Protecting Metadata Servers From Harm Through Application-level I/O Control

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

Abstract—**Modern large-scale I/O applications that run on HPC infrastructures are increasingly becoming metadata-intensive. Unfortunately, having multiple concurrent applications submitting massive amounts of metadata operations can easily saturate the shared parallel file system's metadata resources, leading to unresponsiveness of the storage backend and overall performance degradation. To address these challenges, we present PADLL, a storage middleware that enables system administrators to proactively control and ensure QoS over metadata workflows in HPC storage systems. We demonstrate its performance and feasibility by controlling the rate of both synthetic and realistic I/O workloads. Results show that PADLL can dynamically control metadata-aggressive workloads, prevent I/O burstiness, and ensure I/O fairness and prioritization.**

Index Terms—**Storage, metadata, QoS, PFS, data plane.**

I. INTRODUCTION

Modern supercomputers are establishing a new era in high-performance computing (HPC), providing unprecedented compute power that enables large-scale parallel applications to run at massive scale [1]–[3]. However, contrary to long-lived assumptions about HPC workloads where applications were predominately compute-bound and write-dominated, modern applications (*e.g.,* Deep Learning training) are data-intensive, read-dominated, and generate massive bursts of metadata operations [4]–[7]. In fact, several HPC centers have already observed a surge of metadata operations in their clusters, and they expect this to become more severe over time [7], [8].

While these workloads demand scalable, high throughput, and low latency storage, most TOP500 [9] supercomputers rely on Lustre-like parallel file systems (PFSs), which provide a centralized metadata management service [10]–[12]. In these data centers, having multiple concurrent jobs competing for shared I/O resources can lead to severe I/O contention and overall performance degradation [8], [13], [14]. For example, existing studies report that it is common for even a single user's I/O operations to saturate Lustre metadata resources, leading to unresponsiveness of the file system, reduced speed of computations of all running jobs, and even failures of metadata servers [8], [14], [15].

While there are numerous solutions to assess the bottlenecks generated from data workflows in HPC clusters [13], [14], [16]–[23], the metadata counterpart has not received the same level of attention, and existing approaches are suboptimal.

Manual intervention. In several HPC research facilities, system administrators stop jobs with aggressive I/O behavior (*e.g.,* datasets made of small-sized files, unnecessary file system requests) and temporally suspend job submission access for users that do not comply with the cluster's guidelines [8], [15]. While this helps to protect the file system from metadata-aggressive users, this is a *reactive approach* that is only triggered when the job has already slowed the storage system and the other jobs in execution.

Intrusive to I/O layers. While solutions like GIFT [13] and TBF [14] are designed to mitigate I/O contention and variability, these are tightly coupled to the system implementation and require high intrusiveness to several layers of the HPC software stack, including the shared file system, job scheduler, and I/O libraries. Such an approach requires deep understanding of the system's internal operation model and 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, like BeeGFS or PVFS, as even though they share a similar high-level design, the internal logic differs across implementations.

Partial visibility and I/O control. Some solutions overcome the previous challenge by actuating at the compute node level, enabling QoS control from the application-side, thus not requiring changes to core layers of the I/O stack [24]. However, these act in isolation (*i.e.,* agnostic of other jobs), being unable to holistically coordinate the I/O generated from multiple jobs that compete for shared storage, thus leading to I/O contention and waste of system resources [25], [26].

We propose PADLL, an application and file system agnostic storage middleware that enables QoS control of 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 from all running jobs in the HPC system. PADLL adopts ideas from the Software-Defined Storage (SDS) paradigm [27], following a decoupled design made of two planes of functionality: *control* and *data*. The data plane is a multi-stage component distributed over compute nodes, where each stage mediates the I/O requests between a given application and the shared file system. Specifically, stages transparently handle application's requests by intercepting POSIX calls (*e.g.,* `open`, `close`, `read`) and dynamically rate limiting them before being submitted to the PFS. This enables general applicability

and cross-compatibility with POSIX-compliant file systems.

