

A Tale of Scale: Enabling Scalable and Holistic Storage QoS for Exascale HPC Systems

Mariana Miranda
INESC TEC & University of Minho
Braga, Portugal
mariana.m.miranda@inesctec.pt

Yusuke Tanimura
AIST
Tokyo, Japan
yusuke.tanimura@aist.go.jp

Jason Haga
AIST
Tokyo, Japan
jh.haga@aist.go.jp

Amit Ruhela
Texas Advanced Computing Center
Austin, Texas, USA
aruhela@tacc.utexas.edu

Stephen Lien Harrell
Texas Advanced Computing Center
Austin, Texas, USA
sharrell@tacc.utexas.edu

John Cazes
Texas Advanced Computing Center
Austin, Texas, USA
cazes@tacc.utexas.edu

José Pereira
INESC TEC & University of Minho
Braga, Portugal
jop@di.uminho.pt

Ricardo Macedo
INESC TEC & University of Minho
Braga, Portugal
ricardo.g.macedo@inesctec.pt

João Paulo
INESC TEC & University of Minho
Braga, Portugal
joao.t.paulo@inesctec.pt

Abstract

Modern supercomputers increasingly host multiple data-intensive jobs that compete for shared storage resources, leading to I/O interference and performance degradation. Software-defined storage (SDS) has emerged as a promising approach to ensuring holistic storage Quality of Service (QoS), but existing solutions fail to meet the requirements of today's exascale supercomputers.

In this work, we empirically examine the limitations of state-of-the-art SDS control planes. Based on our findings, we introduce SHIO, a novel distributed and dependable hierarchical control plane design tailored to the scale of modern exascale supercomputers. We also introduce a hierarchical proportional sharing algorithm (PROSHA), which balances and leverages the computational resources across multiple controllers, improving scalability and control latency while providing holistic storage QoS guarantees.

Our large-scale experiments, using up to 2,000 compute nodes from the Frontera supercomputer, show that SHIO enables the coordination of up to 100,000 data plane stages, demonstrating that a scalable control plane is feasible for modern and next-generation HPC systems. Furthermore, our novel design and algorithm enable reducing the latency of control cycles from 1 second to 55 ms, a 17× improvement compared to state-of-the-art designs.

CCS Concepts

• **Information systems** → **Storage management**; • **Computer systems organization** → **Distributed architectures**.

Keywords

Software-defined storage, Scalability, HPC, Storage management

ACM Reference Format:

Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, John Cazes, José Pereira, Ricardo Macedo, and João Paulo. 2026. A Tale of Scale: Enabling Scalable and Holistic Storage QoS for Exascale HPC Systems. In *27th International Middleware Conference (Middleware '26)*, December 14–18, 2026, Tarragona, Spain. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3801927.3858650>

1 Introduction

We have been witnessing a significant increase in the number of nodes and the computational power of HPC supercomputers. LineShine and El Capitan, with 20,480 and 11,520 compute nodes, can deliver 2.198 and 1.809 exaFLOPS, respectively [24, 29]. Fugaku stands out for its scale, comprising 158,976 nodes [13]. Operating infrastructures at such a scale is challenging, especially when ensuring Quality of Service¹ (QoS) for up to thousands of simultaneous user jobs running on these systems [36, 47].

The central problem this paper explores is how to achieve QoS for storage I/O in exascale clusters. Modern supercomputers rely on shared Parallel File Systems (PFS), such as Lustre, DAOS, GPFS, and BeeGFS, which concurrently serve I/O requests from thousands of jobs [4, 5, 10, 20]. As compute node count grows, the number of simultaneous jobs accessing the PFS naturally increases, leading to I/O contention, performance degradation, and even infrastructure downtime [8, 12, 33]. This problem is intensified by the rise of data-centric workloads, including deep learning and LLM training, which stress the PFS's data and metadata paths [6, 7, 9, 14]. While numerous works aim to mitigate storage interference and guarantee QoS for user jobs, they often fall short when faced with the demands of large-scale HPC deployments.

Cloud computing environments face analogous challenges in managing storage at scale, which have been addressed through centralized orchestration platforms, such as Kubernetes and policy-driven resource management systems [15, 16, 23, 39, 42]. However, cloud-based control systems target virtualized infrastructures and

¹By storage QoS, we refer to ensuring a target latency and/or throughput to jobs simultaneously accessing shared storage resources, in our case, the PFS.



assume elastic, on-demand resource provisioning. This does not hold in HPC environments, where workloads are tightly coupled, performance-sensitive, and run on bare-metal hardware with fixed resource allocations [27, 36, 47]. When focusing on HPC requirements, we can categorize existing storage QoS solutions as follows.

From intrusive solutions. Systems, including GIFT [34], CAL-CioM [12], and TBF [36], address storage performance variability by modifying fundamental components of the HPC I/O stack. These changes span I/O libraries, job schedulers, and the PFS itself, which are components that are complex, performance-critical, and tightly coupled to the underlying infrastructure. Such intrusive modifications introduce considerable maintenance burdens and hinder portability across HPC systems [26].

To static-based and individual control. Techniques such as OOOPS [19] avoid system-level modifications by intercepting and throttling POSIX requests at the job level. As these operate independently per job, they lack global visibility (*i.e.*, across multiple jobs) and cannot ensure cluster-wide I/O coordination. Also, they assume static operating conditions, ignoring the highly dynamic nature of HPC workloads, where jobs continuously start and finish with varying I/O patterns. As a result, individualized static-based approaches lead to suboptimal resource utilization and are unable to avoid I/O contention under dynamic environments [26].

Towards Software-Defined Storage (SDS). To overcome the aforementioned limitations, SDS solutions provide non-intrusive, dynamic, and coordinated control across HPC jobs, ensuring that storage QoS policies are met at all times [21, 22, 26, 27]. Their design is based on two decoupled planes of functionality: the *data plane* and the *control plane*, as shown in Fig. 1a. The data plane consists of middleware stages that run on the compute nodes, sitting between the job and the PFS client. In this paper, we consider a design where each compute node has a single stage mediating all I/O requests from the job instance deployed there. Distributed jobs, with multiple instances running on different compute nodes, have a stage managing I/O requests at each node. Stages transparently intercept and rate-limit I/O requests according to policies defined by the control plane (*e.g.*, limiting I/O bandwidth and/or the number of metadata operations submitted to the PFS). The control plane acts as an independent service with cluster-wide visibility. It orchestrates all stages by collecting metrics from these (❶), coordinating their actions (❷) through control algorithms to ensure holistic storage policies (*e.g.*, I/O priority, fairness) across the HPC infrastructure, and enforcing configuration rules across the system (❸).

While there is extensive research on how to build data plane stages [16, 26, 28, 35], current solutions, either targeting cloud computing or HPC infrastructures, leave the design of the control plane as a future endeavor, especially in terms of its scalability. This is an acknowledged challenge but poorly addressed and assessed, for instance, with simple centralized or hierarchical designs that are only tested with a small scale of data plane stages, far from real deployments [16, 21, 22, 26, 42].

The main insight of this work is that the control plane and its control algorithms must be carefully designed to efficiently and promptly enforce holistic storage QoS policies for jobs running on large-scale HPC systems with up to hundreds of thousands of compute nodes.

To achieve such a goal, we make the following contributions.

Limitations of existing control plane designs. We empirically study the scalability of *centralized* (Fig. 1b) and *hierarchical* (Fig. 1c) designs proposed by prior work [16, 21, 22, 26, 40, 42] on an HPC supercomputer, revealing important trade-offs (§2.2). *Centralized designs* suit only small-to-medium infrastructures (*i.e.*, a few thousand nodes), due to hardware constraints (*e.g.*, a maximum number of allowed connections) and control latency inflation caused by funneling metric collection, computation, and enforcement through a single controller. *Hierarchical designs* support larger infrastructures, but intermediate network hops and centralized decision-making introduce substantial delays, as lower-level controllers only act as relays while the top-level controller remains a bottleneck.

New control plane design and control algorithm. Motivated by these findings, we introduce SHIO (*Scalable and holistic HPC I/O management*), an SDS control plane that decentralizes decision-making across all levels of the hierarchy while providing global I/O orchestration (*i.e.*, system-wide storage QoS policies) across jobs (§3). The key idea is that offloading computation to lower-level controllers enables better usage of their available compute resources and allows them to make autonomous decisions, without the intervention of a top-level controller, sparing extra network hops and reducing overall control latency (*i.e.*, improving responsiveness of the control plane to dynamic HPC workloads). Achieving this, however, is not trivial and requires redesigning key components and control algorithms of existing control planes.

Design-wise, SHIO introduces two abstractions: low-level *cluster controllers* and a top-level *global controller*. Cluster controllers independently compute and enforce storage policies for the data plane stages under their responsibility. The global controller preserves system-wide I/O orchestration by distributing available resources across cluster controllers and coordinating stages belonging to the same distributed job but managed by different cluster controllers. Our design is fault-tolerant (*i.e.*, failure of one or more controllers) through a hot-swap primary-backup strategy. Further, we propose PROSHA, a new proportional sharing algorithm fitted for the hierarchical design of SHIO, enabling cross-controller (global and cluster) computation and coordination. Unlike state-of-the-art algorithms [26], which require centralizing all control logic at the global controller, PROSHA delegates this logic across hierarchy levels, enabling timely enforcement of QoS policies without incurring the latency overhead of top-level aggregation.

Large-scale experiments and validation. We validate the performance, scalability, and dependability of SHIO's prototype through a comprehensive experimental evaluation on the Frontera supercomputer, using up to 2,000 compute nodes, under both realistic and synthetic workloads (§4). SHIO is the first storage I/O orchestration solution validated at the scale of modern exascale HPC systems. Results show that SHIO can leverage the computational capabilities of the full hierarchy of controllers, efficiently orchestrating I/O on a supercomputer and handling dynamic workloads with minimal impact on job performance and low control latency. Notably, we show that control latency can be reduced from 1 second (state-of-the-art hierarchical designs) to 55 ms for a cluster of 100,000 data plane stages. Under controller failures, SHIO quickly recovers with minimal impact on QoS compliance.

