Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control

Ricardo Macedo, Mariana Miranda, Yusuke Tanimura†, Jason Haga†
Amit Ruhela⋆, Stephen Lien Harrell⋆, Richard Todd Evans‡, José Pereira, João Paulo
INESC TEC & University of Minho †*AIST* ⋆*TACC & UTAustin* ‡*Intel*

***Abstract*—Modern I/O applications that run on HPC infrastructures are increasingly becoming read and metadata intensive. However, having multiple applications submitting large amounts of metadata operations can easily saturate the shared parallel file system's metadata resources, leading to overall performance degradation and I/O unfairness. We present PADLL, an application and file system agnostic storage middleware that enables QoS control of data and metadata workflows in HPC storage systems. It adopts ideas from Software-Defined Storage, building data plane stages that mediate and rate limit POSIX requests submitted to the shared file system, and a control plane that holistically coordinates how all I/O workflows are handled. We demonstrate its performance and feasibility under multiple QoS policies using synthetic benchmarks, real-world applications, and traces collected from a production file system. Results show that PADLL can enforce complex storage QoS policies over concurrent metadata-aggressive jobs, ensuring fairness and prioritization.**

***Index Terms*—QoS, Storage management, PFS, Metadata.**

I. Introduction

Modern supercomputers are establishing a new era in high-performance computing (HPC), providing unprecedented compute power that enables parallel applications to run at large-scale [2], [5], [52]. However, contrary to long-lived assumptions about HPC workloads where applications were predominately compute-bound and write-dominated, modern applications (*e.g.,* Deep Learning (DL) training) are data-intensive, read-dominated, and generate massive bursts of metadata operations [15], [21], [47]. In fact, recent studies have noted that many applications spend 15-40% of their execution time performing storage I/O, and expect this value to increase for exascale systems [18], [22], [47], [48].

Contrary to compute resources (*e.g.,* CPU, GPU), which are exclusively reserved to a given job, storage resources – including both *data* and *metadata* – are often shared across jobs, for example, when accessing the same Parallel File System (PFS). While modern workloads demand scalable, high throughput, and low latency storage, having multiple concurrent jobs competing for shared storage resources can lead to severe I/O contention and overall performance degradation [38], [48], [50]. Thus, not ensuring storage quality of service (QoS) guarantees in large-scale HPC systems means that jobs will unfairly access shared resources and execute without sustained I/O performance [50].

Challenges. Efficiently controlling I/O workflows of large-scale HPC storage systems poses unique challenges, of which existing approaches have been unable to address.

Manual intervention. In several HPC facilities, system administrators stop jobs with aggressive I/O behavior (*e.g.,* accessing large datasets made of small-sized files, overloading the shared storage with unnecessary data or metadata requests) and temporarily suspend job submission access for users that do not comply with the cluster's guidelines [30], [38]. However, this *reactive approach* is triggered when the job has already slowed the storage system and impacted the QoS of other jobs.

Intrusiveness to I/O layers. While solutions like GIFT [48], CALCioM [22], and TBF [50] aim at mitigating I/O contention and variability, these are tightly coupled to the implementation of core layers of the HPC I/O stack, including the shared file system, job scheduler, and I/O libraries. Such an approach requires profound code refactoring, increasing the work needed to maintain and port it to new platforms. For instance, optimizations made at Lustre may not be directly applicable over other file systems (*e.g.,* BeeGFS, PVFS), as even though they share a similar high-level design, the internal I/O logic differs across implementations.

Partial visibility and I/O control. Few solutions enable QoS control from the application-side (*i.e.,* at the compute node level), thus not requiring changes to core layers of the I/O stack [29]. However, these act in isolation (*i.e.,* agnostic of other jobs in execution), being unable to holistically coordinate the I/O generated from multiple jobs that compete for shared storage, ultimately leading to I/O contention and waste of system resources (*e.g.,* unused I/O bandwidth) [43], [55].

Metadata remains overlooked. While existing proposals focus on achieving QoS over data workflows (I/O bandwidth) [14], [22], [25], [31], [48], [50], [60], [61], the metadata counterpart has not received the same level of attention. In fact, several HPC centers are observing a surge of metadata operations in their clusters and expect this to become more severe over time. This is problematic given that even the I/O operations of a single job can saturate the PFS metadata resources, leading to unresponsiveness of the file system and increased execution time for all running jobs [30], [38].

This work. To address these challenges, we present PADLL, an application and file system agnostic storage middleware that enables QoS control of data and metadata workflows in HPC storage systems. Fundamentally, it allows system administrators to proactively and holistically control the rate at which POSIX requests are submitted to the PFS.

PADLL adopts ideas from Software-Defined Storage [42],

979-8-3503-0119-9/23/$31.00 ©2023 IEEE
DOI 10.1109/CCGrid57682.2023.00015

following a decoupled design that separates the I/O logic into a *data plane* and a *control plane*. The *data plane* is a multi-stage component that actuates at the compute node level, where each stage mediates the I/O requests between a given application and the shared file system. Specifically, stages transparently handle applications' requests by intercepting POSIX calls (*e.g.,* `open`, `close`, `read`, `getattr`) and dynamically rate limiting those that are destined towards the PFS. This makes PADLL applicable over multiple applications and cross-compatible with POSIX-compliant file systems, without requiring changes to any core layer of the HPC I/O stack.

Stages are then controlled by a logically centralized manager, the *control plane*, that defines how all I/O workflows should be handled. It acts as a global coordinator with system-wide visibility that continuously monitors and adjusts the I/O rate of data plane stages. It does so by dynamically allocating storage resources (*i.e.,* metadata rate, I/O bandwidth) among jobs upon workload and system variations, ensuring that QoS policies are met at all times. Further, to orchestrate a large number of data plane stages concurrently, the control plane is hierarchically distributed, made of *global* and *local controllers*.

To ensure custom and fine-grained control over I/O workflows, PADLL enables system administrators to specify QoS policies through *control algorithms*, which can be as simple as statically rate limiting a specific type of request (*e.g.,* `open`) of a given job, to more complex ones, as achieving proportional sharing of metadata resources across all active jobs [11], [26].

Implementation and evaluation. To validate the performance and feasibility of our approach, we implemented a PADLL prototype, including multiple control algorithms to enforce different storage QoS policies, namely *uniform* and *priority-based rate distributions*, *proportional sharing*, and a new *max-min fair share* algorithm suited for volatile workloads.

Experiments were conducted using both synthetic benchmarks and real-world applications (IOR [54] and TensorFlow [9], respectively), as well as traces of metadata operations collected from a production Lustre file system of the ABCI supercomputer. Results demonstrate that: (1) PADLL effectively controls the rate of I/O workflows at different granularities, including request type (*e.g.,* `open`, `read`, `getattr`), request class (*e.g.,* metadata, data), and job; (2) it enables enforcing storage QoS policies over distributed, metadata-aggressive jobs holistically; (3) when configured with our new control algorithm, under volatile workloads, PADLL maximizes the use of metadata resources, accelerating the performance of resource-hungry jobs without degrading over-provisioned ones; and (4) a single stage is able to service requests at high throughput rates (up to 3.20 Mops/s), and the control plane can manage the overall system at μs-scale.

In summary, the paper makes the following contributions:

- A **study** that *analyzes traces* from a production Lustre file system at ABCI, highlighting the importance of ensuring QoS over metadata resources (§II).
- **PADLL**, an application and PFS agnostic storage middleware that enables QoS control in HPC storage systems (§III). The system is publicly available at dsrhaslab/padll, dsrhaslab/cheferd, and Zenodo [41] repositories.
- A **new max-min fair share algorithm** that enables differentiated QoS across multiple jobs, while preventing resource over-provisioning under volatile workloads (§IV).
- **Experimental results** demonstrating PADLL's performance and applicability under different scenarios using both synthetic and realistic I/O workloads (§V).

II. Background and Motivation

Parallel file systems are the storage backbone of HPC infrastructures, being used to store and retrieve, on a daily basis, petabytes of data from hundreds to thousands of concurrent jobs. In this paper, we focus on Lustre-like file systems (*e.g.,* Lustre [53], BeeGFS [16], PVFS [13]), which are present in most TOP500 supercomputers. A typical Lustre-like file system consists of several building blocks. *Metadata Servers (MDSs)* maintain the file system namespace (*e.g.,* file names and layouts, permissions, extended attributes) and handle all metadata operations. The namespace is persisted in a single or multiple *Metadata Targets (MDTs)* nodes. Data operations are serviced by *Object Storage Servers (OSSs)* which are connected to compute nodes via high-speed interconnects, and store files on *Object Storage Targets (OSTs)*. Files are typically distributed across multiple OSTs for parallelism and availability. File system *clients* reside at compute nodes (in kernel-level) and access the file system using standard POSIX system calls (*e.g.,* `open`, `read`, `close`, `getattr`).

Depending on the scale of the file system, metadata nodes assume different configurations [7]. In some deployments, the namespace is persisted across multiple MDTs and a single MDS handles all metadata operations, having additional MDS nodes as standby replicas; in others, different MDSs/MDTs manage/persist different parts of the namespace.

Metadata workflow and limitations. Regardless of the application, workload, or job, whenever a file needs to be accessed (*e.g.,* create/open/remove file, access control, extended attributes) the main I/O path always flows through the metadata service. When creating files, the file system client issues a RPC routine to the MDS, which will create a new entry in the namespace and assign OSTs in a capacity-balanced manner to persist the data; for existing files, the MDS retrieves information about the file stripe and OST mappings.

