsim. In *7th IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'11)*. IEEE, 12 pages.

[47] Yonggang Liu, Renato Figueiredo, Yiqi Xu, and Ming Zhao. 2013. On the design and implementation of a simulator for parallel file system research. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*. IEEE, 5 pages. DOI: https://doi.org/10.1109/MSST.2013.6558438

[48] Glenn K. Lockwood, Wucherl Yoo, Suren Byna, Nicholas J. Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. 2017. UMAMI: A recipe for generating meaningful metrics through holistic I/O performance analysis. In *2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. ACM, New York, NY, 55–60. DOI: https://doi.org/10.1145/3149393.3149395

[49] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. 2013. A multi-level approach for understanding I/O activity in HPC applications. In *IEEE International Conference on Cluster Computing (CLUSTER'13)*. IEEE, 5 pages. DOI: https://doi.org/10.1109/CLUSTER.2013.6702690

[50] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, P. Prabhat, and V. Lee. 2018. CosmoFlow: Using deep learning to learn the universe at scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. IEEE, 11 pages. DOI: https://doi.org/10.1109/SC.2018.00068

[51] Rick Mohr, Michael Brim, Sarp Oral, and Andreas Dilger. 2016. Evaluating Progressive File Layouts for Lustre. In *Cray User Group Conference (CUG'16)*.

[52] E. Molina-Estolano, C. Maltzahn, J. Bent, and S. A. Brandt. 2009. Building a parallel file system simulator. *J. Phys.: Confer. Series* 180 (July 2009), 012050. DOI: https://doi.org/10.1088/1742-6596/180/1/012050

[53] Frank Mueller, Xing Wu, Martin Schulz, Bronis R. de Supinski, and Todd Gamblin. 2012. ScalaTrace: Tracing, analysis and modeling of HPC codes at scale. In *Applied Parallel and Scientific Computing*. Springer Berlin, 410–418. DOI: https://doi.org/10.1007/978-3-642-28145-7_40

[54] Lustre Networking. 2008. High-Performance Features and Flexible Support for a Wide Array of Networks.

[55] Sarah Neuwirth. 2018. *Accelerating Network Communication and I/O in Scientific High Performance Computing Environments*. Ph. D. Dissertation. Heidelberg University, Germany.

[56] Sarah Neuwirth, S. Oral, F. Wang, and Ulrich Bruening. 2016. An I/O load balancing framework for large-scale applications (BPIO 2.0). In *Poster at SC'16*.

[57] Sarah Neuwirth and Arnab K. Paul. 2021. Parallel I/O evaluation techniques and emerging HPC workloads: A perspective. In *IEEE International Conference on Cluster Computing (CLUSTER'21)*. IEEE, 671–679. DOI: https://doi.org/10.1109/Cluster48925.2021.00100

[58] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. 2017. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS'17)*. IEEE, 604–613. DOI: https://doi.org/10.1109/ICPADS.2017.00084

[59] NSNAM. 2023. *NS-3 Network Simulator*. Retrieved from https://www.nsnam.org/

[60] James Oly and Daniel A. Reed. 2002. Markov model prediction of I/O requests for scientific applications. In *16th International Conference on Supercomputing (ICS'02)*. ACM, New York, NY, 147–155. DOI: https://doi.org/10.1145/514191.514214

[61] OpenSFS and EOFS. 2021. Lustre Operations Manual 2.x. Retrieved from https://www.lustre.org/documentation/

[62] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, Saurabh Gupta, Devesh Tiwari Sudharshan S. Vazhkudai, James H. Rogers, David Dillow, Galen M. Shipman, and Arthur S. Bland. 2014. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 217–228. DOI: 10.1109/SC.2014.23

[63] Akshita Parekh, Urvashi Karnani Gaur, and Vipul Garg. 2020. Analytical modelling of distributed file systems (GlusterFS and CephFS). In *Reliability, Safety and Hazard Assessment for Risk-Based Technologies*. Springer Singapore, 213–222. DOI: https://doi.org/10.1007/978-981-13-9008-1_18

[64] Tirthak Patel and Suren Byna. 2020. Uncovering access, reuse, and sharing characteristics of I/O-intensive files on large-scale production HPC systems. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 91–101. Retrieved from https://www.usenix.org/conference/fast20/presentation/patel-hpc-systems

