IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 14 January 2025, accepted 3 March 2025, date of publication 20 March 2025, date of current version 4 April 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3553311

RESEARCH ARTICLE

Leveraging Pre-Built Catalogs and Object-Level Scheduling to Eliminate I/O Bottlenecks in HPC Environments

SEOYEONG LEE[1], JUNGHWAN PARK[1], YOOCHAN KIM[1], SAFDAR JAMIL[1], AWAIS KHAN[2], SEUNG WOO SON[3], (Member, IEEE), JAE-KOOK LEE[4], DO-SIK AN[4], TAEYOUNG HONG[4], AND YOUNGJAE KIM[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[2]Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[3]Department of Electrical and Computer Engineering, University of Massachusetts Lowell, Amherst, MA 01003, USA
[4]Korea Institute of Science and Technology Information (KISTI), Daejeon 34141, South Korea

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant RS-2024-00416666; in part by Korea Institute of Science and Technology Information (KISTI) under Grant K25L2M2C2; and in part by Oak Ridge Leadership Computing Facility, located at 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.

ABSTRACT Modern High-Performance Computing (HPC) environments face mounting challenges due to the shift from large to small file datasets, along with an increasing number of users and parallelized applications. As HPC systems rely on Parallel File Systems (PFS), such as Lustre for data processing, performance bottlenecks stemming from Object Storage Target (OST) contention have become a significant concern. Existing solutions, such as LADS with its object-level scheduling approach, fall short in large-scale HPC environments due to their inability to effectively address metadata I/O bottlenecks and the growing number of I/O processes. This study highlights the pressing need for a comprehensive solution that tackles both OST contention and metadata I/O challenges in diverse HPC workloads. To address these challenges, we propose SwiftLoad, an object-level I/O scheduling framework that leverages a metadata catalog to enhance the performance and efficiency of parallel HPC utilities. The adoption of the metadata catalog mitigates the metadata I/O bottlenecks that commonly occur in HPC utilities, a challenge that is particularly pronounced in object-level I/O scheduling. SwiftLoad addresses OST contention and the uneven distribution of I/O processes across different OSTs through mathematical modeling and incorporates a Loader Configuration Module to regulate the number of I/O processes. Evaluated with two representative utilities—data deduplication profiling and data augmentation—SwiftLoad achieved performance improvements of up to 5.63× and 11.0×, respectively, on a production supercomputer.

INDEX TERMS HPC, I/O, parallel file system, parallel processing.

I. INTRODUCTION

Within the High-Performance Computing (HPC) community, a suite of popular HPC utilities is extensively employed to leverage the capabilities of HPC infrastructure and preprocess data efficiently. These utilities, including deduplication, copying, integrity checking, and others, operate in parallel and typically follow a cyclic process of bulk data reading, computing, and, if necessary, writing. This cycle repeats until the processing of the entire dataset is complete.

Scientific data in HPC facilities are often managed using Parallel File Systems (PFS) like GPFS [1] and Lustre [2], with top HPC systems (e.g., Frontier [3], LUMI [4], Perlmutter [5]) utilizing Lustre for storage. The Lustre PFS consists of a Metadata Server (MDS) with Metadata Targets (MDTs) and an Object Storage Server (OSS) with Object Storage Targets (OSTs). This architecture is designed to accommodate multiple user accesses and support large-scale operations, originally tailored for "hero" applications that

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

involve reading and writing a single exceptionally large file at high speeds. Such workloads typically do not encounter performance bottlenecks.

However, the shift towards the modern Big Data and AI/ML era has transformed the datasets stored in PFS from predominantly large files to small files [6]. Additionally, an influx in the number of users and parallelization of applications has led HPC facilities to face the significant challenge of managing hundreds of tasks performing billions of read and write operations to the shared PFS environment, exceeding the capabilities and scalability of conventional PFS design. Consequently, overwhelming shared PFS resources, such as Lustre, can result in notable bottlenecks due to OST contention [7]. This significantly degrades overall system performance.

In parallel HPC utilities, simultaneous access to the PFS can lead to OST contention, necessitating a scalable solution. LADS [7] introduces object-level scheduling to mitigate OST contention by performing I/O scheduling at the object level rather than the file level. However, this method falls short in addressing the metadata I/O bottleneck and does not adequately consider scenarios with an increased number of I/O processes, making it unsuitable for large-scale HPC systems. The limitations of object-level scheduling become evident in the execution sequence of parallel HPC utilities, which begins with metadata operations for directory traversal, followed by read I/O operations based on the extracted metadata. While this approach can enhance data I/O, it is less effective when dealing with numerous small files, where metadata I/O becomes predominant. Additionally, object-level scheduling necessitates extra MDS requests through the Lustre layout retrieval API *llapi_layout_**, which exacerbates the metadata I/O bottleneck. Moreover, object-level scheduling is not always the optimal solution, as its effectiveness depends on the number of I/O processes and the number of OSTs. Consequently, there is a need for a new object-level scheduling approach that addresses both OST contention and the metadata I/O bottleneck bottleneck across varying scenarios.

In this paper, we present SwiftLoad, a metadata catalog-based object-level I/O scheduling approach designed to enhance the performance of parallel utilities in HPC environments.

SwiftLoad introduces a technique for creating a metadata catalog that eliminates the need for directory traversal—a resource-intensive metadata I/O request—by preemptively capturing file metadata. By overlapping metadata collection with file creation in scientific applications, this approach significantly reduces future metadata I/O overhead for HPC utilities. Our evaluation demonstrates that this technique imposes a negligible performance impact on the application.

The implementation of object-level scheduling in SwiftLoad leveraging the metadata catalog effectively mitigates OST contention in HPC utilities. This method utilizes the metadata catalog to identify file layouts, which enables data loading I/O processes to efficiently retrieve objects from each OST. Consequently, this approach resolves OST contention without exacerbating the metadata I/O bottleneck.

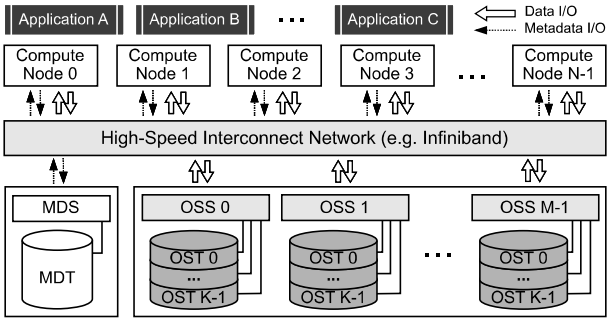


FIGURE 1. An overview of HPC applications running on Lustre PFS in an HPC facility.

We define and analyze OST contention and the imbalance of I/O process access across OSTs, which can lead to the straggler problem, using mathematical modeling. These issues, which were previously challenging to clearly identify, are thoroughly explained and addressed in our work. Based on this analysis and model, we implement a Loader Configuration Module to adaptively adjust the number of I/O processes, thereby avoiding severe OST contention and enhancing resource efficiency.

To demonstrate the effectiveness and feasibility of SwiftLoad, we implemented the parallel (multi-node) version of a data deduplication profiler, FS-C [8], and a data augmentation tool atop SwiftLoad. Extensive evaluations showed that SwiftLoad significantly improved the performance of these HPC utilities. SwiftLoad achieved a 5.63x enhancement in performance for the data deduplication profiler and an 11.0× improvement in performance for the data augmentation tool.