Table 1: Supercomputers of the Top500 rank, LINPACK performance (petaFLOPS), number of nodes, and deployment year.

System	Rank	Rmax	Nodes	Year
LineShine [29]	1	2,198.40	20,480	2025
El Capitan [24]	2	1,809.00	11,520	2024
Frontier [30]	3	1,353.00	9,408	2021
Fugaku [13]	9	442.01	158,976	2020
MareNostrum 5 GPP [3]	61	40.10	6,480	2023
Frontera [41]	90	23.52	8,368	2019

2 Background and study

Next, we provide an overview of the HPC infrastructure landscape and then examine the scalability and performance of state-of-the-art control plane designs.

2.1 Modern HPC infrastructures

The Top500 [45] list (June 2026) ranks the world's 500 most powerful supercomputers according to their performance on the *LINPACK benchmark* [44]. To illustrate the scale of modern supercomputers, Table 1 highlights some of these HPC systems.

LineShine [29] ranks first in the list, achieving 2.198 exaFLOPS. Hosted at the National Supercomputing Centre in Shenzhen, it comprises 20,480 compute nodes. It is followed by El Capitan [24] (1.809 exaFLOPS) deployed at Lawrence Livermore National Laboratory with 11,520 compute nodes and Frontier [30] (1.353 exaFLOPS) at Oak Ridge National Laboratory with 9,408 nodes. Fugaku [13] stands out for its scale. Despite reaching 442 petaFLOPS on the *LINPACK benchmark*, it comprises 158,976 compute nodes, nearly an order of magnitude more than the previous systems.² Even petascale systems, like MareNostrum 5 GPP [3] (40.10 petaFLOPS) and Frontera [41] (23.52 petaFLOPS), operate with thousands of nodes (6,480 and 8,368 nodes, respectively).

Overall, modern HPC systems commonly integrate thousands of compute nodes, and this scale is expected to grow further. Interestingly, all these systems provide shared storage resources, usually through a PFS solution (e.g., Lustre, DAOS, GPFS), accessible by all compute nodes where user jobs execute. Efficiently managing and providing QoS for these shared storage resources remains a pressing concern and an open research challenge [19, 25, 26, 34, 36].

2.2 Scalability issues of HPC controllers

While initially applied to cloud computing infrastructures, SDS has recently been adapted to HPC systems by different research works [16, 21, 22, 26, 27, 40, 42, 46]. These address the differences in terms of infrastructure and software stack between cloud and HPC (e.g., bare-metal access to servers, specialized hardware, libraries), focusing on the data plane component and on providing transparent and general-purpose middleware techniques to intercept and mediate I/O requests from HPC jobs before these reach the PFS.

As depicted in Fig. 1a, data plane stages, deployed alongside the corresponding jobs on compute nodes, are oblivious to other jobs and stages in the system. The control plane enforces global storage policies (e.g., ensuring that the PFS's maximum I/O bandwidth is

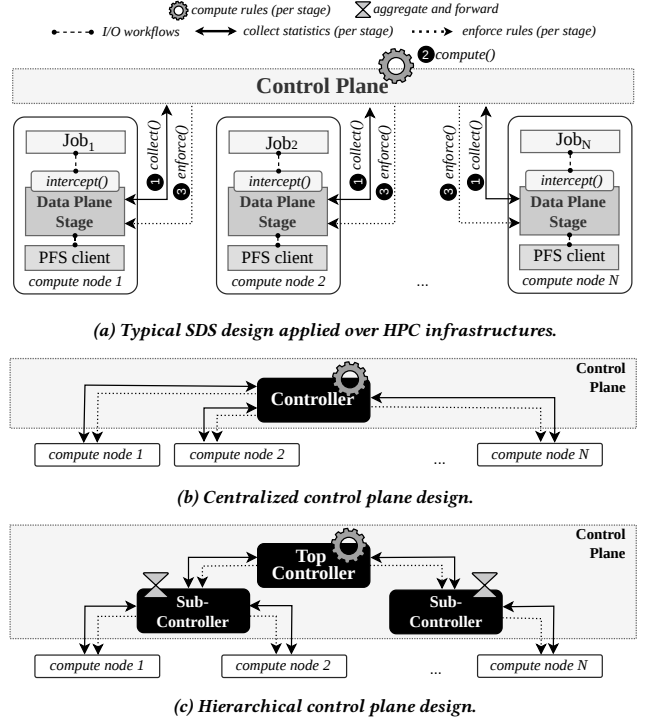


Figure 1: High-level architecture of: (a) typical SDS design applied over HPC infrastructures, (b) centralized control plane design with a single controller, and (c) hierarchical control plane design with controllers organized in a tree-based topology.

never exceeded while fairly sharing its resources among all jobs in the system) by monitoring and coordinating all data plane stages. It periodically collects I/O metrics, computes storage policy decisions using control algorithms, and enforces configuration rules across all stages. These control cycles are repeatedly executed to ensure that storage policies are maintained even under workload variations.

Efficiently monitoring and coordinating tens to hundreds of thousands of data plane stages (i.e., at the scale of modern HPC infrastructures depicted in Table 1) remains a complex and overlooked challenge. To better understand the current limitations of SDS systems and their applicability to large-scale supercomputers, we conduct an empirical study of control plane designs.

Designs and prototypes. State-of-the-art designs fall into two categories, centralized and hierarchical. We implemented a representative prototype for each, which we describe next.

The *Centralized* control plane is based on the state-of-the-art Chefard solution [26], where a single controller with system-wide visibility orchestrates I/O workflows (e.g., data and metadata) for all stages in the HPC system (Fig. 1b), a common design used in other works [16, 42]. The control logic (i.e., control algorithms) is implemented in a feedback control loop, where it continuously collects metrics from stages (e.g., I/O bandwidth of the job's instance), verifies if QoS policies are being met, and computes and enforces new storage rules for non-compliant stages.

The *Hierarchical* control plane extends the centralized prototype by adding a new layer of sub-controllers (Fig. 1c), deployed across different nodes (servers) and sitting between the top controller and

²Fugaku achieves exascale performance on different benchmarks, such as HPL-AI [37].

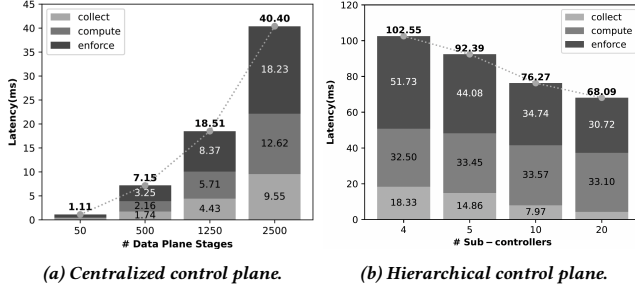


Figure 2: Average latency of control cycles for: (a) a centralized design as the number of stages increases, and (b) a hierarchical design managing 10,000 stages with an increasing number of sub-controllers.

the data plane stages. The top controller is responsible for implementing and computing the control logic, while sub-controllers collect and aggregate metrics from their set of stages, relay them to the top controller, and then forward rules computed by the top controller back to the stages. This design is followed by previous work on hierarchical control planes [21, 22, 26, 40].

At the single controller (centralized design) and the top controller (hierarchical design), a proportional sharing control algorithm (*i.e.*, *proportional sharing without false allocation* (PSFA)) manages I/O resources among jobs [26]. PSFA allows assigning different I/O rates (*e.g.*, I/O bandwidth, metadata rate) to jobs, while guaranteeing that the maximum aggregated I/O rate supported by shared storage resources (*e.g.*, the PFS) is never exceeded. Also, the algorithm considers each job's actual I/O usage and redistributes unused resources among active jobs to avoid waste.

Study goals. We assess the scalability of state-of-the-art control plane designs at the scale of a 10,000-node HPC system, a size representative of modern supercomputers (Table 1), where each compute node hosts a single data plane stage. Specifically, we examine both the hardware constraints that limit scalability and the impact of design choices on control cycle latency at this scale.

Experimental setup. Experiments run on up to 200 compute nodes of the Frontera supercomputer (see §4 for hardware specifications). The deployment supports up to 10,000 data plane stages, matching the target HPC system size for our study. Since exclusive access to 10,000 nodes of a production supercomputer is infeasible, we co-locate 50 stages per physical compute node, each operating independently as if on a distinct node. To ensure the validity of our results, we verify that this co-location does not saturate the node's resources, avoiding any interference in our results. Controllers are deployed on dedicated compute nodes of Frontera.

We use a synthetic stress testing workload (*Configuration B* detailed in §4), in which the control plane executes control cycles continuously: collecting I/O metrics (*i.e.*, data and metadata), computing the PSFA algorithm, and enforcing rate-limiting rules at all data plane stages in each cycle. The data plane stages produce randomly generated metrics without issuing real I/O to the production PFS, preventing interference with other users and disruption to the supercomputer's normal operation. Despite the synthetic workload, each control cycle still requires collecting metrics from all stages, computing the control algorithms, and enforcing rules across stages, faithfully stressing the control plane designs at scale.

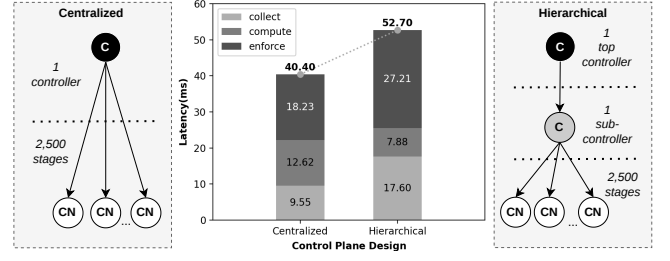


Figure 3: Average control cycle latency for centralized and hierarchical (with a single sub-controller node) designs managing 2,500 stages.

Results. We start by discussing results for the centralized control plane and then analyze the hierarchical design.