[65] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. 2019. Revisiting I/O behavior in large-scale storage systems: The expected and the unexpected. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. ACM, New York, NY, Article 65, 13 pages. DOI: https://doi.org/10.1145/3295500.3356183

[66] Arnab K. Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali R. Butt. 2020. Understanding HPC application I/O behavior using system level statistics. In *IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC'20)*. IEEE, 202–211. DOI: https://doi.org/10.1109/HiPC50609.2020.00034

[67] Arnab K. Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R. Butt, Michael J. Brim, and Sangeetha B. Srinivasa. 2017. I/O load balancing for big data HPC applications. In *IEEE International Conference on Big Data (Big Data'17)*. IEEE, 233–242. DOI: https://doi.org/10.1109/BigData.2017.8257931

[68] Arnab K. Paul, Ahmad Maroof Karimi, and Feiyi Wang. 2021. Characterizing machine learning I/O workloads on leadership scale HPC systems. In *29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'21)*. IEEE, 1–8. DOI: https://doi.org/10.1109/MASCOTS53633.2021.9614303

[69] Arnab Kumar Paul and Bibhudatta Sahoo. 2017. Dynamic virtual machine placement in cloud computing. In *Resource Management and Efficiency in Cloud Computing Environments*. IGI Global, 136–167. DOI: https://doi.org/10.4018/978-1-5225-1721-4.ch006

[70] Arnab K. Paul, Steven Tuecke, Ryan Chard, Ali R. Butt, Kyle Chard, and Ian Foster. 2017. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. Association for Computing Machinery, New York, NY, 49–54. DOI: https://doi.org/10.1145/3149393.3149402

[71] Arnab Kumar Paul, Wenjie Zhuang, Luna Xu, Min Li, M. Mustafa Rafique, and Ali R. Butt. 2016. CHOPPER: Optimizing data partitioning for in-memory data analytics frameworks. In *IEEE International Conference on Cluster Computing (CLUSTER'16)*. IEEE, 110–119. DOI: https://doi.org/10.1109/CLUSTER.2016.41

[72] Gregory F. Pfister. 2001. An introduction to the InfiniBand™ architecture. *High Perform. Mass Stor. Parallel I/O* 42 (2001), 617–632.

[73] Prabhat and Quincey Koziol. 2014. *High Performance Parallel I/O*. CRC Press.

[74] Yingjin Qian, Eric Barton, Tom Wang, Nirant Puntambekar, and Andreas Dilger. 2009. A novel network request scheduler for a large scale storage system. *Comput. Sci.-Res. Devel.* 23, 3-4 (2009), 143–148. DOI: https://doi.org/10.1007/s00450-009-0073-9

[75] Neeraj Rajesh, Hariharan Devarajan, Jaime Cernuda Garcia, Keith Bateman, Luke Logan, Jie Ye, Anthony Kougkas, and Xian-He Sun. 2021. Apollo: An ML-assisted real-time storage resource observer. In *30th International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, New York, NY, 147–159. DOI: https://doi.org/10.1145/3431379.3460640

[76] Ramesh R. Sarukkai. 2000. Link prediction and path analysis using Markov chains. *Comput. Netw.* 33, 1-6 (2000), 377–386. DOI: https://doi.org/10.1016/S1389-1286(00)00044-X

[77] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, 231–244.

[78] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2016. Modular HPC I/O characterization with Darshan. In *5th Workshop on Extreme-Scale Programming Tools (ESPT'16)*. IEEE, 9–17. DOI: https://doi.org/10.1109/ESPT.2016.006

[79] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu Thanh Luu, Surendra Byna, and Prabhat. 2015. Techniques for modeling large-scale HPC I/O workloads. In *6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, New York, NY, 11 pages. DOI: https://doi.org/10.1145/2832087.2832091

[80] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. 2011. A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'11)*. IEEE, 414–423. DOI: https://doi.org/10.1109/CCGrid.2011.26

[81] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On implementing MPI-IO portably and with high performance. In *6th Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*. ACM, New York, NY, 23–32. DOI: https://doi.org/10.1145/301816.301826

[82] TOP500.org. 2022. TOP500 List. Retrieved from https://www.top500.org/lists/top500/2022/06/