When used at scale, this centralized design comprises several limitations that can severely bottleneck the file system and impact the performance of all running jobs. First, different metadata operations carry different costs to the PFS. Depending on the file system implementation, read-only operations such as `getattr` only require acquiring read-locks, while operations like `open`, `close`, and `unlink` require more expensive locking, as they need to update the namespace state [8], [12]. Other operations, such as `mkdir` or `rename`, require even stronger guarantees (*i.e., atomicity*). Second, modern workloads, such as DL training, comprise large-scale datasets that can reach TiB in size and are made of multiple small-sized files, which generate high and continuous bursts

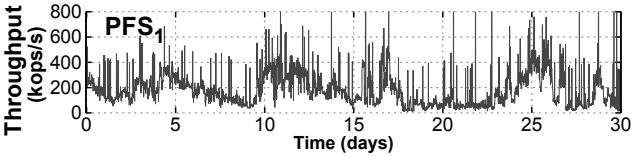

Fig. 1: *Metadata throughput in* PFS_1 *over a 30-day period.*

of metadata operations [20], [34]. Third, the number of file system *clients* is several times higher than available MDSs, which can become saturated when several concurrent jobs have aggressive I/O metadata behavior [38].

A. Analyzing Metadata Operations in Production Clusters

To understand the impact of metadata operations in production, we analyze the logs of a Lustre file system from the ABCI supercomputer. The file system is a DDN ExaScaler Lustre composed of 2 MDSs in a hot-standby configuration, backed by 6 MDTs, and 36 OSTs that provide 9.5 PiB of storage capacity. We refer to this file system as PFS_1.

We monitored the I/O activity of the most frequent metadata operations at MDSs/MDTs, using DDNStorage's LustrePerfMon [4]. We collected per-MDT performance statistics for `open`, `close`, `getattr`, `setattr`, `rename`, `mkdir`, `mknod`, `rmdir`, `statfs`, `sync`, and `unlink` operations. The logs report per-operation performance statistics captured with 1-minute samples over a 30-day observation period. Further, we also monitored the I/O bandwidth (`read` and `write`) observed by OSSs over the same collection period.

Overall metadata load. We first examine the throughput of metadata operations throughout the overall observation period. Fig. 1 depicts the rate of all collected metadata operations at PFS_1. Metadata operations are submitted at a massive rate, averaging 200 kops/s. Over different periods, PFS_1 continuously serves requests over 400 kops/s, which last several hours to days, and experiences bursts that peak at 1 Mops/s. Indeed, the workload is extremely volatile, frequently exhibiting periods of low throughput (50 kops/s or lower) to immediately spike up to 450 kops/s (or higher).

Furthermore, we observe that this load is much higher than those reported in other clusters [47]. For example, a study from NERSC reports that the PFS shared by *Edison* and *Cori* supercomputers had an average rate of 9.7 kops/s and 7 kops/s for `open` and `close` operations, respectively; while PFS_1 experiences 29 kops/s and 43.5 kops/s. While the metadata load depends on different factors (*e.g.,* cluster size, workload, file organization), we suspect that these values mainly stem from the type of jobs conducted at ABCI, which are mostly AI-oriented (*e.g.,* DL training).

> ***Observation:*** *Modern I/O workloads generate massive amounts of metadata operations. Based on previous studies [47] and the results observed from* PFS_1*, it is expected that these values will continue to increase over time. This means that exclusively ensuring QoS over data workflows (i.e.,* `read` *and* `write`*) is no longer enough to achieve fair access to PFS resources, and thus, metadata operations need to be managed as well.*

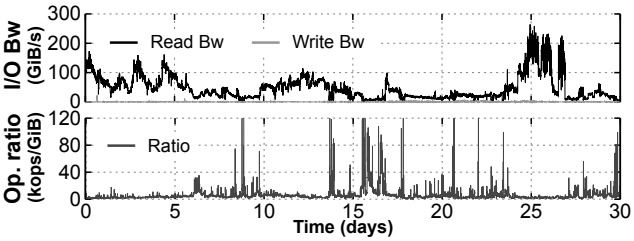

Fig. 2: *Ratio of metadata operations to I/O bandwidth in* PFS_1.

Ratio of metadata operations to I/O bandwidth. We now analyze the correlation between data and metadata operations at PFS_1. As most jobs conducted at ABCI are AI-oriented, PFS_1 experiences low write throughput (average rate of 0.6 GiB/s), while reads are served at an average rate of 48 GiB/s, as depicted in Fig. 2 (top). However, comparing the amount of metadata operations and the I/O bandwidth serviced by PFS_1, as depicted in Fig. 2 (bottom), we observe that, in several periods, metadata operations have significantly higher throughput than GiB/s read/written from/to the PFS. For instance, between days 13 and 20, metadata operations were submitted at a rate over 120 kops for each GiB (or 120 ops/MiB) read/written from/to the PFS. This means that even if hard QoS limits are imposed over data operations, metadata workflows may still remain unchanged.

> ***Observation:*** *We observed several periods where the amount of submitted metadata operations far exceeds the GiBs of data read/written from/to* PFS_1*. This means that there is not a strict dependency between both operation classes, consolidating the need for ensuring QoS over metadata workflows as well.*

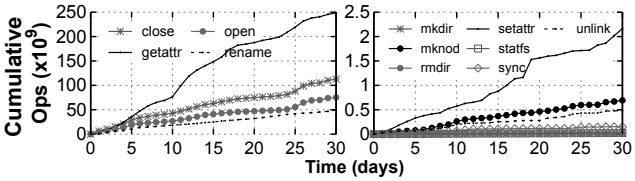

Fig. 3: *Cumulative metadata operations in* PFS_1.

Type and frequency of metadata operations. Fig. 3 shows the type and amount of metadata operations in PFS_1. `open`, `close`, `getattr`, and `rename` are the most frequent operations, accounting for 98% of the total load. Notoriously, several of these are particularly costly to the PFS and prone to cause I/O contention, due to expensive locking (*i.e.,* `open` and `close`) and atomicity guarantees (*i.e.,* `rename`). As for `getattr` operations, while less costly, PFS_1 received almost 250 billion requests throughout the overall observation period (corresponding to ≈47% of the total load), representing an average and continuous rate of 95.8 kops/s.

> ***Observation:*** *The most predominant metadata operations, namely* `open`*,* `close`*,* `rename`*, and* `getattr`*, account for 98% of* PFS_1*'s metadata load. Given that not all operations entail the same cost and I/O pressure over the shared metadata resources, operations should be controlled with fine-granularity.*

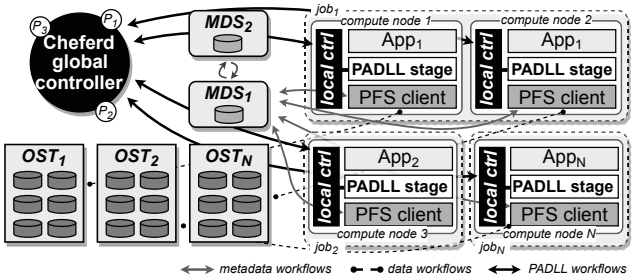

Fig. 4: PADLL *high-level architecture.* *It is composed of a hierarchical control plane and multiple data plane stages.*

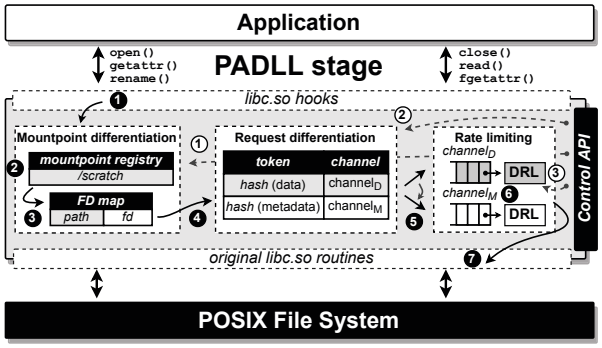

Fig. 5: PADLL *data plane stage design and operation flow.* *Black circles depict the operation flow of intercepted POSIX requests, while white circles depict housekeeping and control operations of* PADLL.

III. PADLL STORAGE MIDDLEWARE

PADLL is a storage middleware that enables system administrators to proactively and holistically control the rate of data and metadata workflows to achieve QoS in HPC storage systems. Its design is built under the following core principles.

Application and PFS agnostic. PADLL does not require code changes to any core layer of the HPC I/O stack, being agnostic of the applications it is controlling as well the file system to which requests are submitted to. This makes PADLL applicable over multiple applications and compatible with POSIX-compliant storage systems, including both local (*e.g.,* `xfs`, `ext4`) and distributed file systems (*e.g.,* Lustre, BeeGFS).

Fine-grained I/O control. PADLL classifies, differentiates, and controls requests at different levels of granularity, including *operation type* (*e.g.,* `open`, `read`), *operation class* (*e.g.,* `data`, `metadata`), *user*, and *job*, which allows applying different types of policies (*e.g.,* only rate limit `open` calls, rate limit all `metadata` operations, rate limit job `xyz` to X ops/s).

Global visibility. PADLL ensures holistic control of all I/O workflows and coordinated access to the PFS, preventing I/O contention and unfair usage of shared storage resources.

Custom QoS specification. PADLL enables system administrators to create custom QoS policies for rate limiting jobs running at the cluster (*e.g.,* uniform [29] and priority-based rate distribution [50], proportional sharing [43], [59], DRF [26]), protecting the PFS from greedy jobs and I/O burstiness.

Fig. 4 outlines PADLL's high-level architecture. It follows a decoupled design that separates the I/O logic into two planes of functionality. The *data plane* (§III-A) is a multi-stage component that provides the building blocks for differentiating and rate limiting I/O workflows. The *control plane* (§III-B) is a global coordinator that manages all data plane stages to ensure that storage QoS policies are met over time and adjusted according to workload variations.

