UMinho | 2022

Ricardo Macedo

User-level Software-Defined Storage Data Planes

Universidade do Minho
Escola de Engenharia

Ricardo Gonçalves Macedo

User-level Software-Defined Storage Data Planes

Setembro, 2022

Universidade do Minho
Escola de Engenharia

Ricardo Gonçalves Macedo

User-level Software-Defined Storage Data Planes

Tese de Doutoramento
Programa Doutoral em Informática
das Universidades do Minho, Aveiro e Porto

Trabalho efetuado sob a orientação de:
João Tiago Medeiros Paulo
José Orlando Roque Nascimento Pereira

Setembro, 2022

Acknowledgements

Este documento marca o fim de uma longa jornada. Jornada essa que se revelou extremamente desafiante mas sobretudo enriquecedora, e que não teria sido possível sem a ajuda de diversas pessoas que estiveram presentes ao longo de todo o seu percurso.

Em primeiro lugar, gostaria de agradecer ao meu orientador, Doutor João Paulo. Sinto-me priveligiado pela sua incansável orientação, pelas inúmeras discussões feitas ao longo de todo o percurso, e sobretudo, por moldar a forma com que encaro a investigação. Agradeço por ter sido o primeiro a acreditar em mim, e espero que esta jornada marque apenas o início de uma longa carreira em colaboração.

Agradeço também ao Professor José Orlando Pereira, que co-orientou este trabalho, por todas as discussões, conselhos, e partilha de conhecimento, que foram determinantes para o sucesso deste trabalho.

I have been fortunate for collaborating with so many people throughout these years. I thank Jason Haga, Yusuke Tanimura, Vijay Chidambaram, Alysson Bessani, Alberto Faria, Marco Dantas, Weijia Xu, Cláudia Correia, Cláudia Brito, Tânia Esteves, Diogo Leitão, Xinlian Liu, Mariana Miranda, Amit Ruhela, Stephen Harrell, Todd Evans, Bernardo Portela, and Danny Harnik. I am also grateful to Oana Balmau for her research advices and comments on papers. None of this work would be possible without the numerous and insightful discussions that I had with each of them.

Aos meus colegas de laboratório, Alberto Faria, Ana Alonso, Catarina Leones, Cláudia Brito, Daniel Cruz, Diogo Couto, Fábio Coelho, Francisco Cruz, Francisco Maia, Francisco Neves, João Dias, João Marco, Luis Meruje, Mariana Miranda, Nuno Machado, Paula Rodrigues, Ricardo Vilaça, Rogério Pontes, Rui Miguel, Tânia Esteves, Vitor Enes, e demais corpo docente, pelos momentos de alegria e companheirismo partilhados ao longo desta jornada. Um especial obrigado à Dona Aurora.

Aos meus pais, Benjamim e Helena, que sempre garantiram as condições necessárias para atingir os meus objectivos e me apoiaram ao longo da minha vida. Muito obrigado por poder contar convosco.

Por fim, quero deixar um agradecimento especial à pessoa que me tem acompanhado ao longo destes últimos anos, e que contribui todos os dias para a minha felicidade. Obrigado Isabel por caminhares sempre ao meu lado, partilhando comigo os bons momentos e apoiando nos maus.

Adicionalmente, os meus agradecimentos às instituições que apoiaram o trabalho aqui apresentado. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através da bolsa de doutoramento SFRH/BD/146059/2019. O Departamento de Informática da Universidade do Minho, o HASLab, e INESC TEC ofereceram-me as condições necessárias para o desenvolvimento deste trabalho.

REPÚBLICA PORTUGUESA
CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

FCT Fundação para a Ciência e a Tecnologia

POCH PROGRAMA OPERACIONAL CAPITAL HUMANO

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Social Europeu

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Braga, September 7th, 2022

Ricardo Gonçalves Macedo

(Ricardo Gonçalves Macedo)

Resumo

Planos de Dados Definidos por Software no Espaço do Utilizador

Os sistemas centrados em dados como bases de dados, sistemas de armazenamento chave-valor, e motores de aprendizagem automática, são hoje componentes fundamentais para as infraestruturas de computação modernas. De forma a atingir bom desempenho, estes sistemas implementam várias otimizações de armazenamento, como escalonamento de E/S, diferenciação, e *caching*. Esta dissertação argumenta que estas otimizações têm vindo a ser implementadas de forma subótima. Em primeiro lugar, as otimizações estão fortemente acopladas à implementação do sistema, e requerem um conhecimento extenso do mesmo por parte de quem as implementa, bem como mudanças significativas no seu código, dificultando a sua manutenção e portabilidade. Em segundo lugar, estas otimizações são maioritariamente implementadas de forma isolada e com visibilidade parcial da infraestrutura, levando-as a competir por recursos de E/S partilhados, a contenção no sistema, e variabilidade no desempenho.

Esta dissertação resolve estes desafios redefinindo a forma como as otimizações de E/S são implementadas. Em específico, as otimizações devem (1) ser desacopladas do sistema; (2) tomar decisões coordenadas sobre os recursos de E/S de forma a garantir controlo holístico; e (3) serem programáveis e adaptáveis de acordo com os requisitos do sistema. Para atingir estes objetivos, defendemos que o paradigma de Armazenamento Definido por *Software* (ADS) fornece um desenho adequado, embora incompleto, para implementar estas otimizações. Assim, começamos por sistematizar o trabalho em ADS, identificando os princípios de desenho comuns entre sistemas, discutimos as características que impulsionaram a aplicabilidade de cada solução, e identificamos as causas que impossibilitam a solução destes desafios por parte dos sistemas atuais. Como contribuição principal, introduzimos o sistema PAIO, um novo plano de dados de ADS que permite construir optimizações de E/S portáveis e genéricas no espaço do utilizador. Por fim, demonstramos o desempenho e a eficácia de otimizações implementadas com o PAIO construindo três planos de dados: o primeiro garante controlo da latência nos percentis altos em sistemas de armazenamento chave-valor, o segundo gere a largura de banda de aplicações num ambiente de armazenamento partilhado, e o terceiro garante controlo na qualidade de serviço das operações de metadados num sistema de ficheiros paralelo. Com estas contribuições, mostramos que é possível construir otimizações de E/S desacopladas do sistema, que atuam com visibilidade global, e que garantem resultados equiparáveis ou melhores que otimizações implementadas de forma tradicional.

Palavras-chave: Armazenamento definido por *software*, Armazenamento programável, Otimizações de armazenamento, Plano de dados.

Abstract

User-level Software-Defined Storage Data Planes

Data-centric systems such as databases, key-value stores (KVS), and machine learning engines have become an integral part of modern I/O infrastructures. Good performance for these systems often requires implementing multiple storage optimizations such as I/O scheduling, differentiation, and caching. This dissertation argues that such optimizations are implemented in a sub-optimal manner. First, optimizations are tightly coupled to the system implementation, and require a deep understanding of the system's internal operation model and profound code refactoring, limiting their maintainability and portability across other systems that would equally benefit from them. Second, optimizations are often implemented in isolation and with partial visibility of the infrastructure, competing for shared I/O resources, and generating I/O contention and performance variation.

This dissertation addresses these challenges by redefining how I/O optimizations are implemented. Specifically, optimizations should (1) be decoupled from the targeted system; (2) perform coordinated decisions over I/O resources to ensure holistic control; and (3) be programmable and adaptable to the requirements of the targeted system. We advocate that the Software-Defined Storage (SDS) paradigm provides a compelling but incomplete design for implementing such optimizations. As such, we start by surveying and systematizing the current body of work on SDS, identifying common design features shared between existing systems, discussing the characteristics that have driven the design and applicability of each solution under a given storage scenario, and uncovering why existing systems do not successfully address these challenges. Then, as our main contribution, we introduce PAIO, a new SDS data plane framework that enables building user-level, portable, and generally applicable storage optimizations. Finally, we demonstrate the performance and effectiveness of complex I/O optimizations implemented with PAIO by building three data plane stages. Namely, the first stage ensures tail latency control in Log-Structured Merge tree KVSs, the second achieves per-application bandwidth control in shared storage settings, and the third ensures QoS control of metadata operations in parallel file systems. With these contributions, this dissertation demonstrates that it is possible to build complex I/O optimizations that are decoupled from the targeted system and actuate with global infrastructure visibility, while achieving similar or better results than traditionally implemented ones.

Keywords: Data Plane, Programmable Storage, Software-Defined Storage, Storage Optimizations.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ABCI	AI Bridging Cloud Infrastructure 4, 6, 57, 60, 75, 76, 78, 79, 81, 83, 84, 101, 104
AI	Artificial Intelligence 75, 76, 84
AIST	National Institute of Advanced Industrial Science and Technology 75, 101, 102
API	Application Programming Interface 4, 13, 15, 43, 46, 52, 53, 54, 59, 73
BLKIO	Block I/O Controller 76, 79, 80
BRF	Bottleneck Resource Fairness 35, 38
CFQ	Completely Fair Queuing 76
CPU	Central Processing Unit 19, 31, 34, 57, 75, 76, 78, 104
DDRR	Distributed Deficit Round Robin 35, 38
DL	Deep Learning 5, 6, 7, 76, 77, 82, 83, 84
DRF	Dominant Resource Fairness 35, 38
DRL	Dynamic Rate Limiter 56, 65, 77, 89
DWRR	Deficit Weighted Round Robin 35, 38
EDF	Earliest Deadline First 35, 38
FIFO	First In, First Out 28, 62
FUSE	File System in User-Space 14, 16, 33, 59, 81, 105
GPU	Graphics Processing Unit 57, 75, 76, 78, 104
HPC	High-Performance Computing 1, 3, 4, 5, 6, 8, 9, 23, 25, 30, 31, 34, 37, 39, 41, 46, 59, 60, 75, 80, 81, 82, 86, 87, 89, 90, 91, 94, 101, 102, 103, 105

IOPS	Input/Ouput Operations per Second x, 19, 53, 57, 58
IoT	Internet of Things 9, 41, 42
KVS	Key-Value Store x, 1, 2, 4, 8, 44, 45, 46, 49, 54, 56, 59, 60, 61, 62, 63, 64, 66, 69, 73, 74, 104
LoC	Lines of Code 1, 44, 61, 65, 73, 77
LP	Linear Programming 29, 35, 36
LSM	Log-Structured Merge tree x, 1, 4, 8, 44, 49, 60, 61, 62, 64, 65, 73, 74, 104
LSTF	Least Slack Time First 35, 38
MDS	Metadata Server 83, 84
MDT	Metadata Target 83, 84, 85, 92, 94, 95, 102
ML	Machine Learning 35, 37, 43, 44
NERSC	National Energy Research Scientific Computing Center 84
NFS	Network File System 88
NIC	Network Interface Controller x, 11
NVMe	Non-Volatile Memory Express xi, 57, 66, 69, 72, 73
OS	Operating System 9, 54, 76, 78
OSS	Object Storage Server xi, 83, 84, 85, 101
OST	Object Storage Target 83
PCIe	Peripheral Component Interconnect Express 32
PFLOPS	10^{15} floating point operations per second 30
PFS	Parallel File System xi, 4, 6, 8, 30, 33, 60, 82, 83, 84, 85, 86, 87, 88, 89, 91, 92, 94, 101, 102, 105
PID	Process Identifier 19, 53, 89
PM	PriorityMeister 26, 28, 35, 37, 38, 39, 40, 58, 59
PMDK	Persistent Memory Development Kit 46, 59, 106
POSIX	Portable Operating System Interface xi, 12, 33, 45, 46, 49, 53, 54, 56, 62, 64, 74, 77, 80, 81, 82, 83, 87, 88, 89, 91, 92, 93, 101, 104, 105
QoS	Quality of Service xi, xii, 2, 4, 6, 10, 15, 27, 29, 30, 34, 42, 55, 58, 59, 60, 80, 82, 85, 86, 87, 90, 91, 94, 95, 101, 102, 105

RAM	Random Access Memory 57, 66, 78, 91
RPC	Remote Procedure Call 13, 32, 37, 54, 55, 83, 91
SATA	Serial Advanced Technology Attachment xi, 28, 57, 66, 71
SDN	Software-Defined Networking 2, 10, 12, 13, 18, 20, 21, 23, 28, 32, 42, 43
SDS	Software-Defined Storage x, xii, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 49, 52, 54, 55, 58, 60, 78, 89, 101, 103, 106
SLO	Service Level Objective 11, 25, 28, 36, 40, 59, 80, 81, 94
SOA	Service-Oriented Architecture 29
SPDK	Storage Performance Development Kit 46, 59, 106
SSD	Solid State Drive xi, 28, 30, 57, 66, 69, 71, 73, 78, 91
SST	Sorted Strings Table 61, 62, 63, 74
VFS	Virtual File System 46, 59, 92
VM	Virtual Machine 27, 29, 36, 37, 40
WAL	Write-Ahead Logging 61, 62, 66, 74
WC	WorkloadCompactor 26, 28, 35, 37, 38, 39, 40
YCSB	Yahoo! Cloud Serving Benchmark 67

1 Introduction

A massive, ever-growing amount of digital information generated from a number of different sources is processed and stored every day in both public and private premises [6, 180]. The consequent continual need for greater storage and processing capacity has significantly increased the complexity of underlying infrastructures. Indeed, modern I/O infrastructures such as enterprise-grade data centers (*e.g.,* cloud computing, serverless computing, application-specific storage stacks) and High-Performance Computing (HPC) supercomputers, feature several layers along the I/O path providing compute, network, and storage functionalities, including operating systems, hypervisors, object stores, databases, I/O libraries, file systems, and device drivers [101, 205, 211]. To achieve good performance in these layers, either from the perspective of the end-user or the overall infrastructure (*e.g.,* throughput, resource management, energy efficiency), these often require implementing multiple important storage optimizations such as scheduling [17, 87, 233], prioritization and differentiation [110, 155, 207], caching [25, 118], replication [121, 202], and more [172, 246]. For instance, Log-Structured Merge tree (LSM) Key-Value Stores (KVS) have become an integral part of modern infrastructures [48, 80, 178, 183], and to achieve high throughput and low latency access to large volumes of data, several I/O optimizations have been implemented over these, including caches [227], checksumming [60], I/O schedulers [17], and tiering [43, 239].

A key problem that emerges from these optimizations however, stems from how these are implemented in the I/O stack, being tightly integrated within the core of each system. Specifically, to implement these optimizations, system designers require a deep understanding of the system's internal operation model and perform profound code refactoring. This is problematic given that multiple systems used in production such as Ceph [224], RocksDB [183], and TensorFlow [1], are made of thousands to millions of Lines of Code (LoC) and are updated with new functionalities and fixes on a regular basis. As a result, this not only increases the work needed to maintain and port optimizations to other systems that would equally benefit from them, but might also jeopardize years of development of the targeted system by introducing bugs in production. For instance, to improve the tail latency performance of RocksDB [183], an industry-standard LSM-based KVS, SILK proposes an I/O scheduler to control the interference among different operations [17]. However, applying this optimization over RocksDB required changing core modules made of thousands of LoC [15, 72]. Furthermore, porting this optimization to other KVSs, such as LevelDB [80] or PebblesDB [178], while feasible, is not trivial, as even though they share the same high-level design, the internal I/O logic differs across implementations (*e.g.,* data structures [48, 178], compaction algorithms [140, 178]).

Another important problem that arises from indiscriminately employing optimizations in the I/O stack, is that these are often implemented in isolation and with partial visibility of the infrastructure. Specifically, optimizations are fine-tuned to attend the requirements of a given system, being oblivious of the remainder I/O layers running in the infrastructure. Under this design, ensuring holistic performance is hard, and if not correctly assessed can lead to high levels of I/O interference and performance degradation [110, 211]. This effect becomes further amplified when I/O resources are accessed concurrently, either by different layers of the I/O stack (*e.g.,* multiple applications sharing a file system and competing for disk bandwidth) or even from internal background activities of a given layer that are competing for resources with the corresponding foreground tasks, as is the case of databases [120], file systems [202], and KVSs [183]. For instance, background tasks such as compaction, checkpointing, and replication are predefined I/O-intensive activities that can rapidly overload shared resources, introducing significant I/O interference and workload burstiness, ultimately impacting overall throughput and latency. To minimize interference with foreground activities, background tasks are processed in a best-effort manner, being executed either periodically or whenever a certain threshold is hit. However, the decision of when and how to execute such operations is taken by the layer itself regardless of the overall load on the infrastructure at the time.

These problems result from how large-scale I/O infrastructures (and corresponding layers) have been traditionally designed, and reflect the *absence of a true programmable I/O stack* and the *uncoordinated control of layers*.

To overcome these challenges, the Software-Defined Storage (SDS) paradigm emerged as a compelling solution that reorganizes traditional I/O stacks by *decoupling the I/O mechanisms from the policies that govern them* into two planes of functionality — *control* and *data* [211]. The control plane is a logically centralized entity that comprises system-wide visibility and acts as a global coordinator, enforcing policies over the I/O stack in holistic manner. Policies are built on top of this centralized controller as control algorithms, and define how I/O requests should be treated in each point of the infrastructure. Examples of such control algorithms are used for achieving Quality of Service (QoS) provisioning [211, 248], performance control [205, 206], and resource fairness [143, 201]. The data plane is a multi-stage component distributed over the I/O stack. Each data plane stage implements custom I/O logic to apply over requests to meet a given policy (defined by the control plane). In particular, stages can provide simple data transformations as encryption and compression schemes [84, 175], or more complex mechanisms such as token-buckets, I/O schedulers, and load balancers [143, 194, 201, 211].

SDS adopts key concepts from the Software-Defined Networking (SDN) paradigm and applies them to storage-oriented environments, bringing new insights to storage infrastructures such as (1) improved system programmability and extensibility, (2) fine-grained resource orchestration, (3) holistic I/O control, and (4) ease in portability and maintainability.

1.1 Problem Statement and Objectives

In spite of the considerable advantages introduced by SDS, the different requirements in performance, scalability, resilience, and resource management imposed over modern infrastructures have driven existing systems to follow a similar path as traditionally implemented I/O optimizations. Namely, current SDS systems are targeted for specific I/O layers, as their design is *tightly coupled to* and *driven by* the architecture and specificities of the software stacks that they are applied to. For instance, while several systems enforce throughput and latency policies under shared storage scenarios, such as IOFlow [211], Crystal [84], and Retro [143], each of these is tightly integrated and co-designed with the internal modules of their targeted I/O layers (*e.g.,* hypervisor, object store, file system). Under this scenario, existing SDS systems cannot be used as a drop-in replacement of one another, nor implement I/O optimizations over layers not targeted by these.

The main objective of this thesis is then to redefine how I/O optimizations are implemented by enabling system designers to build optimizations that are simultaneously (1) decoupled from the targeted system, and do not require significantly changing the I/O layers themselves; (2) perform coordinated decisions over I/O resources to ensure holistic control throughout the overall infrastructure; (3) impose minimal performance overhead; and (4) are programmable and adaptable, so these can be fine-tuned to meet the requirements and storage objectives of the targeted layer. With this, we aim at ensuring that decoupled I/O optimizations can achieve similar or greater levels of performance and control as system-specific ones.