[83] Yuichi Tsujita, Tatsuhiko Yoshizaki, Keiji Yamamoto, Fumichika Sueyasu, Ryoji Miyazaki, and Atsuya Uno. 2017. Alleviating I/O interference through workload-aware striping and load-balancing on parallel file systems. In *ISC High Performance (ISC'17)*. Springer International Publishing, Cham, 315–333. DOI: https://doi.org/10.1007/978-3-319-58667-0_17

[84] Alan Tucker. 1977. A note on convergence of the Ford-Fulkerson flow algorithm. *Math. Oper. Res.* 2, 2 (1977), 143–144. DOI: https://doi.org/10.1287/moor.2.2.143

[85] Andrew Uselton, Mark Howison, Nicholas J. Wright, David Skinner, Noel Keen, John Shalf, Karen L. Karavanic, and Leonid Oliker. 2010. Parallel I/O performance: From events to ensembles. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*. IEEE, 1–11. DOI: https://doi.org/10.1109/IPDPS.2010.5470424

[86] Sudharshan S. Vazhkudai, Ross Miller, Devesh Tiwari, Christopher Zimmer, Feiyi Wang, Sarp Oral, Raghul Gunasekaran, and Deryl Steinert. 2017. GUIDE: A scalable information directory service to collect, federate, and analyze logs for operational insights into a leadership HPC facility. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. ACM, New York, NY, 1–12. DOI: https://doi.org/10.1145/3126908.3126946

[87] Bharti Wadhwa, Arnab K. Paul, Sarah Neuwirth, Feiyi Wang, Sarp Oral, Ali R. Butt, Jon Bernard, and Kirk W. Cameron. 2019. iez: Resource contention aware load balancing for large-scale parallel file systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. IEEE, 610–620. DOI: https://doi.org/10.1109/IPDPS.2019.00070

[88] Stephen R. Walli. 1995. The POSIX family of standards. *StandardView* 3, 1 (Mar. 1995), 11–17.

[89] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. 2020. Recorder 2.0: Efficient parallel I/O tracing and analysis. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'20)*. IEEE, 1–8. DOI: https://doi.org/10.1109/IPDPSW50202.2020.00176

[90] Feiyi Wang, Mark Nelson, Sarp Oral, Scott Atchley, Sage Weil, Bradley W. Settlemyer, Blake Caldwell, and Jason Hill. 2013. Performance and scalability evaluation of the Ceph parallel file system. In *8th Parallel Data Storage Workshop*. ACM, New York, NY, 14–19. DOI: https://doi.org/10.1145/2538542.2538562

[91] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S. Vazhkudai. 2014. Improving large-scale storage system performance via topology-aware and balanced data placement. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. IEEE, 656–663. DOI: https://doi.org/10.1109/PADSW.2014.7097866

[92] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *7th Conference on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, 307–320.

[93] Marc C. Wiedemann, Julian M. Kunkel, Michaela Zimmer, Thomas Ludwig, Michael Resch, Thomas Bönisch, Xuan Wang, Andriy Chut, Alvaro Aguilera, Wolfgang E. Nagel, Michael Kluge, and Holger Mickler. 2013. Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. *Computer Science-Research and Development* 28 (2013), 241–251. DOI: 10.1007/s00450-012-0221-5

[94] Steven A. Wright, Simon D. Hammond, Simon J. Pennycook, Robert F. Bird, J. A. Herdman, Ian Miller, Ash Vadgama, Abhir Bhalerao, and Stephen A. Jarvis. 2013. Parallel file system analysis through application I/O tracing. *Comput. J.* 56, 2 (2013), 141–155. DOI: https://doi.org/10.1093/comjnl/bxs044

[95] Xing Wu and Frank Mueller. 2013. Elastic and scalable tracing and accurate replay of non-deterministic events. In *27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, NY, 59–68. DOI: https://doi.org/10.1145/2464996.2465001

[96] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. 2011. Probabilistic communication and I/O tracing with deterministic replay at scale. In *International Conference on Parallel Processing*. IEEE, 196–205. DOI: https://doi.org/10.1109/ICPP.2011.50