A. Data Plane

PADLL's data plane stages (or *stages* for short) actuate at the compute node level, each sitting between the application and the PFS. PADLL transparently intercepts (using `LD_PRELOAD`) and reimplements multiple POSIX system calls from different operation classes before being submitted to the PFS, including *data* (*e.g.,* `read`, `pwrite`), *metadata* (*e.g.,* `open`, `rename`), *extended attributes* (*e.g.,* `getattr`, `setattr`), and *directory management* (*e.g.,* `mkdir`, `mknod`).

To control the rate of I/O workflows of a given job, multiple PADLL stages may be used. Under single node jobs, a single data plane stage controls all I/O workflows. As depicted in Fig. 4, this is the case of job_2 where App_2 only executes at *compute node 3*. For distributed jobs, where application instances run on separate compute nodes, multiple data plane stages need to be set (*i.e.,* one per instance). For example, as depicted for job_1 in Fig. 4, two stages are required to effectively rate limit all I/O workflows of App_1, since it executes in *compute nodes 1* and *2*. Furthermore, each compute node can also have multiple stages, created from multi-process applications (*i.e.,* one stage per process).

To handle the I/O workflows of a given application, stages are organized in three main components, as depicted in Fig. 5.

Mountpoint differentiation. Internally, compute nodes hold multiple file systems, including *local*, which are used for managing local storage devices (*e.g.,* `xfs`, `tmpfs`), and *remote*, for accessing files in distributed storage systems like Lustre and BeeGFS. Given that PADLL intercepts POSIX requests regardless of the targeted file system, it needs to identify which requests are destined towards the PFS, so these can be treated accordingly. This is achieved in three phases.

Registering mountpoints: first, the system administrator defines which mountpoints should be managed with PADLL, by registering their full path on a mountpoint registry (①). For example, as depicted in Fig. 5, the stage only handles the requests that are destined towards `/scratch`.

Handling path-based operations: all system calls that define the *pathname* of the targeted file, such as `open`, `fopen`, `rename`, and `mkdir`, are then intercepted (❶) and analyzed (❷). Requests that are destined towards the registered mountpoints proceed to the subsequent components (❹); otherwise, these are directly submitted to the corresponding file system without additional processing (❼).

Handling file descriptor (FD) based operations: to determine if a system call that accesses files through FDs (*e.g.,* `read`, `fgetattr`) is destined towards a registered mountpoint, for each `open`-based call, PADLL stores the resulting FD in a

file mapping module (❸). Whenever any of these system calls is intercepted (❶), PADLL verifies if the corresponding FD is valid (❸), proceeding to the subsequent components (❹); otherwise, it is submitted to the corresponding file system without changes (❼). On `close`, the FD is removed from the file mapping. File pointer based operations (*e.g.,* `fread`) are handled in a similar manner.

Request differentiation and rate limiting. Internally, as depicted in Fig. 5, stages are organized in multiple queues (entitled as *channels*), each with a token-bucket that determines the rate of its requests (*DRL*). A token-bucket is a commonly used mechanism for controlling the rate and burstiness of I/O workflows [11]. Each of these *channels* only serves a specific set of requests. For example, $channel_D$ and $channel_M$ handle all `data` and `metadata` operations destined towards `/scratch`, respectively. The type of requests each *channel* handles (②), as well as the rate of each token-bucket are set by the control plane (③).

After validating their associated mountpoint, requests are differentiated based on a specific set of attributes that characterize them, including the *operation type* (*e.g.,* `open`, `getattr`), *operation class* (*e.g.,* `metadata`, `data`), *operation size*, *userID*, and *jobID* (❹). For each request, the stage hashes its attributes into a fixed-size token through a computationally cheap hashing scheme [10], which maps the request to the corresponding *channel* that will enforce it (❺). If no match is found, it means that the request should not be handled by PADLL, being submitted to the file system without additional processing (❼). Once in the *channel*, requests are then processed (*i.e.,* dequeued) and rate limited according to the token-bucket's rate (❻). After this process, requests are then submitted to the targeted POSIX file system (❼).

B. Hierarchical Control Plane

The control plane is a logically centralized component with system-wide visibility that defines how all I/O workflows in the HPC cluster are handled. It does so by continuously communicating with data plane stages to *collect I/O metrics* (*e.g.,* operation rate, I/O bandwidth) and *enforce stage-specific rules* that dynamically adjust the workflow's rate (*i.e.,* at token-buckets) to respond to workload and system variations, as well as new policies set by system administrators.

As depicted in Fig. 4, PADLL's control plane follows a hierarchical distribution. *Local controllers* have local visibility and manage all data plane stages of a given compute node. A *global controller* has global visibility and enforces cluster-wide QoS policies by orchestrating all *local controllers*. Since multiple stages can execute in the same compute node, creating a hierarchy of controllers allows minimizing the number of connections and exchanged messages to the *global controller*.

Control logic. In PADLL, the control logic is specified through storage QoS policies. These can be as simple as individually set the rate for the `open` calls of a given job, to more complex ones such as dynamically reserve shares of metadata operations for all jobs in the cluster. The latter are defined through *control algorithms*, which are implemented in a *feedback control loop*, where the control plane repeatedly performs four main steps, namely *collect*, *compute*, *enforce*, and *sleep*.

Collect: local controllers continuously collect statistics (*e.g.,* per-stage metadata rate) from their assigned data plane stages (*i.e.,* which are co-located in the same compute node). These metrics are aggregated and reported to the *global controller*. For multi-node jobs, the *global controller* aggregates the statistics reported from all *local controllers* where the job is being executed. For instance, in Fig. 4, to observe the metrics of job_1, the *global controller* aggregates statistics from *local controllers* of *compute nodes 1* and *2*.

Compute: the *global controller* then verifies if all policies are being met, by correlating the QoS limits defined by the system administrator (*e.g.,* maximum metadata rate defined for job_1) and the rates reported from stages (*e.g.,* actual rate experienced by job_1). If the imposed limits are not being met, due to workload or system variations, it generates new rates (*i.e., rules*) to the uncompliant stages.

Enforce: all generated rules are then submitted to *local controllers*, which in turn will be submitted to the corresponding stages, where token-buckets will be adjusted with a new rate.

Sleep: as a complementary step, *sleep* defines the periodicity of control cycles (*e.g.,* perform the aforementioned control steps at 1-second intervals). With small intervals stages will be adjusted more frequently and impose higher control overhead (*i.e.,* continuously collect statistics and enforce rules), while with larger intervals jobs may become unsupervised for long periods, which can be harmful under volatile workloads.

Orchestrating stages of distributed jobs. Every time a single or multi-node job starts, its stages are initialized and connected to the corresponding *local controllers*. Each stage sends to the controller information that characterizes the job and the node it is running, such as the *jobID*, `PID`, *hostname*, and *userID*. *Local controllers* synchronize this information with the *global controller*. Based on this, the control plane knows which job each stage respects to, orchestrating the stages that belong to the same `job-ID` as a single one.

C. Implementation

We have implemented PADLL's *data* and *control planes* with 16K and 6K lines of C++ code, respectively.

Transparently intercepting POSIX calls. The data plane uses `LD_PRELOAD` to transparently intercept POSIX calls and handle them before being submitted to the PFS. It supports 42 system calls from different operation classes, including data, metadata, extended attributes, and directory management.

Rate limiting. The logic for rate limiting requests (*e.g.,* queues, token-buckets) was built using PAIO [43], a framework for building custom-made, user-level storage data planes.

Communication. Communication between controllers is established through RPC calls, using the gRPC framework [6], while communication between *local controllers* and data plane stages is established using UNIX Domain Sockets. For the latter, we adopt the control interface proposed in PAIO.

Control delegation. Currently, *local controllers* act as proxies that aggregate statistics from stages before being dispatched to the *global controller*, and forward enforcement rules to the respective stages. We defer the delegation of control logic (*i.e.,* control partitioning) to *local controllers* to future work.

IV. Control Algorithms

We now present state-of-the-art control algorithms supported by PADLL, and introduce a new QoS algorithm for preventing resource over-provisioning. In PADLL, control algorithms can be either *static* (§IV-A) or *dynamic* (§IV-B). For all, we consider the amount of operations serviced by the PFS, either data (bandwidth) or metadata (IOPS), as the shared resource to be distributed among jobs. Further, we define Max_R as the maximum throughput of either data or metadata that a given PFS can service.

A. Static Control Algorithms

Static control algorithms enable defining, for each job, *fixed I/O limits* for accessing shared storage resources.

Uniform. In a *uniform rate distribution* jobs are throttled with a fixed limit throughout their execution, regardless of their size (*i.e.,* number of compute nodes), duration, and workload (*e.g.,* access pattern, I/O load, dataset size). Such an approach is useful to equally distribute resource shares among jobs.

Priority. In a *priority-based rate distribution*, PFS resources are distributed based on a given priority, where jobs with higher priority have access to a larger resource share.

These algorithms follow a similar approach to those of *cgroups blkio* [3] and OOOPS [29]. Analogously, as I/O limits are fixed throughout the jobs' execution, these do not leverage from PADLL's global visibility, which can result in a misuse of system resources. Specifically, jobs cannot be dynamically adjusted when (1) there are leftover resources (*e.g.,* a job ended its execution and released its resource share), leading to *under-provisioning*; or (2) jobs are assigned with shares larger than they need (*e.g.,* job submits operations at a rate lower than the defined limit), experiencing *over-provisioning*.

B. Dynamic Control Algorithms

Dynamic control algorithms enable assigning, to each job, resource shares that change over time (based on *soft* and *hard I/O limits*), being adaptable to workload or system variations (*e.g.,* jobs entering or leaving the system, volatile workloads). These algorithms are implemented in a *feedback control loop* and are executed in the *global controller*.