Moreover, while SDS has gained significant relevance in the research community, many aspects of the paradigm are still unclear, undefined, and unexplored (*e.g.,* what constitutes a SDS system, what are the key design features of the data and control planes, where are SDS systems applied to), leading to an ambiguous conceptualization and a disparate formalization between current and forthcoming solutions. Thus, another objective of this thesis is to systematize the current body of work on SDS, identifying common design features shared between existing systems, and discuss the distinctive characteristics that have driven the design and applicability of each solution under a given storage context.

1.2 Contributions

To achieve the aforementioned goals, this thesis presents three main contributions.

I: SDS survey. As a first contribution of this thesis, we present a comprehensive survey of current SDS systems, explaining and clarifying fundamental aspects of the field. We provide a thorough description of each plane of functionality, and propose a taxonomy and classification of existing systems regarding storage infrastructure type, namely cloud computing, HPC, and application-specific storage stacks, as well as their control and enforcement strategies.

II: PAIO. As the core contribution of this thesis, we present PAIO, a SDS data plane framework that enables building user-level, portable, and generally applicable storage optimizations. The key idea is to implement the optimizations *outside* the targeted system as data plane stages, by intercepting and handling the I/O performed by these. These optimizations are then *controlled* by a logically centralized controller that has the global context necessary to prevent interference among them. To perform complex I/O optimizations that are dependent on the internal system logic, PAIO needs to *propagate context* down the I/O stack, from high-level Application Programming Interfaces (APIs) down to the lower layers that perform I/O in smaller granularities. It achieves this by combining ideas from *context propagation* [144], enabling application-level information to be propagated to data plane stages with minor code changes and without modifying existing APIs.

Due to the different I/O requirements between storage infrastructures, there is no *one-size-fits-all* SDS solution. Thus, PAIO aims at filling this gap by providing the necessary building blocks for system designers to build custom-made data plane stages, fine-tuned for their storage requirements. Using PAIO, one can decouple complex storage optimizations from current systems, such as I/O differentiation and scheduling, while achieving results similar to or better than tightly coupled optimizations.

III: Data plane stages. As a third contribution, we demonstrate how PAIO can be used for implementing complex I/O optimizations by building three data plane stages. Each stage targets a specific I/O layer, and is fine-tuned to cope with the requirements and storage objectives of the targeted system. All scenarios were driven by real use cases that exist in production clusters.

- First, we built a data plane stage that achieves tail latency control in LSM-based KVSs (*i.e.,* RocksDB) by orchestrating the interference between foreground and background operations. Results show that a PAIO-enabled RocksDB improves tail latency by 4× under different workloads, and enables similar performance and I/O control as system-specific optimizations (*i.e.,* SILK [17]).

- Second, we built a data plane stage that ensures per-application bandwidth guarantees under a shared storage environment at the AI Bridging Cloud Infrastructure (ABCI) supercomputer, where applications that execute on the same compute node compete for local disk bandwidth [4]. Results show that all PAIO-enabled applications are provisioned with their bandwidth goals.

- Finally, we built PADLL, a storage middleware that ensures QoS over metadata workflows in HPC storage systems. Results show that PADLL can dynamically control metadata-aggressive workloads, prevent I/O burstiness, and ensure fairness and prioritization between jobs that compete for metadata resources over Parallel File Systems (PFSs).

The first data plane stage demonstrates how PAIO can be used to reimplement complex I/O optimizations that achieve similar performance as system-specific ones, while the second and third stages provide new optimizations that address unsolved challenges present in modern I/O infrastructures.

1.3 Results

Core publications. The work discussed in this thesis resulted in a number of publications in international conferences, journals, and workshops.

Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, João Paulo. **PAIO: General, Portable I/O Optimizations With Minor Application Modifications.** In *20th USENIX Conference on File and Storage Technologies*, 2022.

This paper describes PAIO, a novel SDS framework that enables system-designers to build custom-made data plane stages, applicable over different user-level I/O layers [150]. Rather than implementing system-specific I/O optimizations, tightly coupled within the core of each system, PAIO follows a decoupled design that separates the mechanisms required to implement the I/O logic from the policies that govern them. We demonstrate the performance and applicability of PAIO with two data plane stages, implemented over RocksDB and TensorFlow, to enforce different storage policies. PAIO is publicly available at https://github.com/dsrhaslab/paio.

Ricardo Macedo, João Paulo, José Pereira, Alysson Bessani. **A Survey and Classification of Software-Defined Storage Systems.** In *ACM Computing Surveys 53, 3 (48)*, 2020.

This journal publication provides a comprehensive survey and classification of current SDS systems [149]. It details the main challenges, design principles, and functionalities of the SDS paradigm, and proposes a taxonomy that identifies key design features common to all systems over distinct storage infrastructures, namely cloud, HPC, and application-specific.

Ricardo Macedo, Alberto Faria, João Paulo, José Pereira. **A Case for Dynamically Programmable Storage Background Tasks.** In *38th International Symposium on Reliable Distributed Systems Workshops*, 2019.

This workshop publication presents a detailed study of the impact of storage background tasks over local and distributed storage stacks [147]. It makes an initial case that storage background mechanisms, such as compactions and checkpointing, should follow a SDS design by being dynamically programmable and orchestrated by a control module with global visibility. This vision was further explored in [146, 150]. Results are publicly available at https://rgmacedo.github.io/drss19-website/.

Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, João Paulo. **The Case for Storage Optimization Decoupling in Deep Learning Frameworks.** In *1st Workshop on Re-envisioning Extreme-Scale I/O for Emerging Hybrid HPC Workloads*, co-located with *IEEE International Conference in Cluster Computing*, 2021.

This workshop publication makes a case that I/O optimizations implemented over modern Deep Learning (DL) frameworks are single-purposed, limiting their applicability and portability across other frameworks that would equally benefit from them [146]. The paper proposes a new SDS architecture for implementing I/O optimizations in DL frameworks, as well as Prisma, a framework-agnostic SDS-enabled middleware that implements a parallel data prefetching mechanism. We validate the applicability and portability of our approach by optimizing the training performance of TensorFlow and PyTorch [169]. Prisma is publicly available at https://github.com/dsrhaslab/prisma.

Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, João Paulo. **Protecting Metadata Servers From Harm Through Application-level I/O Control.** In *2nd Workshop on Re-envisioning Extreme-Scale I/O for Emerging Hybrid HPC Workloads*, co-located with *IEEE International Conference in Cluster Computing*, 2022.

This workshop publication discusses existing problems in PFSs that emerge from metadata-aggressive workloads [148]. It proposes an initial prototype of PADLL, a storage middleware that enables system administrators to proactively control and ensure QoS over metadata workflows in HPC storage systems. An experimental evaluation demonstrates that PADLL effectively controls the rate of metadata workflows and prevents I/O burstiness.

Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, João Paulo. **Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control.** In *submission*.

This paper discusses the full design, implementation, and evaluation of PADLL. It proposes an application and file system agnostic SDS storage middleware that enables QoS control over metadata workflows in HPC storage systems. Moreover, the paper provides a comprehensive study that analyzes traces from a production Lustre file system at ABCI, and reveals new insights about metadata operations at scale. Further, it presents a new proportional sharing algorithm that improves the performance of metadata-aggressive jobs while meeting specific storage policies. This paper is currently under *submission*.

Complementary publications. The following work was published in collaboration with multiple researchers from both academia and industry. While complementary to the core contributions of this thesis, these works leveraged from the topics discussed in it.

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

This conference publication proposes Monarch, a framework-agnostic storage tiering middleware for accelerating single-node DL training in HPC infrastructures [52]. It enables DL frameworks to transparently leverage local storage mediums of compute nodes, even for datasets that may not fit entirely on such resources, for improving DL training performance while also reducing the I/O pressure imposed over the shared PFS. Monarch is publicly available at https://github.com/dsrhaslab/monarch.

Alberto Faria, Ricardo Macedo, José Pereira, João Paulo. **BDUS: Implementing Block Devices in User Space.** In *14th ACM International System and Storage Conference*, 2021. *Best paper runner-up.*

This conference publication studies the feasibility of implementing user-level block devices [74]. The paper provides an in-depth performance evaluation over existing user-level block device frameworks, identifies existing problems and limitations, and proposes BDUS, a fully functional framework with low overhead that enables the development of custom block device drivers in user space. BDUS is publicly available at https://github.com/albertofaria/bdus.

Marco Dantas, Diogo Leitão, Cláudia Correia, Ricardo Macedo, Weijia Xu, João Paulo. **Monarch: Hierarchical Storage Management for Deep Learning Frameworks.** In *1st Workshop on Re-envisioning Extreme-Scale I/O for Emerging Hybrid HPC Workloads*, co-located with *IEEE International Conference in Cluster Computing*, 2021.

This workshop publication proposes an initial version of the Monarch system [51]. The paper experimentally demonstrates the benefits of using local storage devices to improve DL training performance, and provides an initial prototype and a preliminary experimental evaluation with TensorFlow.

Alberto Faria, Ricardo Macedo, João Paulo. **Pods-as-Volumes: Effortlessly Integrating Storage Systems and Middleware into Kubernetes.** In *7th International Workshop on Container Technologies and Container Clouds*, co-located with *22nd ACM/IFIP International Middleware Conference*, 2021.

This workshop publication describes Pods-as-Volumes (PaV), a Kubernetes [117] plugin that simplifies the construction of volume provisioners [73]. It enables the straightforward creation of storage middleware components, improving modularity and unlocking the ability to build advanced storage stacks and architectures using Kubernetes. PaV is publicly available at https://github.com/albertofaria/pav.

Tânia Esteves, Ricardo Macedo, Alberto Faria, Bernardo Portela, João Paulo, José Pereira, Danny Harnik. **TrustFS: An SGX-enabled Stackable File System Framework.** In *38th International Symposium on Reliable Distributed Systems Workshops*, 2019.

This workshop publication describes TrustFS, a programmable and modular stackable file system framework for implementing secure content-aware storage functionalities over hardware-assisted trusted execution environments [66]. It extends the original design of the SafeFS system [175] to provide the isolated execution guarantees of Intel Software Guard Extensions [49].

1.4 Outline

The rest of the document is organized as follows:

- **Chapter 2.** We present a detailed survey of SDS systems (§2). We introduce background concepts of the SDS field, and propose a taxonomy and classification of existing systems regarding storage infrastructure, control strategy, and enforcement strategy.
- **Chapter 3.** We introduce the design, implementation, and evaluation of PAIO, a novel SDS framework that enables system designers to build custom-made, user-level data plane stages applicable over different I/O layers (§3).
- **Chapters 4, 5, and 6.** We present the design of three data plane stages built with PAIO. We show how to achieve tail latency control in LSM-based KVSs (§4), ensure per-application bandwidth guarantees in shared storage environments (§5), and ensure metadata control in HPC PFSs (§6).
- **Chapter 7.** We discuss future research work and present the final remarks of this thesis (§7).

2

Software-Defined Storage Background

The ever-growing need for storing massive amounts of digital information with high throughput and low latency has turned modern storage infrastructures extremely complex [6, 180]. Today's data centers such as cloud computing and HPC facilities, are vertically designed and feature several layers along the I/O path that provide compute, network, and storage functionalities, including Operating Systems (OS), hypervisors, caches, schedulers, file systems, and device drivers [211]. Each of these layers includes a predetermined set of I/O mechanisms (*e.g.,* caching, rate limiting, data placement, compression) with strict interfaces and isolated procedures to employ over requests, leading to a complex, limited, and coarse-grained treatment of I/O workflows. To overcome these shortcomings, the Software-Defined Storage paradigm emerged as a compelling solution to ease data and configuration management, while improving end-to-end control functionality of conventional storage infrastructures. This chapter presents the fundamentals of the SDS paradigm and clarifies unclear aspects and misconceptions of the field. First, we identify and discuss the distinctive design principles and characteristics of an SDS-enabled infrastructure (§2.1—§2.4). We then propose a taxonomy and classification of SDS systems, and survey existing solutions grouped by storage infrastructure (§2.5.1), control strategy (§2.5.2), and enforcement strategy (§2.5.3). Finally, we discuss open research challenges of the field (§2.6).

This chapter focuses on programmable and adaptable SDS systems. Namely, we do not address the design and limitations of either specialized storage systems (*e.g.,* file systems, block devices, object stores) or other fields of storage research (*e.g.,* deduplication [172], confidentiality [58], device failures [190]). Autonomic computing systems are also out of the scope of this thesis [98]. Moreover, while other software-defined approaches share similar design principles, they are out of the scope of this work, including but not limited to networking [115], OS [22, 174], data centers [7, 184], cloud [104], flash [165, 193], security [114, 216], and Internet of Things (IoT) [23, 103].

2.1 Overview

Software-Defined Storage is an emerging storage paradigm that breaks the vertical alignment of conventional storage infrastructures by reorganizing the I/O stack to decouple I/O workflows into two planes of

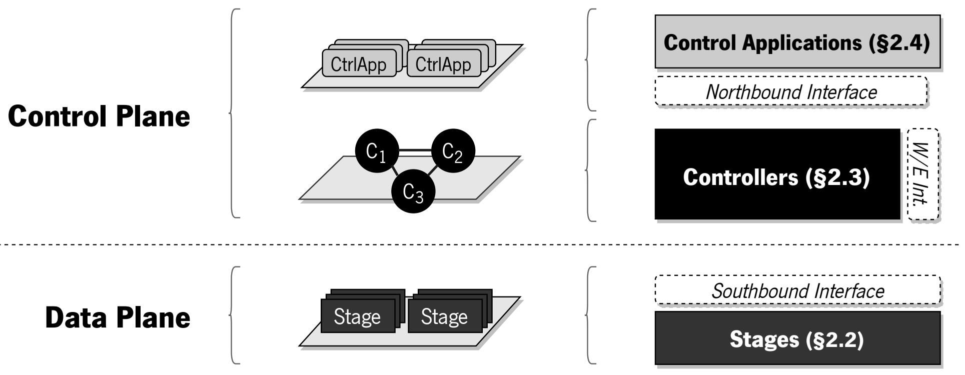


Figure 2.1: Layered view of the SDS planes of functionality, comprehending both control and data tiers.

functionality — *control* and *data*.[1] Figure 2.1 depicts a layered view of such design. The *control plane* provides the control building blocks used for designing system-wide control applications [84, 107, 205, 211]. It holds the intelligence of the SDS system and consists of *(1)* a logically centralized controller (§2.3), which acts as a global coordinator with system-wide visibility and centralized control, and *(2)* several control applications (§2.4) built on top. The *data plane* (§2.2) is composed of several stages that employ specific I/O mechanisms over requests (*e.g.,* rate limiters, caches, encryption and compression schemes) [50, 143, 211]. Communication between components is established through specialized interfaces, namely *Northbound*, *Southbound*, and *Westbound/Eastbound interfaces*. SDS inherits legacy concepts from SDN [115] and applies them to storage-oriented environments, bringing new insights to the storage stack, such as improved system programmability and extensibility [175, 194], fine-grained resource orchestration [143, 158], and end-to-end QoS, maintainability, and flexibility [107, 211].

Unlike traditional storage solutions, which require designing and implementing individual control tasks at each I/O layer such as coordination, metadata management, and monitoring, SDS brings a general system abstraction where control primitives are implemented at a dedicated component. This separation of concerns breaks the storage control into tractable pieces, offering the possibility to programme I/O resources and provide end-to-end adaptive control over large-scale storage infrastructures.

While the storage industry defines SDS as a storage architecture that simply separates software from vendor lock-in hardware [92, 100], existing SDS solutions are more comprehensive than that. As such, we define a SDS-enabled system as a storage architecture with four main principles.

- **Storage mechanisms are decoupled from the policies that govern them.** Instead of designing monolithic, custom-made services at each I/O layer, SDS decouples the control logic (policy) from the storage mechanism to be employed over data.[2]

- **Storage mechanisms are moved to a programmable data plane.** I/O mechanisms (*e.g.,*

[1]We refer to the term *workflow* (or *flow*) as the connection between two I/O layers through where requests are transmitted.

[2]We refer to the term *services* as a set of I/O mechanisms employed over requests to achieve a given policy, such as dynamic rate limiting, caching, load balancing, and more.

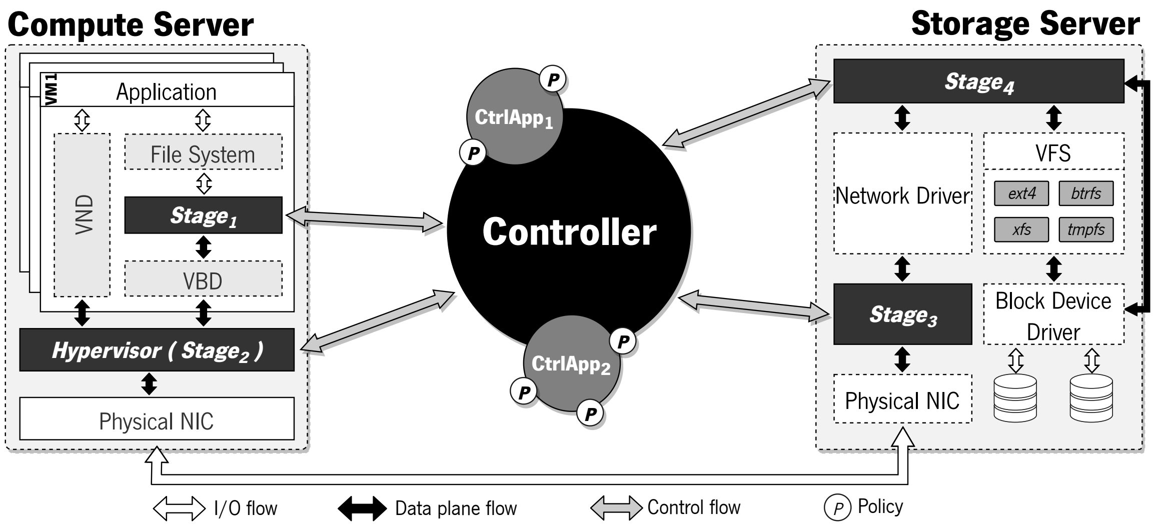


Figure 2.2: SDS-enabled architecture materialized on top of a general-purpose multi-tenant storage infrastructure. Compute servers are virtualized and host virtual machines interconnected to the hypervisor by virtual devices, namely virtual Network Interface Controller (NIC) and virtual hard disk. Storage servers comprehend general network and storage elements.

rate limiters, compression and encryption schemes) to be employed over I/O workflows are implemented over programmable structures, and fine-tuned to meet user-defined requirements.

- **Control logic is moved to an external control plane.** Control logic is implemented at a logically or physically decoupled control plane, and properly managed by applications built on top.
- **Storage policies are enforced over I/O flows.** Service enforcement is data-centric rather than system-centric, being employed over distinct layers and storage resources along the I/O path.