The control plane is a logically centralized entity with global system visibility. It acts as a global coordinator that continuously monitors and manages all running jobs by adjusting the I/O rate of each data plane stage. It does so by enabling system administrators to express rate limiting rules at per-job, group of jobs, or cluster-wide granularity. For example, rules can be as simple as statically rate limiting requests of a given job, to more complex ones, such as dynamically adjusting the metadata rate of all jobs according to workload and system variations, which can be expressed as control algorithms (*e.g.,* proportional sharing [28], dominant resource fairness [29]).

We implemented an early prototype of PADLL, as well as set of rules and a proportional sharing control algorithm. We validate its performance and feasibility through a set of experiments to control the rate of I/O workflows under different scenarios. Experiments were conducted using IOR [30] and real traces of metadata operations collected from a Lustre file system at AIST[1], and demonstrate that PADLL:

- Effectively controls the rate of I/O workflows at different granularities, including request type (*e.g.,* open, close, read), request class (*e.g.,* metadata, data), and job.
- Prevents I/O burstiness and is able to control metadata-aggressive workloads through static and dynamic rates.
- Coordinates the rate of multiple concurrent jobs in holistic fashion, ensuring I/O fairness and prioritization.
- Induces negligible overhead across all testing scenarios.

In summary, this paper provides the following contributions:

- A **study** that *analyzes traces* from a production Lustre file system at AIST, and *reveals new insights* about metadata operations at scale (§II).
- **PADLL**, an application and file system agnostic storage middleware that enables QoS over metadata workflows in HPC storage systems (§III).
- An **experimental evaluation** that showcases PADLL's performance and applicability under different scenarios using both synthetic and realistic I/O workloads (§IV).

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 [10], [31], BeeGFS [11], PVFS [12]), 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 *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
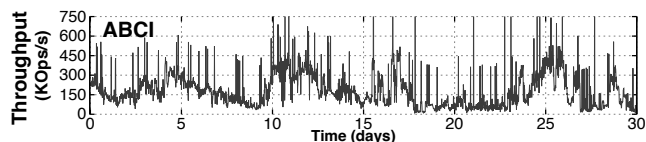
Fig. 1: *Throughput of metadata operations in* PFS$_A$.

at compute nodes and access the file system using standard POSIX system calls (*e.g.,* open, read, close).

Depending on the scale of the file system, metadata nodes assume different configurations [32]. In some deployments, the namespace is persisted across multiple MDTs and a single MDS handles all metadata operations, having additional MDS nodes has 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 remote procedure call (RPC) 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 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 [31], [33]. Other operations, such as mkdir or rename, require even stronger guarantees (*e.g., 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 (*e.g.,* FMA [34], OpenImages [35]), which generate high and continuous bursts of metadata operations. Third, the number of file system *clients* is several times higher than available MDSs, which can easily become saturated when several concurrent jobs have aggressive I/O metadata behavior.

A. Analyzing Metadata Operations in Production Clusters

We analyze the logs of a Lustre file system from the AI Bridging Cloud Infrastructure (ABCI) [36]. The storage at ABCI is made of multiple PFSs. Of these, the /group area is managed by a DDN ExaScaler Lustre file system that is composed of 2 MDSs in a hot-standby configuration, backed by 6 MDTs, and 36 OSTs that provide 9.5 PiB of storage capacity. For simplicity, we refer to this file system as PFS$_A$.

We monitored the I/O activity of the most frequent metadata operations at MDSs/MDTs, using DDNStorage's LustrePerf-Mon [37]. 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. Addition-
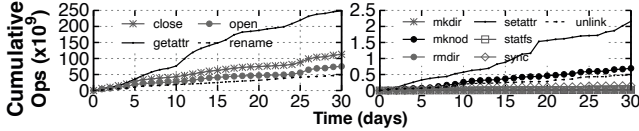
Fig. 2: Cumulative metadata operations in PFS_A.

ally, 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_A. Metadata operations are submitted at a massive rate, with an average of ≈ 200 KOps/s. Over different periods, PFS_A 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 experiencing periods of low throughput (50 KOps/s or lower) to immediately spike up to 450 KOps/s (or higher).