II. BACKGROUND AND RELATED WORK

A. LUSTRE FILE SYSTEM

Figure 1 depicts a conventional high-performance computing HPC infrastructure based on the Lustre PFS. Lustre PFS consists of three primary components: Compute Nodes, the MDS, and OSS. The MDS is responsible for storing and managing file metadata [9], such as file paths, permissions, and layout information, which indicates the physical location of data chunks. Traditionally, Lustre utilizes a single MDS to handle metadata operations. While modern HPC systems can incorporate multiple MDSes, their quantity is considerably lower than that of OSSes. Conversely, OSSes utilize OSTs to store the file contents across multiple OSTs in a distributed manner [9].

Lustre uses file striping to divide files into fixed-size chunks, called "data objects", which are distributed across multiple OSTs in parallel to enhance performance and concurrency [9], [10]. These data objects are the fundamental units of storage managed by OSTs. Consequently, the MDS maintains the file layout information required to reconstruct the files' content and deliver it back to the user or application. This file layout metadata includes the identifiers of OSTs,

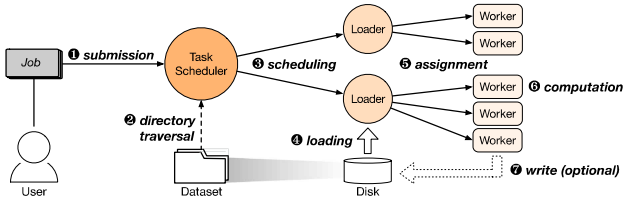


FIGURE 2. Workflow of the HPC utility.

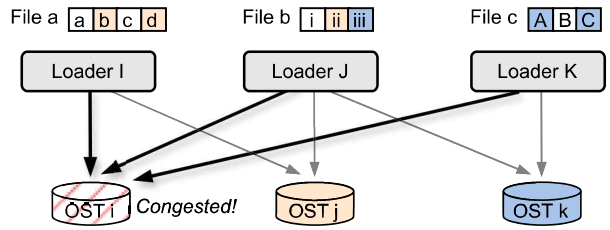


FIGURE 3. Diagram illustrating files (a, b, c) processed by loader modules (I, J, K) into separate OSTs, converging at a congested OST *i*, highlighting I/O bottleneck situation in file-level scheduling.

stripe size, and stripe count. The stripe size, typically set at 1 MB, represents the size of each object. The stripe count indicates the number of OSTs across which the objects are distributed, and it is determined by the file size and configuration parameters set by the system administrator.

B. HPC UTILITIES

In HPC, a variety of parallel utilities are employed, such as copying, restriping, comparison tools [11], integrity checkers [12], deduplication profiler [8], [13], and data augmentation tool. These utilities process, transform, and analyze data produced by scientific applications or datasets obtained from external sources, enabling more efficient data management and preparation for subsequent computational workflows.

Figure 2 illustrates the typical workflow of an HPC utility. When a user submits a job, a task scheduler initiates a directory traversal to resolve the paths of the target files. This involves metadata operations that request information from the MDS. The task scheduler then assigns the data I/O operations of the retrieved file paths to data loaders.

During this phase, data I/O operations are scheduled using a file-level approach, where I/O tasks are performed on a file unit basis across multiple processes. Figure 3 illustrates how the file-level approach handles data I/O. Each assigned file consists of multiple objects striped across OSTs. The data loader performs I/O read operations on the file objects in sequential manner, moving from one OST to another. However, this mismatch between the scheduling unit and the storage structure can lead to OST congestion when multiple I/O processes access the same OST simultaneously. This mismatch is particularly relevant, as this phase involves bulk file reading, which demands intensive I/O operations from the shared PFS. Once the data is read, it is transferred to workers that perform data preprocessing, a compute-intensive task, followed by bulk writing if necessary. A key point in this process is that the data loading performed by the loader and the subsequent computation carried out by the worker form a recurring cycle of read and compute operations [14], [15]. This cycle underscores the critical role of I/O performance in HPC utilities, as evidenced by the I/O bottlenecks observed in both our simulation results and experimental environment (Sections III and V).

C. RELATED WORK

LADS [7] introduces object-level scheduling for data transfer, thereby enhancing data I/O. However, it lacks detailed evidence on the causes and escalation of OST contention and does not address the critical metadata I/O bottleneck, which is essential for large-scale HPC utilities. Additionally, its solution is confined to small-scale scenarios with fewer I/O processes than OSTs, making it unsuitable for larger deployments.

Several previous research efforts have focused on mitigating the metadata server bottleneck in PFS. XFast [16] employs prediction and prefetching techniques to anticipate required file metadata, thereby reducing the number of metadata requests to the server. HVAC [17] aims to alleviate the General Parallel File System (GPFS) metadata I/O bottleneck in large-scale training on HPC systems by utilizing a local storage cache layer. XFast primarily focuses on scaling directory traversal at the system level, while HVAC is tailored specifically for deep learning applications.

While these studies provide valuable insights, we identify several critical points that require further attention. First, a detailed analysis of OST contention is necessary, along with a solution for managing a large number of I/O processes relative to the number of OSTs in object-level scheduling. Second, the metadata I/O bottleneck is intertwined with OST contention and needs to be addressed. Third, previous studies on the metadata I/O bottleneck are either designed at the system level or for specific-purpose applications. None of said studies addresses the metadata I/O bottleneck from the perspective of HPC utilities, particularly the challenges that arise when resolving OST contention. These gaps highlight the need for new object-level scheduling solutions that can efficiently manage both metadata I/O and data I/O for generalized utilities in HPC environments.

III. ANALYSIS OF I/O BOTTLENECKS IN HPC UTILITIES

A. OST CONTENTION AND IMBALANCE ANALYSIS

OST contention refers to the situation where multiple processes attempt to perform I/O operations on the same OST, leading to performance degradation by increasing I/O times and causing potential bottlenecks. Figure 4 shows the slowdown in read time relative to a single process performing I/O on an OST, illustrating the impact of multiple processes on OST performance. As the number of processes accessing a single OST increases, the read time increases exponentially. This slowdown occurs because multiple processes compete for the same I/O resources, causing contention and resulting in delays. Additionally, the graph shows that as the number of

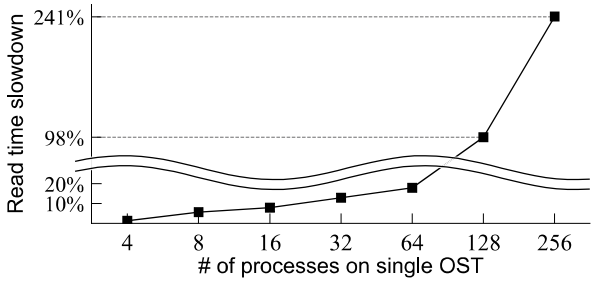


FIGURE 4. ***Pread*** **time slowdown relative to the number of processes performing I/O on a single OST. As the number of processes increases, the *pread* time increases exponentially.**

processes increases, the read time slowdown becomes more severe, emphasizing the impact of increased contention. The experimental settings are detailed in Section V.