A) Centralized. Fig. 2a shows the latency for the different phases of a control cycle (*i.e.*, metric collection, PSFA computation, and rule enforcement), for an increasing number of stages: 50, 500, 1,250, and 2,500. A single controller performs well in small- to medium-scale infrastructures, achieving a control cycle latency of 40.40 ms for 2,500 stages, with latency increasing proportionally with the number of stages. Rule enforcement is the most demanding phase, followed by computation and metrics collection. A single controller is inevitably limited by scale, as its capacity is bound by the physical limits of the underlying hardware on which it is deployed. In our experiments, the node (server) where the controller is deployed can handle only 2,500 concurrent network connections, which is a limitation of Frontera's networking equipment. This value may vary, but similar constraints are inherent to other systems as well.

Observation #1. A centralized controller is suitable for small to medium HPC infrastructures where SDS solutions must react to I/O workload variations at the tens of milliseconds or larger time windows. However, it is inherently limited in scale.

B) Hierarchical. Fig. 2b shows the control cycle latency for 10,000 stages, divided into equal-sized subsets assigned across 4, 5, 10, and 20 sub-controllers. For instance, with 4 sub-controllers, each must manage 2,500 stages, meeting the hardware limit on network connections discussed above. This setup can respond to workload variations within 102 ms. As the number of sub-controllers increases, each manages fewer stages. Adding sub-controllers reduces the number of stages each one manages and parallelizes metric collection/aggregation and rule enforcement, lowering the latency of the collect and enforce phases. With 20 sub-controllers, control cycle latency is kept below 70 ms. However, this comes with a trade-off, as additional sub-controllers require extra servers and resources.

Observation #2. Hierarchical designs can scale to large systems (10,000 stages) and respond to I/O changes within hundreds of milliseconds. Adding more sub-controllers reduces latency, but it requires extra resources (nodes) for these.

Adding sub-controllers requires an additional network round-trip, as all aggregated metrics and enforcement rules must be forwarded to/from the top-level controller. This is needed because, in state-of-the-art designs, the top-level controller requires metrics

from all data plane stages to run a centralized control algorithm (PSFA in our case) to ensure global coordination across the full cluster. Fig. 3 analyzes the latency overhead of having extra network round-trips by comparing a centralized controller with a two-level hierarchical design (1 sub-controller and 1 top-level controller), both managing 2,500 stages. As expected, control cycle latency is higher in the hierarchical design (≈ 12 ms, 30% overhead), affecting mostly the collect and enforce phases. Notably, the compute phase is faster in the hierarchical setup, a consequence of metric aggregation being done at the sub-controllers.

Observation #3. State-of-the-art hierarchical control planes rely on centralized control algorithms that run at the top-level controller, requiring extra network round-trips and increasing the latency of control cycles. Leveraging the computational power of sub-controllers (e.g., for metric aggregation) allows taking advantage of spare resources at these while reducing the compute time phase at the top-level controller.

3 SHIO control plane

We propose SHIO, a hierarchical SDS control plane for managing storage QoS in large-scale HPC systems. It follows a two-level hierarchy of controllers (a *global controller* and multiple *cluster controllers*) where control logic is delegated across all levels, rather than centralized at a single point. This design allows SHIO to scale to the size of modern exascale supercomputers, enforcing QoS guarantees across thousands of compute nodes while keeping control latency low. Building on the observations and takeaways of §2, SHIO’s design is guided by the following core design principles.

Holistic and dynamic QoS control. SHIO maintains a global view of the HPC infrastructure, enabling coordinated system-wide storage policies across all jobs regardless of their placement. These policies are continuously enforced and adjusted as jobs with varying I/O demands enter and exit the system, ensuring QoS compliance under the highly dynamic and volatile nature of HPC workloads.

Scalable and dependable design. SHIO adopts a hierarchical design that scales to systems with tens to hundreds of thousands of data plane stages. Further, it incorporates fault tolerance and recovery mechanisms to ensure robustness and high availability in large-scale deployments, where controller failures must be handled transparently and with minimal impact on QoS compliance.

Control logic delegation. SHIO’s control logic is delegated across all levels of the hierarchy, enabling cluster controllers to directly manage jobs under their responsibility and reducing the burden on the global controller.

Data plane integration. SHIO’s design and interface are built to work in tandem with state-of-the-art data plane solutions [26], ensuring transparent integration with existing HPC I/O stacks.

3.1 Overview and request workflow

Fig. 4 illustrates the high-level design of SHIO. It follows a hierarchical design, comprising a top-level *global controller* and multiple sub-level *cluster controllers*, each assigned to a configurable subset (cluster) of compute nodes at the HPC system. Data plane stages are placed on each compute node, sitting between the job and the PFS

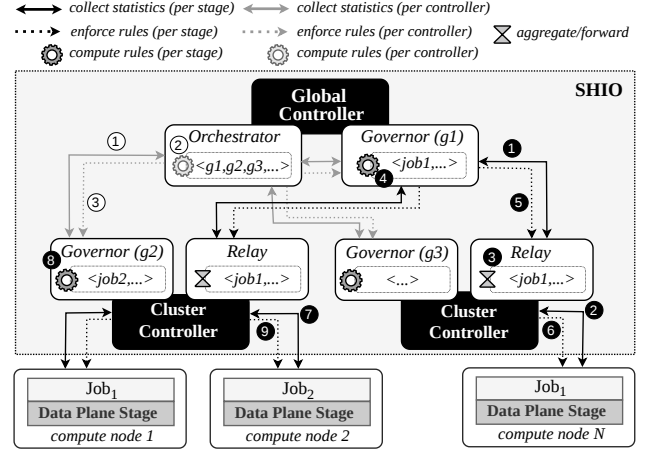


Figure 4: SHIO high-level architecture, consisting of a hierarchy of controllers (a global controller and multiple cluster controllers).

client, and transparently intercept and mediate I/O requests. Together, controllers and stages enforce storage QoS policies defined by system administrators: for instance, ensuring that the aggregate I/O submitted to the PFS never exceeds a global threshold, while fairly sharing available bandwidth across all jobs. These policies are translated by controllers into enforcement rules applied at the data plane stage(s) of each job, such as rate limits on data and metadata operations. Since a distributed job will span several compute nodes, their stages must be coordinated coherently by the control plane.

Cluster controllers. Cluster controllers serve a dual role depending on the scope of the jobs they manage. For jobs whose stages fall entirely within their assigned compute nodes, cluster controllers operate autonomously: they collect I/O metrics from the corresponding stages (⑦), compute storage rules using local control algorithms (⑧), and enforce them directly (⑨), without involving the global controller. This avoids unnecessary network round-trips and network hops on the critical path of control cycles and balances computation evenly across the hierarchy.

For distributed jobs that span compute nodes assigned across multiple cluster controllers, those controllers act as mediators: they collect and aggregate I/O metrics from their stages (②), forward them to the global controller (③), and disseminate the enforcement rules computed by the global controller back to the stages (⑥).

Global controller. The global controller assumes two complementary roles. First, it periodically and asynchronously distributes resource shares among cluster controllers in a feedback control loop (①–②–③), ensuring that system-wide storage policies are respected across the infrastructure. For instance, when enforcing a global I/O bandwidth with fair sharing, the controller distributes available bandwidth across all cluster controllers, each of which can then independently manage its jobs’ stages within the allocated share.

Second, it directly manages distributed jobs whose stages span multiple cluster controllers. In such cases, for example *job₁* in Fig. 4, cluster controllers act as metric/rule forwarders and aggregators (⑧), while the global controller runs the feedback control loop: collecting I/O metrics (①–③–②–③–①), computing enforcement rules (④), and enforcing them across those stages (⑤–⑧–⑥).

Handling dynamic job allocation. In HPC systems, jobs enter (new) and exit (finished) the system, while their compute node assignment is determined at scheduling time by a job scheduler (e.g., Slurm [48]). Consequently, SHIO dynamically determines, for each job, whether its stages fall under the responsibility of a single cluster controller (in which case the cluster controller manages them independently) or span multiple controllers (requiring the global controller's intervention). The impact of this dynamic placement on control plane performance is evaluated in §4.

3.2 Controller architecture and interfaces

SHIO's global and cluster controllers share a common internal architecture. Each controller comprises four main components: a *control center*, a *connection manager*, *sessions*, and *interfaces*.

Control center. The *control center* is the core of each controller. It maintains storage policy information, including QoS targets, shared resource limits, and job/user priorities, as well as topology information mapping jobs to the compute nodes and data plane stages they occupy. The global controller additionally tracks the set of cluster controllers. Finally, the control center hosts control applications, which implement the logic for collecting I/O metrics and computing enforcement rules based on the active storage policies.

Connection manager and sessions. The *connection manager* serves as the entry point for new connections, both from stages joining as new jobs and from controllers connecting across hierarchy levels (e.g., a cluster controller connecting to the global, or a backup controller registering with its primary). Once a connection is established, the controller creates a dedicated *session* that serves as the communication channel for all subsequent interactions.

Interfaces. Controllers expose two communication interfaces. The *northbound interface* of the global controller receives storage policy updates from system administrators, such as changes to global I/O limits or job/user priorities. Its *southbound interface* connects to cluster controllers and is used to distribute resource shares, collect I/O metrics, and forward enforcement rules to jobs managed by the global controller. Cluster controllers mirror this structure: the *northbound interface* communicates with the global controller, while the *southbound interface* collects metrics from and enforces rules at the stages deployed on their assigned compute nodes.

3.3 Control applications

A control application is a component within the control center that implements a feedback control loop: it continuously collects I/O metrics from stages, computes compliance against storage policies using a dedicated algorithm, and enforces the storage rules to satisfy those policies. SHIO introduces three control applications (Fig. 4) which together enable holistic QoS across the entire HPC system while distributing computation across all levels of the hierarchy.

Governor. The *governor* is the core control application of SHIO, running on every controller since both global and cluster controllers may directly manage stages. It implements a feedback control loop for the stages under its responsibility, and supports pluggable algorithms that can be configured by the system administrator.