Interestingly, we observe that this load is much higher than those reported in other clusters [7]. For example, a study from NERSC reports that the PFS shared by the 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_A experiences 29 KOps/s and 43.5 KOps/s. While the metadata load may depend on different factors, 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 #1.** *Modern I/O workloads are generating massive amounts of metadata operations, with high throughput rates, and bursts that reach 1 MOps/s. Based on previous studies [7] and the results observed from PFS_A, it is expected that these values will continue to increase over time. This means that only ensuring QoS for data workflows is not enough, and metadata operations should be handled as well.*

Type and frequency of metadata operations. Fig. 2 shows the type and amount of metadata operations in PFS_A. The most predominant operations are `open`, `close`, `getattr`, and `rename`, which account for 98% of the total load. Notoriously, several of these are particularly costly to the PFS and more prone to cause I/O contention. Specifically, `open` and `close` system calls may require acquiring several locks in the namespace to update internal state of the namespace; `rename` needs to ensure *atomicity*, which is particularly expensive, for example, when moving files between MDT servers [31], [33]. As for `getattr` operations, even though they are less costly than previous ones, PFS_A received almost 250 billion requests during the observation period (average rate of 95.8 KOps/s), which can still bottleneck the system.

> **Observation #2.** *The most predominant metadata operations (i.e., `open`, `close`, `rename`) entail higher costs to the PFS due to namespace housekeeping and locking, being very likely to saturate metadata resources. As such, operations should be controlled with fine-granularity, ensuring that operations with different costs have different QoS levels.*
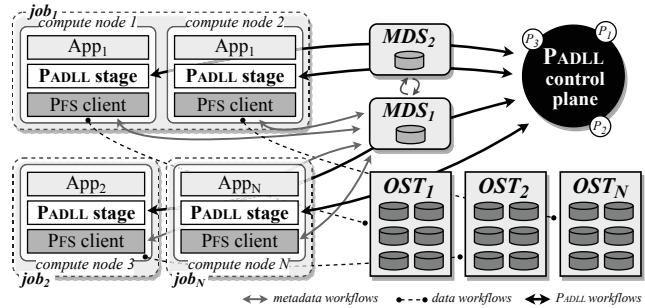

Fig. 3: **PADLL** *high-level architecture. It is composed of a control plane and multiple data plane stages.*

III. PADLL STORAGE MIDDLEWARE

PADLL is a storage middleware that enables system administrators to proactively control and ensure QoS over metadata workflows of all running jobs in HPC infrastructures. Its design is built under the following core principles.

Decoupled design. PADLL follows a decoupled design that separates the I/O logic into two planes of functionality – the *data plane* implements the mechanisms that are applied over requests (*e.g.,* rate limiting), while the *control plane* defines the policies that control them (*e.g,* static rate, I/O fairness).

Application and PFS-agnostic. PADLL does not require code changes to any core layers of the HPC I/O stack, being agnostic of the applications it is controlling as well the PFS to which the requests will be submitted to. This makes PADLL applicable over different applications and compatible with POSIX-compliant storage systems.

Fine-grained I/O control. PADLL classifies and differentiates requests at different levels of granularity, which allows applying different types of policies (*e.g.,* only rate limit `open` calls, rate limit all metadata operations).

Global visibility. PADLL ensures that all stages operate in coordination, controlling the rate of all running jobs holistically.

Fig. 3 outlines PADLL's high-level architecture, which is composed of a data plane (§III-A) and a control plane (§III-B).

A. Data Plane

The data plane is a multi-stage component that actuates at the compute node level. Each stage sits between the application and the file system client, and transparently intercepts POSIX requests before being submitted to the PFS.

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, which is the case of Fig. 3's job_2 where App_2 only runs at *compute node 3*. For distributed jobs, where application's instances run on separate compute nodes, multiple data plane stages need to be set (one per instance). For example, as depicted in Fig. 3's job_1, two stages are needed to rate limit App_1's I/O workflows since it executes in *compute nodes 1* and *2*.

Rate limiting I/O workflows with PADLL requires two steps.