In an SDS-enabled architecture, such as depicted in Figure 2.2, control applications are the entry point of the control environment (Figure 2.2: *CtrlApp*$_1$ and *CtrlApp*$_2$) and the *de facto* way of SDS users (*e.g.,* system designers, system administrators) to express different storage policies to be enforced over the storage infrastructure. Policies are sets of rules that declare how I/O workflows are managed, being defined at control applications, disseminated by controllers, and installed at data plane stages. For example, to ensure sustained I/O performance, SDS users may define minimum disk bandwidth guarantees for a particular set of tenants [211], or define request prioritization to ensure X^{th} percentile latency Service Level Objective (SLO) [128]. Since controllers share a centralized view, applications resort to centralized algorithms to implement the control logic, which are simpler and less error prone than designing the corresponding decentralized versions [211]. The scope of applications (and policies enforced by these) is broad, ranging from performance-related services [139, 175, 211], to resource and data management functionalities [143, 194, 200]. These user-defined policies are shared between applications and controllers through a *Northbound interface*, which defines an instruction set that abstracts the distributed control environment into a centralized one.

Controllers continuously monitor the status of the storage infrastructure, including data plane stages, storage devices, and other storage-related resources, and orchestrate the overall storage services holistically. Having centralized control enables an efficient enforcement of policies and simplifies storage configuration [211]. Policies are handled by a planning engine that translates centralized policies into stage-specific rules and operation logic, which are disseminated to the targeted data plane stages and synchronized with other controllers. Communication with the data plane (*control flow*) is achieved through a *Southbound interface* (illustrated in Figure 2.2 with gray-toned arrows), that allows controllers to exercise direct control over data plane stages through policy dissemination and monitoring. Moreover, a *Westbound/Eastbound interface* establishes the communication between controllers to ensure coordination and agreement [84, 205].

The data plane is a multi-stage component distributed along the I/O path (Figure 2.2: *Stage*$_1$ to *Stage*$_4$) that holds fine-grained storage services dynamically adaptable to the infrastructure status. Each stage employs a distinct storage service over intercepted data flows, such as performance management (*e.g.,* prioritization) [128, 206], data management (*e.g.,* compression, encryption) [175, 177], and data routing (*e.g.,* flow customization) [101, 205]. For all intercepted requests, any matching policy will employ the respective service over filtered data, which will then be forwarded to the rest of the execution path (*e.g.,* Figure 2.2: data flow between *File System* $\leftrightarrow$ *Stage*$_1$ $\leftrightarrow$ *Block Device*).

2.2 Data Plane

Data plane stages are multi-tiered components distributed along the I/O path that perform storage-related operations over incoming I/O requests. Stages are the end-point of SDS systems and abstract complex storage mechanisms (*e.g.,* rate limiters, encryption and compression schemes, caches) into a seamless design that allows user-defined policies to be enforced over I/O workflows. Each of these stages establishes a flexible enforcement point for the control plane to specify fine-grained control instructions. As presented in Figure 2.2, stages can be transparently placed between two layers of the I/O stack, acting as a middleware (*Stage*$_1$, *Stage*$_3$, and *Stage*$_4$), or within an individual I/O layer (*Stage*$_2$). Moreover, stages comprehend *I/O interfaces* to handle the I/O workflows, and a *core* that encompasses the storage primitives to be enforced over such workflows. To preserve the I/O stack semantics, *input* and *output* interfaces marshal and unmarshal data workflows to be both enforced at the *core* and correctly forwarded to the next I/O layer. For instance, SafeFS exposes a Portable Operating System Interface (POSIX) compliant interface to enable the transparent enforcement of policies over file systems [175]. The stage *core* comprises a *(i) policy store* to orchestrate enforcement rules installed by the control plane (similar to a SDN flow table [151]); an *(ii) I/O filter*, that classifies and differentiates requests based on the installed policies; and *(iii)* an *enforcement structure* that employs the respective storage features over filtered requests. A thorough description of this component is presented in §2.2.2.

While overlooked in current SDS literature, the *Southbound interface* is the *de facto* component that defines the separation of concerns in software-defined systems. This interface is the bridge between control and data planes, and establishes the operators that can be used to directly control each stage (*e.g.,* policy propagation, share monitoring information, fine-tune storage mechanisms). It exposes to the control plane an API that defines the instructions a stage understands, so it can be configured to perform local decisions (*e.g.,* manage storage policies, fine-tune configurations). Such an API can be represented in a variety of formats. For example, IOFlow [211] and sRoute [205] use tuples comprising human-friendly identifiers (*e.g.,* operation type, hostname, stage identifier) to control stages, while Crystal [84] resorts to a system-agnostic domain-specific language to simplify stage administration. Moreover, the *Southbound interface* acts as a communication middleware, and defines the communication model between these two planes of functionality (*e.g.,* publish-subscribe, Remote Procedure Call (RPC), Representational State Transfer, Remote Direct Memory Access).

Despite the SDN influence, the divergence between storage and networking areas has driven SDS to comprehend fundamentally different design principles and system properties. First, each field targets distinct stack components, leading to significantly different policy domains, services, and data plane designs. Second, contrarily to an SDN data plane, whose stages are simple networking devices specialized in packet forwarding [115], such as switches, routers, and middleboxes, SDS-enabled stages hold a variety of storage mechanisms, leading to a more comprehensive and complex design. Third, the simplicity of SDN stages eases the placement strategy when introducing new functionalities to be enforced [115], while SDS ones demand accurate enforcement points, otherwise it may disrupt the SDS environment and introduce a significant performance penalty.

2.2.1 Properties

We now define the properties that characterize SDS data planes, namely *programmability*, *extensibility*, *stage placement*, *transparency*, and *policy scope*. These properties are not mutually exclusive (*i.e.,* a data plane can comprise several of them), and are contemplated as part of the taxonomy for classifying SDS systems (§2.5).

Programmability. Programmability refers to the ability of a data plane to adapt and programme existing storage mechanisms provided by the stage's enforcement structure (*e.g.,* stacks, queues, storlets) to develop fine-tuned and configurable storage services [194]. Specifically, in SDS data planes, programmability is usually exploited to ensure I/O classification and differentiation [84, 211], service isolation and customization [107, 205, 206], and development of new storage abstractions [194]. Conventional storage infrastructures are often tuned with monolithic setups to handle multiple applications with time-varying requirements, leading them to experience the same service level [8, 55]. SDS programmability prevents this by adapting data plane stages to provide differentiation of I/O requests and ensure service isolation (further description of this process in §2.2.2). Moreover, programmable storage systems can ease service development by re-purposing existing abstractions of the storage stack, and exporting the configurability aspect of specialized storage systems to a more generalized environment, *i.e.,* expose runtime configurations (*e.g.,* replication factor, cache size, queue depth) and existing storage subsystems of specific services (*e.g.,* load balancing, durability, metadata management) to a more accessible environment [194]. For example, Mantle [195] decouples cache management mechanisms of storage systems into independent policies so they can be dynamically adapted and re-purposed.

Extensibility. Extensibility refers to how easy is for data plane stages to support additional storage mechanisms or customize existing ones. An *extensible* data plane comprises a flexible and general-purpose design suitable for heterogeneous storage environments, and allows a straightforward implementation of storage mechanisms. Such a property is key for achieving a comprehensive SDS environment, capable of attending different requirements of a variety of applications, as well as to broad the policy spectrum (*i.e.,* number and type of policies) supported by the SDS system. The extensibility of the data plane strongly relies on the actual implementation of its architecture. In fact, as presented in the literature, highly extensible data plane implementations are built atop flexible and *extensible by design* storage systems (*e.g.,* File System in User-Space (FUSE) [132], OpenStack Swift [11]). For instance, SafeFS [175] allows developers to extend its design with new self-contained storage services (*e.g.,* encryption, replication, erasure coding) without requiring changing its core codebase. However, behind this flexible and generic design, lies a great deal of storage complexity that if not properly assessed can introduce significant performance overhead. On another hand, an *inextensible* data plane typically holds a rigid implementation and hard-wired storage mechanisms, tailored for a predefined subset of storage policies. Such a design bears a more straightforward and fine-tuned system implementation, and thus comprehends a more strict policy domain only applicable to a limited set of scenarios.

Placement. The placement of data plane stages refers to the overall position on the I/O path a stage can be deployed (for instance, as depicted in Figure 2.2, *Stage*$_1$ is placed between a file system and block device, while *Stage*$_2$ is placed within the hypervisor). It defines the control granularity of SDS systems and is a key enabler to ensure efficient policy enforcement. Each stage is considered as an enforcement point. Less enforcement points lead to a coarse-grained treatment of I/O workflows, while more points allow for a fine-grained management. Since the control plane has system-wide visibility, broadening the enforcement domain allows controllers to accurately determine the most suitable place to enforce a specific storage policy [211]. Improper number and placement of stages may disrupt the control environment, and therefore, introduce significant performance and scalability penalties to the overall infrastructure.

Depending on the context and size of the storage infrastructure, stages are often deployed individually *i.e.,* presenting a single enforcement point to the SDS environment. This *single-point* placement is often associated to local storage environments, being tightly coupled to a specific I/O layer or storage component, as in SafeFS [175] (file system) and Mesnier *et al.* [155] (block layer). However, this setting narrows the available enforcement strategies, which may lead to control inefficiencies and conflicting policies (*e.g.,* enforcing X^{th} percentile latency under throughput-oriented services).

A similar placement pattern can be applied in distributed settings. Distributed placement of stages (*i.e., distributed single-points*) is associated to distributed storage components of an individual I/O layer (*e.g.,* distributed file systems [143], object stores [84, 158]). In this scenario, each enforcement point is a data plane stage deployed at the same I/O layer as the others. In contrast to the prior placement strategy, this design displays more enforcement points for the control plane to decide the enforcement strategies. It is, however, still limited to a particular subset of storage components and may suffer similar drawbacks as the *single-point* approach.

Another placement alternative is *multi-point* data plane stages, which can be placed at several points of the I/O path, regardless the I/O layer [205, 206, 211]. This design provides a fine-grained control over stages and is key to achieve end-to-end policy enforcement. However, it can introduce significant complexity to the data plane development, and often requires direct implementation over I/O layers *i.e.,* following a more intrusive approach.

Transparency. The transparency of a data plane reflects on how seamless is its integration with I/O layers. A *transparent* stage is often placed between storage components and preserves the original I/O flow semantics [175]. For instance, Moirai [206] provides direct control over the distributed caching infrastructure, being applicable across different layers of the I/O path. Such an integration however may require substantial marshaling and unmarshaling activities, directly impacting the latency of I/O requests. Contrarily, an *intrusive* stage implementation is tailored for specific storage contexts and can achieve higher levels of performance since it does not require semantic conversions [84, 211]. However, this may entail significant changes to the original codebase (*e.g.,* modify internal and external APIs, adapt internal system components and data structures), imposing major challenges in developing, deploying, and maintaining such stages, ultimately reducing its flexibility and portability.

Policy scope. The policy scope of a data plane categorizes the different storage mechanisms and objectives employed over I/O requests. SDS systems can be applied in different storage infrastructures to achieve several purposes, namely performance, security, and resource and data management optimizations. To cope with these objectives, SDS systems comprehend a large array of storage policies categorized in three main scopes, namely *performance management*, *data management*, and *data routing*. Noticeably, the support for different scopes relies on the data plane implementation and storage context. *Performance management* mechanisms are associated to performance-related policies to ensure isolation and QoS provisioning [107, 205, 206, 211] (*e.g.,* cache management, bandwidth aggregation, I/O prioritization). *Data management* assembles mechanisms oriented to the management of data requests, such data reduction [84, 158], security [175], and redundancy [21, 50]. *Data routing* primitives encompass routing mechanisms that redefine the data flow, such as I/O path customization [205], replica placement strategies [226], and data staging [101]. Even though mainly applied in networking contexts [115], *data routing* mechanisms are now contemplated as another storage primitive in order to dynamically define the path and destination of an I/O workflow at runtime [205].

As SDS systems are employed over different storage scenarios, data planes can include additional

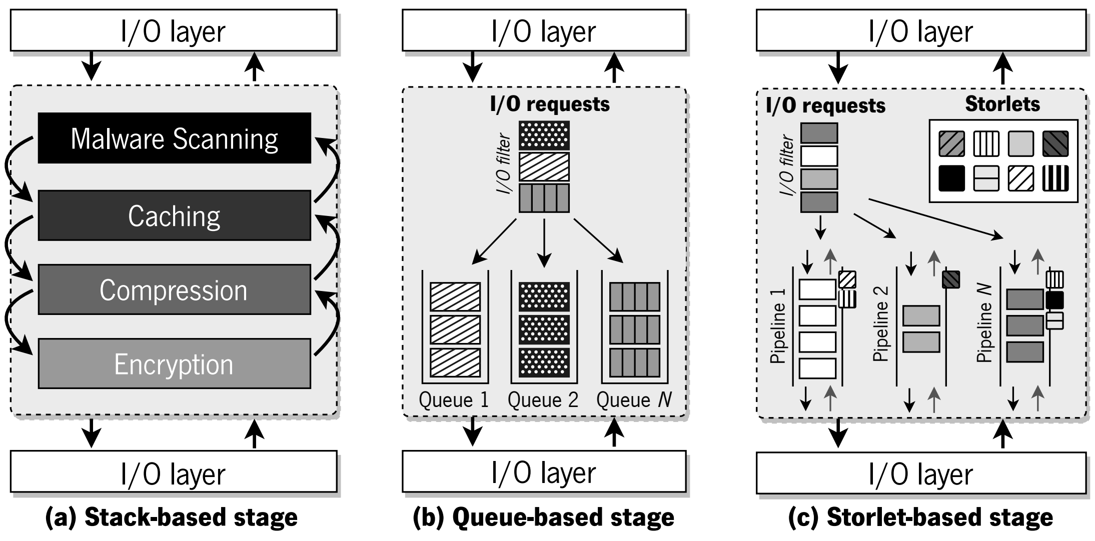


Figure 2.3: SDS data plane stage designs, namely (a) Stack-, (b) Queue-, and (c) Storlet-based.

properties (*e.g.*, dependability, simplicity, generality) that portray other aspects of SDS [112, 211]. However, as they are not covered by the majority of systems, they are not contemplated in the taxonomy presented in this chapter. One such property is dependability, which refers to the ability of a data plane to ensure availability and tolerate faults of the storage mechanisms implemented at stages, regardless of employing performance management, data management, or data routing objectives [84, 211].

2.2.2 Stage Design

We now categorize data plane stages in three main designs. Each design respects to the internal organization of a stage's enforcement structure, regardless of being applicable at different points of the I/O path. Figure 2.3 illustrates such designs, namely *(a) Stack-*, *(b) Queue-*, and *(c) Storlet-based* data plane stages. These designs are contemplated as part of the taxonomy for classifying SDS systems (§2.5).

Stack-based stages. *Stack-based* data plane stages provide an interoperable layer design where layers correspond to specific storage mechanisms and are *stacked* according to installed policies [175]. This design enables an independent and straightforward layer development, introducing a modular and programmable storage environment. The organization of layers is established by the control plane, and may result in a number of stacking configurations tuned to satisfy the installed policies and attend the I/O requirements. Figure 2.3 *(a)* depicts an abstract design of a *stack-based* data plane stage. It comprehends a four-layer stacking configuration, each with a specific storage service to be applied over I/O workflows, namely *malware scanning*, *caching*, *compression*, and *encryption*. The I/O workflow follows a passthrough layout, ensuring that all requests traverse all layers orderly, such that each layer only receives requests from the layer immediately on top, and only issues requests to the layer immediately below. SafeFS [175], for example, provides a framework for developing stackable storage services by re-purposing existing FUSE-based file system implementations to employ different policies over I/O flows, such as encryption and erasure coding.

Stacking flexibility is key to efficiently reuse and adapt layers to different environments. However, this vertical alignment may limit the ability to enforce specific policies (*e.g.,* data routing), limiting the available policy spectrum exposed to the control plane. Current stack-based solutions are attached to specialized I/O layers and usually deployed at lower levels of the I/O stack, such as file systems and block devices [158, 175].

Queue-based stages. *Queue-based* data plane stages provide a multi-queue storage environment, where queues are organized to employ distinct storage functionalities over I/O requests [211]. Each queue is a programmable storage element that comprehends a set of rules to regulate traffic differentiation and define its storage properties. Such properties govern how fast queues are serviced, employ specific actions over data, and forward I/O requests to other points of the I/O path. Examples of such properties include the use of token-buckets [33], priority queues [33, 65], and scheduling mechanisms [81, 199] (these properties are further detailed in §2.5.3). Figure 2.3 *(b)* depicts the design of a *queue-based* data plane stage. Incoming requests are inspected, filtered, and assigned to the corresponding queue. For instance, IOFlow [211] provides programmable queues to employ performance and routing mechanisms over virtual instances and storage servers, enabling end-to-end differentiation and prioritization of I/O workflows.

Such a design makes stages more flexible and modular, and simplifies their overall orchestration [211]. However, as demonstrated by complementary research fields [88, 89], queue structures are primarily used to apply performance-oriented policies. As such, trading customization over a more tailored design turns the integration and extension of alternative storage services a challenging endeavor.

Storlet-based stages. *Storlet-based* data plane stages abstract storage functionalities into programmable storage objects (*storlets*) [84, 194]. Leveraging from the principles of active storage [182] and *OpenStack Swift Storlets* [164], a *storlet* is a piece of programming logic that can be injected into the data plane stage to perform custom storage mechanisms over incoming I/O requests. This design promotes a flexible and straightforward storage development, and improves the modularity and programmability of data plane stages, fostering reutilization of existing programmable objects. Figure 2.3 *(c)* depicts the abstract architecture of a *storlet-based* data plane stage. Stages comprehend a set of pre-installed storlets that comply with initial storage policies. Policies and storlets are kept up-to-date to ensure consistent actions over requests. Moreover, the stage provides several storlet-made pipelines to efficiently enforce storage mechanisms over I/O workflows. At runtime, I/O flows are intercepted, filtered, classified, and redirected to the respective pipeline. Crystal [84], for example, provides a storlet-enabled data plane that allows system designers run customized performance and data management services over I/O requests.

The seamless development of storlets ensures a programmable, extensible, and reusable SDS environment. However, as the policy scope increases, it becomes harder to efficiently manage the storlets at stages. Such an increase may introduce significant complexity in data plane organization and lead to performance penalties on pipeline construction, pipeline forwarding, and metadata and storlet management.