SHIO supports two algorithms. The first is a *static* algorithm that assigns each job a fixed I/O limit (independently for data and

metadata operations), ensuring that shared storage resources (i.e., PFS) are never overloaded regardless of workload or system variations. The second is a proportional sharing algorithm (PSFA [26]), which dynamically adapts each job's I/O share over time based on actual usage and system load, redistributing unused resources among active jobs to avoid waste while maintaining a maximum threshold on the aggregate operations submitted to the PFS.

Relay. The *relay* is a lightweight control application deployed exclusively on cluster controllers. It acts as a proxy, aggregating I/O metrics collected from data plane stages, forwarding them to the global controller, and disseminating enforcement rules (received from the global controller) back to the corresponding stages. Unlike the *governor*, the *relay* performs no control computation; it is solely responsible for relaying information on behalf of the global controller for jobs whose stages span multiple cluster controllers.

Orchestrator. While each controller's *governor* can independently manage the jobs under its responsibility, this alone does not guarantee that system-wide storage policies are respected (e.g., global I/O at the PFS is not exceeded or jobs managed by different cluster controllers receive a fair share of resources). The *orchestrator*, deployed exclusively at the global controller, addresses this by periodically and asynchronously distributing resource shares (e.g., I/O bandwidth, metadata operations) among the *governors* of all controllers, including the global controller's own *governor*, ensuring compliance with the system-wide orchestration objectives. SHIO supports static and dynamic distribution strategies. The static approach divides resources equally among all *governors*, while keeping this allocation fixed over time (referred to as UNIFORM). For the dynamic approach, SHIO provides two algorithms: the first distributes resources proportionally to the number of stages each *governor* manages (referred to as PROPORTIONAL_UNIFORM); the second is the novel proportional sharing algorithm (PROSHA) described next.

3.4 PROSHA: Proportional demand-aware hierarchical resource sharing

PSFA [26] is a proportional sharing algorithm that dynamically distributes I/O resources among jobs based on their priority, demand, and actual usage. Each job is assigned a priority level, determined by factors such as the submitting user or historical behavior, which governs its share of the available I/O resources. When a job's actual I/O usage falls below its allocated demand, unused resources are redistributed among active jobs following a max-min fairness strategy, avoiding waste while respecting priority ordering.

While effective at the job level, PSFA, like all algorithms designed for centralized controllers, requires global state (namely, the demand and usage of every active job in the system) to make allocation decisions. In SHIO's hierarchical design, running any such algorithm centrally at the global controller would require all metrics to be aggregated at the top level before any decision can be made, reintroducing the latency overhead and centralization problem that SHIO is explicitly designed to avoid (§2).

As such, our goal is not to introduce new QoS algorithms, but instead to adapt well-established ones to operate under SHIO. PROSHA (*proportional demand-aware hierarchical resource sharing*) addresses this by shifting the granularity of resource distribution from jobs to controllers. Rather than allocating resources directly to jobs, it

Algorithm 1 Proportional Demand-aware Hierarchical Resource Sharing (PROSHA)

Require: $Max_R = N$; $Priority, Active, demand_p > 0$;
 $usage_{p,i}, nr_stages_{p,i} \geq 0$; $0 \leq \epsilon \leq 1$

```

1:  $\{usage_{0,0}, \dots, usage_{Priority-1,Active-1}\}$ ,
2:  $\{nr\_stages_{0,0}, \dots, nr\_stages_{Priority-1,Active-1}\} \leftarrow collect()$ 
3:  $left_R \leftarrow Max_R$ 
4: for  $p$  in  $[0, Priority-1]$  do
5:   for  $i$  in  $[0, Active-1]$  do
6:      $fair\_share \leftarrow \frac{left_R}{Active-i}$ 
7:      $demand\_share \leftarrow nr\_stages_{p,i} * demand_p$ 
8:     if  $usage_{p,i} \leq demand\_share$  then
9:        $threshold \leftarrow (demand\_share - usage_{p,i}) * \epsilon$ 
10:       $rate \leftarrow \min(usage_{p,i} + threshold, fair\_share)$ 
11:     else
12:       $rate \leftarrow \min(demand\_share, fair\_share)$ 
13:       $rate_i \leftarrow rate_i + rate$ 
14:       $left_R \leftarrow left_R - rate$ 
15:  $total\_usage \leftarrow \sum_{p=0}^{Priority-1} \sum_{i=0}^{Active-1} usage_{p,i}$ 
16: for  $i$  in  $[0, Active-1]$  do
17:    $usage\_proportion \leftarrow \frac{\sum_{p=0}^{Priority-1} usage_{p,i}}{total\_usage}$ 
18:    $rate_i \leftarrow rate_i + (usage\_proportion * left_R)$ 
19:  $enforce(\{rate_0, \dots, rate_{Active-1}\})$ 
20:  $sleep(loop\_interval)$ 

```

distributes resource shares among the *governors* of all active controllers, based on their aggregated job demands and actual usage. Each *governor* can then independently run PSFA for the jobs under its responsibility, within the share allocated to it, without requiring further coordination with the global controller. This decomposition preserves the fairness and efficiency properties of PSFA while eliminating the need for centralized global state.

In more detail, PROSHA operates in three phases, executed at every control cycle by the *orchestrator* at the global controller (Alg. 1).

Initialization. The algorithm begins by collecting, from all active controllers, the number of data plane stages per priority level ($nr_stages_{p,i}$) and their aggregate I/O usage ($usage_{p,i}$) (L. 1–2). The total assignable rate $left_R$ is initialized to Max_R , the maximum rate that the shared PFS can reliably sustain (L. 3).

Priority-aware allocation. PROSHA then iterates over priority levels and active controllers in order (L. 4–5), computing for each controller a *fair_share* (i.e., an equal slice of the remaining assignable rate) and a *demand_share* (i.e., the aggregate I/O demand of that controller’s active stages at the current priority level) (L. 6–7).

Lower-priority controllers are processed first, ensuring they have a fair opportunity to claim available resources before higher-priority demands are applied. For each controller, the algorithm distinguishes two cases. If the controller’s current usage falls below its *demand_share*, it may not yet be fully utilizing its allocation, which is common given the volatile nature of HPC workloads. To avoid underallocating in future cycles, the algorithm assigns a rate slightly above current usage, computed as a fraction ϵ of the gap between *demand_share* and usage, not exceeding *fair_share* (L. 8–10). If the controller’s usage meets or exceeds its *demand_share*, it is assigned the minimum of *demand_share* and *fair_share*, preventing it from exceeding its demand (L. 11–12). In both cases, the assigned rate is deducted from $left_R$ (L. 13–14).

Leftover redistribution. After all priority levels have been processed, any remaining rate in $left_R$ is redistributed among all active controllers proportionally to their aggregate usage (L. 15–18). This usage-proportional redistribution favors controllers that actively consume resources, ensuring that spare capacity is not wasted but directed to where it is most needed. Finally, the computed rates are enforced across all controllers (L. 19), and the algorithm sleeps for *loop_interval* before beginning a new control cycle (L. 20).

3.5 Control plane dependability

SHIO incorporates a fail-stop fault-tolerance model with recovery mechanisms to ensure sustained QoS despite controller failures. Importantly, control plane failure does not imply unavailability of the HPC system; in the worst case, new jobs execute without QoS enforcement, which is equivalent to most existing HPC systems that lack storage QoS guarantees altogether. Jobs already running will continue to follow their last enforced rules until they complete.

Fault model. SHIO considers a distributed system composed of a global controller and multiple cluster controllers, where any controller may experience a node-level crash fault, causing it to stop processing requests and issuing orchestration commands. We assume fail-stop behavior, meaning that once a controller crashes, it ceases operation without producing incorrect outputs or arbitrary behavior. Failures are detectable by backup controllers and the model assumes that multiple controllers may fail simultaneously, though Byzantine faults are outside the scope of this work. During a failure interval, jobs may continue executing with outdated QoS guarantees until the backup controller takes over.

Primary-backup design. SHIO adopts a primary-backup design in which each controller is paired with a dedicated backup that continuously monitors it via heartbeats. Upon detecting a failure, the backup promotes itself to primary and resumes operation. To ensure transparency, the backup assumes the failed primary’s network address, making the failover invisible to the rest of the system. This design applies uniformly to both the global and cluster controllers. Moreover, given the high-end hardware and stability of HPC infrastructures, we assume that a simple failure detection mechanism is sufficient for our fault model, avoiding phenomena such as *split-brain*. Nonetheless, handling more complex failure scenarios is orthogonal to this work, and our design is compatible with alternative failure detection strategies. For instance, if the backup replica fails, an external daemon could automatically restart or replace it. We defer this to future work.

Global controller failures. When the backup for the global controller detects a failure, it faces two challenges. First, it must take over the failed primary’s network address to maintain transparency toward the cluster controllers previously connected to it. Second, it must reconstruct the system state; specifically, which cluster controllers, jobs, and data plane stages the failed primary was managing. Under normal operation, the primary keeps the backup up to date by notifying it of every new cluster controller connection, acknowledged only after the backup registers it. The remaining state (i.e., the jobs and stages under direct management) is reconstructed at takeover time through a collection step in which the backup queries all connected cluster controllers to determine which jobs are currently executing and fall under its responsibility.

Cluster controller failures. Failover for cluster controllers follows the same principles. Each cluster controller is paired with a backup that monitors it via heartbeats and promotes itself upon detecting a failure. The backup assumes both the upstream network address (toward the global controller) and the downstream network address (toward stages) of the failed primary, keeping all other components unaware of the replacement. System topology is similarly reconstructed via a collection step at takeover time.

It is important to remark that the recovery of global and cluster controllers requires synchronizing light state (*i.e.*, direct connections with other peers maintained by the failed controller). The remaining, heavier state (*i.e.*, full system topology, policies, jobs, stages) is recovered from a collection operation to the peers. During this step, controllers perform regular collect operations, thus providing the same scalability guarantees as under normal operation.

3.6 Implementation and deployment

We implemented a prototype of SHIO with $\approx 16K$ lines of C++ code³. Communication between controllers and between the control plane and the data plane stages is established through RPC calls, using the gRPC framework (v1.67.0) [17].