Proportional sharing. To ensure I/O fairness while preventing under-provisioning scenarios, we implemented a max-min fair share control algorithm that enforces *per-job rate reservations*, similar to those in [43], [50], [59]. At any given time, jobs are allocated with I/O resources in order of increasing *demands* (*i.e.,* defined QoS limit), where (1) no job gets a share larger than its *demand* and (2) jobs with unsatisfied demands get equal shares of resources. Then, whenever there are leftover resources – for example, the current metadata rate used by all jobs has not reached Max_R – the algorithm distributes them across active jobs in a proportional manner.

Algorithm 1 *Prop. Sharing without False Resource Allocation*

Initialize: $Max_R = N$; $Active > 0$; $demand_i > 0$; $usage_i > 0$; $0 \leq \varepsilon \leq 1$

1: $\{usage_0, \ldots, usage_{Active-1}\} \leftarrow$ *collect ()*
2: $left_R \leftarrow Max_R$
3: **for** $i = 0$ in [0, $Active-1$] **do**
4: $\quad fair\_share \leftarrow \frac{left_R}{Active-i}$
5: $\quad$ **if** $usage_i \leq demand_i$ **then**
6: $\quad\quad threshold_i \leftarrow (demand_i - usage_i) * \varepsilon$
7: $\quad\quad rate_i \leftarrow min\ (usage_i + threshold_i, fair\_share)$
8: $\quad$ **else**
9: $\quad\quad rate_i \leftarrow min\ (demand_i, fair\_share)$
10: $\quad left_R \leftarrow left_R - rate_i$
11: $total\_usage \leftarrow \sum_{j=0}^{Active-1} usage_j$
12: **for** $i = 0$ in [0, $Active-1$] **do**
13: $\quad usage\_proportion_i \leftarrow \frac{usage_i}{total\_usage}$
14: $\quad rate_i \leftarrow rate_i + (usage\_proportion_i * left_R)$
15: *enforce* $(\{rate_0, \ldots, rate_{Active-1}\})$
16: *sleep (loop_interval)*

While this algorithm is well-suited for workloads with sustained I/O load, it is suboptimal under volatile workloads. Specifically, the algorithm always allocates a share of the I/O resources to jobs, regardless of their I/O load; if a given job follows a volatile workload, the algorithm may assign a share larger than it needs, resulting in over-provisioning. We refer to this behavior as ***false resource allocation***.

Proportional sharing without false allocation. We propose a new proportional sharing algorithm that prevents false resource allocation to ensure storage QoS under volatile workloads (Alg. 1), entitled as PSFA. Briefly, rather than assigning resource shares exclusively based on the number of active jobs in the system and their *demands*, we consider the actual usage (*i.e.,* I/O load) of each job and redistribute resources in a max-min fair share manner based on those observations.

In more detail, the algorithm performs the following steps. First, it collects statistics from each active job's stage to determine its actual rate, given by $usage_i$ (1). For each active job, the algorithm computes its *fair_share* (4) and verifies if the current rate ($usage_i$) is lower than its *demand* (5). Under this scenario, job_i can be serviced at a rate lower than its *demand*. As such, it assigns the minimum between *fair_share* and $usage_i$ + $threshold_i$ (7). $Threshold_i$ is computed based on the product of a configurable ε value and the difference between $demand_i$ and $usage_i$, and is used to absorb the rate of highly volatile workloads (6). If $usage_i$ is higher than $demand_i$, the controller assigns the minimum between $demand_i$ and the *fair_share* (8–9). The algorithm then distributes leftover rate ($left_R$) across actives jobs (11–14). Specifically, it computes the overall rate used by all jobs (11), and assigns $left_R$ based on their usage proportion, given by $usage\_proportion_i$ (13–14). Finally, the *global controller* generates rules (`enforce`) to be submitted to each *local controller* (15), and sleeps for *loop_interval* before beginning a new control cycle (16).

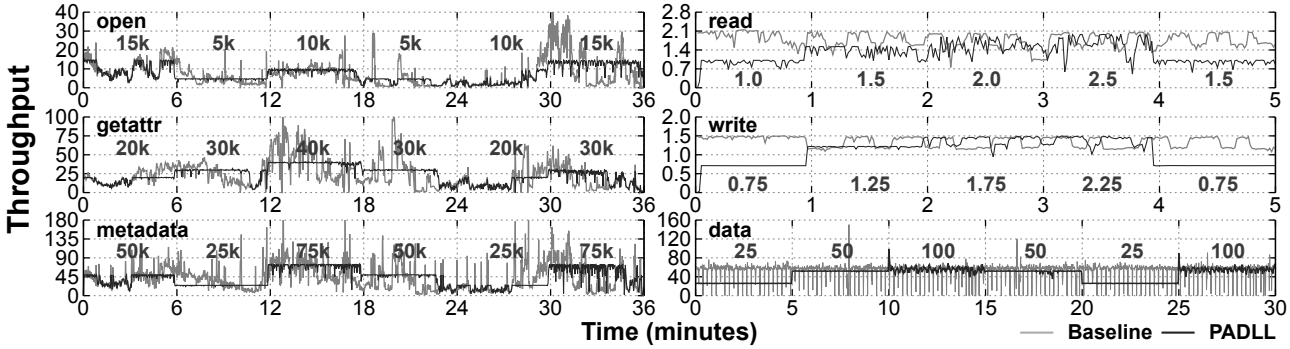

Fig. 6: ***Per-operation type and class rate limiting.*** *Experiments show that* PADLL *can enforce different rate limits over different POSIX operations and granularities, including* `open`*,* `getattr`*,* `read`*,* `write`*,* `metadata`*, and* `data`*. Metadata operations are presented in IOPS (kops/s), while data operations in I/O bandwidth (namely, GiB/s for* `read/write` *and MiB/s for* `data` *experiments).*

V. EVALUATION

Our evaluation seeks to answer the following questions:

- Can PADLL control I/O workflows at different granularities?
- Can PADLL enforce QoS policies over concurrent jobs?
- What is the performance of PADLL control and data planes?

Experimental testbed. Experiments were conducted on three hardware configurations. **Configuration *A*** respects to compute nodes of the ABCI supercomputer [1], equipped with two 20-core Intel Xeon processors, 384 GiB of RAM, and an InfiniBand EDR network card, running CentOS 7.5. The PFS is a *dedicated* DDN ExaScaler Lustre composed of 2 MDSs in a hot-standby configuration, backed by 2 MDTs, and 24 OSTs that provide 359 TiB of storage capacity. **Configuration *B*** respects to compute nodes of the Frontera supercomputer [56], equipped with two 16-core Intel Xeon processors, 128 GiB of RAM, four NVIDIA Quadro RTX 5000 GPUs, and a Mellanox InfiniBand FDR network card, running CentOS 7.9. The production PFS is a Lustre file system composed of 4 MDSs, each with a single MDT, and 32 OST nodes with 22 PiB of storage capacity. **Configuration *C*** respects to compute nodes of the Frontera supercomputer, equipped with two 28-core Intel Xeon processors, 192 GiB of RAM, and a Mellanox InfiniBand HDR-100 network card, running CentOS 7.9. The production PFS is the same as hardware configuration ***B***.

Benchmarks and workloads. We conducted experiments using both data and metadata workloads. For data workloads we used IOR [54], which performs a `read`/`write` workload that sequentially reads/writes from/to a single file with 875 GiB using POSIX-compliant system calls. We also used TensorFlow [9], an AI framework used for DL training that is predominately executed in today's HPC clusters [47], [62].

To generate realistic metadata workloads, we implemented a *trace replayer* that submits metadata operations with an identical request distribution as the one observed from the logs collected at PFS_1 (§II-A). The *replayer* is multi-threaded, and each thread submits operations at a rate that follows the same performance curve as original logs. The rate of each operation was scaled-down to half, due to the difference in size between PFS_1 and configuration ***A***'s PFS. The execution period was also accelerated, and each second of the *replayer* corresponds to a minute's worth of operations in the original trace.

Methodology. For all experiments, the *global controller* runs at a dedicated compute node, while *local controllers* execute co-located with each job instance and respective data plane stages. Metadata experiments were conducted with hardware configuration ***A*** (§V-A–§V-C), and data experiments with configurations ***B*** (§V-A, §V-B) and ***C*** (§V-D). All experiments were conducted using the shared PFS. Across all testing scenarios, two setups were used: ***baseline***, which represents the benchmark without using PADLL, and ***padll***, where POSIX operations submitted by the benchmark are intercepted by PADLL and throttled at a given rate.

A. Per-operation type rate limiting

We first demonstrate how PADLL enables system administrators to control the rate of specific operations. Fig. 6 depicts the results of all setups under different operation types.

Workload configuration. Both *trace replayer* and *IOR* were configured to submit a single operation type – namely, `open` and `getattr` (*left*), and `read` and `write` (*right*).

PADLL configuration. For all experiments, PADLL was configured to throttle operations with a static rate, whose value changes every N minutes (6 minutes for metadata and 1 minute for data operations) upon instruction of the system administrator (*i.e.,* rule defined on the control plane).

Results. At all times, ***padll*** is able to control the rate of all operations, never exceeding the configured limits. Over several periods, ***padll*** follows the same performance curve as ***baseline***, as observed in `open` between 23 and 29 minutes (*i.e.,* periods where the black line is not flat). This is because, the limit set by the system administrator (for that interval) is higher than the operations submitted by the *replayer*. Analogously, we also observe periods where ***padll*** achieves higher throughput than ***baseline***, as observed in `getattr` between 8 and 12 minutes. This happens when operations are being aggressively rate limited (*i.e.,* the original rate is significantly higher than the defined limit), creating a backlog of operations to be executed later when there are enough available resources.