In distributed environments, processes access the necessary OSTs, and the number of processes performing I/O on each OST varies over time. This imbalance can lead to more severe I/O performance issues for processes accessing certain OSTs compared to others due to contention, creating "stragglers." Stragglers are processes that experience significant delays because they are competing for access to heavily contended OSTs, resulting in overall performance degradation. To understand when such imbalances become severe and the resulting performance degradation in HPC utilities, we developed an entropy-based mathematical model representing imbalance and conducted simulations emulating HPC utility scenarios.

First, we apply the concept of entropy from information theory to define the entropy value H as follows:

$$H = \sum_i H_i \text{ where } H_i = \begin{cases} 0 & \text{if } p_i = 0 \\ -p_i \log p_i & \text{otherwise} \end{cases} \quad (1)$$

Here, $p_i = n_i/n$, where n_i denotes the number of processes performing I/O on OST i, and n is the total number of processes. The value H represents the entropy, indicating how uniformly the processes are distributed across the OSTs. In our system, entropy effectively captures the overall distribution and helps identify nonlinear relationships. Higher entropy values indicate a more uniform distribution of processes across the OSTs, reflecting a balanced system. However, simply observing the entropy value does not provide an understanding of how close the distribution is to an ideal state. To facilitate numerical comparison, we introduce the *Contention Imbalance Intensity* (CI), a standardized measure ranging from 0 to 1, using entropy to quantify the degree of imbalance. CI is defined as follows:

$$CI = 1 - \frac{H}{H_{\max}} \quad (2)$$

Here, $H_{\max} = \log m$ represents the maximum entropy, achieved when processes are evenly distributed across all OSTs, where m is the number of OSTs. The CI metric provides a clear numerical representation of the concentration and dispersion of processes accessing the OSTs during utility execution. A higher CI indicates greater imbalance, whereas a lower CI suggests a more balanced distribution. For instance, $CI = 1$ indicates that all processes are accessing a single OST, representing maximum imbalance. The relationship between CI and the number of processes is not straightforward and can vary significantly. A small number of processes might concentrate on a single OST, increasing imbalance, while a larger number of processes might distribute more evenly across multiple OSTs, potentially reducing imbalance.

According to Figure 7(a) in Section V-B, as the number of processes grows, the likelihood of increased imbalance also rises. Consequently, there is a higher probability that more OSTs will experience greater contention compared to others, leading to higher contention imbalance intensity. Furthermore, CI values are not zero regardless of the number of processes. This indicates that in conventional HPC utilities, there is a consistent likelihood of processes favoring specific OSTs, which can invariably result in straggler issues. For example, when the process-to-OST ratio is 32:1, the most balanced distribution would have each OST handling 32 processes. However, if the distribution is imbalanced, the performance degradation becomes more pronounced. For instance, one OST can have three times as many processes as another (e.g., 48 processes on one OST and 16 on another). This imbalance leads to increased read times due to higher contention on more heavily loaded OSTs. According to Figure 7(b) in Section V-B, higher CI values lead to more severe straggler problems, emphasizing the impact of increased contention imbalance on execution time disparities.

Therefore, OST contention and file-level scheduling can substantially degrade read performance and exacerbate imbalance issues respectively. These two factors collectively contribute to increased end-to-end latency and overall performance degradation due to stragglers in distributed HPC environments.

B. LIMITATIONS OF OBJECT-LEVEL SCHEDULING IN HPC UTILITIES

Object-level scheduling [7], which schedules I/O at the object level, has been proposed as an approach to address OST contention. Unlike file-level scheduling, object-level scheduling manages I/O operations by dividing the file into object-sized units. With this methodology, each process can read an object from a specific OST rather than the entire file, allowing processes to be assigned to specific OSTs. This approach helps to avoid contention that occurs when multiple processes access the same OST simultaneously.

Despite its advantages, object-level scheduling is not without limitations. While it effectively mitigates OST contention, it does not fully address the challenges posed by metadata I/O.

1) METADATA I/O BOTTLENECK

In file-level scheduling, particularly when reading a large number of small files, the directory traversal problem arises due to excessive metadata I/O resulting in performance

degradation. Metadata I/O is time-consuming for several reasons. First, parallel file systems such as Lustre and GPFS typically employ centralized metadata servers [18], [19]. Although modern HPC systems are designed with multiple metadata servers and adopt a distributed namespace to enable metadata distribution across these MDSes, the number of MDSes remains significantly smaller compared to the number of OSSes. Consequently, the capabilities of MDSes can become saturated under an excessive number of metadata server requests, even in leading supercomputers [17]. Second, the POSIX interface *readdir*, which is invoked for directory traversal, reads one *dirent* at a time, making it difficult to parallelize at the thread level and thus becomes a sequential operation. Third, most file systems organize directories and files in a hierarchical tree structure. In this structure, acquiring metadata information for a single file requires obtaining a lock on the respective directory through the lock manager. Therefore, when multiple processes simultaneously send detailed metadata requests such as *stat* for files within a directory, lock contention can occur, potentially serializing the requests [19].

However, object-level scheduling fails to resolve these issues. Addressing OST contention alone does not mitigate the Metadata I/O bottleneck. In fact, object-level scheduling necessitates that tens to thousands of processes retrieve object information. This results in additional metadata I/O, as the task scheduler requests file layout information from the MDS. When a substantial number of file information requests are issued, this can become a significant bottleneck. Figure 9 in Section V demonstrates that both approaches have nearly the same metadata I/O time for directory traversal. However, the object-level scheduling is even more slower than file-level scheduling due to layout information requests. These two sources of metadata I/O time negate the benefits of object-level scheduling and can even lead to performance degradation.

2) INEFFICIENCIES OF NUMEROUS PROCESSES ON OSTS

There is a threshold for the number of processes that a single OST can manage effectively. Figure 4 implies the following: when more than 64 processes execute I/O operations on a single OST, the pread time increases substantially. Hence, there is likely to be an optimal number of processes that can leverage parallelism effectively, as determined by pread time. However, traditional object-level scheduling does not consider this issue.

According to Figure 7(c) in Section V-B, the difference between the traditional file-level scheduling method and the ideal object-level scheduling method, where processes are distributed most evenly across all OSTs, decreases as the pread value sharply increases at the threshold of 64. This suggests that evenly distributing processes across OSTs has limitations in improving performance. As the pread time increases rapidly, both file-level and ideal object-level scheduling methods encounter significant bottlenecks, resulting in similar performance levels.

Thus, to mitigate the straggler effect resulting from OST contention and imbalance, a novel object scheduling method that considers metadata server bottlenecks, layout information function call overhead, and an optimal number of processes is required.

IV. DESIGN OF SWIFTLOAD

A. SWIFTLOAD OVERVIEW

SwiftLoad introduces an object-level I/O scheduling approach utilizing a Metadata Catalog. The Metadata Catalog is a prebuilt log file containing metadata information. By using the catalog, the need for directory traversal and path resolution during HPC utility execution is eliminated. Additionally, other metadata system calls can be directed to the catalog without requiring MDS requests. For shared datasets, once the catalog is built, it can be reused by multiple users, enabling faster and more efficient data access for repeated analyses or shared computational tasks, significantly improving overall system performance.

As illustrated in Figure 5, SwiftLoad's architecture encompasses two aspects: the construction of the catalog and its subsequent utilization. When scientific applications or HPC utilities create a file, ❶ a Catalog Agent intercepts *open* and *close* requests, and ❷ the Catalog Server records the file's corresponding metadata information in real-time. This information may include details necessary for directory traversal, as well as other relevant metadata system calls which require MDS calls.