Data plane integration. SHIO integrates with the PADLL data plane [26], which deploys stages that transparently intercept POSIX calls via LD_PRELOAD that enforce rate limiting rules defined by the control plane before forwarding requests to the PFS.

System deployment and bootstrapping. SHIO controllers are deployed on dedicated servers co-located with other HPC management services such as job schedulers and monitoring systems. Data plane stages are deployed directly on compute nodes alongside the jobs they serve, where they intercept and mediate I/O operations.

Operational workflow. System administrators configure the global controller with QoS constraints (such as maximum throughput limits for data and metadata operations at the production PFS), and control application settings, including resource demands per priority level. Cluster controllers receive control application settings at initialization and inherit QoS constraints from the global controller upon connecting to it. Data plane stages are deployed alongside each job and assigned to a cluster controller at scheduling time.

4 Evaluation

Our evaluation seeks to assess the following questions:

- Can SHIO ensure QoS for large-scale clusters, with 10,000 nodes, under realistic HPC I/O workloads (§4.1)?
- How does PROSHA compare with other baselines (§4.1)?
- How does SHIO compare with state-of-the-art (SOTA) hierarchical designs that fully delegate coordination to the global controller (§4.2)?
- Can SHIO scale to even larger clusters (up to 100,000 nodes) under stress-testing workloads (§4.2)?
- How quickly can SHIO recover from failures, and what impact does it have on QoS guarantees (§4.3)?

Experimental testbed. All experiments were conducted on multiple compute nodes of the Frontera supercomputer [41], each equipped with two 28-core Intel Xeon processors, 192 GiB of RAM,

a single 240 GiB SSD, and a Mellanox InfiniBand HDR-100 network card, running CentOS 7.9 with the Linux kernel v3.10.

Collected metrics and workloads. Resource utilization metrics, including CPU, memory, and network usage, are collected using Remora [38]. Experiments run under two workload configurations.

Configuration A. Evaluates SHIO under settings representative of a production environment. *Governors* execute control cycles (*i.e.*, collect, compute, and enforce phases) every 1 second, and the *orchestrator* runs every 5 seconds, using one of the three supported algorithms (*i.e.*, UNIFORM, PROPORTIONAL UNIFORM, PROSHA).

We consider two I/O workloads. The first workload (**A1**) models a realistic HPC environment by reproducing the jobs and workload patterns commonly found in production supercomputers. To devise such a workload, we conducted three main steps. First, we analyzed job traces collected over 1 year from Frontera and characterized them by job size (number of compute nodes) and job duration. Second, we collected I/O traces from four applications representative of supercomputer workloads, namely two scientific (GROMACS [2] and OpenFOAM [31]) and two AI training workloads (PyTorch [32] training ResNet [18] and TensorFlow [1] training ShuffleNet [49] on the ImageNet dataset [11]). Finally, we combined the job and I/O traces to construct a workload that emulates Frontera's production environment, covering both single-node and multi-node workloads (up to 1,024 nodes per job). The second workload (**A2**) comprises jobs that continuously issue I/O operations at a fixed user-defined rate, emulating persistent PFS saturation.

In both cases, I/O operations are replayed through a modified version of PADLL [26] that accurately simulates I/O operation frequency without issuing actual requests to the PFS. Since prior work has shown that PADLL incurs low performance overhead, we do not re-evaluate it. This approach ensures deterministic and reproducible experimental runs across all setups, while accurately capturing the I/O metrics (*i.e.*, data and metadata IOPS) observed by the control plane. More importantly, it avoids issuing real I/O requests to the production PFS, preventing interference with other users and ensuring that our experiments do not disrupt the supercomputer's normal operation.

Configuration B. Evaluates the performance and scalability of SHIO under a stress-testing scenario. The control plane continuously executes control cycles without interruption, while the data plane stages generate randomly produced metrics. Despite the synthetic nature of the metrics, each control cycle still requires collecting metrics from all data plane stages (*i.e.*, data and metadata IOPS), computing the control algorithms, and enforcing rules across stages, faithfully stressing the control plane at scale.

Deployment. The global and cluster controllers are deployed on dedicated compute nodes of Frontera. Since exclusive access to a supercomputer at the scale of tens to hundreds of thousands of compute nodes is not feasible, we simulate larger deployments by co-locating multiple data plane stages on each physical compute node (*i.e.*, 50 stages per node). Each stage instance operates independently, as if deployed on a distinct compute node, reproducing the logical behavior of a large-scale infrastructure. For instance, our experiments simulating 100,000 nodes allocate 2,000 physical compute nodes from Frontera, each hosting 50 stages. To ensure the validity of our results, we verified that this co-location does not

³SHIO is publicly available at <https://github.com/dsrhaslab/shio>.

Table 2: Average delay per job type (GROMACS, OpenFOAM, ResNet, and ShuffleNet) across the evaluated setups: Static, Uniform, Proportional_Uniform, Prosha, and Priority_Prosha. Results compare expected job execution times with measured runtimes.

Job Type	Static	Uniform	Prop. Uniform	Prosha	Priority Prosha
GROMACS [2]	73.75%	8.17%	3.60%	1.80%	1.55%
OpenFOAM [31]	31.21%	4.83%	4.47%	4.02%	0.43%
ResNet [18]	109.70%	6.15%	5.09%	1.45%	1.29%
ShuffleNet [49]	242.35%	111.01%	69.61%	34.88%	19.51%

saturate the node’s resources, thus avoiding interference with the control plane measurements and the overall experimental results.

4.1 Large-scale QoS policy comparison

We start by showing SHIO’s ability to enforce global QoS storage policies under realistic HPC workloads using *Configuration A (A1)* on a cluster with 10,000 stages. We focus on managing storage data requests (*i.e.*, `read()`, `write()`) and set Max_R to 190 GiB/s, representing the system-wide limit for data operations that the PFS can sustainably handle across concurrent jobs, which we define as our QoS goal.

Experimental setup. Experiments are conducted on a system simulating 10,000 nodes, with stages deployed across 200 physical compute nodes. These are managed by a global controller and four cluster controllers, each of the latter connected to 2,500 stages. We consider six setups. In Baseline, jobs execute without any QoS enforcement, reflecting the default behavior of most supercomputers. In Static, all *governors* assign each stage a fixed rate of 19.5 MiB/s (derived from evenly splitting Max_R across all stages) without the *orchestrator* intervention. This represents a decentralized approach to QoS enforcement that requires no global controller.

For the remaining four setups, *governors* use PSFA, while the *orchestrator* applies different resource distribution algorithms, allowing us to isolate the impact of each approach on overall QoS.

For Uniform, the *orchestrator* distributes Max_R among the five *governors*, assigning each a fixed rate of 38 GiB/s. In Proportional_Uniform, each *governor* receives a share proportional to the number of stages it manages (*i.e.*, $19.5 \text{ MiB/s} \times nr_stages$). In Prosha, the *orchestrator* applies PROSHA with uniform job priorities, assigning each stage a demand of 19.5 MiB/s, but dynamically redistributes unused I/O resources over time (§3.4). Finally, Priority_Prosha also applies PROSHA but assigns higher priority to ShuffleNet jobs (specifically, a $5\times$ demand boost to 97.5 MiB/s). We prioritize ShuffleNet workloads because, as shown in Table 2, it is the most heavily throttled job type, a consequence of producing a considerably larger volume of data operations in the replayed traces.

Results. Fig. 5 depicts the aggregate I/O bandwidth usage for each setup over a 90-minute experiment, while Table 2 reports the average runtime delay per job type relative to Baseline.

In Baseline, throughput exhibits significant variability and frequent peaks exceeding the established 190 GiB/s system-wide PFS limit, resulting in periods of both high throughput and underutilization, and confirming the need for storage QoS guarantees. Static eliminates these bursts and enforces a stable utilization pattern,

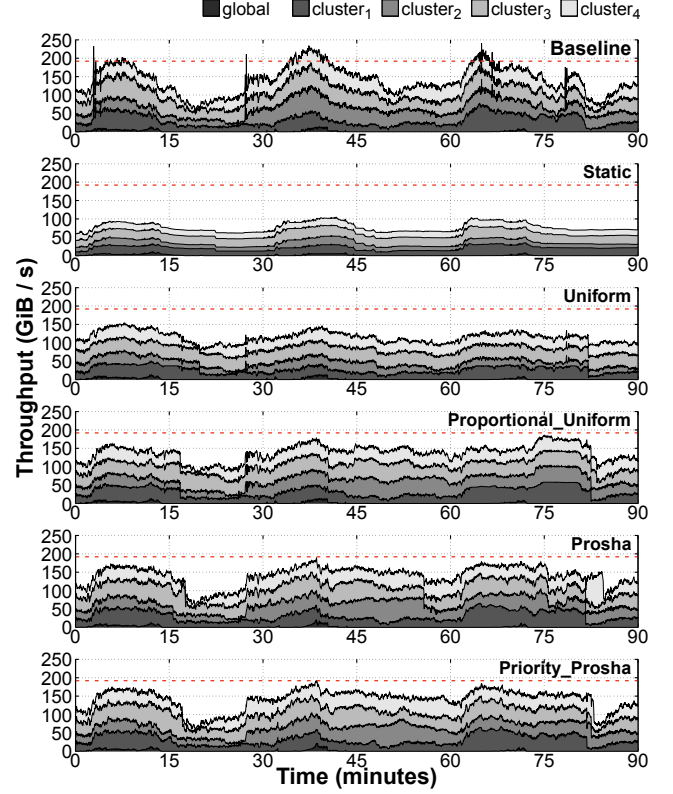


Figure 5: I/O throughput collected at each controller’s governor over a 90-minute experiment for the evaluated setups: Baseline, Static, Uniform, Proportional_Uniform, Prosha, and Priority_Prosha. The red line indicates the system-wide limit (190 GiB/s).

but at the cost of over-throttling jobs whose actual demand is well below their assigned limit, resulting in average runtime delays ranging from $\approx 31\%$ to $\approx 242\%$ across job types. Uniform reduces these delays but still throttles jobs more aggressively than needed, keeping system-wide utilization below the configured threshold. Proportional_Uniform improves resource utilization further, yet jobs remain unnecessarily constrained at times, as observed between the 80 and 90 minute marks, where active jobs are throttled despite having available bandwidth in the system. In both cases, the lack of demand-aware allocation leads to suboptimal resource use, delaying job execution by up to $\approx 111\%$.