Request differentiation. Given that applications may submit POSIX requests that are not destined towards the PFS, PADLL

needs to identify which requests it should handle. Requests are differentiated based on a specific set of attributes that characterize them, including the request type (*e.g.,* `open`, `getattr`), request class (*e.g.,* `metadata`, `data`), path name, and others. For each intercepted request, the stage verifies its attributes and determines if the request should be rate limited or be directly submitted to the file system without any throttling (for example, requests that are submitted to POSIX file systems – `xfs`, *NFS server* – other than the PFS).

Rate limiting. Internally, stages are organized in multiple queues, each with a token-bucket that determines the rate of its requests. A token-bucket is a commonly used mechanism for controlling the rate and burstiness of I/O workflows [28]. Each of these queues only serves a specific set of requests. For example, $queue_1$ handles *metadata*, while $queue_2$ handles *data* operations; $queue_3$ only handles `open` calls; $queue_4$ throttles the requests submitted to `/scratch/foo`. The type of requests each queue handles, as well as the rate set to each token-bucket, are defined by the control plane. After being throttled, requests are then submitted to the PFS.

B. Control Plane

The control plane is a logically centralized component with system-wide visibility that orchestrates how the I/O workflows of all running jobs should be handled. It does so by continuously communicating with stages to collect I/O metrics (*e.g.,* workflows' rate) and enforce new rates to respond to workload variations or new rules set by system administrators.

It enables system administrators to define how the system should act (*i.e.,* control logic), either by specifying *simple policies* such as individually set the rate for `open` calls of a given job, or through *control algorithms*, such as dynamically reserve shares of metadata operations for all jobs in the cluster.

In particular, control algorithms are implemented in a *feedback control loop* manner [28], where the control plane repeatedly *1)* collects metrics from stages, *2)* verifies if all policies are being met, and *3)* adjusts stages with a new rate.

Orchestrating stages from the same job. Every time a job starts, its corresponding stages are initialized and connected to the control plane. Stages send to the control plane information that characterizes the job and the node it is running (*e.g.,* `job-ID`, `PID`, hostname, user). 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. Prototype Implementation

We have implemented a PADLL prototype with 16K (*data plane*) and 6K (*control plane*) lines of C++ code. The data plane exposes a POSIX interface that reimplements 42 calls from different operation classes, including data, metadata, extended attributes, and directory management. It uses `LD_PRE-LOAD` to transparently intercept I/O requests, which are then differentiated and rate limited before going to the PFS. The logic for rate limiting requests (*e.g.,* queues, token-buckets) was built using PAIO, a framework for building custom-made,

user-level storage data planes [25]. The control plane implements the necessary building blocks for specifying policies (and control algorithms) and controlling data plane stages (*e.g.,* collect statistics, submit rules). Communication between the control plane and data plane stages is established through RPC calls, using the gRPC framework [38].

IV. EVALUATION

Our evaluation seeks to answer the following questions:
- Can PADLL enforce policies at different granularities?
- Can PADLL control I/O burstiness?
- Can PADLL enforce I/O prioritization and proportional sharing over multiple concurrent jobs?
- What is the overhead of using PADLL?

Experimental testbed. Experiments were conducted on compute nodes of the Frontera supercomputer [39], equipped with two 28-core Intel Xeon processors, 192 GiB of RAM, and a single 240 GiB SSD. Software-wise, it uses CentOS 7.9 with the Linux kernel v3.10 and the `xfs` file system. The production PFS is a Lustre file system.

Benchmarks and workloads. We conducted experiments using both data and metadata workloads. For data workloads, we used IOR [30]. To generate realistic metadata workloads, we implemented a *trace replayer* that submits (*"replays"*) metadata operations with an identical request distribution as the one observed from the logs collected at PFS_A. The replayer is multi-threaded, and each thread submits a specific operation type (*i.e.,* `open`, `close`, `getattr`) at a rate that follows the same performance curve as the original logs. The rate of each operation was scaled-down to half, to ensure that the file system could serve them without bottlenecking. The execution period was also accelerated, where each second of the *replayer* corresponds to a minute's worth of operations in the original log. The trace used in the experiments corresponds to the metadata operations of a single MDT server of PFS_A.