Parallel HPC utilities employ multiple processes and nodes. In the SwiftLoad design, each process includes a Data Loader, an Object Scheduler, and a Computational Module. The Object Scheduler is comprises of the Catalog Loader and Mapper. Additionally, a single Loader Configuration Module is employed. When metadata I/O is requested, ❸ the Catalog Loader reads the pre-built metadata catalog and loads it into memory. Once loaded, any request that would typically go to the MDS is first checked against the catalog, thereby reducing the need for direct MDS queries.

❹ The Mapper within the Object Scheduler maps the OSTs and Data Loaders, taking into account the results provided by the Loader Configuration Module regarding the optimal number of processes for a given number of OSTs. It then retrieves the metadata information for directory traversal and file layout from the catalog. ❺ This layout information is subsequently sent to the corresponding Data Loader. With this information, the Data Loader reads objects from its explicitly paired OST with minimal contention. Following this, ❻ the Computational Module performs necessary computations, such as calculating the deduplication ratio for deduplication analysis tools or computing a checksum for integrity checkers.

B. CATALOG CONSTRUCTION

The Metadata Catalog delivers critical metadata, enabling HPC utilities to circumvent the MDS. It encompasses two

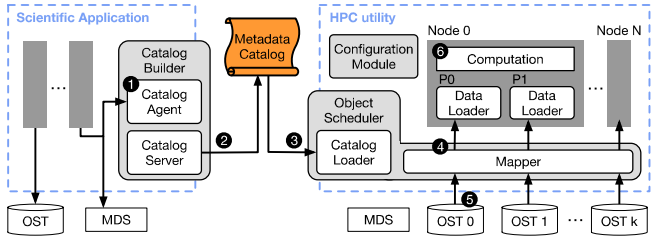


FIGURE 5. Architectural overview of SwiftLoad.

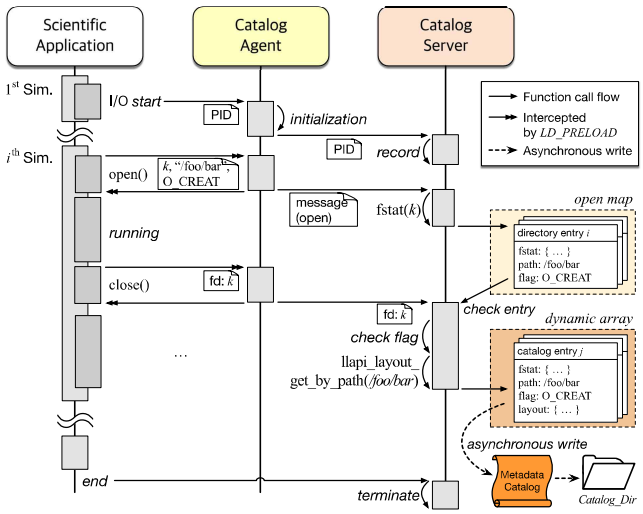


FIGURE 6. Catalog construction process during scientific application execution. The Catalog Agent intercepts I/O calls, logs file paths, and updates metadata. The Catalog Server dynamically stores and asynchronously writes catalog entries, optimizing performance.

primary categories of information: metadata system call data pertinent to file operations and layout information necessary for object-level scheduling. This includes *readdir*, *stat*, and *lstat* for providing directory entries, file status, and link status respectively, as well as file path, stripe size, stripe count, and OST index for layout information. The catalog is stored and retrieved in the SwiftLoad_Catalog_Dir, allowing multiple reuses and minimizes the number of MDS calls.

Catalog Builder comprises two modules, Catalog Server and Catalog Agent. Catalog Server is a separate process running parallel to the application on each node, whereas Catalog Agent is a shared library loaded into the application process. The server listens for and receives messages from the clients, and executes the appropriate metadata operations and records the results in the catalog. Each client monitors and intercepts specific system calls related to file operations by sending messages containing information captured from system calls, such as file paths or file descriptors, using *LD_PRELOAD*.

Figure 6 illustrates the catalog construction process while the scientific simulation application is running. When the application calls *open*, the client intercepts it and logs the file path to mitigate path-resolution overhead associated with directory-level checks, such as those performed by *readdir*. The flags are then checked; if the flags are *O_CREAT*, *O_RDWR*, or *O_WRONLY*, indicating that the file is newly created or updated, metadata updates are required. Hence, a message containing the original file path, flags, and file descriptor is written to the shared memory buffer and sent to the server. If the flags are such that metadata (other than access time) do not change, as in *O_RDONLY*, no message is sent to the server. The server continuously monitors the shared memory buffer for new messages and initially verifies the message type upon detection. If the message type is *open*, *fstat* is called to obtain information about the file. This is done as the message includes the file descriptor, eliminating the need to call the *stat* function and reducing the load on the MDS. The acquired file information, combined with the initial message, forms a "directory entry". This entry is stored in a structure termed the open map, which can be referenced later using the file descriptor.

When the application issues *close*, the client intercepts this call and logs the corresponding fd to the buffer. When the server detects a *close* type message, it checks whether the received file descriptor is recorded on the open map. If not included, the message is ignored. Otherwise, the server checks if the flag is *O_CREAT* to determine if the file was newly created. For newly created files, layout information is gathered using the Lustre API.

If the layout configuration has not changed, layout information collection is not performed. Our observations indicate that the application does not perform explicit layout changes, such as with *lfs setstripe*, making this approach valid. The server creates a "catalog entry" from the layout information and the directory entry and stores them dynamically in memory. When the number of elements in the dynamic array reaches a certain threshold, the recorded catalog entries are asynchronously written in the catalog file. This method mitigates I/O overhead and prevents performance degradation that would result from frequent catalog file updates.

The Catalog Builder is designed to operate seamlessly with scientific applications, enabling the creation of a catalog at runtime as raw data is generated. This ensures that the catalog is readily available when the data is later processed, transformed, or analyzed by HPC utilities or during subsequent computational phases of the application. When the application starts I/O, the client sets environment variables, initializes shared memory, and writes the application's PID to it for server monitoring. The server periodically checks this PID, ensuring that all data is properly logged into the catalog upon application termination.

This catalog construction methodology decouples metadata collection from the scientific application processing. It asynchronously performs metadata I/O, optimizing file operations and minimizing application impact. Performance evaluation is discussed in Section V.

To utilize the catalog, each file's metadata must be loaded into the catalog. However, data creation occurs in various ways, such as being generated by scientific simulation applications or downloaded from external sources. Therefore, a single method for creating a catalog is insufficient. SwiftLoad addresses this challenge by supporting not only

runtime catalog construction but also a post-process catalog construction method. The latter is designed for files that already exist in storage or are downloaded from the web. In such cases, post-process catalog construction is scheduled prior to the application's execution, similar to data preprocessing. At an extremely large scale where datasets are vast, this post-process construction could become time-intensive. In such cases, adequate time must be allocated to ensure catalog construction is completed before performing dependent operations. However, this represents a one-time cost, as the created catalog can be reused multiple times thereafter. To further mitigate this one-time cost, future work could investigate system-level catalog construction, where the catalog is generated immediately as files are created from external sources. This approach could significantly reduce the overhead of post-processing catalog construction and enhance usability.