Table I: *Per-Job QoS control testing scenarios.*

	Test. scenario #1	Test. scenario #2	Test. scenario #3
Job$_1$	25% – 15 kops/s	15% – 15 kops/s	15% – 40 kops/s
Job$_2$	25% – 25 kops/s	20% – 25 kops/s	20% – 25 kops/s
Job$_3$	25% – 30 kops/s	20% – 30 kops/s	20% – 30 kops/s
Job$_4$	25% – 40 kops/s	45% – 40 kops/s	45% – 15 kops/s

We observe similar results for data-oriented operations, namely `read` and `write`. However, contrarily to configuration ***A*** where requests are submitted to a dedicated PFS, we observe more variability, as experiments were conducted over a file system shared with multiple concurrent jobs.

B. Per-operation class rate limiting

We now demonstrate how PADLL controls the workflows of a given operation class, namely `metadata` and `data`.

Workload configuration. We followed a similar methodology as in §V-A. For `metadata`, operations are submitted from a multi-node job made of 4 *trace replayer* instances, each running on a dedicated compute node. Each instance spawns 8 threads that submit different types of metadata operations (same as those discussed in §II-A). To demonstrate PADLL's general applicability, we conducted the `data` experiments using a distributed TensorFlow job over 4 compute nodes, running version 2.3.2 with the LeNet training model [35] and configured with a batch size of 128 TFRecords. We used the ImageNet dataset (≈150 GiB) [51], hosted at the shared PFS.

PADLL configuration. Similar to §V-A, PADLL was configured to throttle operations with a static rate, whose value changes every 6 minutes for metadata workloads and 5 minutes for data workloads. Fig. 6 (*bottom*) depicts the obtained results. The throughput corresponds to the accumulated rate of all *replayer* instances (in IOPS) or TensorFlow workers (in I/O bandwidth).

Results. In both `data` and `metadata` experiments, ***padll*** effectively controls the rate of all workflows throughout the overall observation. In several periods, ***padll*** matches or achieves higher throughput performance than ***baseline***; we draw similar observations as in §V-A. Complementary, these experiments also demonstrate that PADLL can achieve QoS limits even for distributed jobs.

C. Per-job QoS control

We now demonstrate how PADLL achieves per-job QoS control in HPC storage systems by orchestrating the metadata workflows of all active jobs. Under this scenario, metadata operations are treated as a finite and shared I/O resource, and for the PFS to provide sustained I/O performance, jobs need to meet specific metadata service level objectives (SLOs).

Workload configuration. At all times, there are at most four jobs in the system, each running 4 *trace replayer* instances in dedicated compute nodes (16 in total) and submitting metadata operations. Jobs are incrementally added to the system every 3 minutes. We consider that the system administrator defines a maximum rate of metadata operations (Max_R) that can be submitted to the targeted PFS, being set at 110 kops/s (red dashed lines). The trace used in the experiments corresponds to the metadata operations of all MDT servers of PFS$_1$.

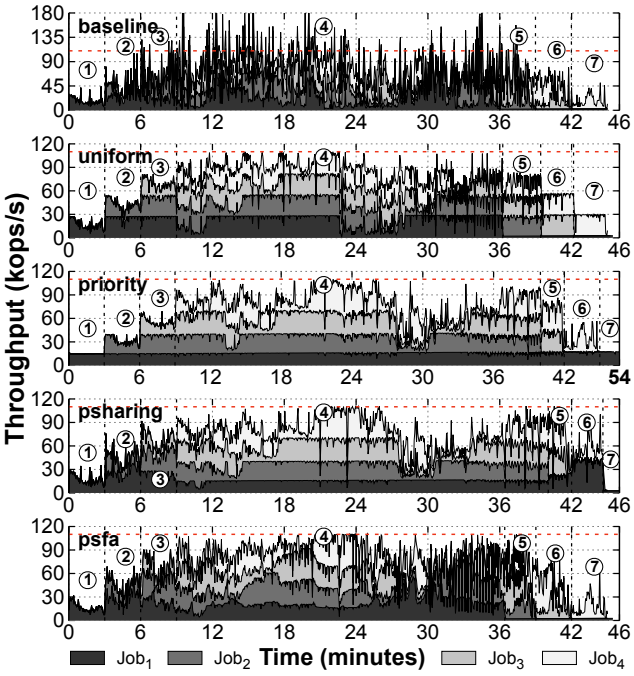

Fig. 7: ***Per-job metadata control*** *over Baseline, Uniform, Priority, PSharing, and PSFA setups under* ***testing scenario #1****.*

Testing scenarios. To provide a comprehensive evaluation testbed, we consider three testing scenarios with varying load proportions and rate limits. Table I depicts the load proportion and the metadata rate limit for each combination of testing scenario and job. *Testing scenario #1:* jobs follow the same workload but are assigned with different priorities. *Testing scenario #2:* jobs have different load proportions and rate limits are assigned proportionally to each job's load (*i.e.,* jobs with lower load are assigned with lower priority). *Testing scenario #3:* jobs follow the same load proportions as *testing scenario #2*, but the rate limits of jobs 1 and 4 are switched.

Setups. Experiments were conducted under five setups. *Baseline* represents the current setup supported at most supercomputers, where jobs execute without any throttling. The remainder setups are rate limited with PADLL, with a maximum combined rate of Max_R, and respect to the control algorithms discussed in §IV. In *Uniform*, each job is rate limited to 27.5 kops/s, while *Priority*, *Proportional sharing (PSharing)*, and *PSFA*, jobs are assigned different rates, as depicted in Table I. For *PSharing*, these limits represent the per-job maximum rate when all jobs are active, while for *PSFA* represent the per-job maximum rate when all jobs are active and each job's *usage* is higher than its *demand*.

Test. scenario #1. Fig. 7 depicts, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #1*. Experiments include seven phases (①–⑦), each marking when a given job enters or leaves the system.

Baseline. Experiments were executed over 45 minutes. Each

job executes over 36 minutes and leaves the system in the same order as it entered. Throughout the entire execution, we observe that the workload is extremely volatile and bursty, with peaks that reach 300 kops/s and several periods where the file system continuously serves requests over Max_R.

Uniform. Experiments were executed over 45 minutes. Whenever a new job is added, it is provisioned with its assigned rate (27.5 kops/s). PADLL ensures, to all jobs, sustained metadata throughput and eliminates existing burstiness. However, while this setup is useful to equally distribute metadata rate among jobs, it does not allow them to execute with different priorities.

Priority. Experiments were executed over 54 minutes. Similarly to *Uniform*, PADLL ensures that all jobs are provisioned with their assigned rate. However, when a job is set with low priority, its execution may take longer than its corresponding unthrottled version since metadata operations are rate limited more aggressively. We observe this in Job_1, where its execution takes 18 minutes longer than in previous setups, and occurs because the algorithm does not leverage from leftover metadata rate (*i.e.,* ①-③ and ⑤-⑦), as discussed in §IV-A.

PSharing. Experiments were executed over 45 minutes. Whenever a new job enters (①-③) or leaves the system (⑤-⑦), it is assigned with its proportional share. When all jobs are running (④), they are assigned with their demanded rate. Compared to *Priority*, *PSharing* distributes leftover metadata rate, which enables improving Job_1's performance by 9 minutes. However, since the workload is volatile and bursty, we observe several periods with *false resource allocation* (§IV-B), being particularly noticeable in the 12–20 minutes and 25–37 minutes intervals. During these periods one or more jobs are over-provisioned, and the exceeding resources could be used to improve the performance of the remainder jobs (*e.g., Job_1* still took 9 minutes longer to execute than in *Baseline*).

PSFA. Experiments were executed over 45 minutes. All jobs complete their execution in the same time as their corresponding unthrottled versions. Throughout the overall execution, the *PSFA* algorithm continuously adjusts the limit of each job based on the actual rate it is using, preventing over-provisioning. Specifically, in the 12–20 minutes period, *PSFA* assigns unused metadata rate to Job_1, Job_2, and Job_3, temporarily having more rate than their *demand*. The same is observed for Job_1 and Job_2 in the 25–37 minutes period.

Test. scenario #2. Fig. 8 depicts, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #2*.

Baseline. Experiments executed over 45 minutes. As the overall metadata load is the same across all testing scenarios, we observe similar volatility and burstiness as in *testing scenario #1*. The key difference is that now Job_4 generates a major part of the metadata load, while Job_1 has significantly lower load.

Uniform. Experiments executed over 62 minutes. Similarly to *testing scenario #1*, whenever a new job is added it is provisioned with its static rate (27.5 kops/s). However, as Job_4 now generates 45% of the overall metadata load, PADLL aggressively rate limits it, resulting in a longer execution period (*i.e.,* 17 minutes longer than *Baseline*). On the other hand,

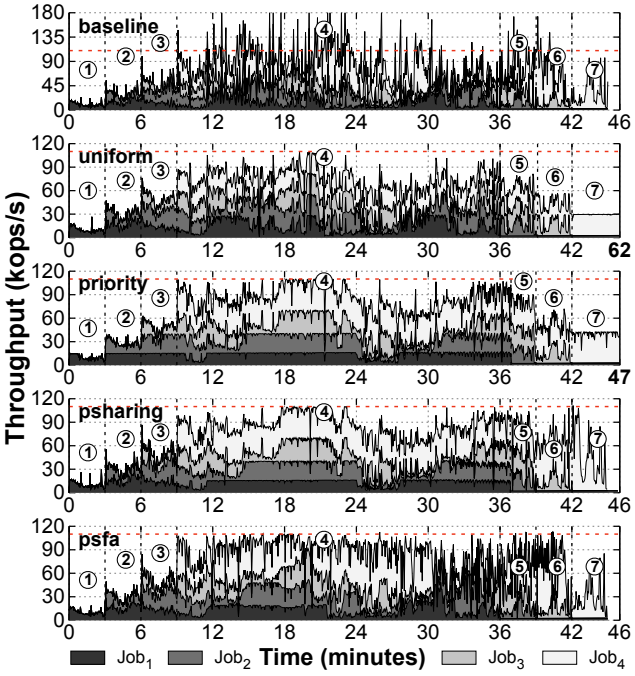

Fig. 8: ***Per-job metadata control*** *over Baseline, Uniform, Priority, PSharing, and PSFA setups under* ***testing scenario #2****.*

Job_1 experiences over-provisioning for most of its execution, given that it only generates 15% of the overall workload.