Methodology. Unless stated otherwise, experiments were conducted under three setups: *baseline*, which represents the benchmark (*IOR* or *trace replayer*) without using PADLL; *passthrough*, where POSIX operations submitted by the benchmark are intercepted by PADLL but are not rate limited; and *padll*, where POSIX operations submitted by the benchmark are intercepted by PADLL and throttled at a given rate.

For all experiments, the control plane runs at a dedicated compute node, and each job respects to the execution of *IOR* or the *trace replayer* under a specific workload. To prevent overloading the production PFS, and cause I/O contention and interference to concurrent jobs in the system, all metadata workloads were submitted to the local file system. *IOR* experiments were conducted using the PFS.

A. Per-operation Type and Class Rate Limiting

Per-operation type. First, we demonstrate how PADLL enables system administrators to control the rate of specific operations. Under this scenario, both *IOR* and *trace replayer* were configured to submit a single operation type. For all
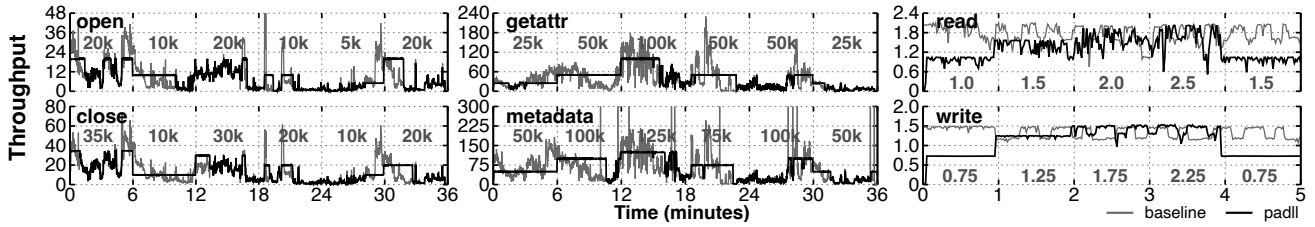
Fig. 4: *Per-operation type/class rate limiting. Experiments show that* PADLL *can enforce different rate limits over different POSIX operations.*

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). Fig. 4 depicts the results of all setups under different operation types, namely open, close, and getattr.

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 12 and 18 minutes. This is because, the limit set by the system administrator (for that interval) is higher than the rate at which the *replayer* submits operations. Analogously, we also observe periods where the *padll* setup achieves higher throughput than *baseline*, as observed in getattr between 6 and 12 minutes. This occurs 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. Experiments were also conducted for the rename operation type, of which we report similar findings as the aforementioned operations.

We observe similar results for data-oriented operations, namely read and write. However, since these are being submitted to the PFS, we observe more variability.

Per-operation class. We now demonstrate how PADLL controls the workflows of a given operation class (*i.e.*, metadata). The *trace replayer* spawns four threads, one for each operation type. Fig. 4 *(metadata)* depicts the obtained results. The throughput corresponds to the accumulated rate of all *replayer* threads. Again, *padll* effectively controls the rate of all metadata operations throughout the overall testing period.

Overhead. To evaluate the overhead imposed by PADLL, we conducted a set of experiments with the *passthrough* setup. When comparing *passthrough* with *baseline*, the overhead is negligible, never degrading performance more than 0.9% across all experiments. For figure clarity, *passthrough* is not depicted in Fig. 4, as its performance line practically overlaps with the *baseline* one.

B. Per-job Rate Limiting and QoS Control

We now demonstrate a scenario where PADLL controls the I/O workflows of multiple jobs. Under this scenario, we consider that the system administrator defines a maximum rate of metadata operations that can be submitted to the PFS (from all jobs). At all times, there are at most four jobs in the system, each running the same workload (same as in §IV-A's *per-operation class*). Jobs are incrementally added to the system every 3 minutes.