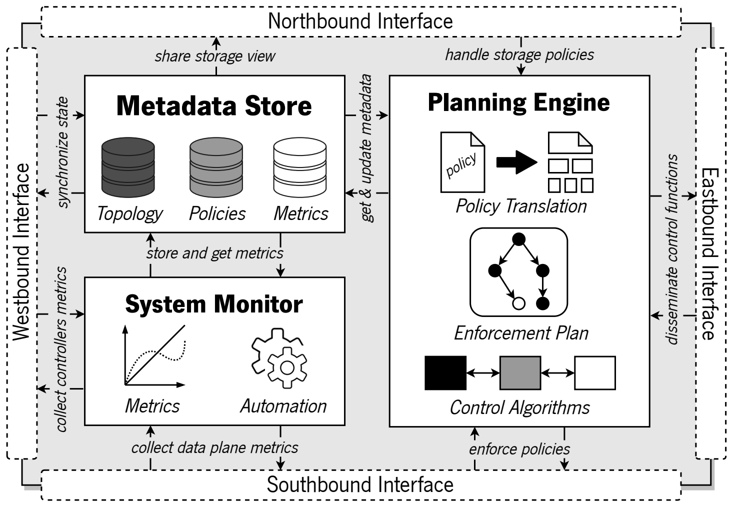
2.2.3 Summary

The SDS data plane is a multi-stage component distributed along the I/O path, where each stage mediates the I/O workflows between two layers. Stages intercept I/O requests and enforce specific storage mechanisms (*e.g.,* rate limiters, encryption and compression schemes, caches) to comply with the specified policies. Moreover, stages communicate with the control plane through a *southbound interface*. Internally, stages can be organized with different designs, including *stack*, *queue*, and *storlet*-based. Further, depending on the infrastructure requirements and policies to be enforced, stages can expose different properties, such as *programmability*, *extensibility*, *placement*, *transparency* and *policy scope*.

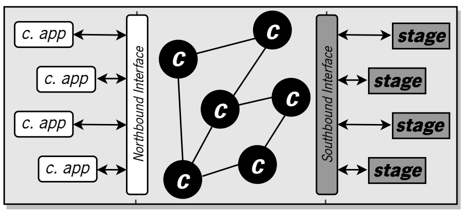
2.3 Control Plane — Controllers

Similarly to SDN, SDS control planes provide a logically centralized controller with system-wide visibility that orchestrates a number of data plane stages. However, even though identical in principle, the divergence between SDN and SDS research objectives may impact the entailed complexity on designing and implementing production-quality SDS systems. First, the introduction of a new (storage) functionality to employ over I/O workflows cannot be arbitrarily assigned to stages, since it may introduce significant performance penalties and compromise the enforcement of other policies. For instance, SDN data planes are mainly composed with simple forwarding services, while SDS data planes may comprehend performance functionalities, which are sensitive to I/O processing along the I/O path, but also data management ones, which entail additional computation directly impacting the processing and propagation time of I/O workflows. Thus, controllers are required to perform extra computations to ensure the efficient placement of storage features, preventing policies to conflict with each other, and ensuring a correct execution of the SDS environment. Second, since the domain of both I/O mechanisms and policies is broader than in SDN, ensuring transparent control and policy specification introduces increased complexity to the design of controllers (*e.g.,* decision making, service placement).

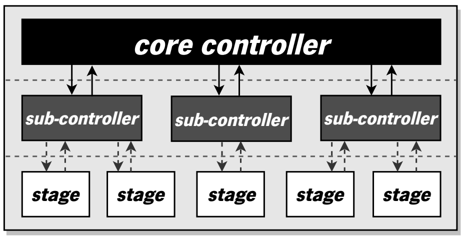
As depicted in Figure 2.1, controllers are the mid-tier of a SDS system and provide the building blocks for orchestrating data plane stages according to the actions of control applications built on top. Despite distributed, the control plane shares its control logic through a logically centralized controller that comprehends system-wide visibility. This eases the design and development of general-purpose control applications, provides a simpler and less error prone development of control algorithms, ensures an efficient distribution and enforcement of policies, and enables data plane stages to be adjusted holistically (*i.e.,* encompassing the global storage environment) [84, 101, 205, 211, 226]. Unless stated otherwise, we define a *controller* as a logically centralized component, even though physically distributed. A controller can be partitioned in three functional modules, namely a *metadata store*, a *system monitor*, and a *planning engine*, as illustrated in Figure 2.4 (a). Each of these modules comprises a particular set of control features shared between controllers, or designed for a specific control device. Moreover, these modules are programmable and allow SDS users (*e.g.,* system designers, system administrators) to install, at



(a) Internal organization of a SDS controller.



(b) Controllers organized in a flat distribution.



(c) Controllers organized in a hierarchical distribution.

Figure 2.4: SDS controllers architecture: (a) presents the organization of the internal components of a SDS controller; (b) and (c) depict the design of SDS control plane controllers, namely *Flat* and *Hierarchical*.

runtime, custom control actions to employ over the system (*e.g.,* control algorithms for accurate policy enforcement, collection of monitoring metrics).

The *metadata store* holds the essential metadata of the storage environment and ensures a synchronized view of the infrastructure. As depicted in Figure 2.4 (a), different types of metadata are stored in separate instances, namely *topology*, *policies*, or *metrics*. The *topology* instance maintains a topology graph that comprehends the distribution of data plane stages, along the assigned storage mechanisms and information about the resources of each node a stage is deployed (*e.g.,* available storage space, Central Processing Unit (CPU) usage, Process Identifier (PID), hostname). For instance, sRoute's controller maintains an up-to-date topology graph with the capacity of physical resources, and shares it with control applications to define accurate storage policies [205]. The *policies* instance holds the storage policies submitted by applications, as well as those installed at data plane stages. The *metrics* instance persists digested monitoring metrics and statistics of both control and data flows, which are used to adapt data plane stages to meet applications' requirements. Further, to ensure a dependable SDS environment, this metadata is usually synchronized among controllers [205, 211].

The *system monitor* collects, aggregates, and transforms unstructured storage metrics and statistics into general and valuable monitoring data [84, 94, 226]. It captures relevant metrics of the physical storage environment (*e.g.,* device and network performance, available storage space, Input/Ouput Operations per Second (IOPS), bandwidth usage) from controllers, data plane stages, and existing monitoring tools (*e.g.,* dstat [167], `/proc` file system [166]), ensuring that it comprises a clear understanding on the performance of the storage stack. Such metrics are then analyzed and correlated, bringing significant insights about the system status that allow the controller to optimize the infrastructure by (re)configuring data plane stages, and assist other modules in policy enforcement and feature placement activities. For example, Mirador [226] collects device and network load and traces workload profiles to build an accurate model of the infrastructure to assist in workflow customization and data placement enforcement.

The *planning engine* implements the control logic responsible for policy translation, policy enforcement, and data plane configuration. Policies submitted by control applications are parsed, validated, and translated into stage-specific rules, which will be installed at the respective data plane stage. Policy enforcement is achieved through different control algorithms and strategies that specify how the data plane handles I/O workflows and define the most suitable place for policies to be enforced [211]. Examples of such control algorithms and control strategies include proportional sharing, I/O isolation and priority, feedback control, and performance modeling (further detail in §2.5.2). Both translation and enforcement operations may lead the controller to interact with a single data plane stage (*e.g.,* to install a particular rule) or a number of stages to perform distributed enforcement, such as cluster-wide bandwidth aggregation and I/O prioritization [206, 211]. Furthermore, the *planning engine* can also exercise automation operations without additional application input, ranging from simple management activities such as increasing or decreasing the number of controllers, to more complex tasks (*e.g.,* fine-tuning storage policies, reconfiguring data plane stages) [84].

To provide a seamless integration with the remainder SDS layers, the controller connects to a *northbound* and a *southbound interfaces* to interact with control applications and data plane stages, respectively. Similar to other software-defined approaches, the control plane comprehends a network of controllers connected through a *westbound/eastbound interface*, as illustrated in Figures 2.1 and 2.4 (a). This interface defines the instruction set and communication protocols between controllers, being used for synchronization, exchange data, fault tolerance, monitoring, and depending on the control plane architecture, assign control tasks to other controllers (further description is presented in §2.3.2). Further, this interface aims at achieving interoperability between different controllers [115]; however, despite its clear position in the SDS environment, current literature does not explore nor details about such an interface.

2.3.1 Properties

Similarly to other software-defined approaches, designing and implementing production-quality SDS systems requires solving important challenges at the control plane [112]. We now define the properties that characterize SDS controllers, namely *scalability*, *dependability*, and *adaptability*, which are also contemplated as part of the taxonomy for classifying SDS systems (§2.5).

Scalability. Scalability refers to the ability of a control plane to efficiently orchestrate and monitor a number of data plane stages. Similarly to SDN, the control plane can either be physically centralized or distributed [19]. A physically centralized control plane consists of a single SDS controller that orchestrates the overall storage infrastructure, which is an attractive design choice in terms of simplicity [162, 201, 215]. However, physical control centralization imposes severe scalability, performance, and dependability requirements that are likely to exhaust and saturate underlying system resources, largely dictating the end performance of the storage environment. As the amount of stages increases so does the control traffic destined towards the centralized controller, bounding the system performance to the processing, memory, and network power of this single control unit. Hence, despite the obvious limitations in scale and reliability, such a design may be only suitable to orchestrate small-to-medium storage infrastructures [115].

Production-grade SDS controllers must be designed to attend the scalability, performance, and dependability requirements of today's production storage systems, meaning that any limitations should be inherent to the overall infrastructure and not from the actual SDS implementation. Thus, physically distributed controllers can be scaled up to attend such requirements. While sharing a logically centralized service, multiple connected controllers orchestrate the storage infrastructure by sharing control responsibility, and thus alleviating overall control load. Leveraging from existing classifications in the SDN literature [19], distributed SDS controllers can follow a *flat* or a *hierarchical* distribution (§2.3.2). Flat designs (Figure 2.4 (b)) imply horizontal control partitioning to provide a replicated control service, forming a reliable, fault tolerant, highly available cluster of controllers [84, 211, 226]. On the other hand, hierarchical-based designs (Figure 2.4 (c)) imply the vertical control partitioning to provide a scalable SDS control plane [107, 205].

Dependability. Dependability refers to the ability of a control plane to ensure sustained availability, resiliency, and fault tolerance of the control service [13]. Physically centralized controllers represent a single point of failure, leading to the unavailability of the control service upon a failure. As a result, the overall system becomes unsupervised, incapable of regulating incoming storage policies and orchestrate the data plane tier. The system should handle failures gracefully, avoiding single points of failure and enabling fault tolerance mechanisms. Similarly to SDN [24, 32, 112], physically distributed SDS solutions provide coordination facilities for detecting and recovering from control instance failures [84, 205, 226]. In this model, controllers are added to the system to form a replicated, fault tolerant, and highly available SDS environment. Moreover, existing distributed controllers consider the different trade-offs of performance, scalability, and state consistency, and provide distinct mechanisms to meet fault tolerance and reliability requirements. For instance, controllers may assume a clustered format to achieve fault tolerance through active or passive replication strategies by resorting to state machine replication [122, 163] or implementing a primary-backup approach where one main controller orchestrates all data plane elements while remainder control instances are used for replication of the control service [31].

While some solutions comprehend a strong consistency model to ensure correctness and robustness of the control service, others resort to more relaxed models where each controller is assigned to a subset of the storage domain and holds a different view of the infrastructure. Regarding control distribution, flat controllers are designed to ensure sustained resilience and availability [84, 211], while hierarchical controllers focus on the scalability challenges of the SDS environment [101, 205].

Depending on the storage context, the dependability offered by the control plane can be coupled to a specific I/O layer. Specifically, as some SDS systems are directly implemented over existing storage systems, such as Ceph (*e.g.,* Mantle [195], SuperCell [215]) and OpenStack (*e.g.,* Crystal [84]), the control plane's dependability can be bounded by the dependability of the respective storage system.

Adaptability. Adaptability refers to the ability of a control plane to respond, adapt, and fine-tune enforcement decisions under requirements that change over time. The high demand for virtualized services has driven data centers to become extremely heterogeneous, leading storage components and data plane stages to experience volatile workloads [8, 55, 211]. Moreover, designing heterogeneity-oblivious SDS systems with monolithic and homogeneous configurations can severely impact the storage infrastructure, hindering the ability to accurately enforce policies [84].

Therefore, SDS controllers must comprehend a self-adaptive design, capable of being dynamically adjusted and tuned (*e.g.,* policy values, data plane stage configurations) to provide responsive and accurate enforcement decisions [84]. As enforcement strategies directly impact I/O workflows, employing self-adaptive and autonomous mechanisms over SDS controllers brings a more accurate and dynamic enforcement service. Moreover, due to the fast changing requirements of the storage environment, data plane configurations rapidly become subpar, and thus, automated optimizations of data plane resources is key to ensure efficient policy enforcement and resource usage. Current strategies to provide adaptable SDS control include control-theoretic mechanisms, such as *feedback controllers* that orchestrate system state based on continuous monitoring and data plane tuning [108, 139, 211], and *performance modeling,* such as heuristics [225], linear programming [158, 248], and machine learning [107] techniques (further detail on these strategies is presented in §2.5.2).

2.3.2 Controller Distribution

SDS literature classifies the distribution of controllers as logically centralized, despite being physically distributed for the obvious reasons of scale and resilience [205, 211]. We now categorize distributed control planes regarding their controller distribution (topology). Figure 2.4 illustrates such designs, namely (b) *flat* and (c) *hierarchical* controllers. Both designs are contemplated as part of the taxonomy for classifying SDS control elements (§2.5).

Flat. Flat control planes provide a horizontally partitioned control environment, where a set of interconnected controllers act as a coordinated group to ensure a reliable and highly available control service while preserving logical control centralization. Depending on the infrastructure requirements (*e.g.,* performance, resiliency), controllers may be organized differently. For instance, some implementations may provide a cluster-like distribution, where a single controller orchestrates the overall storage domain, while others are used as backups that can take over in case the primary fails. Under this scenario, the centralized controller handles all stage-related events (*e.g.,* collect reports and metrics), disseminates policies, generates comprehensive enforcement plans, and enforces policies. Moreover, the control plane provides the coordination facilities to ensure fault tolerance and strong consistency by relying on state machine replication [205, 226] or simple primary-backup strategies [31]. Mirador [226], follows such an approach, by resorting to a coordination service to ensure a highly available control environment [99]. This design allows distributed controllers to comprise strong consistency properties, ensure high availability of the control service, and ease control responsibility. However, since control remains centralized, this cluster-based distribution falls short when it comes to scalability, limiting its applicability to small-to-medium sized storage infrastructures [211].

Other solutions, as depicted in Figure 2.4 (b), may provide a network-like flat control plane, where each controller is responsible for a subset of the data plane [84, 112]. Namely, each controller orchestrates a different part of the infrastructure, synchronizing its state with remainder controllers with strong or eventual consistency mechanisms. Upon the failure of a controller, another may assume its responsibilities until it becomes available. For instance, Crystal [84] holds a set of autonomous controllers, each running a separate control algorithm to enforce different points of the storage stack. This network-like design ensures an efficient control service and provides a flexible consistency model that allows the SDS system to scale to larger environments than cluster-based approaches. However, this control model hardens the control plane's ability to share a logical centralized setup and synchronized view to control applications, hindering its applicability to large-scale production storage infrastructures. Further, with the emergence of novel computing paradigms composed of thousands of nodes, such a serverless cloud computing [188] and Exascale supercomputers [61], this design may hold severe scalability and performance constraints.

Hierarchical. The continuous communication between controllers and data plane stages, through enforcement of policies and statistic collection, hinders the scalability of the control plane [107, 205]. To alleviate the load on the centralized controller, both control and management workflows must be handled closer to data plane resources, and minimized as possible without compromising system correctness. Thus, similarly to distributed SDN controllers [102, 235, 236], hierarchical distributions address such a problem by organizing SDS controllers in a hierarchical disposition [101, 107, 205]. Controllers are hierarchically ranked and grouped by control levels, each of them respecting to a cumulative set of control services. This approach distributes the control responsibility to alleviate the load imposed over centralized services, enabling a more scalable SDS control environment.

As depicted in Figure 2.4 (c), the control plane is vertically partitioned into *core controllers* and *sub-controllers*. *Core controllers* are placed at the top of the hierarchy, and comprehend overall control power and system-wide visibility. While maintaining a synchronized state of the SDS environment, *core controllers* manage control applications, and orchestrate both data plane and *sub-controller* elements. Moreover, *core controllers* share part of their responsibilities with underlying tiers, propagating the control logic hierarchically. *Sub-controllers* are placed closer to data plane stages and comprehend a subset of control services. Each of these controllers manages a part of the data plane, as well as the control activities that do not require global knowledge nor impact the overall state of the control environment. For example, Clarisse [101] implements a hierarchical control plane over HPC infrastructures that groups control activity through global, application, and node controllers. Since *core controllers* hold global visibility, they perform accurate and holistic decisions over the SDS environment. However, maintaining a consistent view is costly, causing significant performance overhead even when performing simple and local decisions [101, 205]. On the other hand, *sub-controllers* are tailored for specific control operations, providing faster and

Table 2.1: Classification of SDS control applications regarding storage objectives.

		Cloud	**HPC**	**Application-specific**
Performance	*Perf. guarantees*	[128, 201, 205, 206, 211, 241, 248]	[40, 107]	[84, 108, 143, 155, 222]
	Prioritization	[128, 206, 208, 211, 225, 241, 247, 248]	[101, 107]	[143, 155, 222]
	Perf. control	[139, 158, 200, 206, 208, 211, 247]	[107]	[84, 143, 194, 215, 222]
Resource Management	*Fairness*	[201]		[143, 155]
	Cache management	[205, 206]		[84, 195]
	Device management	[158]		[50, 215]
	Flow customization	[139, 205, 225]	[101]	[21, 226]
Data Management	*Data redundancy*	[177]		[21, 50, 175]
	Data reduction	[158, 177]		[50, 84]
	Data placement	[158, 162, 177, 205]	[101]	[21, 194, 195, 215, 226]
	(Meta)Data org.	[162, 177]		[21, 194, 195]
Security	*Encryption*	[177]		[175]
	Malware scanning	[211]		
Other Storage Objectives		[158, 162, 177]	[101]	[21, 175, 194]

fine-grained local decisions over stages. In case a *sub-controller* cannot perform certain control actions over its elements, it passes such responsibility to higher ranking controllers.

Communication between control instances is achieved through the *westbound/eastbound interface*, and is used for establishing the control power and policy dissemination, periodic state propagation for synchronization, and health monitoring events.

2.3.3 Summary

The SDS control plane provides a logically centralized controller with system-wide visibility that holistically orchestrates all data plane stages. Controllers can be physically distributed, and follow a *flat* or *hierarchical-based* distribution. Control applications are built on top of controllers and interact through a *northbound interface*. Controllers continuously communicate with data plane stages, through a *southbound interface*, for collecting I/O statistics and enforcing stage-specific rules to ensure storage policies are met at all times. Communication between controllers is established through a *westbound/eastbound interface*. Further, depending on the infrastructure requirements, controllers can have different properties, such as *scalability*, *dependability*, and *adaptability*.