Priority. Experiments executed over 47 minutes. Due to the decreased metadata load, Job_1 finishes approximately at the same time as in *Baseline*. On the other hand, Job_4 takes 2 minutes longer to complete. Specifically, even though Job_4 is set with a larger resource share (40 kops/s), due to its large metadata load, PADLL aggressively rate limits it throughout the overall execution period, resulting in a large backlog of metadata operations to be performed, as observed in ⑦.

PSharing. Experiments executed over 45 minutes. Contrarily to static setups, since *PSharing* distributes leftover metadata rate whenever it is available, jobs complete their execution in 36 minutes (as in *Baseline*). However, similarly to the observations made in *testing scenario #1*, this setup experiences periods with *false resource allocation*, being especially noticeable in the 12–18, 23–35, and 36–42 minutes intervals.

PSFA. Experiments executed over 45 minutes. *PSFA* maximizes the use of metadata resources by reallocating unused rate from over-provisioned jobs. For instance, during the 23–30 minutes period, Job_4 improves its performance by leveraging from unused metadata rate of the other jobs. Interestingly, during the 31–36 interval, *PSFA* demonstrates lower usage of resources compared to *Priority* and *PSharing*. This is because up to the 31-min mark, *PSFA* allocates enough rate to all jobs (either from leftovers or over-provisioning) that allowed them to conduct any accumulated backlog of metadata operations.

Test. scenario #3. Fig. 9 depicts, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #3*. Experiments conducted for *Baseline* and *Uniform* setups are

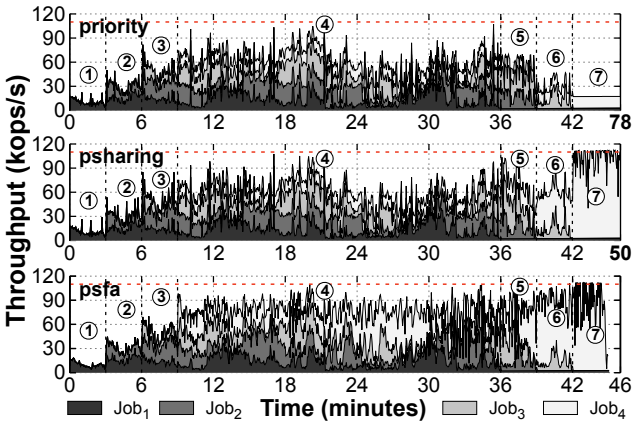

Fig. 9: ***Per-job metadata control*** *over Priority, PSharing, and PSFA setups under* ***testing scenario #3****.*

the same as in *testing scenario #2*, as both metadata load and rate limits remain unchanged. We draw identical observations.

Priority. Experiments executed over 78 minutes. As a result of being assigned with the lowest priority while having the highest load, Job_4 takes 33 minutes longer to complete its execution. Noticeably, during ⑦, PADLL aggressively rate limits operations at a constant rate of 15 kops/s, not leveraging from the remainder 95 kops/s available in the system.

PSharing. Experiments executed over 50 minutes. Since Job_4 is the last to enter the system (④), it only leverages from leftover rate when the other jobs complete their execution (⑤–⑦). Thus, due to accumulated backlog (⑦), the job submits metadata operations at a constant rate of Max_R, being 28 minutes faster than *Priority* but still requiring an additional 5 minutes to complete when compared to *Baseline*. On the other hand, Job_1 is over-provisioned for most of its execution.

PSFA. Experiments executed over 45 minutes. Since *PSFA* prevents *false resource sharing*, during the 12–42 minutes interval, a large share of unused metadata rate is assigned to Job_4, which enables executing all accumulated backlog just under the 45-min mark (⑦). Note that *PSFA* is able to reassign unused resources without compromising other policies; for instance, Job_1 demonstrates the same performance curve as in setups with more strict policies, namely *Priority* and *PSharing*.

Summary. Results demonstrate that PADLL enforces different QoS policies over distributed metadata-aggressive jobs without exceeding Max_R. *Uniform* is suited for scenarios where jobs have similar I/O load (#1), while *Priority* is appropriate for assigning larger metadata shares to jobs with higher load (#2). *PSharing* prevents under-provisioning, improving job execution time and overall resource usage by allocating leftover metadata rate (#1, #2). *PSFA* maximizes the use of metadata rate, accelerating the performance of resource-hungry jobs without degrading over-provisioned ones (#1 – #3).

D. PADLL *performance, resource usage, and overhead*

Finally, we demonstrate the performance of PADLL control and data planes, their impact on computational resources, and evaluate the overhead imposed over applications.

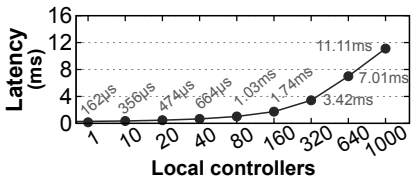

Fig. 10: ***Average latency of control*** *cycles (in global controller) when the number of local controllers increases.*

Control plane. We conducted a set of experiments where the *global controller* performs 250k iterations of *PSFA* control algorithm's main phases, namely *collect*, *compute*, and *enforce* (§III-B). In particular, the *collect* and *enforce* phases involve traversing the network multiples times per iteration. The number of *local controllers* submitting metrics and receiving enforcement rules is increased from 1 to 1,000.

Fig. 10 depicts the obtained results, reporting the average latency of each control iteration. Results demonstrate that latency increases with the number of connected controllers. Up to 40 *local controllers*, PADLL performs at μs-scale ranging between $162\mu s$ and $664\mu s$. After that mark, the average latency per control cycle ranges between 1.03ms and 11.11ms.

These results show that the *global controller* can orchestrate the overall system at *ms*-scale, which fits the requirements of production workloads where these control cycles are tuned with larger time frames, as otherwise stages would be adjusted for every minimal workload change in the system. For instance, the experiments discussed in §V-C were conducted with control cycles of 1-second intervals. When reaching 1,000 nodes, we start to notice a higher utilization of network resources at the *global controller*, and an increase in response time. This highlights the need to further research the scalability of this component as future work.

Data plane. To evaluate the maximum performance achievable with a single PADLL stage, we implemented a benchmark that submits POSIX requests in a closed loop, under a varying number of threads (1–256). Each thread submits 100M requests. The stage is configured with the same number of *channels* as client threads, and token-buckets are set with a rate large enough to not perform any throttling. Requests are intercepted by the stage and follow the same path as discussed in §III-A. Results report that a single stage can service requests at a high rate, ranging between 1.28 Mops/s and 3.20 Mops/s.

Overhead. To evaluate the overhead imposed by PADLL, we conducted experiments with a ***passthrough*** setup, which respects to a scenario where POSIX operations submitted by the benchmark are handled by PADLL but are not rate limited. We repeated §V-A and §V-B workloads. When compared to ***baseline***, the overhead is negligible, never degrading performance more than 0.9% across all experiments.

Resource usage. We now discuss the resource usage impact imposed by PADLL. In terms of *network bandwidth*, the payload of the messages exchanged in each control cycle between the *global* and each *local controller* is approximately 200 bytes, which is negligible compared to the capacity of modern

network devices. As for *network latency*, as referred in the *control plane performance* discussion (Fig. 10), PADLL is able to manage multiple *local controllers* (and corresponding data plane stages) at μs-scale. Regarding *CPU* and *memory* usage, the components that run co-located with the targeted jobs (*i.e.,* data plane stages and local controller) impose minimal overhead, only increasing CPU by at most 5% and memory by $\approx$100 MiB. As such, PADLL imposes minimal overhead to all targeted jobs, being suited for the computational requirements of modern I/O infrastructures.

VI. Related Work

HPC storage QoS. Many works, such as GIFT [48], CALCioM [22], IOrchestrator [60], UShape [61], and *Gainaru et al*. [25], are designed to mitigate I/O contention in HPC storage stacks but ignore the impact that metadata workflows have over the overall system performance. PADLL is able to control the rate of both data and metadata workflows. Other systems are directly implemented within core layers of the HPC I/O stack, including the PFS [28], [31], [50], [60], [61], scheduler [25], and I/O libraries [14], [22]. These solutions are intrusive and offer limited maintainability and portability. PADLL actuates at the compute node level and does not require any changes to core layers of the HPC I/O stack.

Similarly to PADLL, OOOPS transparently intercepts and rate limits POSIX requests at compute nodes [29]. However, it does not provide global visibility, being only capable of enforcing static policies that remain unchanged throughout the job execution. On the other hand, PADLL can enforce dynamic and cluster-wide QoS policies that require global visibility.

SDS systems. PADLL builds on a large body of work on SDS [42]. Systems like IOFlow, sRoute, and PSLO, actuate at the virtualization and block device layers, only controlling the rate of `read` and `write` requests [36], [45], [57], [59]. Others, like Retro and Crystal, enforce resource management policies over distributed storage systems, but are directly implemented within the storage system itself, offering limited maintainability and portability [27], [40]. SIREN enforces bandwidth policies over OrangeFS [13], [31]. PADLL is a bare-metal solution that actuates at the compute node level and transparently intercepts and enforces POSIX requests, both data and metadata, before being submitted to the PFS. This makes it applicable over different applications and compatible with POSIX-compliant storage systems.