Setups. Experiments were conducted under four setups. *Baseline* represents the current setup supported at most supercomputers, where all jobs execute without being rate limited. The remainder setups are rate limited with PADLL. For these, we set the PFS's maximum metadata rate to 300 KOps/s. In the *Static* setup, each job is rate limited to 75 KOps/s throughout the entire execution. In *Priority*, jobs are also statically rate limited but are given different rates; namely, job_1 to job_4 are assigned with 40, 60, 80 and 120 KOps/s, respectively. In *Proportional sharing*, we implemented a control algorithm that enforces *per-job metadata rate reservations*, similar to those in [25], [40]. At any given time, the algorithm ensures that all jobs have access to a reserved metadata rate. However, whenever there are leftover metadata operations (*i.e.,* the current metadata rate has not reached the maximum limit), the algorithm distributes it among all active jobs in a proportional manner. For this setup, we assign the reserved rate of each job as in *Priority*. Fig. 5 depicts the results of all setups.

Baseline. Experiments were executed over 45 minutes. Throughout the entire execution, we observe that the workload is *volatile* and *bursty*, with peaks that reached 800 KOps/s. When all jobs are executing, there are several periods where the file system continuously serves requests over 400 KOps/s.

Static. Throughout the entire execution, whenever a new job is added, it is provisioned with its assigned rate (75 KOps/s). PADLL ensures that the throughput of all jobs is sustained and eliminates existing burstiness. Further, we observe that all jobs finish in the same time as in *Baseline*. While this setup is useful to equally distribute metadata rate across all jobs, it has two main limitations: first, it does not allow jobs to execute with different priorities; second, given that there are several time periods where there is leftover metadata rate (*e.g.,* ①–③), jobs may be rate limited more aggressively than needed.

Priority. Similarly to *Static*, PADLL ensures that all jobs are provisioned with their rate throughout the entire execution. 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 phenomenon for job_1's execution, which takes 20-min longer than in the *Baseline* and *Static* setups.

Proportional sharing. Under this scenario, all jobs finish under 45 minutes. Whenever a new job enters (①–③) or leaves the system (⑤–⑦), it is assigned with its proportional metadata share. When all jobs are running (④), they are assigned with their demanded rate. Compared to *Baseline*, this setup eliminates I/O burstiness and provides sustained
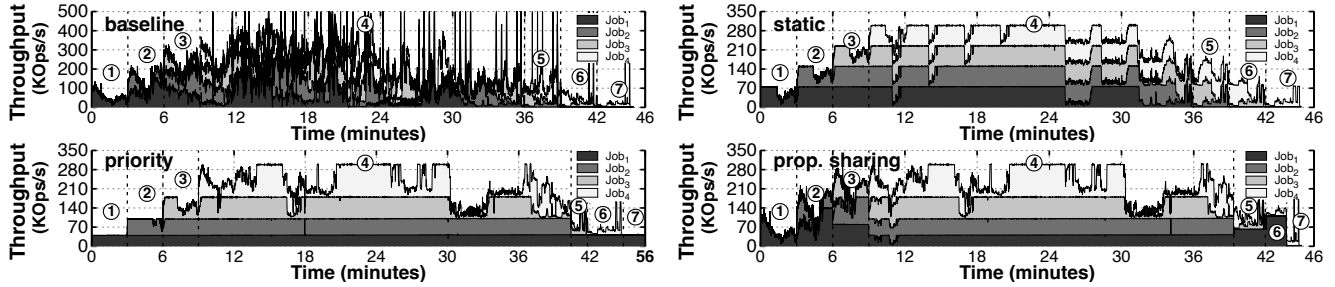
577

Fig. 5: Per-job metadata control over Baseline, Static, Priority, and Proportional Sharing setups. *All jobs (job_1 to job_4) execute the same workload and are spawned at different times. ① to ⑦ mark the moment when a job enters or leaves the system.*

metadata performance. Compared to **Static** and **Priority**, it ensures different priority rates, and whenever leftover metadata rate is available, it shares it across active jobs.