2.4 Control Plane — Control Applications

Control applications are the entry point of the SDS environment and the *de facto* way of expressing how I/O workflows should operate. Applications exercise direct control over controllers by defining the control logic through policies and control algorithms, which are further translated into fine-grained stage-specific rules to be employed over I/O requests. Examples of control algorithms include proportional sharing, prioritization and isolation, and shares and reservations, which are further detailed in §2.5.2. Similar to other software-defined approaches [104, 115], control applications introduce a *specification* abstraction into the SDS environment to express the desired storage behavior without being responsible for implementing the behavior itself. Moreover, the logical centralization of control services allows control applications to leverage from the same control base, leading to an accurate, consistent, and efficient policy creation.

Existing control applications are designed for a variety of storage contexts and cover a wide array of functionalities, including *performance*, *resource* and *data management*, *security*, and *other storage objectives*. *Performance* objectives aim at enforcing performance guarantees (*e.g.,* throughput and latency SLOs [128]), prioritization (*e.g.,* bandwidth allocation according to applications' priority [211]), and performance control (*e.g.,* I/O isolation). On the management side, resource-centric objectives enforce fairness between applications accessing shared storage systems, as well as caching and device management policies (*e.g.,* caching schemes [206], storage quotas [50]), and I/O flow customization (*e.g.,* modify the I/O endpoints of a layer [205]). Data-centric objectives, on the other hand, enforce objectives directly applicable over data and metadata such as data redundancy, data reduction, data placement, and data and metadata organization [196]. *Security*-based objectives enforce encryption and malware scanning rules to ensure privacy and confidentiality of sensitive data. *Other storage objectives* such as energy efficiency and elasticity control seek to provide additional properties to storage systems. Table 2.1 classifies existing SDS control applications regarding storage objectives, while organized by storage infrastructure.

Finally, a *Northbound interface* connects control applications and controllers by abstracting the distributed control environment into a language-specific communication interface, hiding unnecessary infrastructure details, while allowing straightforward application-building and policy specification. Such a design fosters the integration and reutilization of different control applications between SDS technologies, enabling an interoperable control design. However, current work on SDS lacks a standard *Northbound interface*, which limits the ability to combine different control applications throughout distinct storage contexts and SDS technologies.

2.5 Survey of Software-Defined Storage Systems

This section presents an overview of existing SDS systems regarding storage infrastructure (§2.5.1), control strategy (§2.5.2), and enforcement strategy (§2.5.3). §2.5.4 discusses key differences between SDS systems. Systems are classified according to the taxonomies described in §2.2 and §2.3.

2.5.1 Survey by Infrastructure

Storage infrastructures comprise different requirements and restrictions, and thus, the design and combination of SDS properties may vary significantly with the targeted infrastructure. To provide a comprehensive survey of SDS systems, we describe them in a twofold. Table 2.2 classifies SDS systems according to the taxonomies described in §2.2 and §2.3, while grouping them by storage infrastructure, namely cloud, HPC, and application-specific storage stacks. This table highlights the design space of each infrastructure and depicts current trends and unexplored aspects of the paradigm that require further investigation.

Then, the textual description presented at each section (§2.5.1.1– §2.5.1.3) draws focus on the environment and context where each system is applied, as well as the enforced storage objectives and other aspects that differentiate these solutions. The classification considers systems from both academia and industry.[3] Commercial solutions whose specification is not publicly disclosed are not considered. Systems that follow the SDS design principles and storage functionalities described in §2.1, targeting at least one of the planes of functionality, are contemplated in this classification.[4]

2.5.1.1 Cloud Infrastructures

Cloud computing infrastructures offer enterprise-grade computing and storage resources as public utilities so customers can deploy and execute general-purpose services in a flexible pay-as-you-go model. Cloud premises consist of hundreds to thousands of compute and storage servers. Compute servers are virtualized and abstract multiple physical servers into a pool of resources exposed through Virtual Machine (VM) or containers. Resources are shared between tenants and mediated by hypervisors. Storage servers accommodate heterogeneous storage systems and devices with distinct levels of granularity and performance. These servers persist all data and are exposed to VMs as virtual devices. Compute and storage servers are connected through high-speed network links that carry all infrastructure traffic. However, behind this virtualized environment lie complex and preconfigured I/O stacks, making the enforcement of storage requirements and end-to-end control over storage resources harder [87, 211]. While several systems have been proposed to partially address this problem (*e.g.,* QoS provisioning and scheduling [27, 87–89, 233]), these have not considered end-to-end enforcement nor holistic orchestration of resources. To address such shortcomings, SDS-enabled systems have moved towards cloud infrastructures.

The term Software-Defined Storage was first introduced by IOFlow [211]. Specifically, IOFlow enables end-to-end policy enforcement under multi-tenant architectures. Queue-based stages employ performance and routing primitives from VMs to shared storage, allowing efficient performance isolation, differentiated I/O treatment, and end-to-end performance control. A reactive flat-based control plane discovers stages and dynamically configures their I/O mechanisms to attend the manifold objectives of applications built on top (*e.g.,* bandwidth aggregation, prioritization, throughput and latency objectives). While originally designed to enforce policies over storage-related layers, it was later extended to support caching and networking [205, 206]. Moirai [206] extends IOFlow's design to exercise direct and coordinated control over the distributed caching infrastructure to improve resource utilization and achieve performance isolation and QoS guarantees. Stages are deployed as stackable and programmable caching instances to employ performance mechanisms over incoming I/O requests, such as workload aggregation and maximization of cache hit ratios. At the control plane, a logically centralized controller, built on top IOFlow's traffic classification mechanism, orchestrates stages holistically, and maintains (cache) consistency and coherence across the I/O stack. Further, it continuously monitors the infrastructure and maintains key performance

[3]Industrial SDS systems are marked with an ***i*** in the classification table.

[4]Systems not originally defined as SDS but follow the same design principles are marked with ⋆ in the classification table.

Table 2.2: Classification of Software-Defined Storage systems regarding storage infrastructure.

	Control Plane				Data Plane						Interface
	Distribution	Scalability	Dependability	Adaptability	Design	Programmability	Extensibility	Placement	Transparency	Scope	
IOFlow [i] [211]	*F*	◐	●	◐	*Q*	◐	◐	*MP*	○	*PR*	↑ ↓
Moirai [206]	*F*	◐	●	◐	*S*	◐	◐	*MP*	●	*P*	↓
sRoute [205]	*H*	●	●	◐	*Q*	●	◐	*MP*	○	*PR*	↑↔↓
JoiNS [225]	*H*	●	---	◐	*Q*	◐	○	*DSP*	○	*PR*	↔ ↓
*Façade** [139], *AVATAR** [241]	*C*	○	○	◐	*Q*	◐	○	*SP*	●	*P*	↓
*Pisces** [201], *Libra** [200]	*C*	○	○	◐	*Q*	○	○	*DSP*	○	*P*	↓
*PM** [248], *WC** [247]	*F*	◐	●	●	*Q*	○	○	*DSP*	○	*P*	↓
*PSLO** [128]	*C*	○	○	●	*Q*	◐	○	*DSP*	○	*P*	↑ ↓
*Wisp** [208]	*D*	●	---	◐	*Q*	◐	○	*DSP*	●	*P*	↔ ↓
*Wiera-Tiera** [162, 177]	*C*	○	○	◐	*St*	●	●	*DSP*	●	*PDR*	↓
flexStore [158]	*F*	◐	●	◐	*S*	◐	○	*DSP*	◐	*PD*	↑ ↓
Clarisse [101]	*H*	●	◐	○	*Q*	◐	○	*DSP*	○	*PR*	↑↔↓
SIREN [107]	*H*	●	◐	◐	*Q*	◐	○	*MP*	○	*P*	↔ ↓
Retro [143]	*F*	◐	●	◐	*Q*	◐	◐	*MP*	○	*P*	↑ ↓
*Cake** [222]	*C*	○	○	◐	*Q*	◐	○	*MP*	●	*P*	↑ ↓
Crystal [84], *Mass* [45]	*F*	◐	●	◐	*St*	●	●	*DSP*	◐	*PD*	↑↔↓
Coho Data [i] [50, 226]	*F*	◐	●	◐	*S*	◐	○	*DSP*	●	*PDR*	↑ ↓
Mantle [195]	*C*	○	○	◐	*St*	●	○	*DSP*	○	*P*	↑ ↓
SuperCell [215]	*C*	○	○	◐	*St*	○	○	*DSP*	●	*PD*	↓
Malacology [194]	—	—	—	—	*St*	●	◐	*DSP*	○	*PDR*	↓
SafeFS [175]	—	—	—	—	*S*	◐	●	*SP*	●	*PD*	↓
*Triage** [108]	*F*	◐	●	●	*Q*	○	○	*DSP*	●	*P*	↓
PADS * [21]	—	○	○	○	*St*	●	●	*DSP*	○	*PR*	↑ ↓
*Mesnier et al.** [155]	—	○	○	○	—	◐	◐	*SP*	○	*P*	↓

Properties	**Distribution**	**Design**	**Placement**	**Scope**	**Interfaces**
○ Absent	*C* - Centralized	*S* - Stack	*SP* - Single-point	*P* - Performance	↑ Northbound
◐ Limited	*F* - Flat	*Q* - Queue	*DSP* - Distributed SP	*D* - Data	↔ West/Eastbound
● Manifested	*H* - Hierarchical	*St* - Storlet	*MP* - Multi-point	*R* - Routing	↓ Southbound
--- Unspecified	*D* - Decentralized				
— Not Applicable					

metrics of each workload running on the system (*e.g.,* throughput, read-write proportion, *hit ratio* curves).

To override the rigid and predefined I/O path of cloud infrastructures, sRoute [205] goes towards combining storage and networking primitives. It extends IOFlow's design to employ routing mechanisms throughout the I/O path, turning the storage stack more programmable and dynamic. The data plane comprises programmable switches (*sSwitch* stages), that provide flow regulation and customization, and queue-based stages that implement performance management and I/O differentiation at hypervisors.

Such a design allows I/O workflows to be redirected to any point of the I/O stack (*e.g.,* controller, *sSwitch-enabled* stages). The control plane holds a hierarchical distribution made of a centralized controller and several *control delegates*, which are restricted control daemons that contain a part of the control logic and installed at *sSwitches* for control plane efficiency. Each of these *delegates* performs decisions locally, alleviating the load of the centralized component, and thus providing a more scalable environment. Similarly, JoiNS [225] orchestrates storage and network layers over networked storage premises to achieve strict latency SLOs. While sharing similar control principles, JoiNS leverages from existing SDN data planes [151] and programmable switches to enforce routing primitives over the storage infrastructure, while storage data plane stages implement predetermined performance features over block device drivers (via Internet Small Computer Systems Interface).

While IOFlow-enabled systems are designed to achieve end-to-end optimization in cloud storage, several works address specific problems and layers of the I/O stack in a SDS fashion. Façade [139] and AVATAR [241] propose a virtualization layer that sits between clients (hosts) and the storage devices of shared storage systems, and enforces throughput and latency objectives in the presence of bursty and volatile workloads. Even though not physically decoupled, Façade provides a centralized controller that employs a non-linear feedback loop to allocate bandwidth shares for each workload and adjust stages according targeted workload latencies, and a queue-based data plane that governs the queue depth of a storage device. On the other hand, AVATAR proposes a two-level scheduling framework that enforces 95th percentile latency objectives. At a high level, a centralized controller orchestrates per-workload First In, First Out (FIFO) queues and regulates workflows to achieve isolation, while at a lower level, a queue-based data plane rate limits requests before dispatching them to the storage device. At multi-tenant cloud environments, Pisces [201] and Libra [200] provide system-wide performance isolation and fair resource allocation. In both systems, control and data are not physically decoupled. In Pisces, a centralized controller provides per-tenant weighted fair-shares to enforce throughput objectives, while queue-based stages schedule per-tenant rules over network resources of storage servers. On the other hand, while sharing the same design primitives as Pisces, Libra's stages enforce per-tenant application request reservations over Serial Advanced Technology Attachment (SATA) Solid State Drive (SSD) devices.

Despite providing isolation and fairness over network and storage resources, previous systems focus on sharing storage bandwidth, which is simpler to control than tail latency, as bandwidth is an average over time not affected by I/O path's cumulative interactions. Moreover, single resource enforcement either over storage [139, 200, 241] or network [201], limits the ability to enforce end-to-end storage policies. This led to the design of multi-resource SDS systems [247, 248]. PriorityMeister (PM) [248] combines prioritization and rate limiting over network and storage resources to meet tail latency at the 99.9th and 99.99th percentiles. A proactive flat-based controller automatically orchestrates the data plane under varying degrees of workload burstiness, while queue-based stages deployed over storage and network devices provide per-workload latency differentiation. Each stage comprises multiple rate limiting queues, that even though improving burstiness, introduce increased computation time and number of required computing servers. To address this, WorkloadCompactor (WC) [247] extends the design of PM to consolidate multiple workloads onto a storage server. WC's controller automatically selects rate limiting and priority profiles, enforcing them over storage and network to minimize the number of instances that cloud providers use to serve all workloads. While enforcing high latency percentile objectives, PM and WC cannot simultaneously serve throughput-based services. As such, PSLO [128] provides an efficient storage environment that simultaneously enforces tail latency and throughput objectives over consolidated VMs under shared storage infrastructures. Deployed at the hypervisor-level, PSLO holds a centralized controller and queue-based stages, that employ an integral feedback control loop combined with Linear Programming (LP) models to govern the arrival rate of I/O requests in a per-VM basis.

Multi-tenant systems composed of hundreds of small partitioned services (*e.g.,* Service-Oriented Architecture (SOA)) are often used on cloud premises to build large-scale web applications [208]. These systems comprise fine-grained and loosely coupled services, each running on a physical or virtual machine. However, its limited visibility hinders the ability to provide efficient storage enforcement. Wisp [208] proposes a distributed framework for building efficient and programmable SOAs that adapt storage resources under multi-tenancy. It provides a fully decentralized design, where each SOA holds a controller and data plane stages. Each controller gathers local information and propagates it to other peers to execute distributed control algorithms, while queue stages enforce local performance services over SOA resources.

Cloud providers offer a wide-array of storage services with trade-offs in performance, cost, and durability, leading applications to opt for simplicity instead of resorting to different services with conflicting properties. The Wiera-Tiera SDS system provides a geo-distributed cloud environment that facilitates the use and specification of multi-tiered storage across data centers. At the data plane, Tiera [177] provides a programmable storlet-based middleware that encapsulates and re-purposes existing services into an optimized interface that can be glued to comply with data management and routing policies (*e.g.,* encryption, compression, data placement). At the control plane, Wiera [162] provides a centralized controller that enforces storage policies across multi-tiered data centers. It allows the combination of storage features available at different tiers of the cloud, enabling the creation of new services via composition.

Storage features besides QoS provisioning and performance isolation can be also explored in SDS fashion. For instance, flexStore [158] provides a framework for dynamically adapting a data center to cope with QoS and energy consumption objectives. At the data plane, stack-based stages are employed over storage systems (*e.g.,* object stores) and hypervisors. Stages placed at the object store adjust the data layout of storage devices and collect performance metrics to enforce energy-related policies (*e.g.,* reduce number of storage devices), while hypervisor-level stages employ performance and data management services, such as deduplication and caching management. On the control side, a flat-based controller that leverages from LP models enforces QoS and energy-related policies under multi-tenancy by managing the life cycle of dedicated storage volumes and allocating them to VMs.

Cloud-based SDS has become an active research topic for improving the performance and resource efficiency of cloud storage infrastructures. Enterprise-grade systems such as Red Hat Gluster Storage [92] and Microsoft Windows Server [156], have been paving the way of the paradigm in industry, fostering its adoption at a global scale. Moreover, while several storage optimizations have been proposed to improve the performance of cloud premises, such as proportional sharing and I/O scheduling, these can now be re-purposed as control algorithms to be used in existing SDS systems [27, 87–89, 109, 133, 223].

2.5.1.2 High-Performance Computing Infrastructures

HPC infrastructures are composed of thousands of nodes capable of generating hundreds of 10^{15} floating point operations per second (PFLOPS) at peak performance [213]. Supercomputers are the cornerstone of scientific computing and the *de facto* premises for running compute-intensive applications. Modern infrastructures are composed of compute and storage nodes. Compute nodes perform computational-related tasks through manycore processors that deliver massive parallelism and vectorization. Storage nodes persist applications' data in a shared Petabyte-scale Parallel File System (PFS) (*e.g.,* Lustre [34, 192], GPFS [189], BeeGFS [47], PVFS [39]) that offers high-performance storage on top of hundreds of storage drives. Communication across nodes is made through specialized high-performance interconnects. Further, many of current *Top500* supercomputers comprehend a third group of nodes, namely I/O forwarding nodes (or I/O nodes), that act as a middleware between compute and storage nodes, and are responsible for receiving compute nodes' requests and forward them to storage ones [30, 213]. I/O nodes hold the intermediate results of applications, either in memory or high-speed SSDs, and enable several optimizations over I/O workflows (*e.g.,* request ordering, aggregation, data staging).

The long and complex I/O path of HPC infrastructures make performance isolation, end-to-end control of I/O workflows, and I/O optimizations increasingly challenging [231]. The variety of access patterns exhibited by applications has led HPC clusters to observe high levels of I/O interference and performance degradation, inhibiting their ability to achieve predictable and controlled I/O performance [136, 237]. While several efforts were made to prevent I/O contention and performance degradation (*e.g.,* QoS provisioning [230, 245], job scheduling optimization [105, 203]), neither have considered the path of end-to-end enforcement of storage policies nor system-wide flow optimizations. To this end, SDS systems have been recently introduced to HPC environments.

Clarisse [101] provides the building blocks for designing coordinated system-wide cross-layer mechanisms, such as parallel I/O scheduling, load balancing, and elastic collective I/O. A queue-based data plane stages data between applications and storage nodes, and implements performance and routing mechanisms for transferring data between compute and storage nodes at the middleware layer (*e.g.,* MPI-IO). A hierarchically distributed control plane offers the mechanisms for coordinating and controlling routing-related activities. Controllers are hierarchically deployed over compute premises and perform different levels of control and enforcement. Similarly, SIREN [107] enforces end-to-end performance objectives by dynamically allocating resources according to applications' demands. It introduces the concept of SDS resource enclaves for resource management of HPC storage systems, allowing users to specify I/O requirements via reservation and sharing of compute and storage resources between applications. A hierarchy-based controller efficiently enforces performance objectives over managed resources, while data plane stages deployed throughout the I/O stack (*e.g.,* request schedulers, PFS) dispatch I/O requests through a queue-based structure that enforces the reservations and shares specified by control instances.

The recent efforts on designing and implementing SDS-enabled HPC infrastructures have proven its utility and feasibility on high-performance technologies. As we move to the Exascale era [61, 76], the adoption of the paradigm by the scientific community is key to ensure end-to-end I/O performance over large-scale supercomputers. When compared to other infrastructures, HPC premises comprise different requirements in terms of architecture and hardware, turning unfeasible the applicability of non HPC-based SDS systems over such environments. First, HPC storage backends are generally composed of a shared file system [192], that becomes a major performance bottleneck when concurrently used by hundreds to thousands of jobs competing for shared resources, leading to high-levels of I/O interference and performance degradation [176, 231]. Second, HPC applications generate complex workflows (*e.g.,* scientific simulations, real-time visualizations), that are translated into different storage objectives and services to be employed over I/O flows [101]. Third, since HPC infrastructures are not virtualized (*i.e.,* jobs are executed over bare-metal resources), solutions that actuate over virtualized environments such as IOFlow [211], PSLO [128], or sRoute [205] cannot be directly applied.