C. SCHEDULING WITH CATALOG

The object-level scheduling process begins with the catalog loader reading the catalog into memory. For *readdir* information, the data is stored in an array, while other metadata, such as *stat*, is sorted by file path to allow efficient retrieval via binary search. Along with these metadata operations, layout information is also loaded and sorted. The object scheduler retrieves the object layout information of each file, distinguishes them based on the OST index, and creates multiple object tasks accordingly. The scheduler maps I/O processes to the OSTs using a round-robin approach and places the object tasks into the task queues of the I/O processes bound to the corresponding OSTs. Binding processes to OSTs implies that each I/O process is dedicated to a specific OST and does not switch to another OST. As illustrated in Figure 5, this approach mitigates OST contention by ensuring that each process consistently accesses only its designated OST. Consequently, this reduces the CI value and minimizes overall slowdown. The ratio of I/O processes mapped to each OST is adjusted based on the results from the Loader Configuration Module to ensure that the potential OST contention does not outweigh the benefits of parallelism. Any remaining I/O processes that are not mapped to OSTs remain idle.

D. LOADER CONFIGURATION MODULE

As mentioned in Section III, increasing the number of data loaders does not always guarantee optimal performance. This remains an issue even when applying object-level scheduling, particularly when a large number of processes access a single OST. Consequently, beyond a certain threshold, additional loaders can degrade performance rather than improve it. Even when the data loaders are evenly distributed across OSTs, the absolute number of loaders can still be excessive, leading to degraded performance due to overall system contention. To address this issue, SwiftLoad introduces a *Loader configuration module*. This module profiles the system's read slowdown in advance to regulate the number of loaders accessing each OST. By leveraging these profiling results, the module dynamically adjusts the number of active loaders to optimize performance. The module prevents performance degradation due to excessive contention on a single OST while allowing the utilization of more loaders when system resources are available. This method balances efficient resource utilization, mitigating risks of both over-contention and under-utilization of available loaders.

Algorithm 1 Optimal Loader Management

Input: Profiled slowdown g, Loaders $L = \{L_1, L_2, \cdots\}$
Function `configureLoaders`(g, L)**:**
  Let x be the number of loaders
  $h(x) \leftarrow \frac{x}{g(x)}$
  Find x where $h'(x) = 0$
    Solve $g(x) = x \cdot g'(x)$
  $|L|_{opt} \leftarrow \lceil x \rceil$
  if $|L|_{opt} < \frac{|L|}{|OST|}$ **then**
    Deactivate L_i where $i > |L|_{opt} \cdot |OST|$
  end
return

For example, consider a system with 3 OSTs labeled A, B, and C, and 9 loaders evenly distributed across the OSTs, with 3 loaders per OST. If the threshold for optimal performance per OST is 2 loaders, having 3 loaders per OST would exceed this threshold, resulting in contention and degraded performance despite the even distribution. The Loader Configuration Module identifies this issue and adjusts the number of loaders to ensure only 2 loaders per OST are active, maintaining optimal performance.

Algorithm 1 illustrates the method for determining the optimal number of loaders. Here, $\frac{|L|}{|OST|}$ represents the expected average number of loaders per OST. The optimal number of loaders is determined by identifying the point where the marginal benefit of additional loaders equals the marginal slowdown caused by contention. This mechanism optimizes resource utilization and mitigates the adverse effects of OST contention, thereby maintaining optimal I/O performance.

E. CATALOG CONSISTENCY

The metadata catalog is intentionally designed to offer weak consistency rather than strong consistency, as its primary purpose is to serve as a utility for user convenience rather than a critical component that necessitates strict consistency guarantees. There are a few consistency scenarios to consider. First, if a fault occurs during the building or updating of the catalog, the catalog may become incomplete. In this case, since the original metadata resides on the MDS, the user can simply rebuild the catalog to restore its integrity. The second scenario involves the catalog becoming inconsistent due to metadata updates. However, this scenario is relatively rare.

Typically, scientific applications, which are among the most commonly run workloads in HPC environments, generate data once, and this data is used for analysis without modification. Additionally, users rarely alter preprocessed data once it is downloaded. Therefore, even if there is a time gap between when the catalog is built and when it is used, it generally does not pose a significant issue. Nonetheless, if users are concerned about consistency, they can calculate a checksum for each object when initially building the catalog and store these checksums in a separate checksum catalog file. Later, when the user reads data using the catalog, they can recalculate the checksum for the data content and compare it with the stored checksum to verify consistency. If an inconsistency is detected, the user can update the catalog by retrieving the latest metadata from the MDS, ensuring that subsequent data I/O operations are based on the updated information.

It should be noted that layout modifications, such as those performed using *lfs setstripe*, are not detected through checksum calculations. While the application does not perform such layout changes explicitly, if a user does make such modifications, it is their responsibility to update the catalog accordingly.

F. CATALOG SIZE ANALYSIS

Given the critical role of the metadata catalog, concerns may arise regarding the feasibility of this approach in terms of catalog file size. However, it is important to emphasize that the catalog size remains well within manageable limits, which is crucial for maintaining system efficiency. Our observations indicate that a catalog file of approximately 8 MB is sufficient to store the metadata for 100,000 files. For instance, a user with 1 TB of storage and an average file size of 1 MB would require only about 80 MB to catalog 1,000,000 files. This relatively small size ensures that loading the catalog into memory during utilization is not resource-intensive. Even with limited memory, the Catalog Loader can implement a replacement algorithm to manage catalog entries effectively. While catalog sizes may vary depending on usage, the system is designed to adapt by generating additional catalog files as needed. Furthermore, in HPC environments where files are frequently archived or purged from PFS, strategies such as compressing the catalog for tape storage archiving, particularly with application-generated datasets, can further mitigate concerns regarding catalog file size.

V. EVALUATION

A. EXPERIMENTAL SETUP

1) TESTBED

We used a petascale supercomputer, Nurion [20], hosted by Korean Institute of Science and Technology Information (KISTI). We conducted experiments using one of the Lustre partitions in Nurion. The partition contains 2 MDSes and 24 OSTs. The compute and storage nodes are connected via a 100G interconnect. Further details are listed in Table 1.

TABLE 1. Testbed setup specifications.

Processor	Intel Xeon Phi 7250 (KNL) processor 1.4 GHz, # of Cores per Node: 68
Main Memory	DDR4-2400, 16 GB x 6 per Node
Parallel File System	Lustre 2.7.21.3, 0.76 PB Bandwidth: 0.3 TB/, RAID6(8D+2P)
OS Kernel	Linux version 3.10.0-1062.el7.x86_64

It is important to note that this partition was not isolated and was shared with other users. To mitigate potential bias due to variability, the experiments were repeated multiple times, and the results were averaged. Additionally, the experiments were conducted during similar time periods to ensure consistency.