I/O optimizations. Many works propose I/O optimizations to reduce the amount of operations submitted to the PFS by resorting to storage tiering and node-local storage [19], [23], [32], [46], remote burst buffers [23], [33], [37], data reduction techniques [44], and optimized data formats [24], [39], or improve the metadata management of large-scale file systems [17], [49], [58]. While these can reduce the I/O pressure imposed over the PFS, they can still expose it to burstiness and unfairness, since I/O workflows are not rate limited. On the other hand, PADLL rate limits all workflows destined towards the PFS. Further, contrary to PADLL, several of these works are also intrusive to core layers of the HPC I/O stacks [17], [33], [37], [49], [58]. While complementary to our work, these can be combined with PADLL to further improve the control of I/O workflows in HPC clusters.

VII. Conclusion

We have presented PADLL, a storage middleware that enables QoS control of data and metadata workflows in HPC storage systems. PADLL does not require changing any core layer of the HPC stack, and enforces storage policies with fine-granularity and global system visibility. Results demonstrate that PADLL can enforce complex storage policies over concurrent metadata-aggressive jobs in holistic fashion, achieving I/O fairness, prioritization, and performance isolation.

With PADLL, we aim at supporting system administrators, researchers, and practitioners to effectively provide QoS control over large-scale HPC systems, and assist bridging the convergence of cloud and HPC infrastructures.

Acknowledgements

We thank AIST for providing access to computational resources of ABCI. We thank Cláudia Brito and Tânia Esteves for reviewing initial versions of this work. This work is financed by: the ERDF - European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme under the Portugal 2020 Partnership Agreement, and by National Funds through the FCT - Portuguese Foundation for Science and Technology, I.P. on the scope of the UT Austin Portugal Program within project BigHPC, with reference POCI-01-0247-FEDER-045924 (Mariana Miranda); through PhD Fellowships SFRH/BD/146059/2019 and PD/BD/151403/2021; and the UT Austin-Portugal Program, a collaboration between the Portuguese Foundation of Science and Technology and the University of Texas at Austin, award UTA18-001217. The first two authors contributed equally to this work.

References

[1] AI Bridging Cloud Infrastructure. https://abci.ai/.
[2] Aurora Supercomputer. https://www.alcf.anl.gov/aurora.
[3] BLKIO: Cgroup's Block I/O Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.
[4] DDNStorage/LustrePerfMon: Lustre Monitoring System. https://github.com/DDNStorage/LustrePerfMon.
[5] Frontier Supercomputer. https://www.olcf.ornl.gov/frontier/.
[6] gRPC: A high performance, open source universal RPC framework. https://grpc.io/.
[7] Lustre Metadata Service (MDS). https://wiki.lustre.org/Lustre_Metadata_Service_(MDS).
[8] Lustre MDC: mdc_reint.c. https://github.com/lustre/lustre-release/blob/master/lustre/mdc/mdc_reint.c, 2022.
[9] 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 Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.
[10] Austin Appleby. appleby/smhasher: SMHasher test suite for MurmurHash family of hash functions. https://github.com/aappleby/smhasher, 2010.

[11] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer Science & Business Media, 2001.

[12] Peter Braam. The Lustre Storage Architecture, 2019.

[13] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase & Conference*. USENIX Association, 2000.

[14] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E. Singh, and Nicolas Vidal. Mapping and Scheduling HPC Applications for Optimizing I/O. In *34th ACM International Conference on Supercomputing*. ACM, 2020.

[15] Steven WD Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing Deep-Learning I/O workloads in TensorFlow. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 54–63. IEEE, 2018.

[16] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In *48th International Conference on Parallel Processing*. ACM, 2019.

[17] Dong Dai, Yong Chen, Philip Carns, John Jenkins, Wei Zhang, and Robert Ross. GraphMeta: A Graph-Based Engine for Managing Large-Scale HPC Rich Metadata. In *2016 IEEE International Conference on Cluster Computing*, pages 298–307. IEEE, 2016.

[18] Christopher S. Daley, Devarshi Ghoshal, Glenn K. Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas J. Wright. Performance Characterization of Scientific Workflows for the Optimal Use of Burst Buffers. *Future Generation Computer Systems*, 110:468–480, 2017.

[19] Marco Dantas, Diogo Leitão, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and João Paulo. Accelerating Deep Learning Training Through Transparent Storage Tiering. In *2022 IEEE/ACM 22nd International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2022.

[20] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. FMA: A Dataset For Music Analysis, 2016.

[21] Hariharan Devarajan, Huihuo Zheng, Anthony Kougkas, Xian-He Sun, and Venkatram Vishwanath. DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*, pages 81–91. IEEE, 2021.

[22] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164. IEEE, 2014.

[23] Hatem Elshazly, Jorge Ejarque, and Rosa M. Badia. Storage-Heterogeneity Aware Task-based Programming Models to Optimize I/O Intensive Applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3589–3599, 2022.

[24] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An Overview of the HDF5 Technology Suite and Its Applications. In *EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[25] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC Applications Under Congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2011.

[27] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, et al. Crystal: Software-Defined Storage for Multi-tenant Object Stores. In *15th USENIX Conference on File and Storage Technologies*, pages 243–256. USENIX Association, 2017.

[28] Yusheng Hua, Xuanhua Shi, Hai Jin, Wei Liu, Yan Jiang, Yong Chen, and Ligang He. Software-defined QoS for I/O in exascale computing. *CCF Transactions on High Performance Computing*, 1(1):49–59, 2019.

[29] Lei Huang and Si Liu. OOOPS: An Innovative Tool for IO Workload Management on Supercomputers. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems*, pages 486–493. IEEE, 2020.

[30] Lei Huang, Yinzhi Wang, Chun-Yaung Lu, and Si Liu. Best Practice of IO Workload Management in Containerized Environments on Supercomputers. In *Practice and Experience in Advanced Research Computing*. ACM, 2021.

[31] Suman Karki, Bao Nguyen, and Xuechen Zhang. QoS Support for Scientific Workflows using Software-Defined Storage Resource Enclaves. In *2018 IEEE International Parallel and Distributed Processing Symposium*, pages 95–104. IEEE, 2018.

[32] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System. In *27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2018.

[33] Anthony Kougkas, Hariharan Devarajan, Xian-He Sun, and Jay Lofstead. Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-volatile Burst Buffers. In *2018 IEEE International Conference on Cluster Computing*, pages 290–301. IEEE, 2018.

[34] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The Open Images Dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[36] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *11th European Conference on Computer Systems*, pages 28:1–28:14. ACM, 2016.

[37] Weihao Liang, Yong Chen, Jialin Liu, and Hong An. CARS: A contention-aware scheduler for efficient resource management of HPC storage systems. *Parallel Computing*, 87:25–34, 2019.

[38] Si Liu, Lei Huang, Hang Liu, Amit Ruhela, Virginia Trueheart, Susan Lindsey, and Quan Yuan. Practice Guideline for Heavy I/O Workloads with Lustre File Systems on TACC Supercomputers. In *Practice and Experience in Advanced Research Computing*. ACM, 2021.

[39] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *6th International Workshop on Challenges of Large Applications in Distributed Environments*, pages 15–24, 2008.

[40] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 589–603. USENIX Association, 2015.

[41] Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen L. Harrell, Todd Evans, José Pereira, and João Paulo. Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control (Artifact Description/Evaluation). https://doi.org/10.5281/zenodo.7627949, February 2023.

[42] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. A Survey and Classification of Software-Defined Storage Systems. *ACM Computing Surveys*, 53(3), May 2020.

[43] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. PAIO: General, Portable I/O Optimizations With Minor Application Modifcations. In *20th USENIX Conference on File and Storage Technologies*, pages 413–428. USENIX Association, 2022.

[44] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[45] Michael Mesnier, Feng Chen, Tian Luo, and Jason Akers. Differentiated Storage Services. In *23rd ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[46] Adam Moody, Danielle Sikich, Ned Bass, Michael J. Brim, Cameron Stanavige, Hyogi Sim, Joseph Moore, Tony Hutter, Swen Boehm, Kathryn Mohror, Dmitry Ivanov, Teng Wang, Craig P. Steffen, and US-DOE National Nuclear Security Administration. UnifyFS: A Distributed Burst Buffer File System, 2017.

[47] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. 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*. ACM, 2019.

[48] Tirthak Patel, Rohan Garg, and Devesh Tiwari. GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems. In *18th USENIX Conference*

on File and Storage Technologies, pages 103–119. USENIX Association, 2020.

[49] Arnab K. Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R. Butt. Efficient Metadata Indexing for HPC Storage Systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, pages 162–171. IEEE, 2020.

[50] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017.

[51] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.

[52] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. Co-Design for A64FX Manycore Processor and "Fugaku". In *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[53] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003, pages 380–386, 2003.

[54] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008.

[55] David Shue, Michael Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 349–362. USENIX Association, 2012.

[56] Dan Stanzione, John West, R. Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. Frontera: The Evolution of Leadership Computing at the National Science Foundation. In *Practice and Experience in Advanced Research Computing*, pages 106–111. ACM, 2020.

[57] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sRoute: Treating the Storage Stack Like a Network. In *14th USENIX Conference on File and Storage Technologies*, pages 197–212. USENIX Association, 2016.

[58] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. SoMeta: Scalable Object-Centric Metadata Management for High Performance Computing. In *2017 IEEE International Conference on Cluster Computing*, pages 359–369. IEEE, 2017.

[59] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, et al. IOFlow: A Software-Defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[60] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010.

[61] Xuechen Zhang, Kei Davis, and Song Jiang. QoS Support for End Users of I/O-Intensive Applications Using Shared Storage Systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011.