2.5.1.3 Application-specific Infrastructures

Application-specific infrastructures are storage stacks built from the ground up, designed for specialized storage and processing purposes to achieve application-specific I/O optimizations [194]. Production-grade clusters include multi-tenant distributed storage systems such as Hadoop [202], Ceph [224], and OpenStack Swift [11], being mainly composed of proxy and storage servers. Proxy servers map application requests to the respective data location, and provide global infrastructure visibility, system-wide management services (*e.g.,* load balancing, lease management), and high availability. Storage servers are user-space daemons that persist application's data. While these systems are built to run on commodity hardware, enterprise-grade infrastructures may hold hundreds to thousands of storage servers interconnected with dedicated network links. Each server accommodates several multicore processors and storage drives hierarchically organized. Such a specialized environment leads to hard-coded designs and predefined I/O stacks, making the programmability of such systems challenging [194]. Further, the absence of performance guarantees and isolation leads to greedy tenants and background tasks (*e.g.,* garbage collection, replication) to consume a large quota of resources, impacting the overall system performance [143, 147]. While several mechanisms have been proposed to address different system intricacies (*e.g.,* workload-awareness, availability), neither have considered end-to-end enforcement of storage policies or improve programmability of specialized stacks. As such, several SDS-enabled systems have been proposed to address such challenges. Even though these infrastructures can be seen as a subfield of cloud computing (or even HPC), for the purpose of this work and to provide a more granular classification, we classify these in a separate category.

The requirements of isolation and fairness in distributed storage systems have led researchers to shift from hard-coded single-purpose implementations to software-defined approaches. Retro [143] enforces performance guarantees and fairness over multi-tenant Hadoop stacks (*e.g.* HDFS [202], HBase [79], YARN [218]), by identifying and rate limiting workflows bottlenecking shared resources. A reactive flat-based controller enforces fine-tuned policies in the presence of bursty and volatile workloads, while queue-based stages abstract arbitrary system resources (*e.g.,* storage devices, CPU, thread pools) and employ performance management features over priority queues, fair schedulers, and token-buckets implemented along the I/O path. Further, Cake [222] introduces end-to-end enforcement of 99^{th} percentile latency objectives over Hadoop storage stacks. Similarly to AVATAR [241], Cake proposes a two-level scheduling framework, where a centralized controller continuously monitors latency performance and orchestrates queue-based stages to perform per-tenant prioritization through proportional sharing and reservations. Stages are deployed at RPC layers, providing differentiated scheduling of I/O requests and enabling multi-resource control throughout the storage stack.

While these systems focus on the performance and resource management of Hadoop stacks, others focus on multi-tenant object stores. Crystal [84] provides a SDS-enabled object store that supports resource sharing and isolation in the presence of heterogeneous workloads. Implemented over the *OpenStack Swift* object store [11], a storlet-based data plane injects user-defined mechanisms over I/O workflows, such as compression, caching, encryption, and bandwidth control. At the control plane, flat-based controllers dynamically adapt stages according to tenants' requirements. Controllers are twofold, divided in *global controllers* with system-wide visibility that continuously control, monitor, and disseminate storage policies to data plane stages and other controllers, and *automation controllers* with limited visibility that enforce dedicated control actions over selected points of the I/O path. Further, Mass [45] extends the design of Crystal to enable the enforcement of workload-aware policies, improving the performance of *OpenStack Swift* under highly volatile workloads.

Commercial storage systems have also experienced a thrust towards the software-defined domain. For instance, Coho Data [50, 226] proposes a SDS enterprise storage architecture that provides efficient, scalable, and highly-available control over high-performance storage devices (*e.g.,* Peripheral Component Interconnect Express (PCIe) storage drives). At the control plane, Mirador [226] provides a flat-based dynamic storage placement service that orchestrates heterogeneous scale-out storage systems. To enforce routing and data management activities, the control plane continuously collects resource metrics and workload profiles of the cluster, and uses solvers to calculate enforcement plans. At the data plane, Strata [50] implements a stack-based network-attached object store that manages high-performance storage devices under multi-tenancy. Stages are both deployed over SDN-enabled switches, for flow customization and data placement, and PCIe flash devices, to employ striping, replication, and deduplication over requests.

As several systems provide a rich spectrum of storage functionalities (*e.g.,* resource sharing, durability, load balancing), some SDS systems rely on these to improve the control functionality of the storage environment [194, 196, 215]. For instance, Mantle [195, 196] decouples management-based policies from the storage implementation, allowing users to fine-tune and adapt the storage environment under volatile requirements. At the control plane, a heuristic-based policy engine injects management policies into distributed storage systems, such as Ceph [224], while a storlet-based data plane abstracts the underlying storage system through a data management language, allowing users to build flexible and fine-grained policies, such as programmable caching and metadata management. On the other hand, SuperCell [215] relies on the flexibility and availability of Ceph, and proposes a SDS-based recommendation engine that measures and provides cluster configurations under varied workload settings. A centralized controller measures workload characteristics (*e.g.,* I/O size, read/write proportion) and generates enforcement strategies tailored to meet the user's requirements in a cost-effective manner. At the data plane, SuperCell fine-tunes storage settings and configurations of Ceph deployments, at runtime, to cope with different performance and data objectives.

Differently, other systems propose novel abstractions and storage features over application-specific stacks [175, 194]. Malacology [194] is a controllerless SDS system that provides novel storage abstractions by exposing and re-purposing code-hardened storage services into a more programmable environment. Rather than creating storage systems from the ground up, Malacology encapsulates existing system functionalities into reusable building blocks that enable general-purpose systems to be programmed and adapted into tailored storage applications via composition. Implemented over Ceph, Malacology decouples policies from storage mechanisms through a storlet-based data plane, which exposes commonly used services as programmable interfaces that hold the main primitives for developing comprehensive storage applications, namely service metadata, data, resource sharing, load balancing, and durability. Following these same principles, SafeFS [175] aims at re-purposing existing FUSE-based file system implementations into stackable storage mechanisms to employ over I/O requests. Specifically, SafeFS provides a flexible and extensible stack-based data plane that abstracts the file system layer to enable the development of POSIX-compliant file systems atop FUSE. Its stackable organization enables layer interoperability and allows system designers to simply stack independent layers to enforce different storage objectives, such as encryption, replication, erasure coding, and caching.

Previous works on application-specific storage have already crossed the path of software-defined principles [21, 108, 155]. Specifically, as a first attempt towards SDS, Triage [108] introduced an adaptive control architecture to enforce throughput and latency objectives over the Lustre PFS [192], in the presence of bursty and volatile workloads. Adaptive flat-based controllers orchestrate per-client I/O workflows, and regulate request queues according to user-defined performance objectives. At the data plane, queue-based stages rate limit requests before dispatching them to Lustre storage servers. Differently, PADS [21] provides a policy-based architecture to ease the development of custom distributed storage systems. Control and data planes are not physically decoupled, and part of the control logic is shared with a storlet-based data plane. Control applications hold routing and blocking policies to define the logic of the correspondent storage system. Routing policies define data flows, while blocking policies specify consistency and durability objectives. At the data plane, stages accommodate a set of common storage services (*e.g.,* replication, consistency, storage interface) that allow system designers to develop tailored systems via composition, by simply defining a set of policies rather than implementing them from the ground up. Further, Mesnier et al. [155] proposes a classification architecture to achieve I/O differentiation at kernel-level. As the performance of compute servers is often determined by the I/O interference and performance degradation of storage servers, it proposes a classification framework that is able to classify I/O requests at compute instances (through *tagging*), and differentiate them at storage servers (block layer) according to user-defined policies, thus ensuring performance isolation and resource fairness.

The introduction of software-defined principles into application-specific storage infrastructures has led to significant improvements in terms of programmability and resource efficiency. Its design allows users to experience sustained QoS provisioning and performance isolation in multi-tenant settings, instead of the formerly predefined and single-purposed approaches. However, as these infrastructures typically provide a homogeneous I/O stack, employing application-specific SDS systems over cloud or HPC can be a challenging endeavor, due to their wide-array of storage subsystems, which do not operate holistically [101], and heterogeneous workloads [107, 201].

2.5.2 Survey by Control Strategy

As SDS systems are employed over different storage contexts, controllers may assume different control strategies to adapt existing services to the specified objectives. We now survey SDS systems regarding their control strategy employed at the control plane, namely *feedback control* and *performance modeling*. Table 2.3 highlights the control strategies and algorithms used by SDS controllers, and depicts the current trends and unexplored aspects of the paradigm.

2.5.2.1 Feedback Control

Control-theoretic approaches have been widely used to provide sustained storage performance [93]. A feedback-based controller avoids the need of accurate performance modeling by dynamically adjusting I/O workflows to meet different storage objectives. It does so through a control loop, that depends on *input metrics*, *control actions*, and *control intervals*. The controller continuously monitors system metrics (*e.g.,* throughput, latency), and validates them with installed storage policies. In case of policy violation, the controller adjusts the data plane stages through control actions, which rely on the enforcement strategy employed at the stage (*e.g.,* adjust arrival rate of I/O, increase queue depth). Monitoring is made periodically in a predefined control interval. Large intervals result in longer unsupervised control periods, leading to policy violations and performance degradation in case of burstiness or volatile workloads. Small intervals lead the controller to react to performance outliers, resulting in fine-grained adjustments that inhibit sustained storage efficiency.

Reactive SDS controllers employed by IOFlow and sRoute continuously collect throughput and latency metrics of different points of the I/O path. For each stage, the controller enforces control actions over a token-bucket that rate limits queues according to max-min fairness algorithms [26, 33], efficiently providing distributed and dynamic I/O enforcement. Differently, controllers of Wisp and Crystal rely on throughput-only observations. Wisp rate limits request queues of micro-services according to different scheduling policies, while Crystal observes per-tenant throughput at *OpenStack Swift* nodes and allocates proportional bandwidth shares to ensure performance guarantees. In proportional sharing, processes are assigned

Table 2.3: Classification of SDS systems regarding control and enforcement strategies.

	Control Strategy			**Enforcement Strategy**			
	Feedback	*Modeling*	*Algorithms*	*T. Bucket*	*Scheduling*	*P. Queues*	*Injection*
IOFlow [211]	*Reactive*		*PS*	●		●	
Moirai [206]	*Reactive*		*S*				
sRoute [205]	*Reactive*		*PS*	●			
JoiNS [225]		*H*				●	
Façade [139], *AVATAR* [241]	*Non-linear*		*PI*		*EDF*		
Pisces [201]			*PS*		*DRF*, *DWRR*		
Libra [200]			*S*		*DDRR*		
PM [248], *WC* [247]	*Proactive*	*LP*	*I*	●		●	
PSLO [128]	*Integral*	*LP*			●		
Tiera [177], *Malacology* [194]							●
Wisp [208]	*Reactive*				*DRF*, *BRF*, *EDF*, *LSTF*		
flexStore [158], *Mirador* [226]		*LP*	*S*				
SIREN [107]		*ML*	*S*		●		
Retro [143]	*Reactive*		*P*	●	*DRF*, *BRF*	●	
Cake [222]	*Reactive*		*PS*		●		
Crystal [84],*Mass* [45]	*Reactive*		*P*				●
Mantle [195], *SuperCell* [215]		*H*					●
Triage [108]	*Adaptive*	*LP*	*I*		●		

Modeling. *(H)euristic, (L)inear (P)rogramming, (M)achine (L)earning.* **Algorithms.** *(P)roportional sharing, (I)solation and priority, (S)hares and reservations.*

with a notion of weight, and resources are proportionally allocated based on it [221]. Mass periodically collects information over system (*e.g.*, CPU and memory usage) and workload-specific metrics (*e.g.*, bytes read/written to *OpenStack Swift*), and based on these, it allocates per-tenant bandwidth shares. Further, Moirai uses average latency and hit ratio curves to adjust the configurations of stack-based caching stages.

Reactive approaches are also used to enforce tail latency objectives over complex storage settings. For instance, Cake provides a two-level scheduling framework that continuously collects latency and resource utilization metrics over distributed storage stacks, and dynamically adjusts per-client queues according to proportional shares and reservations. In these algorithms, shares define the resource allocation that a given process receives, while reservations express the lower bound of I/O performance reserved to a process [107]. Similarly, Retro observes per-workflow latency and resource usage, and employs control actions over token-buckets, schedulers, and priority queues, according to max-min fair shares. The collection of heterogeneous metrics, along the multiple enforcement points deployed over the I/O path, allow Cake and Retro to differentiate workflows and enforce 99^{th} percentile latency objectives over Hadoop stacks.

Nonetheless, while able to enforce different performance objectives over varied storage stacks, reactive controllers cannot sustain efficient performance at high latency percentiles under bursty workloads, as they experience several policy violations before beginning a new control loop and adjusting stages accordingly [248]. As such, several systems follow a proactive control strategy to enforce 99^{th}, 99.9^{th} and

99.99th percentile latencies. For example, PM and WC provide a proactive feedback controller that models per-workload worst-case latency, and enforces different control actions over multiple rate limiters and priority queues, according to isolation and priority rules. Each stage comprises per-workload token-buckets and priority queues, and efficiently enforces services over network and storage resources.

While these systems are designed to either provide throughput or tail latency objectives, PSLO achieves both by providing an integral feedback controller backed by a forecast model. Integral control ensures that the output of a given system sets a value that is resilient to noise or variation in system parameters [93]; in the case of PSLO, the system throughput converges to the SLO target. It does so by continuously monitoring per-VM X^{th} percentile latency and throughput, and adaptively configuring the level of I/O concurrency and arrival rates, providing isolated and differentiated service levels.

Other approaches follow a non-linear feedback control to enforce proportional sharing and isolation in the presence of bursty and volatile workloads [139, 241]. For instance, Façade collects the average latency of requests accessing the storage device of a shared storage system, and dynamically adjusts the depth of the device queue. AVATAR follows a similar approach but enforces 95th percentile latency. Differently, adaptive feedback controllers backed by self-tuning estimators, such as the one proposed by Triage, provide predictive and differentiated storage performance under varying workloads. Triage periodically observes latency perceived by Lustre-deployed applications and throttles per-client request queues to provide sustained throughput and latency.

2.5.2.2 Performance Modeling

Other strategies often used by SDS systems to efficiently control the storage environment are *heuristics*, which control and adjust selected enforcement points to meet a specific storage objective [196, 215, 225], and *performance models*, that characterize the behavior of the system and its workloads [107, 108, 128, 158, 226, 247, 248].

Heuristics. SDS controllers resort to heuristic-based mechanisms to estimate throughput or latency performance of selected points of the I/O stack. For instance, JoiNS continuously monitors latency and bandwidth utilization at network and storage stages, and provides a simple heuristic that estimates network latency of networked storage systems. From this estimation, the controller adjusts priority queues installed at programmable switches to meet average and tail latency requirements. Similarly, SuperCell observes read and write latencies of Ceph storage nodes, and implements a bandwidth-centered heuristic that calculates per-workload maximum bandwidth to provide adaptive configuration under read- and write-intensive workloads. Mantle, on the other hand, supports user-defined heuristics to provide programmable metadata management and load balancing over Ceph deployments.

Linear Programming. Linear Programming (LP) mechanisms are also frequently used to support SDS control actions. For instance, flexStore resorts to an integer linear program to enforce adaptive replica consistency under varied energy constraints, and network and disk bandwidth; while Triage continuously collects latency measurements to serve a recursive least-squares estimator that supports the feedback controller actions [135]. Differently, general-purpose solvers used by Mirador estimate the performance of network-attached storage systems according to user-defined objectives. These solvers leverage from the continuous observations of network and storage resources, as well as periodic workload profiles, to optimize network traffic and data placement.

Latency analysis models are also used to enforce tail latency objectives under bursty scenarios. Leveraging from network calculus principles, PM and WC propose a model that estimates per-workload worst-case latencies. It models multiple system endpoints and induces time, workflow, rate limit, and work conservation constraints to maximize the available time to serve a workload. Similarly, PSLO provides a forecast model that predicts per-VM high percentile latency violations to simultaneously enforce X^{th} percentile latencies and throughput objectives.

Machine Learning. The use of Machine Learning (ML) to implement control strategies has just been recently adopted by SDS controllers. Specifically, SIREN uses a ML-based algorithm (*i.e.,* classification and regression trees [35]) to assign proportional shares and reservations of compute and storage resources to HPC applications. While SIREN proposes resource enclaves for the efficient management of HPC infrastructures, the algorithm identifies opportunities for enclave migrations, due to workload and I/O demand variance. The introduction of such storage automation mechanisms allows more accurate enforcement strategies and fine-grained control over storage infrastructures.

2.5.3 Survey by Enforcement Strategy

The need to enforce varied storage policies throughout the I/O path leads SDS systems to assume different enforcement strategies. We now survey SDS systems regarding enforcement strategy employed at data plane stages, namely *token-buckets*, *schedulers*, *priority queues*, and *logic injection*. Table 2.3 highlights the enforcement strategies used by SDS data planes, and depicts the current trends of the paradigm.[5]

2.5.3.1 Token-Bucket

A token-bucket is an abstract structure used by queues to control the rate and burstiness of I/O workflows. A bucket is configured with a *bucket size*, that delimits the maximum token capacity, and a *bucket rate*, that defines the rate at which new tokens are added. When an I/O request arrives at the queue, it consumes tokens to proceed. If the bucket is empty, the request waits until sufficient tokens are in the bucket. Each bucket executes locally but is configured by SDS controllers according to existing storage policies and the current system state. Several SDS systems resort to token-buckets mechanisms for enforcing performance-oriented policies [143, 205, 211, 247, 248].

Per-queue token-buckets, such as those in IOFlow and sRoute, enforce max-min fair shares over I/O workflows. As stages are deployed throughout the I/O path, queues are adjusted with different rates

[5]Systems that enforce scheduling mechanisms but do not detail about their policies are marked with ●.

and sizes, providing differentiated I/O treatment and dynamic end-to-end control. Similarly, Retro proposes multi-point per-workflow token-buckets, employed over thread pools and RPC queues to achieve performance guarantees and resource fairness objectives in Hadoop stacks.

Per-workload token-buckets enforce tail latency objectives under bursty environments [247, 248]. To better bound the workload burstiness, PM implements multiple token-buckets per-workload at each data plane stage, which in turn are continuously controlled and modeled by a proactive feedback controller. On the other hand, WC optimizes the choice of bucket parameters through a *rate-bucket size curve* that characterizes workload burstiness, while consolidating workloads into a storage server, in order to both meet tail latency objectives and minimize overall resource usage.