Prosha achieves a better balance, maintaining high overall utilization while substantially reducing variability (compared to Baseline) through dynamic allocation adaptation based on demand and actual I/O usage, reducing both over- and under-provisioning in most cases. However, ShuffleNet still experiences significant delays ($\approx 35\%$), as its demand is not differentiated from that of less intensive jobs. Priority_Prosha addresses this directly: by assigning higher priority to ShuffleNet, the I/O backlog visible in the 80-90 minute window is reduced, and overall job delays decrease across all job types, with ShuffleNet delays dropping to $\approx 20\%$.

Overall, these results show a clear progression: Static and Uniform achieve stable throughput at the cost of excessive, demand-agnostic throttling (delays up to $\approx 242\%$). Proportional_Uniform

improves utilization but still throttles jobs unnecessarily, whereas Prosha substantially reduces these inefficiencies through demand-aware allocation, though intensive workloads like ShuffleNet remain underserved ($\approx 35\%$ delay). Priority_Prosha resolves this by prioritizing such workloads, cutting ShuffleNet's delay to $\approx 20\%$. Together, these results confirm that SHIO's hierarchical design, combined with PROSHA and priority-aware resource distribution, achieves effective and adaptive QoS enforcement at scale.

4.2 Scalability and job placement

We now evaluate SHIO's scalability under stress-testing (*Configuration B*), evaluating the effect of progressively delegating control responsibility from the global controller to the cluster controllers, and the impact of increasing system scale up to 100,000 nodes. *Governors* and the *orchestrator* use the PSFA and PROSHA control algorithms, respectively. Moreover, control cycle latency is the primary metric, measuring the time a controller takes to collect metrics, compute adjustments, and enforce rules (*i.e.*, how quickly the control plane can react to workload variations). We report latency for all control applications: *orchestrator* and *governor* at the global controller (Fig. 6) and *governor* and *relay* at the cluster controllers (Fig. 7).

Experimental setup. We evaluate SHIO under 2 axes: *control responsibility distribution* and *system size*, as depicted in Table 3. For *control responsibility distribution*, we consider 5 setups. SOTA corresponds to the state-of-the-art design of hierarchical control planes, where all stages are coordinated by the global controller and cluster controllers serve only as relays [26]. We then progressively transfer stage management responsibility to the cluster controllers (for instance, in 75-25, 75% of stages are managed by the global controller and 25% are evenly handled by cluster controllers) up to 0-100, where all stages are managed exclusively by cluster controllers. These experiments allow validating scenarios in which cross-cluster jobs are more or less prominent: in 0-100, each job's stages are contained within the compute nodes of a single cluster controller, eliminating the need for global controller intervention.

We evaluate these distributions for *system sizes* ranging from 10,000 to 100,000 stages (each compute node runs 50 stages, resulting in 200 to 2,000 physical nodes). As the system grows, so does the number of cluster controllers, with each fixed at 2,500 stages.

Results. Fig. 6 and Fig. 7 illustrate how the different setups and system sizes affect the average control cycle latency, broken down by control phase (*i.e.*, collect, compute, enforce).

SOTA. When all stages are coordinated by the global controller, control latency is dominated by the global controller's *governor* (Fig. 6b), which includes the time it takes cluster controllers to forward metrics and enforcement rules through their *relay* applications (Fig. 7b). As expected, *governor* latency increases with the number of coordinated stages, driven by the growing volume of collected metrics and the higher computational complexity of the control algorithm. Control latency ranges from 111.4 ms with 10,000 nodes to 952 ms with 100,000 nodes. Since each cluster controller is always connected to 2,500 stages, regardless of system size, relay latency remains stable at ≈ 42 ms across all configurations.

0-100. When control logic is fully delegated to cluster controllers, latency is measurable at the *orchestrator* at the global controller (Fig. 6a) and the cluster *governors* (Fig. 7a). As the system grows,

Table 3: Number of stages assigned to global and cluster controllers across systems of 10K, 20K, 50K, and 100K stages, under five control responsibility distribution setups (SOTA, 75-25, 50-50, 25-75, 0-100).

#Stages (#Physical Nodes)	Control Layer	Control Responsibility Setup				
		SOTA	75-25	50-50	25-75	0-100
10K (200)	global	10K	7.5K	5K	2.5K	0
	clusters ($\times 4$)	0	2.5K	5K	7.5K	10K
	per cluster	0	625	1250	1875	2.5K
20K (400)	global	20K	15K	10K	5K	0
	clusters ($\times 8$)	0	5K	10K	15K	20K
	per cluster	0	625	1250	1875	2.5K
50K (1000)	global	50K	37.5K	25K	12.5K	0
	clusters ($\times 20$)	0	12.5K	25K	37.5K	50K
	per cluster	0	625	1250	1875	2.5K
100K (2000)	global	100K	75K	50K	25K	0
	clusters ($\times 40$)	0	25K	50K	75K	100K
	per cluster	0	625	1250	1875	2.5K

the *orchestrator* manages more cluster controllers (*i.e.*, collects and computes more metrics), and its latency increases slightly, reaching 8.5 ms at 100,000 stages. *Governor* latency at the cluster controllers remains stable at ≈ 55 ms across all system sizes, since each cluster controller always manages the same number of stages (2,500). Importantly, in this setup, the control cycle latency is determined solely by the cluster controller *governor*, as the orchestrator operates asynchronously and does not lie on the critical path. This yields a control latency of 55.4 ms for a system of 100,000 stages.

Intermediate setups. In the 75-25, 50-50, and 25-75 configurations, control cycle latency is jointly determined by the *governors* at both the global and cluster controllers. The *orchestrator* maintains its asynchronous behavior, and *relay* latency is subsumed within the global controller's *governor* control cycle. As more stages are delegated to cluster controllers, the global's *governor* latency decreases, from 733.5 ms (75-25) to 205.6 ms (25-75) at 100,000 stages, while cluster *governors* take on more responsibility and their latency increases accordingly, from 13.3 ms (75-25) to 41 ms (25-75). Relay latency decreases as fewer metrics and rules are forwarded to and from the global controller, from 42.1 ms (75-25) to 20 ms (25-75).

A trend worth noting is that, for intermediate setups, *governor* latency at cluster controllers slightly decreases as system size increases. In 75-25, it drops from 17.9 ms at 10,000 stages to 13.3 ms at 100,000 stages. This is a consequence of the *orchestrator* polling behavior: as the system grows and control cycles become longer, the *orchestrator* polls cluster controllers less frequently, reducing synchronization overhead at those controllers. This effect diminishes as more control logic is delegated to cluster controllers, since the *orchestrator*'s polling represents a smaller fraction of the overall control cycle. In the 0-100 setup, where the orchestrator never intervenes in stage management, this trend disappears entirely.

Takeaway. When all control logic is centralized at the global controller, as in state-of-the-art designs, SHIO can intervene across 100,000 stages under 1 second. When control logic is fully delegated to cluster controllers, SHIO reduces this to 55.4 ms, a $17\times$ improvement. In intermediate configurations, control latency for 100,000 stages ranges from 205.6 ms (25-75) to 733.5 ms (75-25), reflecting the trade-off between the degree of delegation and the prevalence of cross-cluster jobs in the workload.

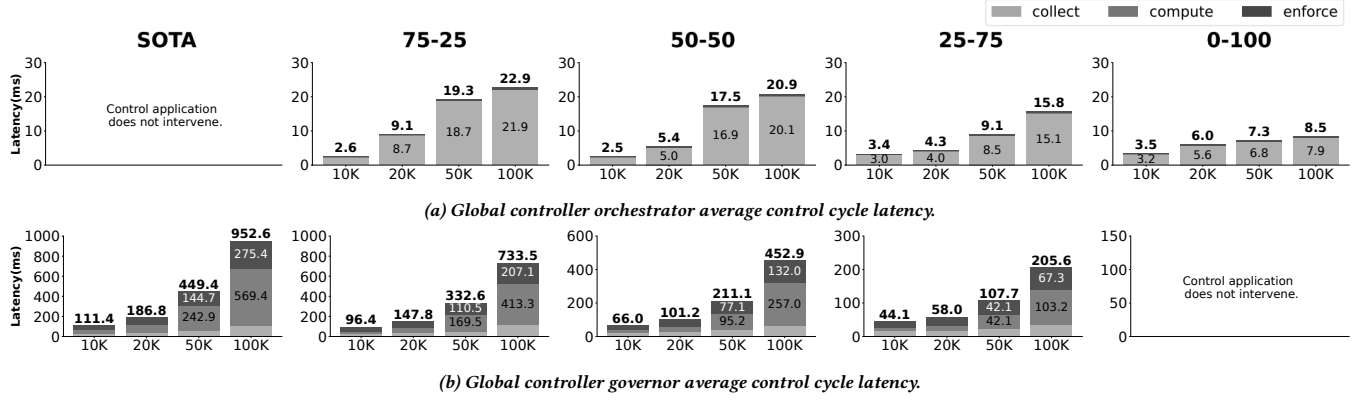


Figure 6: Average control cycle latency at the global controller for 10,000, 20,000, 50,000, and 100,000 stages under five control responsibility distribution setups (SOTA, 75-25, 50-50, 25-75, 0-100). Results are reported for (a) orchestrator and (b) governor control applications.