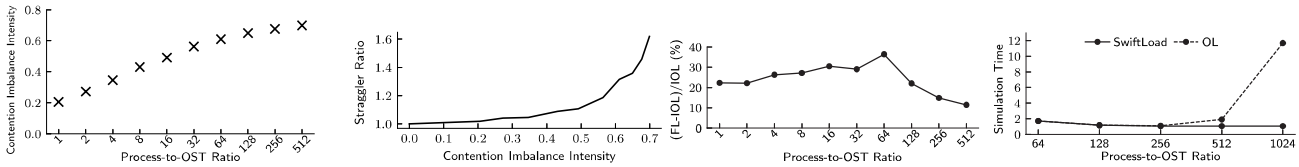
Implementation: To evaluate SwiftLoad, we implemented a parallel (multi-node) version of a data deduplication profiler, FS-C [8], and a data augmentation tool within the SwiftLoad framework. The parallel deduplication profiler, written in C++, operates in two phases: the file system walk phase and the analysis phase. During the file system walk phase, it traverses directories and performs I/O reads. The read content is then divided into smaller chunks, and a hash value is calculated for each chunk. The deduplication ratio is determined by dividing the number of duplicated chunks by the total number of chunks. In this setup, one process handles the directory traversal while the other processes execute the remaining steps in parallel. The data augmentation tool is developed using C, C++, and Python, and leverages the Keras preprocessing library for image augmentation. To simulate OST performance degradation based on loader count, we measured the reduction in OST read times in an actual HPC environment as the number of loaders increased. We mathematically modeled this relationship and conducted simulations to emulate file reading behavior. The simulation was developed using Python. This allowed us to evaluate the impact of varying loader numbers on CI values and overall system performance. Data Loaders (I/O processes) were generated in multiples of the number of OSTs, with each data loader bound to a specific OST. Number of Data Loaders per node was 4.

2) APPROACHES

We compare the following approaches.

- **SwiftLoad**: The SwiftLoad model we propose. A model scheduling I/O work at the object level using a catalog.
- **Object-level (OL):** A model scheduling I/O work at the object level without using a catalog.
- **File-level (FL):** A model scheduling I/O work at file level.

Workloads: For the Deduplication Profiler, we used a synthetic dataset consisting of 100,000 files, each 1 MB in size. For the data augmentation tool, we employed the CIFAR-10 dataset, which comprises 60,000 files with an average size of approximately 2.7 KB. To assess the overhead of catalog construction, we utilized a dataset of 400,000 files, each 1 MB in size, as a makespan of 100,000 files proved insufficient for a meaningful comparison.



(a) Contention Imbalance Intensity (b) Straggler Slowdown (c) FL vs IOL Perf. (d) SwiftLoad vs OL Perf.

FIGURE 7. Simulation results. (a) Contention imbalance intensity increases as the process-to-OST ratio rises. (b) Straggler ratio, the time difference between longest and shortest process, grows nonlinearly with contention intensity. (c) Difference in performance between file-level and ideal-object-level scheduling diminishes as the process-to-OST ratio exceeds 64, showcasing OL scalability limits. (d) Comparison of SwiftLoad and OL simulation times shows SwiftLoad's consistent efficiency across different process-to-OST ratios.

B. SIMULATION STUDY

We conducted HPC simulations based on the read time results modeled in Fig. 4. The simulation was designed to mimic the Nurion storage system environment. Specifically, we assumed a scenario with 24 OSTs, adopted a Progress File Layout (PFL) for the file striping policy, and utilized a Round Robin allocator for OST allocation. The simulation involved 1 million files, each with sizes representative of real HPC workloads. We assumed that a single process performing one object I/O read on an OST would take a single time unit. Using the results from Fig. 4, which reflect how read performance degrades with an increasing number of contending processes in our simulated environment, the simulation was designed to model scenarios where multiple processes reading simultaneously would collectively read less than one object per time unit on an OST. This simulation modeled scenarios where multiple processes accessed data concurrently. The file placement (OST configuration) was re-generated for each simulation run, ensuring a fresh dataset layout for every iteration of the experiment.

At each time unit, we calculated H_i to derive H and subsequently CI. Here, H represents the entropy, indicating the uniformity of process distribution across OSTs, while CI (Contention Imbalance Intensity) quantifies the degree of imbalance, ranging from 0 to 1, as previously discussed in Section III. The overall CI value was obtained by averaging the CI values over multiple unit times. Through this simulation, we analyzed the imbalance in process distribution during data reads for various file placements, examining how processes were concentrated on OSTs and assessing the temporal overhead caused by contention on heavily loaded OSTs.

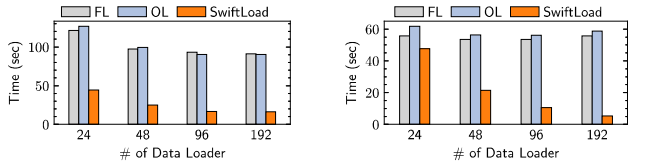
The results are shown in Figure 7. Figure 7(a) shows CI as a function of the process-to-OST ratio. As the number of processes increases, the CI value consistently rises. Figure 7(b) illustrates the straggler ratio, defined as the ratio of the execution time of the longest process to the shortest process as a function of CI in the simulation. For instance, at CI = 0.204, the straggler ratio is 1.019, indicating a 1.9% increase in the longest process time relative to the shortest. As CI increases to 0.701, the straggler ratio rises significantly to 1.756, indicating a 75.6% increase. Figure 7(c) shows the performance difference between file-level (FL) scheduling and ideal object-level (IOL) scheduling through simulation. In this figure, FL denotes the traditional scheduling method that concentrates processes on specific OSTs, whereas IOL signifies the ideal scheduling approach that distributes an equal number of I/O processes across all OSTs. Refer to Section III for the interpretation of these three simulation results.

Figure 7(d) illustrates the simulation times of SwiftLoad and OL as a function of the number of processes. SwiftLoad maintains a stable read time even as the process-to-OST ratio increases. This demonstrates the effectiveness of the loader configuration module in managing the number of loaders performing I/O operations to stay within the optimal range. In contrast, the OL method exhibits a significant increase in time when the process-to-OST ratio exceeds 512, due to an excessive number of loaders assigned to a single OST, which prolongs the simulation duration. This trend highlights the limitations of traditional scheduling approaches when faced with multiple concurrent I/O processes and underscores the necessity of SwiftLoad's loader configuration module to maintain efficiency in high-load scenarios.

C. PERFORMANCE EVALUATION

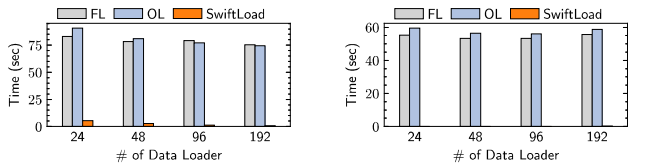
1) OVERALL PERFORMANCE

Figure 8(a) presents the makespan for the deduplication profiler when FL, OL, and SwiftLoad are applied. SwiftLoad demonstrates a significantly lower makespan compared to the other models, FL and OL. When the number of data loaders is 24 or 48, OL shows a higher makespan than FL due to increased metadata operations required for object-level scheduling. However, at 96 and 192 data loaders, there is a slight decrease or similar performance to FL. This is because at this point, the benefits of object-level scheduling outweigh the additional metadata request overhead. In contrast, SwiftLoad exhibits a 2.73 times lower latency than FL when the number of data loaders is 24, and a 5.63 times lower makespan than FL when the number of data loaders reaches 192. This improvement is attributed to the metadata catalog introduced by SwiftLoad, which nearly eliminates the metadata I/O time—a substantial portion of the utility's makespan. For FL and OL, metadata operation time becomes the dominant factor, resulting in minimal makespan reduction beyond 48 data loaders. However, SwiftLoad, which effectively removes metadata I/O time, consistently scales up from 24 data loaders, achieving 2.73 times better performance at 192 data loaders compared to 24 data loaders. However, while this represents consistent scaling,



(a) Deduplication Profiler (b) Data Augmentation Tool

FIGURE 8. Makespan of each utility with an increasing number of data loaders.