[97] Peter Xenopoulos, Jamison Daniel, Michael Matheson, and Sreenivas Sukumar. 2016. Big data analytics on HPC architectures: Performance and cost. In *IEEE International Conference on Big Data (Big Data'16)*. IEEE, 2286–2295. DOI: https://doi.org/10.1109/BigData.2016.7840861

[98] Shaocheng Xie, Wuyin Lin, Philip J. Rasch, Po-Lun Ma, Richard Neale, Vincent E. Larson, Yun Qian, Peter A. Bogenschutz, Peter Caldwell, Philip Cameron-Smith, Jean-Christophe Golaz, Salil Mahajan, Balwinder Singh, Qi Tang, Hailong Wang, Jin-Ho Yoon, Kai Zhang, and Yuying Zhang. 2018. Understanding cloud and convective characteristics in version 1 of the E3SM atmosphere model. *Journal of Advances in Modeling Earth Systems* 10, 10 (2018), 2618–2644. DOI: 10.1029/2018MS001350

[99] Cong Xu, Suren Byna, Vishwanath Venkatesan, Robert Sisneros, Omkar Kulkarni, Mohamad Chaarawi, and Kalyana Chadalavada. 2016. LIOProf: Exposing Lustre File System Behavior for I/O Middleware. In*Cray User Group Meeting (CUG'16)*.

[100] Pengfei Xuan, Walter B. Ligon, Pradip K. Srimani, Rong Ge, and Feng Luo. 2017. Accelerating big data analytics on HPC clusters using two-level storage. *Parallel Comput.* 61 (2017), 18–34. DOI: https://doi.org/10.1016/j.parco.2016.08.001

[101] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. End-to-end I/O monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association 379–394. Retrieved from https://www.usenix.org/conference/nsdi19/presentation/yang

[102] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. 2023. End-to-end I/O monitoring on leading supercomputers. *ACM Trans. Storage* 19, 1, Article 3 (Jan. 2023), 35 pages. DOI: https://doi.org/10.1145/3568425

[103] Bin Yang, Yanliang Zou, Weiguo Liu, and Wei Xue. 2022. An end-to-end and adaptive I/O optimization tool for modern HPC storage systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'22)*. IEEE, 1294–1304. DOI: https://doi.org/10.1109/IPDPS53621.2022.00128

[104] Qian Yingjin, Wang Di, and Nirant Puntambekar. 2009. Lustre Simulator. Retrieved from https://github.com/yingjinqian/Lustre-Simulator

[105] Zeng Zeng and Bharadwaj Veeravalli. 2008. On the design of distributed object placement and load balancing strategies in large-scale networked multimedia storage systems. *IEEE Trans. Knowl. Data Eng.* 20, 3 (2008), 369–382. DOI: https://doi.org/10.1109/TKDE.2007.190694

[106] G. Zheng, Gunavardhan Kakulapati, and L. V. Kale. 2004. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium*. IEEE, 10 pages. DOI: https://doi.org/10.1109/IPDPS.2004.1303013

[107] Mingfa Zhu, Guoying Li, Li Ruan, Ke Xie, and Limin Xiao. 2013. HySF: A striped file assignment strategy for parallel file system with hybrid storage. In *IEEE 10th International Conference on High Performance Computing and Communications and IEEE International Conference on Embedded and Ubiquitous Computing (HPCC & EUC'13)*. IEEE, 511–517. DOI: https://doi.org/10.1109/HPCC.and.EUC.2013.79

[108] Zhaobin Zhu, Niklas Bartelheimer, and Sarah Neuwirth. 2023. MAWA-HPC: Modular and Automated Workload Analysis for HPC Systems. In *Poster at ISC High Performance Conference (ISC'23)*. DOI: https://doi.org/10.13140/RG.2.2.10671.92325

[109] Zhaobin Zhu, Sarah Neuwirth, and Thomas Lippert. 2022. A comprehensive I/O knowledge cycle for modular and automated HPC workload analysis. In *IEEE International Conference on Cluster Computing (CLUSTER'22)*. IEEE, 581–588. DOI: https://doi.org/10.1109/CLUSTER51413.2022.00076

[110] Laura Zingaretti and Miguel Pérez-Enciso. 2022. Deep Learning for Genomic Prediction (DeepGP). Retrieved from https://github.com/lauzingaretti/DeepGP

Received 18 January 2023; revised 7 September 2023; accepted 26 December 2023