[62] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 145–156. IEEE, 2018.

APPENDIX
ARTIFACT DESCRIPTION/EVALUATION

The paper proposes a new storage middleware that enables system administrators to proactively and holistically control the rate of data and metadata workflows to achieve QoS in HPC storage systems. It is organized in two main components: the *data plane* (PADLL) and the *control plane* (CHEFERD).

PADLL. The data plane, named PADLL, is a multi-stage component that provides the building blocks for differentiating and rate limiting POSIX requests that are destined towards a given file system (*e.g.,* Lustre, `ext4`). Stages actuate at the compute node level, and use `LD_PRELOAD` to transparently intercept POSIX requests (*i.e.,* `libc.so` calls) from a given application and handling them before being submitted to the file system. PADLL is written in C++17 and is publicly available at the `dsrhaslab/padll`[1] GitHub repository under a *BSD 3-Clause* license. The logic for differentiating and rate limiting requests was built using the PAIO data plane framework [43].[2] Communication between PADLL stages and the control plane (*local controllers*) is established using *UNIX Domain Sockets*.

CHEFERD. The control plane, named CHEFERD, is a logically centralized component with system-wide visibility that defines how all I/O requests should be handled. It follows a hierarchical distribution, made of *global* and *local controllers*. Local controllers are placed at compute nodes and manage all PADLL stages that are executing there. The global controller executes at a dedicated compute node, and communicates with all local controllers with RPC calls through the gRPC framework. Moreover, CHEFERD is written in C++17 and is publicly available at the `dsrhaslab/cheferd`[3] GitHub repository under a *BSD 3-Clause* license.

Artifacts. Both PADLL and CHEFERD, alongside a *trace replayer* and several scripts to reproduce the experiments of the paper are publicly available at Zenodo's artifact repository (**https://zenodo.org/record/7627949/**) [41].

- PADLL and CHEFERD have the same version as their corresponding publicly available GitHub repositories.
- The *trace replayer* (`mdreplayer`) is used to generate realistic metadata workloads. It is multi-threaded, and each thread submits operations at a variable rate that is defined by the trace distribution.
- The *scripts* folder includes bash scripts for installing and deploying all systems, as well as scripts to (1) reproduce experiments in the paper (§V-A–§V-C) at the Frontera supercomputer, or (2) test a subset of them in a commodity setup (namely, §V-A `getattr` and §V-B `metadata`).

Further, all repositories include `README` files that describe how to install, configure, and test each system.

REQUIREMENTS AND DEPENDENCIES

System requirements. PADLL and CHEFERD were built and tested using `g++9.3.0` and `cmake-3.16`, and were successfully deployed in Ubuntu Server 20.04 LTS, CentOS 7.5, and CentOS 7.9 Linux distributions. Operating system wise, the main requirements of these artifacts lie on the use of `LD_PRELOAD` (PADLL) and *UNIX Domain Sockets* (PADLL and CHEFERD).

[1] PADLL: https://github.com/dsrhaslab/padll
[2] PAIO: https://github.com/dsrhaslab/paio
[3] CHEFERD: https://github.com/dsrhaslab/cheferd

PADLL dependencies. PADLL was built using the PAIO v1.0.0 and `spdlog v1.8.1`[4] libraries. The former needs to be manually installed (installation steps are detailed in the PADLL repository), and the latter is installed at compile time (defined with a CMake rule). Moreover, PADLL also uses `xoshiro-cpp`[5], `tabulate`[6], and `better-enums`[7] third-party libraries, which are embedded as single-header files.

CHEFERD dependencies. CHEFERD was built using the `spdlog v1.8.1`, `grpc v1.37.0`[8], `gflags v2.2.2`[9], `asio v1.18.0`[10], and `yaml-cpp v0.6.3`[11] libraries. All dependencies are installed at compile time, and all rules are defined at CHEFERD's `CMakeLists.txt` file.

COMMODITY HARDWARE EXPERIMENTS

To ease the reproducibility of the paper experiments, we created a set of scripts to test the artifacts under a commodity hardware testbed. For this scenario, we consider the *per-operation type rate limiting* (`getattr`) and *per-operation class rate limiting* (`metadata`) use cases, which are discussed in §V-A and §V-B of the paper, respectively.

Experimental testbed. Experiments were conducted in a server equipped with a single 6-core Intel Core i5-9500 processor, 16 GiB of memory, and a Samsung NVMe SSD 970 EVO Plus 250 GiB disk. Software-wise it used Ubuntu Server 20.04 LTS, with kernel 5.4.0 and an `ext4` file system.

Methodology. All systems were executed under the same server. Experiments were conducted under the following steps:

1) Execute the *global controller* using the script `launch_core_controller.sh`. The *global controller* will run at a user-level process, exposing a communication endpoint (gRPC server) for *local controllers* to connect.
2) Execute the *local controller* using the script `launch_local_controller.sh`. The *local controller* will connect to the already running *global controller*, while also creating a *UNIX Domain Socket* for data plane stages to connect.
3) Finally, execute the application (*trace replayer*) with the script `launch_padll_application.sh`. The application will be spawned with a `LD_PRELOAD` hook for the data plane stage (*i.e.,* replace `libc.so` calls with those exposed by PADLL).

For the experiments described at §V-A, the *trace replayer* replays the `getattr_log.txt` file, while for §V-B's experiments, it simultaneously replays `getattr_log.txt`, `rename_log.txt`, and `open_log.txt` log files.[12] All traces are synthetic (*i.e.,* not the original PFS_1 logs discussed in §II), but follow a variable metadata rate distribution. Both experiments are executed over 1 minute, and PADLL adjusts the rate at which POSIX operations are submitted to the file system every 20 seconds.

Testing scripts. The scripts discussed in this section can be found at the `scripts/commodity_scripts` folder of the Zenodo public artifacts: `/per_type_getattr` for the §V-A and `/per_class_metadata` for the §V-B experiments.

FRONTERA EXPERIMENTS

All experiments presented in the paper can be reproduced using the scripts available at `scripts/frontera_scripts` folder of the Zenodo public artifacts, which are ready to be executed at TACC's Frontera supercomputer.

Experimental testbed. Experiments were conducted over two hardware configurations. For the §V-A, §V-B `metadata`, and §V-C experiments, we used compute nodes of Frontera's `normal` job queue, which are equipped with two 28-core Intel Xeon processors, 192 GiB of RAM, and a Mellanox InfiniBand HDR-100 network card, running CentOS 7.9. The production PFS is a Lustre file system composed of 4 MDSs, each with a single MDT, and 32 OST nodes with 22 PiB of storage capacity. The §V-B `data` (TensorFlow) were executed at computes nodes of Frontera's `rtx` job queue, which are equipped with two 16-core Intel Xeon processors, 128 GiB of RAM, four NVIDIA Quadro RTX 5000 GPUs, and a Mellanox InfiniBand FDR network card, running CentOS 7.9.

Workloads. Metadata workloads were conducted using the *trace replayer*. All traces are synthetic (*i.e.,* not the original PFS_1 logs), but follow a variable metadata rate distribution.

Data experiments were twofold: (1) for §V-A `read` and `write` testing scenarios, we used IOR (commit #1076c89) sequential read/write workloads (IOR scripts can be found at `per_type/padll_ior_job.sh`); and (2) for the §V-B `data` experiment, we used TensorFlow v2.3.2 with the LeNet training model, configured with a batch size of 128 TFRecords, and ImageNet dataset (TensorFlow scripts can be found at `per_class/data/test_tensorflow.sh`).[13]

Methodology. For all experiments, the *global controller* runs at a dedicated compute node (`core_job.sh`). For §V-A and §V-B, we use an additional compute node to host the *local controller* (`local_job.sh`), the application (*trace replayer*), and the data plane stage.

For §V-C experiments, we use four additional compute nodes, each hosting a *local controller*, the application (*trace replayer*), and the data plane stage.

Testing scripts. For each evaluation scenario, we used the following scripts:

- **§V-A:** `frontera_scripts/per_type`
  - **open:** `launch_open_test.sh`
  - **getattr:** `launch_getattr_test.sh`
  - **read:** `launch_read_test.sh`
  - **write:** `launch_write_test.sh`
- **§V-B:** `frontera_scripts/per_class`
  - **metadata:** `metadata/launch_test.sh`

[4] spdlog: https://github.com/gabime/spdlog/tree/v1.8.1
[5] Xoshiro-cpp: https://github.com/Reputeless/Xoshiro-cpp
[6] tabulate: https://github.com/p-ranav/tabulate
[7] better-enums: https://github.com/aantron/better-enums
[8] gRPC: https://github.com/grpc/grpc/tree/v1.37.0
[9] gflags: https://github.com/gflags/gflags/tree/v2.2.2
[10] asio: https://github.com/chriskohlhoff/asio/tree/asio-1-18-0
[11] yaml-cpp: https://github.com/jbeder/yaml-cpp/tree/yaml-cpp-0.6.3
[12] All logs can be found at `mdreplayer/logs/`.
[13] ImageNet: https://www.image-net.org/challenges/LSVRC/2012/.

  - **data:** `data/launch_test.sh`
- **§V-C:** For the §V-C experiments, we only provide scripts for the testing scenario #1 (the remainder are just a variation of the load and job priorities). Further, we provide five subdirectories, each with the corresponding scripts for the different algorithms discussed in §IV, including: *baseline*, *uniform*, *priority*, *proportional sharing*, and *PSFA*. All scripts can be found in the `frontera_scripts/per_job` folder.
  - **baseline:** `baseline/launch_test.sh`
  - **uniform:** `static/launch_test.sh`
  - **priority:** `priority/launch_test.sh`
  - **psharing:** `proportional/launch_test.sh`
  - **psfa:** `psfa/launch_test.sh`