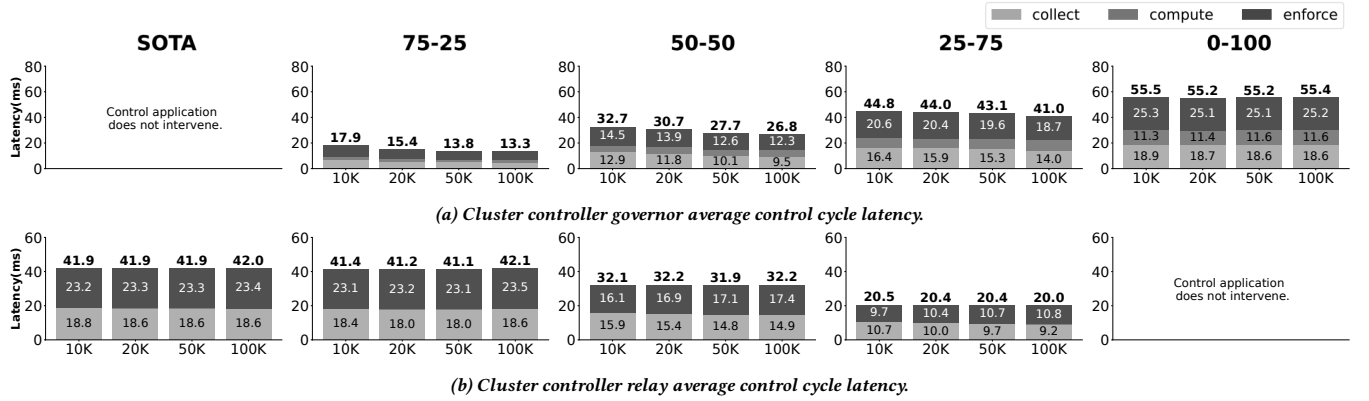


Figure 7: Average control cycle latency at a cluster controller for 10,000, 20,000, 50,000, and 100,000 stages under five control responsibility distribution setups (SOTA, 75-25, 50-50, 25-75, 0-100). Results are reported for (a) governor and (b) relay control applications.

Table 4: Resource utilization (min-max) for CPU, memory, and network at the global and cluster controllers.

Type	Size	CPU (%)	Mem (MB)	Tx (MB/s)	Rx (MB/s)
Global	10K	0.05–1.50	40.9–92.1	0.06–1.13	0.05–0.86
	20K	1.52–2.05	51.2–143.3	0.24–1.41	0.19–1.08
	50K	1.83–2.37	60.7–296.9	0.06–1.46	0.05–1.12
	100K	1.92–2.48	122.8–542.7	0.24–1.26	0.20–0.96
Cluster	10K	7.45–19.21	286.7–337.9	3.26–6.08	2.20–4.91
	20K	4.64–17.98	133.1–358.4	1.50–5.51	1.33–4.70
	50K	1.91–15.60	296.9–399.3	0.97–4.79	0.68–4.22
	100K	0.92–12.19	276.4–440.3	0.32–3.27	0.32–2.97

Resource usage. Table 4 summarizes CPU, memory, and network bandwidth usage of SHIO’s controllers across different system sizes, reporting the minimum and maximum values observed during the scalability and job placement experiments. Overall, SHIO imposes a modest resource footprint, with CPU utilization peaking at $\approx 19\%$ at cluster controllers, memory consumption remaining below 600 MB across all configurations, and network bandwidth staying under 7 MB/s and 5 MB/s for transmitted and received data, respectively.

Moreover, a notable trend is that CPU and network usage at cluster controllers counterintuitively decrease as system size grows. This is because larger systems have longer control cycles, so controllers run fewer cycles within the same time frame, reducing the frequency of metric collection, computation, and rule enforcement.

At cluster controllers, this effect is primarily driven by the *orchestrator*’s polling behavior: although each cluster controller manages a fixed number of stages and its control cycle is largely unaffected by system scaling, the orchestrator polls them less frequently as the system grows, reducing the associated communication overhead.

4.3 Controller failure recovery

When a controller fails, jobs continue to execute under potentially outdated storage rules, which can compromise QoS compliance. We evaluate the resilience of SHIO’s primary-backup design by assessing the impact of one or more controller failures on QoS compliance under realistic job workloads (A1) and sustained PFS saturation (A2), using *Configuration A*. Experiments are conducted on a large-scale cluster of 10,000 stages, enforcing a system-wide throughput limit of 190 GiB/s (Max_R), with PSEA and PROSHA running at the *governors* and *orchestrator*, respectively.

Experimental setup. For each scenario, we compare two setups: *Baseline* and *With Faults*, representing fault-free and faulty conditions. We test multiple combinations of controller failures, terminating a set of controllers every 3 minutes in the following progressive sequence: (1) a single cluster controller, (2) the global controller, (3) all cluster controllers, (4) the global controller together with a single cluster controller, and (5) all controllers simultaneously. Each primary controller is paired with a dedicated backup, and upon

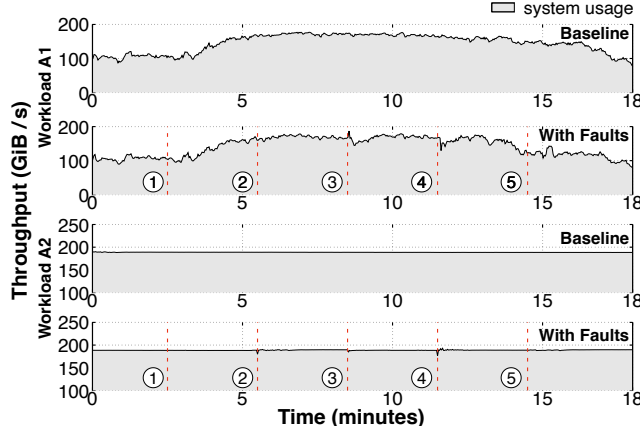


Figure 8: Throughput for workloads A1 and A2 under fault-free (Baseline) and faulty scenarios (With Faults). Failure events are represented by a number and marked with a red vertical line.

each promotion (*i.e.*, fault of the primary), a new backup is launched to maintain continuous dependability throughout the experiment.

Results. Fig. 8 depicts the throughput under each failure combination, marked with ① through ⑤. Across all testing scenarios, SHIO recovers quickly with minimal impact on QoS requirements, as backup controllers resume normal operation within 1 to 2 seconds of detecting a primary failure. During this short recovery window, data plane stages continue enforcing the last rules issued by the failed controller, preventing abrupt throughput disruption and maintaining the system within a safe operating region until the backup assumes control. Isolated failures of a single cluster (①) or the global controller (②) have a negligible impact on throughput, remaining virtually undetectable in the traces. Performance deviations become more apparent when all cluster controllers fail simultaneously (③), and are further compounded when the global controller fails alongside a cluster controller (④). In the most extreme case, where all controllers are terminated at once (⑤), concurrent recovery processes introduce transient coordination overhead before normal operation fully resumes, resulting in the most visible (yet still short-lived) disruption to QoS compliance. Overall, workload A2 is always reaching its QoS limit (190 GiB/s), making injected faults almost imperceptible, resulting in the workload’s throughput appearing as a nearly continuous line on the plot. For workload A1, since it has more headroom below the QoS limit, small throughput drops are noticeable but the impact remains almost negligible.

5 Related work

SHIO targets the scalability and dependability of SDS control planes that enforce dynamic storage QoS across large-scale infrastructures. We position it against prior work along four dimensions.

Centralized designs. IOFlow [42] introduces a centralized control plane designed for small-to-medium data centers (*i.e.*, tens to hundreds of nodes), deferring the exploration of scaling for larger data centers as orthogonal work. Crystal [16] implements a flat design in which the control plane comprises multiple controllers, each responsible for a distinct set of applications and storage policies. Controllers can be dynamically provisioned or removed as new applications and policies enter or leave the infrastructure. Since

each controller manages independent storage policies, none require system-wide visibility or coordination among themselves. Further, Crystal’s control plane is only evaluated with tens of stages. In contrast, SHIO provides system-wide, holistic control across all jobs and controllers, while scaling to hundreds of thousands of stages.

Hierarchical designs. Prior systems, including CLARISSE [21], SIREN [22], and Cheferd [26], propose hierarchical control planes specifically designed for HPC infrastructures, but none are validated beyond 1,024 stages. Across all these designs, control computation is centralized at the top-level controller, which becomes a bottleneck as the system grows. SHIO differs from these by distributing control computation across all levels of the hierarchy, reducing control latency and enabling scalability to hundreds of thousands of stages.

Dependable designs. Existing works largely treat controller dependability as an afterthought; some propose a conservative default policy when a controller becomes unavailable [42], while others suggest replicating the controller using coordination protocols [40], but none actually implement or evaluate these mechanisms [16, 22, 40, 42]. SHIO introduces a simple yet effective primary-backup design that is fully implemented and evaluated under multiple simultaneous failure scenarios, demonstrating rapid recovery with minimal impact on QoS compliance.

Cluster-level management. SHIO complements cluster schedulers such as Borg [43] and Slurm [48]. These schedulers allocate resources once, at admission time, deciding which jobs run on which nodes. SHIO, instead, runs continuously post-scheduling, dynamically controlling jobs’ I/O rate to a shared storage backend.

6 Conclusion

We presented SHIO, a hierarchical storage control middleware that delivers system-wide QoS for large-scale HPC systems. By distributing control logic across all levels of the hierarchy, it enables decentralized decision-making to improve scalability and control latency, reducing reliance on a central controller and improving responsiveness to the dynamic, bursty nature of HPC workloads. SHIO is the first SDS control plane to enable and validate storage QoS enforcement at the scale of modern exascale supercomputers. Our large-scale evaluation, conducted on a real HPC system, shows that SHIO successfully coordinates up to 100,000 data plane stages, reducing control cycle latency from ≈ 1 second to ≈ 55 ms, a $17\times$ improvement over state-of-the-art designs. These results establish that a scalable, dependable SDS control plane is not only feasible but practical for modern and next-generation HPC infrastructures.