(a) Deduplication Profiler (b) Data Augmentation Tool

FIGURE 9. Metadata I/O time of each utility with an increasing number of data loaders.

the improvement is not linear in relation to the increase in data loaders. This is because the bottleneck shifts from metadata I/O to data I/O. Although SwiftLoad addresses this by introducing object-level scheduling, as the number of loaders increases, the process-to-OST ratio also rises, diminishing the effectiveness of object-level scheduling over time.

Figure 8(b) shows the makespan for the data augmentation tool under FL, OL, and SwiftLoad. The results follow a similar pattern to those observed in the deduplication profiler. FL consistently exhibits a lower makespan than OL. For example, at 24 data loaders, FL has a makespan of approximately 55.6 seconds, which is about 9.9% lower than OL's makespan of 61.7 seconds. At 192 data loaders, FL's makespan is approximately 56.0 seconds, which is about 4.9% lower than OL's 58.9 seconds. For both FL and OL, increasing the number of data loaders does not significantly reduce the makespan, as the overall makespan is primarily determined by the directory traversal time. Conversely, SwiftLoad shows a linear decrease in makespan as the number of data loaders increases, with a makespan of 47.8 seconds at 24 loaders and a drastic reduction to just 5.3 seconds at 192 loaders, demonstrating its efficiency.

2) METADATA I/O LATENCY

Figure 9(a) and (b) show the metadata I/O time for the two utilities using FL, OL, and SwiftLoad. Metadata I/O time encompasses the combined duration spent on open and close operations for each data loader and the directory traversal time. For SwiftLoad, this also includes the time required to load the catalog into memory. In Figure 9(a), When the number of data loaders is 24, FL exhibits a traversal time of 83.0 seconds, while OL shows a slightly higher time of 90.6 seconds. At 192 data loaders, these times are reduced to 75.3 and 74.4 seconds, respectively. In contrast, SwiftLoad demonstrates significantly lower traversal times, with only 5.3 seconds at 24 data loaders and a mere 0.6 seconds at 192 data loaders. In (b), the FL time was 55.6 seconds,

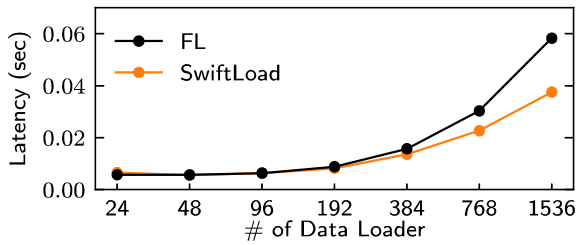


FIGURE 10. I/O read latency of FL and SwiftLoad for the data deduplication profiler with an increasing number of Data Loaders.

OL time was 58.9 seconds, and SwiftLoad completed the operation in less than a second. This substantial reduction is due to SwiftLoad only requiring the catalog to be loaded into memory, bypassing traditional directory traversal.

For the data augmentation tool, the minimal differences in the overall makespan and the metadata I/O time between FL and OL can be attributed to the parallelization of metadata, data I/O, and computation. As these processes are parallelized, the overall makespan becomes heavily dependent on the metadata I/O time. However, in the case of SwiftLoad, the metadata I/O time is negligible, as shown in Figure 9(b), leading to a scenario where the overall makespan in Figure 8(b) is primarily dependent on data I/O time. The reason the deduplication profiler exhibits some metadata I/O while the data augmentation tool does not is due to the different workloads they handle. The deduplication profiler manages files of varying sizes, from tiny files to those reaching tens of GBs. In such cases, multiple processes may simultaneously open and close the same files to perform reads, leading to potential lock contention. Conversely, the data augmentation tool primarily handles smaller files, which avoids such scenarios, making the metadata I/O-particularly open and close operations-negligible.

These results demonstrate that the metadata I/O time for both utilities is nearly eliminated through the introduction of the proposed catalog. By eliminating directory traversal time and calls to functions that retrieve stat or file layout information in the applications, the proposed metadata catalog removes the need to access the metadata server, except for essential file handling metadata I/Os such as open/close operations.

3) I/O READ LATENCY

Figure 10 represents the comparison of average read latency of SwiftLoad and FL across different number of data loaders. When the numbers of data loaders is small, the I/O latency of OL shows little difference compared to FL. Although with 24 data loaders, the OL exhibits slightly higher latency than FL, it remains within the error bounds. This indicates that with a small number of data loaders, the metadata I/O bottleneck is dominant. However, as the number of data loaders increases, the difference in read latency between FL and OL becomes significantly more pronounced. Specifically, with 192 data loaders, OL achieves a 7% lower latency than FL. This advantage increases to 33% and 55%

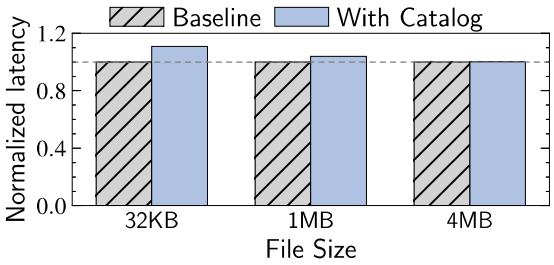


FIGURE 11. Normalized latency of the baseline application and the baseline application with runtime catalog construction for different file sizes.

with 768 and 1536 data loaders, respectively. The reason for this improvement is that OL induces less OST contention as I/O scales up compared to FL scheduling. These experimental results validate that the proposed method is significantly more effective than file-level scheduling in scalable environments.

4) OVERHEAD ANALYSIS ON CATALOG CONSTRUCTION

To investigate the potential overhead of runtime catalog construction, we examine the additional performance impact of system call interception. Figure 11 compares the normalized latency between the baseline application and the application with catalog construction. The application simulates a scientific workload by opening a file, calculating the hash of dummy content, and writing the hash value to an output file. For this experiment, we configured the application to write 400,000 files. As the created file size from the application increases, the overhead becomes negligible. When the file size is 32KB, the normalized latency is about 1.1; for 1MB, it is 1.04; and for 4MB, it is 1.002. This is because when the file size is small, the application writes and computes less, resulting in a high density of open/close requests which causes a slight delay due to the Catalog Server's throughput. At the same time, the benefit of using the catalog is greater for smaller files, as they intensify MDS bottlenecks. This is particularly evident in HPC utilities, where metadata I/O dominates due to shorter computational and data I/O phases relative to metadata operations. Consequently, there is a trade-off between the slight overhead introduced during catalog construction and the significant performance gains achieved by alleviating MDS bottlenecks. However, even in the case of 32KB files, the absolute overhead time is only about one to two seconds.

VI. CONCLUSION

In this paper, we present SwiftLoad, a novel approach designed to enhance the performance of parallel utilities in HPC environments by addressing both metadata I/O bottlenecks and OST contention through metadata catalog-based object-level I/O scheduling. We evaluated SwiftLoad using two representative utilities: a parallel data deduplication profiler, FS-C [8], and a data augmentation tool in a production HPC environment. The results demonstrate that SwiftLoad significantly improves performance, achieving a 5.63-fold and 11-fold increase in the makespan of the deduplication profiler and data augmentation tool, respectively.