2.5.3.2 Scheduling

Scheduling has been a long-term strategy of storage systems to govern how I/O requests are serviced. In SDS-enabled architectures, scheduling is generally made with queues (at data plane stages) to employ proportional sharing algorithms, prioritization and isolation of requests, and enforce performance objectives over storage and network resources. For instance, single queue scheduling systems, such as Façade and AVATAR, manage per-workload requests to meet average and tail latencies objectives. Requests are dispatched to a queue and served to a storage device following an Earliest Deadline First (EDF) policy. As latency objectives are enforced at per-workload granularity, the deadline of a workload is the deadline of its older pending request [139].

Other solutions implement multi-point resource scheduling mechanisms to achieve fairness and sustained latency performance. Retro, for example, orchestrates per-workflow requests, employing a Dominant Resource Fairness (DRF) policy [81] to ensure resource fairness, and a Bottleneck Resource Fairness (BRF) policy [143] to throttle aggressive I/O workflows and ensure proportional use of resources. Similarly, Pisces employs a per-node scheduler that implements DRF and Deficit Weighted Round Robin (DWRR) policies [199, 201] to achieve system-wide fairness in multi-tenant cloud environments. Under a DRF policy, per-node schedulers track the resource usage of each tenant, and recompute its resource allocation to continuously ensure max-min fair shares, while in DWRR, Pisces ensures per-tenant weighted fair shares of throughput. Libra, on the other hand, follows a similar approach but provides throughput reservations over disk resources through a Distributed Deficit Round Robin (DDRR) scheduling policy [129]. Further, Wisp rate limits micro-service workflows through BRF and DRF policies to achieve throughput objectives, and simultaneously prioritizes individual requests through EDF and Least Slack Time First (LSTF) [208] to enforce latency-related objectives.

2.5.3.3 Priority Queues

A number of SDS systems ensure prioritization and performance control through priority queues [143, 211, 225, 247, 248]. Controllers define and adjust the priority of queue-based data planes to provide different levels of latency among workloads according to installed storage policies. For instance, IOFlow, PM, and WC define the priority of token-bucket enabled queues. Specifically, token-buckets serve first the highest priority queues until no token is left, serving next lower priority queues upon the replenish of the bucket. Moreover, PM and WC specify per-workload priority queues over both storage and network resources. Retro, on its turn, enforces per-workflows priority queues over multi-point data plane stages, while JoiNS provides per-workload priority queues over programmable network switches.

2.5.3.4 Logic Injection

Storlet-based stages implement programmable enforcement structures to allow system designers to inject custom control logic over I/O workflows [84, 194, 196, 215]. For instance, Mantle and Malacology leverage from existing storage subsystems of Ceph, such as durability, load balancing, and resource sharing, and inject control logic to enforce performance and data management storage policies. Mantle decouples policies from storage services by letting administrators inject metadata migration code to dynamically adjust metadata distribution of Ceph deployments. On the other hand, Malacology encapsulates existing system functionalities into reusable building blocks and injects Lua scripts to enable general-purpose systems to be programmed and adapted into tailored storage applications via composition. Further, SuperCell continuously monitors read and write requests of Ceph storage nodes and provides different storage reconfigurations to adapt storage settings under volatile workloads.

Differently, Crystal provides a programmable framework that allows the injection of programming logic to perform custom computations over object requests. This design allows administrators to implement a wide-array of storage services to cope with different performance and data management policies, such as compression, cache optimization, and bandwidth differentiation.

2.5.4 Discussion

The SDS paradigm has drawn major focus on providing controlled I/O performance and fairness over cloud and application-specific infrastructures. While generally providing centralized and flat distributions, the next step towards scalable environments is to foster the development of hierarchical and decentralized control planes in such infrastructures. SDS-enabled HPC infrastructures, which are still at an early research stage, comprise hierarchical control designs due to the scale and performance requirements of supercomputers.

The centralized control distribution assumed by several SDS systems presents obvious limitations in scale and resilience. However, systems experience these limitations at different magnitudes, as they may provide different number of stages and enforce policies with different levels of complexity. Specifically, Façade and AVATAR enforce performance-oriented objectives over a single enforcement point, while Pisces, Libra, PSLO, and Cake need to orchestrate multiple points. Further, Wiera, SuperCell, and Mantle enforce data-oriented policies by injecting control logic at stages, which are less demanding than continuously adjusting stages for performance policies.

Differently, other systems follow a flat control distribution, with a prevalence on performance-based policies. While providing a more dependable design, the control centralization leads to clear scalability limitations. Interestingly, as systems enforce different policies over infrastructures, they implement different control strategies to adapt data plane stages to the specified objectives. For instance, some systems like IOFlow resort to feedback control to continuously adjust the storage environment, while others, such as flexStore and Mirador, employ performance modeling strategies. As both provide a single control strategy, they are unable to provide a fully adaptable SDS environment. On the other hand, PM, WC, Triage, and PSLO combine feedback control and performance modeling strategies, thus providing a more adaptable storage environment, capable of enforcing complex storage policies under volatile environments.

Regarding hierarchical controllers, while providing a scalable design, some solutions present dependability limitations. For instance, the failure of a controller in Clarisse and SIREN leads to unsupervised control points in the infrastructure. Contrarily, sRoute and JoiNS issue control delegates to enforce policies in specific points of the I/O path, which in case of failure, can be replaced by another.

On the data plane side, stack-based approaches focus on data management services, since data workflows follow a passthrough layout and do not employ enforcement strategies. Existing stack-based systems may provide dedicated stacks designed for a specific objective such as flexStore and Moirai, or multiple stacking layers as done in SafeFS and Strata to provide a variety of storage services.

Queue-based data planes, on the other hand, are mainly designed for performance objectives. Nonetheless, even though operating over similar storage structures, existing approaches may differ from each other in several aspects. For instance, single queue systems, such as Façade and AVATAR, enforce average performance policies and are unable to meet SLOs at high latency percentiles. Differently, Pisces and Libra provide system-wide performance isolation and fairness over multi-tenant cloud environments by enforcing per-tenant max-min fair shares. Other systems, such as PM and WC, provide multi-resource scheduling to enable complex storage policies to be enforced over network and storage resources. Further, IOFlow and PSLO provide multiple enforcement queues, each serving at different rates in a per-VM basis, providing distributed and dynamic policy enforcement.

Finally, storlet-based systems introduce novel storage abstractions and programmability to existing storage systems. For instance, Crystal, Tiera, and SuperCell, re-purpose existing storage subsystems and configurations to enforce data and routing activities. Malacology, Mantle, and PADS, on the other hand, abstract underlying storage systems to ease the development of custom storage systems. As opposed to stack-based designs, which are transparent as stacks sit between two I/O layers, storlet-based stages introduce increased complexity to the design of storage solutions.

2.6 Lessons Learned and Open Research Challenges

We now discuss the key insights provided by this chapter, grouped by storage infrastructure (***SI***x), planes of functionality, namely data (***D***x) and control planes (***C***x), SDS interfaces (***I***x), and other aspects of the field (***O**x*). We focus on the design space and characteristics of current SDS systems and in possible future research directions of the paradigm.

SI1: SDS research is widely explored over cloud and application-specific infrastructures. SDS research has drawn major focus on cloud and application-specific designs. However, the former are generally composed of centralized and flat controllers and queue-based data plane stages, while the latter focus on storlet-based designs. With the continuous increase of data centers complexity, further research on hierarchical and decentralized control distributions will be needed to attend the ever-growing scalability and dependability requirements.

SI2: HPC-based SDS systems are at an early research stage. The increasing requirements of scale and performance of supercomputers have led to the first advances towards SDS-enabled HPC systems, being composed of hierarchical control distributions. Given the requirements of Exascale infrastructures [6, 61, 76], as well as the approximation of the *computer continuum* [20, 159], novel contributions are expected to foster research in the SDS–HPC field.

SI3: SDS systems for emerging computing paradigms are unexplored. SDS-enabled systems have been employed over modern storage infrastructures to achieve different objectives. However, with the emergence of novel computing paradigms such as serverless cloud computing [188] and IoT [12], a number of challenges (*e.g.,* scalability, performance, resiliency) need to be addressed to ensure sustained storage efficiency. As such, the research and development of novel decentralized SDS architectures (*e.g.,* wide-area SDS systems, gossip-based control protocols), as well as the convergence of different software-defined technologies (*e.g.,* storage, networking, security) will be essential to provide a fully programmable storage environment and attend the requirements of emerging computing paradigms.

D1: Stage design impacts programmability and extensibility. Several SDS data planes rely on queue-based designs, trading customization and transparency for performance. This performance-focused development has led queue-based solutions to experience limited programmability and extensibility. Storlet-based solutions however, comprehend a more programmable and extensible design, being able to serve general-purpose storage requirements.

D2: End-to-end enforcement is hard to ensure. Most SDS systems provide distributed enforcement points bounded to a specific layer of the I/O stack (*e.g.,* hypervisor, file system). Systems that ensure efficient end-to-end policy enforcement comprehend specialized queue-based stages fine-tuned for specific storage services, and require significant code changes to the original codebase. As such, the ability to enforce end-to-end policies in storage infrastructures is tightly coupled to the placement property, and directly influences the transparency of data plane stages.

D3: Performance management services dominate SDS systems. Performance-oriented services have dominated the spectrum of storage policies and mechanisms supported by SDS systems. This design has led to a large research gap for the remainder policy scopes. Nevertheless, the advent of modern storage technologies, such as kernel-bypass [232, 240], storage disaggregation [90, 111, 198], and new storage hardware [14, 28, 185], along with the emergence of novel computing paradigms require significant attention and further investigation in order to adapt, extend, and implement novel storage features over SDS data planes [6]. As such, there is a great research opportunity to explore these new technologies in SDS architectures.

D4: End-to-end storlet data planes are unexplored. Despite the acknowledged programmability and extensibility benefits of storlet-based data planes, end-to-end enforcement has not yet been explored with such design. In fact, there are few proposals on storlet data planes, and several contributions and combinations of storage mechanisms are possible.

D5: Re-purposing of existing storage subsystems is overlooked. Existing services installed at data plane stages are mainly designed from the ground up and fine-tuned for a specific data plane solution. While some solutions already encapsulate existing storage systems as reusable building blocks [175, 194], there is no SDS system that leverages from existing storage subsystems (*e.g.,* QoS provisioning, I/O scheduling) to be re-purposed as programmable storage objects and reused in different storage contexts throughout the I/O path (*e.g.,* key-value stores, distributed file systems, machine learning engines). Such a design would open research opportunities towards programmable storage stacks and foster reutilization of complementary works [109, 176, 233] and existing storage subsystems [62].

D6: Heterogeneous data planes are unexplored. Despite the number of possible configurations and design flavors of SDS data planes, the combination of different stage designs has not yet been explored. This turns the data plane domain mostly monolithic, tailored for specific storage objectives and suboptimal enforcement efficiency. As such, following the steps of the SDN paradigm [115], novel contributions towards heterogeneous data plane environments that explore the different trade-offs of combining stack-, queue-, and storlet-based designs should be pursued.

C1: Current systems are unsuitable for larger environments. A large quota of SDS controllers follow a flat distribution to serve small-to-medium sized storage infrastructures [211]. However, the emergence of novel computing paradigms made of complex and highly heterogeneous storage stacks (*e.g.,* serverless computing, IoT), make current control centralization assumptions unsuitable. As such, leveraging from the initial efforts of decentralized controllers [208], it is essential to further investigate this topic and provide novel contributions towards control decentralization.

C2: Controllers lack programmability. Current hierarchical controllers resort to delegate functions or micro-services to improve control scalability, providing limited control functionality to sub-controllers. Instead, it would be interesting to follow similar design principles as SDS data planes and make control functionality more programmable. Researching different paths of scalability and programmability in SDS would bring major benefits for incoming storage infrastructures [6].

C3: Scalability and dependability are overlooked. SDS systems use a *"logically centralized"* controller to orchestrate the storage environment [211]. However, behind this simple but ambiguous assumption lies a great deal of practical complexity of dependability, leaving no clear definitions on its practical challenges and actual impact in performance and scalability at the overall storage infrastructure. Similarly to other software-defined approaches [125], these assumptions leave several open questions regarding controllers dependability that require further investigation such as fault tolerance and consistency [24, 31, 32, 112], load balancing and control dissemination [59], controller synchronization [187], and concurrency [91].

C4: Controllers are self-adaptable. Several controllers resort to feedback control mechanisms to dynamically adjust data plane stage mechanisms, while few proposals rely on performance modeling techniques to provide a more accurate and comprehensive automation model. However, the storage landscape is changing at a fast pace, with new computing paradigms and emerging hardware technologies vested with novel workload profiles. As such, it is essential to advance the research of autonomous mechanisms for supporting control decisions of SDS controllers, by combining and providing novel control strategies. For instance, exploring distributed ML techniques [127] would be of great utility to attend the needs of both modern and emerging infrastructures, not only for the obvious reasons of scale but also for ensuring new levels of accuracy in heterogeneous and volatile environments.

I1: Communication protocols are tightly coupled to planes of functionality. SDS interfaces are used as simple communication APIs, making communication protocols to be tightly coupled to either the control or data plane implementation. This design prevents the reutilization of alternative technologies and inhibits SDS systems to be adaptable to other storage contexts without significant code changes at the communication codebase. As such, decoupling the communication from control and/or data plane implementations would improve the transparency between the two planes of functionality, foster reutilization of communication protocols, and open research opportunities to attain the communication challenges of novel storage paradigms [12, 188] and network fabrics [6].

I2: Interfaces lack standardization and interoperability. Contrarily to SDN [115], SDS literature does not provide any standard interface to achieve interoperability between SDS technologies. Indeed, this lack of standardization leads researchers and practitioners to implement custom interfaces and communication protocols for each novel SDS proposal, tailored for specific software components and storage purposes. Such a design inhibits interoperability between control and data plane technologies, and hinders the independent development of each plane of functionality. As such, novel contributions towards standard and interoperable SDS interfaces should be expected.

O1: Non-intrusive and end-to-end monitoring are unexplored. Current monitoring mechanisms of SDS controllers are predefined and static. Integrating non-intrusive [67, 160] and end-to-end monitoring systems [145, 231] in these would bring novel insights to the field and would allow assisting controllers to define accurate enforcement strategies over I/O workflows. Further, non-intrusive approaches do not require *a priori* knowledge of the I/O stack and comprehend near-zero changes to the original codebase.

O2: SDS paradigm lacks proper methodologies and benchmarking platforms. Current evaluation methodology of SDS systems is mostly made through trace replaying and benchmarking of selected points of the I/O path, either with specialized or custom-made benchmarks. Thus, there is no comprehensive SDS benchmarking methodology that systematically characterizes the end-to-end performance and design trade-offs of general SDS systems. As such, these considerations motivate for novel contributions in SDS evaluation.

3

PAIO: A Software-Defined Storage Data Plane Framework

As previously mentioned, there are several open challenges to be addressed in the SDS field. In this thesis, we focus on advancing the research on the SDS data plane by designing general applicable, programmable, and adaptable data plane stages, mitigating the challenges **SI2**, **D3**, **D5**, and **D6** (§2.6). Due to the different requirements of performance, scalability, resilience, and resource management between storage infrastructures, there is no *one-size-fits-all* solution that can meet all of these requirements. This thesis aims at filling this gap through a novel SDS system that enables system designers to build custom-made data plane stages fine-tuned for data-centric systems such as databases [38, 191, 209], KVSs [80, 178, 183, 214], streaming processing systems [42, 161], and ML engines [1, 131, 169]. In particular, data-centric systems have become an integral part of modern I/O infrastructures, and to achieve good performance, these systems often implement multiple storage optimizations such as I/O scheduling, differentiation, caching, storage tiering, and replication. However, we argue that these optimizations are implemented in a suboptimal manner, as these are *tightly coupled to the system implementation* and can *interfere with each other due to lack of global context*.

Problem 1: tightly coupled optimizations. Most I/O optimizations are single-purposed as they are tightly integrated within the core of each system [17, 43, 118, 200]. Implementing these optimizations requires deep understanding of the system's internal operation model and profound code refactoring, limiting their maintainability and portability across other systems that would equally benefit from them. For instance, to reduce tail latency spikes at RocksDB [183], an industry-standard Log-Structured Merge tree KVS, SILK proposes an I/O scheduler to control the interference between foreground and background tasks [17]. However, applying this optimization over RocksDB required changing several core modules made of thousands of Lines of Code, including *background operation handlers*, *internal queuing logic*, and *thread pools* [15, 72]. Furthermore, porting this optimization to other KVSs, such as LevelDB [80], PebblesDB [178], or SplinterDB [48] is not trivial, as even though they share the same high-level design, the internal I/O logic differs across implementations (*e.g.*, data structures [48, 178], compaction algorithms [140, 178]).

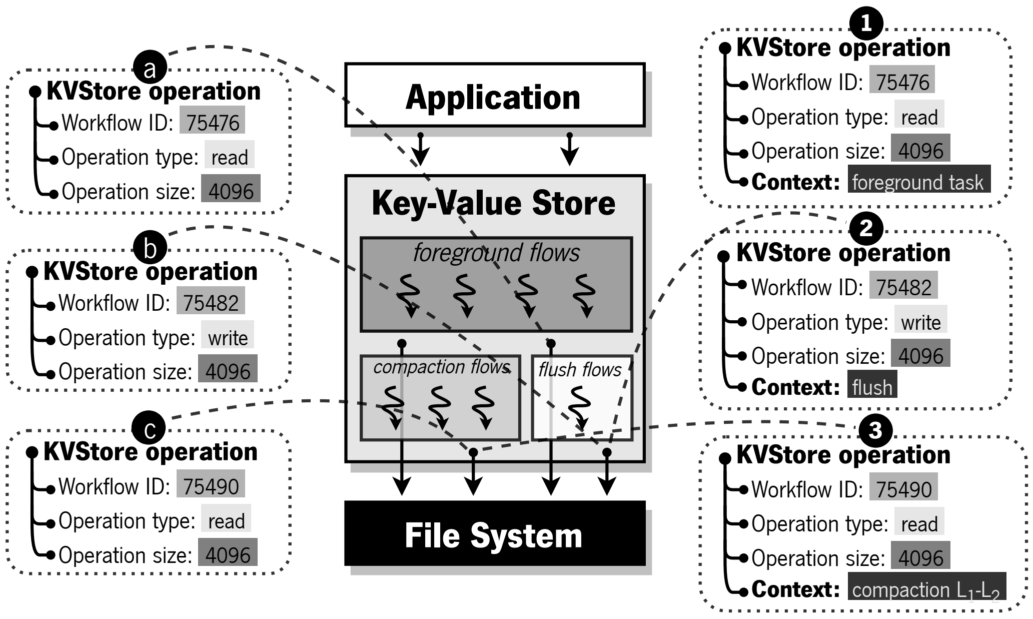


Figure 3.1: Example of the operation workflow of a multi-layered I/O stack made of an Application, KVS, and File System. Left side depicts the regular information that can be extracted from operations between the KVS and File System, while the right side propagates additional request information throughout layers.