Acknowledgments

We thank our shepherd, Timothy Wood, and the anonymous reviewers for their insightful feedback. This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (<https://doi.org/10.54499/UID/50014/2025>) (Mariana Miranda and João Paulo) and PhD Fellowship PD/BD/151403/2021 (Mariana Miranda). It has been carried out within the scope of the projects BCD.S+M, reference 14436 (NORTE2030-FEDER-00584600), CDMS, reference 17409 (COMPETE2030-FEDER-01193000), and the WISE Extra Exploratory Research Project (UTA18-001217; 2024.14682.UTA) from the UT Austin Portugal Program.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, USA) (OSDI '16). USENIX Association, Berkeley, CA, USA, 265–283. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>
- [2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1–2 (Sept. 2015), 19–25. doi:10.1016/j.softx.2015.06.001
- [3] Barcelona Supercomputing Center. 2026. MareNostrum 5. <https://www.bsc.es/marenostrum/marenostrum-5>. Accessed June 4, 2026.
- [4] BeeGFS. 2026. BeeGFS: The Leading Parallel Cluster File System. <https://www.beegef.io/c/>. Accessed June 4, 2026.
- [5] Peter Braam. 2019. The Lustre Storage Architecture. arXiv:1903.01955
- [6] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. 2018. Characterizing Deep-Learning I/O Workloads in TensorFlow. In *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems* (Dallas, TX, USA) (PDSW-DISCs '18). IEEE, Piscataway, NJ, USA, 54–63. doi:10.1109/PDSW-DISCs.2018.00011
- [7] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) (ICPP '19). Association for Computing Machinery, New York, NY, USA, Article 80, 10 pages. doi:10.1145/3337821.3337902
- [8] Christopher S. Daley, Devarshi Ghoshal, Glenn K. Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas J. Wright. 2020. Performance characterization of scientific workflows for the optimal use of Burst Buffers. *Future Generation Computer Systems* 110 (Sept. 2020), 468–480. doi:10.1016/j.future.2017.12.022
- [9] Marco Dantas, Diogo Leitão, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and João Paulo. 2022. Accelerating Deep Learning Training Through Transparent Storage Tiering. In *Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing* (Taormina, Italy) (CCGrid '22). IEEE, Piscataway, NJ, USA, 21–30. doi:10.1109/CCGrid54584.2022.00011
- [10] DAOS. 2026. DAOS: Distributed Asynchronous Object Storage. <https://daos.io/>. Accessed June 4, 2026.
- [11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Miami, FL, USA) (CVPR '09). IEEE, Piscataway, NJ, USA, 248–255. doi:10.1109/CVPR.2009.5206848
- [12] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. 2014. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium* (Phoenix, AZ, USA) (IPDPS '14). IEEE, Piscataway, NJ, USA, 155–164. doi:10.1109/IPDPS.2014.27
- [13] Fujitsu. 2026. Supercomputer Fugaku. <https://global.fujitsu/en-global/technology/research/fugaku>. Accessed June 4, 2026.
- [14] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. 2024. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 96, 13 pages. doi:10.1145/3597503.3639232
- [15] Raúl Gracia-Tinedo, Pedro García-López, Marc Sánchez-Artigas, Josep Sampé, Yosef Moatti, Eran Rom, Dalit Naor, Ramon Nou, Toni Cortés, William Oppermann, and Pietro Michiardi. 2016. IOStack: Software-Defined Object Storage. *IEEE Internet Computing* 20, 3 (May 2016), 10–18. doi:10.1109/MIC.2016.46
- [16] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. 2017. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '17). USENIX Association, Berkeley, CA, USA, 243–256. <https://www.usenix.org/conference/fast17/technical-sessions/presentation/gracia-tinedo>
- [17] gRPC. 2026. gRPC: A high performance, open source universal RPC framework. <https://grpc.io/>. Accessed June 4, 2026.
- [18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV, USA) (CVPR '16). IEEE, Piscataway, NJ, USA, 770–778. doi:10.1109/CVPR.2016.90
- [19] Lei Huang and Si Liu. 2020. OOPS: An Innovative Tool for IO Workload Management on Supercomputers. In *Proceedings of the 26th IEEE International Conference on Parallel and Distributed Systems* (Hong Kong) (ICPADS '20). IEEE, Piscataway, NJ, USA, 486–493. doi:10.1109/ICPADS51040.2020.00069
- [20] IBM. 2026. IBM Storage Scale. <https://www.ibm.com/products/storage-scale>. Accessed June 4, 2026.
- [21] Florin Isaila, Jesus Carretero, and Rob Ross. 2016. CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (Cartagena, Colombia) (CCGrid '16). IEEE, Piscataway, NJ, USA, 346–355. doi:10.1109/CCGrid.2016.24
- [22] Suman Karki, Bao Nguyen, Joshua Feener, Kei Davis, and Xuechen Zhang. 2018. Enforcing End-to-End I/O Policies for Scientific Workflows Using Software-Defined Storage Resource Enclaves. *IEEE Transactions on Multi-Scale Computing Systems* 4, 4 (Oct. 2018), 662–675. doi:10.1109/TMSCS.2018.2879096
- [23] George Kelantonakis, Fallia Kourou, and Kostas Magoutis. 2025. Proportional Fairness and Isolation for Serverless Applications over FaaS Platforms. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering* (Toronto, ON, Canada) (ICPE '25). Association for Computing Machinery, New York, NY, USA, 58–68. doi:10.1145/3676151.3719363
- [24] Lawrence Livermore National Laboratory. 2026. El Capitan Supercomputer. <https://hpc.llnl.gov/hardware/compute-platforms/el-capitan>. Accessed October 3, 2026.
- [25] Si Liu, Lei Huang, Hang Liu, Amit Ruhela, Virginia Trueheart, Susan Lindsey, and Quan Yuan. 2021. Practice Guideline for Heavy I/O Workloads with Lustre File Systems on TACC Supercomputers. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions* (Boston, MA, USA) (PEARC '21). Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. doi:10.1145/3437359.3465570
- [26] Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, José Pereira, and João Paulo. 2023. Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control. In *Proceedings of the 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing* (Bangalore, India) (CCGrid '23). IEEE, Piscataway, NJ, USA, 47–61. doi:10.1109/CCGrid57682.2023.00015
- [27] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2020. A Survey and Classification of Software-Defined Storage Systems. *Comput. Surveys* 53, 3, Article 48 (May 2020), 38 pages. doi:10.1145/3385896
- [28] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. 2022. PAIO: General, Portable I/O Optimizations With Minor Application Modifications. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '22). USENIX Association, Berkeley, CA, USA, 413–428. <https://www.usenix.org/conference/fast22/presentation/macedo>
- [29] National Supercomputing Centre in Shenzhen. 2026. LineShine Supercomputer. <https://www.nscsz.cn/>. Accessed September 15, 2026.
- [30] Oak Ridge National Laboratory. 2026. Frontier Supercomputer. <https://www.olcf.ornl.gov/frontier/>. Accessed June 4, 2026.
- [31] OpenFOAM. 2026. OpenFOAM: The Open Source Computational Fluid Dynamics Toolbox. <https://www.openfoam.org>. Accessed June 4, 2026.
- [32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems* 32 (Vancouver, BC, Canada) (NeurIPS '19). Curran Associates, Inc., Red Hook, NY, USA, 8024–8035. https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html
- [33] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. 2019. Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) (SC '19). Association for Computing Machinery, New York, NY, USA, Article 65, 13 pages. doi:10.1145/3295500.3356183
- [34] Tirthak Patel, Rohan Garg, and Devesh Tiwari. 2020. GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '20). USENIX Association, Berkeley, CA, USA, 103–119. <https://www.usenix.org/conference/fast20/presentation/patel-gift>
- [35] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. 2017. SafeFS: A modular architecture for secure user-space file systems: One FUSE to rule them all. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) (SYSTOR '17). Association for Computing Machinery, New York, NY, USA, 12 pages. doi:10.1145/3078468.3078480
- [36] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 2017. A Configurable Rule based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *Proceedings of the*

- International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) (SC '17). Association for Computing Machinery, New York, NY, USA, 12 pages. doi:10.1145/3126908.3126932
- [37] RIKEN Center for Computational Science. 2026. HPL-AI. <https://www.r-ccs.riken.jp/labs/lpncrt/projects/hpl-ai/index.html>. Accessed June 13, 2026.
- [38] Carlos Rosales, Antonio Gómez-Iglesias, and Andrew Predoehl. 2015. Remora: a resource monitoring tool for everyone. In *Proceedings of the 2nd International Workshop on HPC User Support Tools* (Austin, TX, USA) (HUST '15). Association for Computing Machinery, New York, NY, USA, 8 pages. doi:10.1145/2834996.2834999
- [39] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (Hollywood, CA, USA) (OSDI '12). USENIX Association, Berkeley, CA, USA, 349–362. <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/shue>
- [40] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. 2016. sRoute: Treating the Storage Stack Like a Network. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '16). USENIX Association, Berkeley, CA, USA, 197–212. <https://www.usenix.org/conference/fast16/technical-sessions/presentation/stefanovici>
- [41] Texas Advanced Computing Center. 2026. Frontera Supercomputer. <https://www.tacc.utexas.edu/systems/frontera>. Accessed June 4, 2026.
- [42] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, PA, USA) (SOSP '13). Association for Computing Machinery, New York, NY, USA, 182–196. doi:10.1145/2517349.2522723
- [43] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. doi:10.1145/3342195.3387517
- [44] TOP500. 2026. The LINPACK Benchmark. <https://www.top500.org/project/linpack/>. Accessed September 15, 2026.
- [45] TOP500. 2026. TOP500 Supercomputers. <https://www.top500.org/>. Accessed June 4, 2026.
- [46] Jake Wires and Andrew Warfield. 2017. Mirador: An Active Control Plane for Datacenter Storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) (FAST '17). USENIX Association, Berkeley, CA, USA, 213–228. <https://www.usenix.org/conference/fast17/technical-sessions/presentation/wires>
- [47] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. End-to-end I/O Monitoring on a Leading Supercomputer. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) (NSDI '19). USENIX Association, Berkeley, CA, USA, 379–394. <https://www.usenix.org/conference/nsdi19/presentation/yang>
- [48] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing* (Seattle, WA, USA) (Lecture Notes in Computer Science, Vol. 2862). Springer, Berlin, Heidelberg, 44–60. doi:10.1007/10968987_3
- [49] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Salt Lake City, UT, USA) (CVPR '18). IEEE, Piscataway, NJ, USA, 6848–6856. doi:10.1109/CVPR.2018.00716