REFERENCES

[1] *IBM Spectrum Scale (Formerly General Parallel File System, GPFS)*. Accessed: 2021. [Online]. Available: https://www.ibm.com/docs/en/spectrum-scale
[2] (Oct. 2023). *Lustre File System*. [Online]. Available: https://www.lustre.org/
[3] (Oct. 2023). *The Frontier*. [Online]. Available: https://www.olcf.ornl.gov/frontier/
[4] (Oct. 2023). *Supercomputer Lumi*. [Online]. Available: https://www.lumi-supercomputer.eu/
[5] (Oct. 2023). *Supercomputer Perlmutter*. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/
[6] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol. (MSST)*, May 2013, pp. 1–12.
[7] Y. Kim, S. Atchley, G. Vallée, and G. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, Jan. 2015, pp. 1–14.
[8] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in HPC storage systems," in *SC : Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–11.
[9] P. J. Braam and P. Schwan, "Lustre: The intergalactic file system," in *Proc. Ottawa Linux Symp.*, Jan. 2002, pp. 3429–3441.
[10] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proc. Linux Symp.*, 2003, pp. 380–386.
[11] D. Sikich, G. D. Natale, M. Legendre, and A. Moody, "MpiFileUtils: A parallel and distributed toolset for managing large datasets," in *Proc. PDSW-DISCS*, Oct. 2017, pp. 1–4.
[12] S. Xiong, F. Wang, and Q. Cao, "A Bloom filter based scalable data integrity check tool for large-scale dataset," in *Proc. 1st Joint Int. Workshop Parallel Data Storage Data Intensive Scalable Comput. Syst. (PDSW-DISCS)*, Nov. 2016, pp. 55–60.
[13] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang, "P-dedupe: Exploiting parallelism in data deduplication system," in *Proc. IEEE 7th Int. Conf. Netw., Archit., Storage*, Jun. 2012, pp. 338–347.
[14] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "Middleware support for many-task computing," *Cluster Comput.*, vol. 13, no. 3, pp. 291–314, Sep. 2010.
[15] A. Kougkas, H. Devarajan, J. Lofstead, and X. Sun, "LABIOS," in *Proc. 28th Int. Symp. High-Performance Parallel Distrib. Comput.*, Jun. 2019, pp. 1–14.
[16] Y. Qian, W. Cheng, L. Zeng, X. Li, M.-A. Vef, A. Dilger, S. Lai, S. Ihara, Y. Fan, and A. Brinkmann, "Xfast: Extreme file attribute stat acceleration for lustre," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2023, pp. 1–12.
[17] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing I/O bottleneck for large-scale deep learning applications," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2022, pp. 324–335.
[18] J. Liao, G. Xiao, and X. Peng, "Log-less metadata management on metadata server for parallel file systems," *Scientific World J.*, vol. 2014, no. 1, 2014, Art. no. 813521.
[19] L. Wang, Y. Lu, W. Zhang, and Y. Lei, "Distributed and scalable directory service in a parallel file system," *IEICE Trans. Inf. Syst.*, vol. E99.D, no. 2, pp. 313–323, 2016.
[20] J.-K. Lee and T. Hong, "Analysis of traffic and attack frequency in the NURION supercomputing service network," *KIPS Trans. Comput. Commun. Syst.*, vol. 9, no. 5, pp. 113–120, Jan. 2020.

SEOYEONG LEE received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2023. She is a member with the Data-Intensive Computing and Systems Laboratory, Department of Computer Science and Engineering, Sogang University. Her research interests include I/O performance optimization, parallel and distributed systems, and systems for artificial intelligence (AI).

JUNGHWAN PARK received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2023. He is currently a member with the Data-Intensive Computing and Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include parallel and distributed systems and systems for artificial intelligence (AI).

YOOCHAN KIM received the B.S. degree in computer science and mathematics from Sogang University, Seoul, South Korea, in 2023, where he is currently pursuing the M.S. degree with the Department of Computer Science and Engineering. His research interests include I/O optimization for AI, cloud computing solutions, mathematical modeling, and distributed deep learning.

SAFDAR JAMIL received the B.E. degree in computer systems engineering from the Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing the M.S. leading to Ph.D. integrated program degree with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. He is a member with the Data-Intensive Computing and Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include scalable indexing data structures and algorithms, key-value stores, zoned namespace storage, and storage optimization techniques.

AWAIS KHAN received the B.S degree in bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan, and the Ph.D. degree in computer science and engineering from Sogang University, Seoul, South Korea, in 2021. He was with the Digital Research Laboratories as a Software Engineer, from 2012 to 2015. Currently, he is an HPC Systems Scientist with Oak Ridge National Laboratory, Oak Ridge, TN, USA. Prior to his role at ORNL, he was a Senior Systems Performance Engineer with Micron Tech. His research interests include service optimizations for AI/ML applications, large-scale data checkpointing, network topology and cost modeling, memory-centric computing and HPC, data management services in HPC, object storage systems, cluster-scale deduplication, and parallel and distributed file systems.

SEUNG WOO SON (Member, IEEE) received the Ph.D. degree in computer science and engineering from The Pennsylvania State University, USA, in 2008. Since 2014, he has been with the University of Massachusetts Lowell. He is currently an Associate Professor. Before UMass Lowell, he was a Postdoctoral Researcher with the Electrical Engineering and Computer Science Department, Northwestern University, and the Math and Computer Science Division, Argonne National Laboratory. His research interests include high-performance computing with an emphasis on parallel I/O and storage systems, computer architecture, compilers, embedded systems, most recently, and data compression for various datasets, including HPC and IoT datasets, and systems and machine learning. He was a recipient of the National Science Foundation CAREER Award, in 2018, and the Amazon Research Award, in 2020.

JAE-KOOK LEE received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Chungnam National University, South Korea, in 2002, 2004, and 2012, respectively. He has been a Senior Researcher with the Supercomputing Infrastructure Center, Korea Institute of Science and Technology Information (KISTI), South Korea, since 2013. His research interests include HPC systems and network security.

DO-SIK AN received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Jeonbuk National University, South Korea, in 2007, 2010, and 2017, respectively. He has been a Senior Researcher with the Supercomputing Infrastructure Center, Korea Institute of Science and Technology Information (KISTI), South Korea, since 2017. His research interests include HPC systems and interconnect networks.

TAEYOUNG HONG received the B.S. and M.S. degrees in physics from Sungkyunkwan University, South Korea, in 1999 and 2002, respectively. He has been the Director of the Supercomputing Infrastructure Center, Korea Institute of Science and Technology Information (KISTI), South Korea, since 2003. His research interests include HPC system operation and parallel file systems.

YOUNGJAE KIM (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2001, the M.S. degree in computer science from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from The Pennsylvania State University, University Park, PA, USA, in 2009. He is currently a Professor with the Department of Computer Science and Engineering, Sogang University. Before joining Sogang University, he was a Research and Development Staff Scientist with U.S. Department of Energy's, Oak Ridge National Laboratory, from 2009 to 2015, and an Assistant Professor with Ajou University, Suwon, South Korea, from 2015 to 2016. His research interests include operating systems, file and storage systems, parallel and distributed systems, computer systems security, and performance evaluation.