C. Discussion

The purpose of this evaluation is to demonstrate that PADLL can effectively control the rate of workflows, both data and metadata, of multiple HPC jobs. Experiments involving metadata operations were conducted over the local file system to prevent the **baseline** setup to cause harm to the production-based PFS and negatively impact the performance of concurrent jobs in the cluster. Nevertheless, because PADLL actuates at the system call level (user-space), it can intercept and handle POSIX operations submitted to any in-kernel file system registered in the Virtual File System layer, including local and remote file systems (*i.e.,* Lustre kernel client). As such, we expect PADLL to achieve the same level of effectiveness and performance when used over PFSs. Furthermore, since PADLL orchestrates I/O workflows in holistic fashion, we expect that control algorithms like **Proportional sharing** (§IV-B) can improve the performance of jobs, over **Baseline**, when the PFS is saturated (in terms of metadata operations).

V. RELATED WORK

HPC storage QoS. Many works are designed to mitigate I/O contention in HPC storage stacks but ignore the impact of metadata workflows [13], [16], [18], [20]–[22]. 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 [14], [18], [20], [22], [23], scheduler [21], and I/O libraries [16], [17]. 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 I/O stack. Similarly to PADLL, OOOPS transparently intercepts and rate limits POSIX requests at compute nodes [24]. However, it does not provide global visibility, being unable to enforce dynamic and cluster-wide policies as those demonstrated in §IV-B.

SDS systems. PADLL builds on a large body of work on SDS systems. Systems like IOFlow, sRoute, and PSLO, actuate at the virtualization and block device layers, only controlling the rate of `read` and `write` requests [40]–[43]. Others, such as Retro and Crystal, implement resource management policies over distributed storage systems [44]–[46], but are directly

implemented within the storage system itself, offering limited maintainability and portability. 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.

VI. CONCLUSION

We have presented PADLL, an application and PFS-agnostic storage middleware that enables enforcing QoS policies over metadata workflows in HPC clusters. With it, system administrators can proactively and holistically control the I/O rate of all running jobs, and thus, prevent metadata-aggressive ones from harming the PFS, as well as other jobs in execution. Preliminary results show that PADLL can: *i)* control the rate of I/O workflows at different granularities, *ii)* prevent I/O burstiness, and *iii)* ensure I/O fairness and prioritization.

Moreover, the work presented in this paper opens the path to interesting research directions including:

Control algorithms. Given the advantages of using control algorithms as those discussed in §IV-B, it would be interesting to explore others and analyze their impact in PFSs in production.

Control plane scalability. Currently, the control plane is a centralized component that monitors and controls multiple stages. To ensure PADLL can be used at scale, it is fundamental to investigate the control plane's scalability and dependability.

Additional experiments. To further assess PADLL's contributions, it should be used under production-based PFSs, and evaluated with large-scale I/O applications (*e.g.,* TensorFlow [47]) with different I/O workloads and access patterns.

REFERENCES

[1] "Frontier Supercomputer," https://www.olcf.ornl.gov/frontier/.

[2] "Aurora Supercomputer," https://www.alcf.anl.gov/aurora.

[3] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-Design for A64FX Many-core Processor and "Fugaku"," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[4] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2021, pp. 81–91.

[5] M. Dantas, D. Leitão, P. Cui, R. Macedo, X. Liu, W. Xu, and J. Paulo, "Accelerating Deep Learning Training Through Transparent Storage Tiering," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2022, pp. 21–30.

[6] S. W. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. 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*. IEEE, 2018, pp. 54–63.

[7] T. Patel, S. Byna, G. K. Lockwood, and D. 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.

[8] S. Liu, L. Huang, H. Liu, A. Ruhela, V. Trueheart, S. Lindsey, and Q. 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.

[9] "The Top 500 List," https://www.top500.org/.

[10] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003, pp. 380–386.

[11] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning," in *48th International Conference on Parallel Processing*. ACM, 2019.

[12] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *4th Annual Linux Showcase & Conference*. USENIX Association, 2000.

[13] T. Patel, R. Garg, and D. 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*. USENIX Association, 2020, pp. 103–119.

[14] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. 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.

[15] L. Huang, Y. Wang, C.-Y. Lu, and S. Liu, "Best Practice of IO Workload Management in Containerized Environments on Supercomputers," in *Practice and Experience in Advanced Research Computing*. ACM, 2021.

[16] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 155–164.

[17] J. Carretero, E. Jeannot, G. Pallez, D. E. Singh, and N. Vidal, "Mapping and Scheduling HPC Applications for Optimizing I/O," in *34th ACM International Conference on Supercomputing*. ACM, 2020.

[18] X. Zhang, K. Davis, and S. 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.

[19] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-side I/O Coordination for Parallel File Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2011.

[20] X. Zhang, K. Davis, and S. 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.

[21] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1013–1022.

[22] S. Karki, B. Nguyen, and X. Zhang, "QoS Support for Scientific Workflows using Software-Defined Storage Resource Enclaves," in *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2018, pp. 95–104.

[23] Y. Hua, X. Shi, H. Jin, W. Liu, Y. Jiang, Y. Chen, and L. He, "Software-defined QoS for I/O in exascale computing," *CCF Transactions on High Performance Computing*, vol. 1, no. 1, pp. 49–59, 2019.

[24] L. Huang and S. Liu, "OOOPS: An Innovative Tool for IO Workload Management on Supercomputers," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems*. IEEE, 2020, pp. 486–493.

[25] R. Macedo, Y. Tanimura, J. Haga, V. Chidambaram, J. Pereira, and J. Paulo, "PAIO: General, Portable I/O Optimizations With Minor Application Modifcations," in *20th USENIX Conference on File and Storage Technologies*. USENIX Association, 2022, pp. 413–428.

[26] D. Shue, M. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-Tenant Cloud Storage," in *10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 349–362.

[27] R. Macedo, J. Paulo, J. Pereira, and A. Bessani, "A Survey and Classification of Software-Defined Storage Systems," *ACM Computing Surveys*, vol. 53, no. 3, May 2020.

[28] J.-Y. L. Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer Science & Business Media, 2001, vol. 2050.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2011.

[30] H. Shan, K. Antypas, and J. 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.

[31] P. Braam, "The Lustre Storage Architecture," 2019.

[32] "Lustre Metadata Service (MDS)," https://wiki.lustre.org/Lustre_Metadata_Service_(MDS).

[33] "Lustre MDC: mdc_reint.c," https://github.com/lustre/lustre-release/blob/master/lustre/mdc/mdc_reint.c, 2022.

[34] M. Defferrard, K. Benzi, P. Vandergheynst, and X. Bresson, "FMA: A Dataset For Music Analysis," 2016.

[35] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov *et al.*, "The Open Images Dataset v4," *International Journal of Computer Vision*, vol. 128, no. 7, pp. 1956–1981, 2020.

[36] "AI Bridging Cloud Infrastructure," https://abci.ai/.

[37] "DDNStorage/LustrePerfMon: Lustre Monitoring System," https://github.com/DDNStorage/LustrePerfMon.

[38] "gRPC: A high performance, open source universal RPC framework," https://grpc.io/.

[39] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The Evolution of Leadership Computing at the National Science Foundation," in *Practice and Experience in Advanced Research Computing*. ACM, 2020, pp. 106–111.

[40] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron *et al.*, "IOFlow: A Software-Defined Storage Architecture," in *24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196.

[41] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sRoute: Treating the Storage Stack Like a Network," in *14th USENIX Conference on File and Storage Technologies*. USENIX Association, 2016, pp. 197–212.

[42] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage," in *11th European Conference on Computer Systems*. ACM, 2016, pp. 28:1–28:14.

[43] M. Mesnier, F. Chen, T. Luo, and J. Akers, "Differentiated Storage Services," in *23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 57–70.

[44] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted Resource Management in Multi-tenant Distributed Systems," in *12th*

USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 2015, pp. 589–603.

[45] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling High-level SLOs on Shared Storage Systems," in *3rd ACM Symposium on Cloud Computing*. ACM, 2012, pp. 14:1–14:14.

[46] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López *et al.*, "Crystal: Software-Defined Storage for Multi-tenant Object Stores," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 243–256.

[47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283.