Solution: decouple optimizations. A possible solution for this challenge is to disaggregate I/O optimizations from the system's internal logic and move them to a dedicated I/O layer. This way, optimizations become generally applicable and portable across different scenarios (*e.g.*, other systems and I/O layers).

Resulting challenge: rigid interfaces. Decoupling I/O optimizations comes with a cost, since we lose the granularity and internal application knowledge present in system-specific optimizations. Specifically, the operation model of conventional I/O stacks requires layers to communicate through rigid interfaces that cannot be easily extended, discarding information that could be used to classify and differentiate requests at different levels of granularity [5]. For instance, consider the I/O stack depicted in Figure 3.1 made of an *Application*, a *KVS*, and a POSIX-compliant *File System*. POSIX operations submitted from the *KVS* can be originated from different I/O workflows, including foreground (ⓐ) and background flows *i.e.,* flushes (ⓑ) and compactions (ⓒ). The *File System* however, can only observe the size and type (*i.e.,* read and write) of the request, making it impossible to infer its origin. Implementing SILK's I/O scheduler at a lower layer — for example at the *File System* or as a new layer between the *KVS* and the *File System* — would make the optimization portable to other KVS solutions. However, it would be ineffective since it could not differentiate operations submitted from foreground and background workflows (*e.g.,* foreground-based `read` (ⓐ) *vs.* background-based `read` (ⓒ)), or between different background operations (*e.g.,* flush-based `write` *vs.* compaction-based `write`).

Solution: information propagation. To address this, application-level information (*i.e.,* information that is only known to the application itself) must be propagated throughout I/O layers to ensure that decoupled optimizations can ensure the same level of control and performance as system-specific ones.

Resulting challenge: kernel-level layers. While implementing SILK's I/O scheduler at the kernel (*e.g.,* file system, block layer) would promote its applicability across other KVS solutions, it poses several disadvantages. First, for application-level information to be propagated to these layers, it requires breaking user-to-kernel (*i.e.,* POSIX) and kernel-internal interfaces (*e.g.,* Virtual File System (VFS), block layer, page cache), decreasing portability and compatibility [5]. Further, kernel-level development is more restricted and bug prone than in user-level [157, 217]. Finally, these optimizations would be ineffective under kernel-bypass storage stacks (*e.g.,* Storage Performance Development Kit (SPDK) [36, 234], Persistent Memory Development Kit (PMDK) [173, 232]), since I/O requests are submitted directly from the application (user-space) to the storage device.

Solution: actuate at user-level. To address this, I/O optimizations should be implemented at a dedicated user-level layer, promoting portability across different systems and scenarios, and easing information propagation throughout I/O layers.

Problem 2: partial visibility. Optimizations implemented in isolation are oblivious of other systems that compete for the same storage resources. Under shared infrastructures (*e.g.,* cloud, HPC), this lack of coordination can lead to conflicting optimizations [110, 233], I/O contention, and performance variation for both applications and storage backends [201, 229].

Solution: global control. Optimizations should be aware of the surrounding environment and operate in coordination to ensure holistic control of I/O workflows and shared resources.

We address these challenges with PAIO, a SDS data plane framework that enables building user-level, portable, and generally applicable storage optimizations.[1] The key idea is to implement the optimizations *outside* the applications as data plane stages, by intercepting and handling the I/O performed by these. These optimizations are then controlled by a logically centralized controller that has the global context necessary to prevent interference among them. PAIO does not require any modifications to the kernel. Using PAIO, one can decouple complex storage optimizations from current systems, such as I/O differentiation and scheduling, while achieving results similar to or better than tightly coupled optimizations.

Building PAIO is not trivial, as it requires addressing multiple challenges that are not supported by current solutions. To perform complex I/O optimizations outside the application, PAIO needs to *propagate context* down the I/O stack, from high-level APIs down to the lower layers that perform I/O in smaller granularities. It achieves this by combining ideas from *context propagation* [144], enabling application-level information to be propagated to data plane stages with minor code changes and without modifying existing APIs (§3.2). PAIO requires the design of new abstractions that allow differentiating and mediating I/O requests between user-space I/O layers. These abstractions must promote the implementation and portability of a variety of storage optimizations. PAIO achieves this with four main abstractions (§3.1.1). The *enforcement object* is a programmable component that applies a single user-defined policy, such as rate limiting or scheduling, to incoming I/O requests. PAIO characterizes and differentiates requests using *context objects*, and connects I/O requests, enforcement objects and context objects through *channels*.

[1]PAIO stands for Programmable and Adaptable I/O workflows.

To ensure coordination (*e.g.,* fairness, I/O prioritization) across independent storage optimizations, the control plane, with global visibility, fine-tunes the enforcement objects by using *rules*.

3.1 PAIO in a Nutshell

PAIO is a framework that enables system designers to build custom-made SDS data plane stages. A data plane stage built with PAIO targets the workflows of a given user-level I/O layer, enabling the classification and differentiation of requests and the enforcement of different storage mechanisms according to user-defined storage policies. Examples of such policies can be as simple as rate limiting greedy tenants to achieve resource fairness, to more complex ones as coordinating workflows with different priorities to ensure sustained tail latency. PAIO's design is built over five core principles.

General applicability. To ensure applicability across different I/O layers, PAIO stages are disaggregated from the internal system logic, contrary to tightly coupled solutions.

Programmable building blocks. PAIO follows a decoupled design that separates the I/O mechanisms from the policies that govern them, and provides the necessary abstractions for building new storage optimizations to employ over requests.

Fine-grained I/O control. PAIO classifies, differentiates, and enforces I/O requests with different levels of granularity, enabling a broad set of policies to be applied over the I/O stack.

Stage coordination. To ensure stages have coordinated access to resources, PAIO exposes a control interface that enables the control plane to dynamically adapt each stage to new policies and workload variations.

Low intrusiveness. Porting I/O layers to use PAIO requires none to minor code changes.

3.1.1 Abstractions in PAIO

PAIO uses four main abstractions, namely *enforcement objects*, *channels*, *context*, and *rules*.

Enforcement object. An enforcement object is a self-contained, single-purposed mechanism that applies custom I/O logic over incoming I/O requests. Examples of such mechanisms can range from *performance control* and *resource management* such as token-buckets and caches, *data transformations* as compression and encryption, to *data management* (*e.g.,* data prefetching, storage tiering). This abstraction provides to system designers the flexibility and extensibility for developing new mechanisms tailored for enforcing specific storage policies.

Channel. A channel is a stream-like abstraction through which requests flow. Each channel contains one or more enforcement objects (*e.g.,* to apply different mechanisms over the same set of requests) and a *differentiation rule* that maps requests to the respective enforcement object to be enforced.

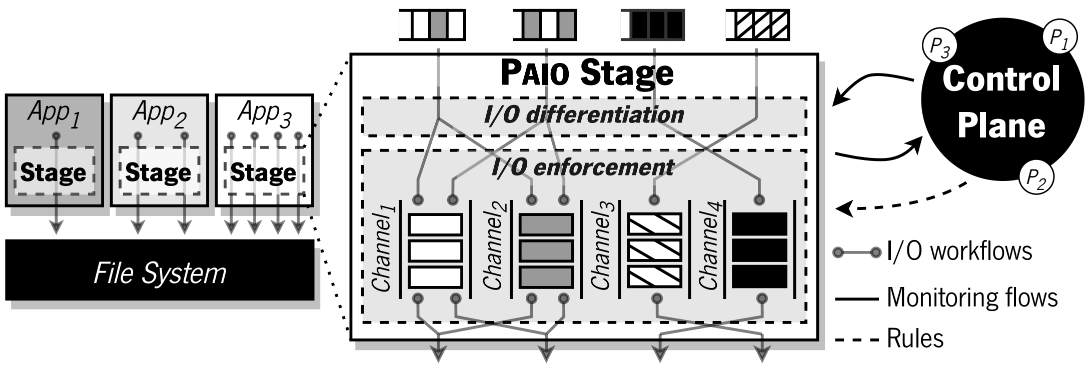


Figure 3.2: High-level design of a PAIO data plane stage.

Context object. A context object contains metadata that characterizes a request. It includes a set of elements (or *classifiers*), such as the *workflow id* (*e.g.,* thread-ID), *request type* (*e.g.,* `read`, `open`, `put`, `get`), *request size*, and the *request context*, which is used to express additional information of a given request, such as determining its origin, context, and more. For each request, PAIO generates the corresponding *Context* object that is used for classifying, differentiating, and enforcing the request over the respective I/O mechanisms.

Rule. In PAIO, a rule represents an action that controls the state of a data plane stage. Rules are submitted by the control plane and are organized in three types: *housekeeping rules* manage the internal stage organization, *differentiation rules* classify and differentiate I/O requests, *enforcement rules* adjust enforcement objects upon workload variations.

3.1.2 High-level Architecture

Figure 3.2 outlines PAIO's high-level architecture. It follows a decoupled design that separates policies, implemented at an external control plane, from the mechanisms that enforce them, implemented at the data plane stage. PAIO targets I/O layers at the user-level. Stages are embedded within layers, intercepting all I/O requests and enforcing user-defined policies. To achieve this, PAIO is organized in four main components.

Stage interface. Applications access stages through a stage interface (§3.4) that routes all requests to PAIO before being submitted to the next I/O layer (*i.e., App*$_3$ →*PAIO Stage* → *File System*). For each request, it generates a *Context* object with the corresponding I/O classifiers.

Differentiation module. The differentiation module (§3.2) classifies and differentiates requests based on their *Context* object. To ensure requests are differentiated with fine-granularity, we combine ideas from *context propagation* [144] to enable application-level information, only accessible to the layer itself, to be propagated to PAIO, broadening the set of policies that can be enforced.

Enforcement module. The enforcement module (§3.3) is responsible for applying the actual I/O mechanisms over requests. It is organized with channels and enforcement objects. For each request, the

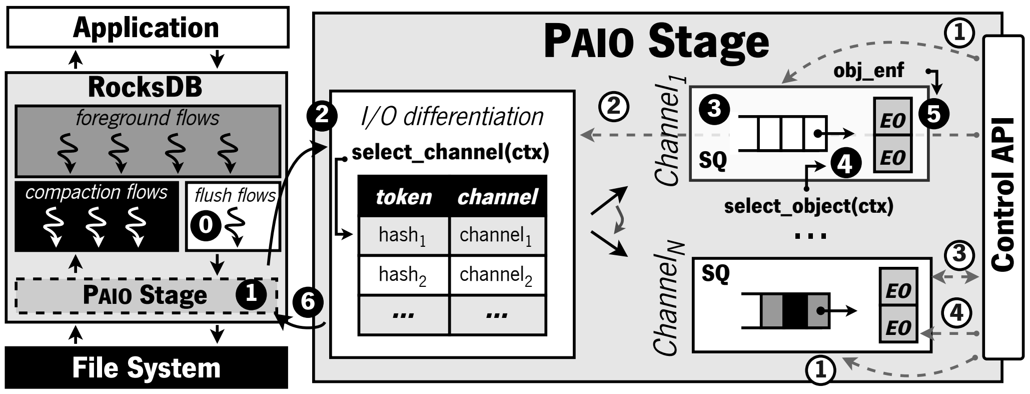


Figure 3.3: Operation flow in a PAIO-enabled I/O stack. Black circles depict the execution flow of a request in the PAIO stage; white circles depict the control flow between the SDS control plane and the stage.

module selects the channel and enforcement object that should handle it. After being enforced, requests are returned to the original data path and submitted to the next I/O layer (*File System*).

Control interface. PAIO exposes a control interface (§3.4) that enables the control plane to (1) orchestrate the stage lifecycle by creating channels, enforcement objects, and differentiation rules, and (2) ensure all policies are met by continuously monitoring and fine-tuning the stage. The control plane provides global visibility, ensuring that stages are controlled holistically. Exposing this interface allows stages to be managed by existing control planes [84, 143, 211].

3.1.3 A Day in the Life of a Request

Before delving into PAIO's internal modules, we first illustrate how it orchestrates the workflows of a given layer. We consider the I/O stack depicted on Figure 3.3, which is made of an *Application*, *RocksDB* (a LSM-based KVS [183]), a PAIO stage, and a POSIX-compliant *File System*; and the enforcement of the following policy: *"limit the rate of RocksDB's flush operations to X MiB/s"*. *RocksDB*'s background workflows generate flush and compaction jobs, which are translated into multiple POSIX operations that are submitted to the *File System*. Flushes are translated into `write` system calls, while compactions into `reads` and `writes`.

At startup time, *RocksDB* initializes the PAIO stage, which connects to an already deployed control plane. The control plane submits housekeeping rules to create a channel and an enforcement object that rate limits requests at X MiB/s (①). It also submits differentiation rules (②) to determine which requests should be handled by the stage, namely flush-based `writes`. Details on how the *differentiation* and *enforcement* processes work are given in §3.2 and §3.3, respectively.

At execution time, *RocksDB* propagates the context at which a given operation is created (⓿) (*i.e.*, if it is a flush or a compaction) and redirects all `write` operations to PAIO (❶). Through ❶, we ensure that only `write` operations are enforced at PAIO, while with ⓿, we differentiate flush-marked `writes` from others that can be triggered by compactions jobs. Upon a flush-based `write`, a *Context* object is created

Table 3.1: Examples of the type of requests a channel receives.

Channel	Workflow ID	Request context	Request type
channel$_1$	*flow*$_1$	–	–
channel$_2$	–	*background tasks*	`read`
channel$_3$	*flow*$_5$	*compaction*	`write`

with its *request type* (`write`), *context* (`flush`), and *size*, and submitted, along the request, to the stage (❶). Then, the stage selects the channel (❷) to be used, enqueues the request (❸), and selects the enforcement object to service the request (❹), which in turn rate limits the request at X MiB/s (❺). After enforcing the request (❻), the original `write` operation is submitted to the *File System* .

The control plane continuously monitors and fine-tunes the data plane stage. Periodically, it collects from the stage the throughput at which requests are being serviced (③). Based on this metric, the control plane may adjust the enforcement object to ensure flush operations flow at X MiB/s, generating enforcement rules with new configurations (④).

3.2 I/O Differentiation

PAIO's differentiation module provides the means to classify and differentiate requests at different levels of granularity, namely *per-workflow*, *request type*, and *request context*. The process for differentiating requests is achieved in three phases.

Startup time. At startup time, the user defines *how* requests are differentiated and *who* should handle each request. First, it defines the granularity of the differentiation, by specifying which I/O classifiers should be used to differentiate requests. For example, to provide per-workflow differentiation PAIO only considers the *Context's workflow id* classifier, while to differentiate requests based on their context and type, it uses both *request context* and *request type* classifiers. Second, the user attributes specific I/O classifiers to each channel to determine the set of requests that a given channel receives. Specifically, it defines the exact request's *workflow id*, *context*, and/or *type* that a channel receives. Table 3.1 provides examples of this specification: *channel*$_1$ only receives requests from *flow*$_1$, while *channel*$_2$ only handles `read` requests originated from *background tasks*; *channel*$_3$ receives compaction-based `writes` from *flow*$_5$. To generate a unique identifier that maps requests to channels, the classifiers can be concatenated into a string or hashed into a fixed-size token (§3.5). Further, this process can be set by the control plane (*i.e.,* differentiation rules) or configured at stage creation.

Execution time. The second phase differentiates the I/O requests submitted to the stage and routes them to the respective channel to be enforced. This is achieved in two steps.

Channel selection. For each incoming request, which is accompanied by its *Context* object, PAIO selects the channel that must service it (depicted in Figure 3.3, step ❷). PAIO verifies the *Context*'s I/O classifiers and maps the request to the respective channel to be enforced. This mapping is done as described in the first phase of the differentiation process.

Enforcement object selection. As each channel can contain multiple enforcement objects (*e.g.,* apply different I/O mechanisms for requests of the same channel), analogously to channel selection, PAIO selects the correct object to service the request (depicted in Figure 3.3, step ❹). For each request, the channel verifies the *Context*'s classifiers and maps the request to the respective enforcement object, which will then employ its I/O mechanism (§3.3).

Context propagation. Several I/O classifiers, such as *workflow id*, *request type*, and *size*, are accessible just from observing raw I/O requests. However, application-level information, that is only accessible to the layer that submits the I/O requests, could be used to expand the policies to be enforced over the I/O stack. An example of such information, as depicted in Figure 3.1, is the *operation context*, which allows to determine the origin or context that a given request was created, *i.e.,* if it comes from a foreground or background task, flush or compaction, or other.

As such, PAIO enables the propagation of additional information from the targeted I/O layer to the data plane stage. It combines ideas from *context propagation*, a commonly used technique that enables a system to forward context along its execution path [144, 145, 155, 233], and applies them to ensure fine-grained control over requests. To achieve this, systems designers instrument the data path of the targeted layer where the information can be accessed, and make it available to the stage through the process's address space, shared memory, or thread-local variables [63]. The information is included at the creation of the *Context* object as the *request context* classifier. Propagating the context without this method would require changing all core modules and function signatures between where the information can be found and its submission to the stage.

As an example, consider the I/O stack of Figure 3.3. To determine the origin of POSIX operations submitted by *RocksDB*'s background workflows, system designers instrument the *RocksDB*'s critical path responsible for managing flush or compaction jobs (⓿) to capture their context. This information is then propagated to the *stage interface*, where the *Context* object is created with all I/O classifiers, including the *request context*, and submitted to the stage (❶).

Note that this step (*i.e.,* context propagation) is *optional*, as it can be skipped for policies that do not require additional information from the application to be enforced.

3.3 I/O Enforcement

The enforcement module provides the building blocks for developing the actual I/O mechanisms that will be employed over requests. It is composed of several channels, each of which contains one or more enforcement objects. The enforcement process begins after the channel selection.

As depicted in Figure 3.3, requests are moved to the selected channel and placed in a *submission queue* (❸). For each dequeued request, PAIO selects the correct enforcement object (❹) and applies its

Table 3.2: Interface definitions of PAIO.

1†	`paio_init()`	Initialization of PAIO stage
	`enforce(ctx,r)`	Enforce context `ctx` and request `r`
	`obj_init(s)`	Initialize enforcement object with state `s`
2*	`obj_enf(ctx,r)`	Enforce I/O mechanism over `ctx` and `r`
	`obj_config(s)`	Configure enforcement object with state `s`
	`stage_info()`	Get data plane stage information
	`hsk_rule(t)`	Housekeeping rule with tuple `t`
3⋆	`dif_rule(t)`	Differentiation rule with tuple `t`
	`enf_rule(id,s)`	Enf. rule over enf. object `id` with state `s`
	`collect()`	Collect statistics from data plane stage

†**Stage API**; ***Enforcement object API**; ⋆**Control API**.

I/O mechanism (❺). Examples of these mechanisms include token-buckets, caches, encryption schemes, and more; we discuss how to build enforcement objects in §3.4.3. Since several mechanisms can change the original request's state, such as data transformations (*e.g.,* encryption, compression), during this phase, the enforcement object generates a *Result* that encapsulates the updated version of the request, including its content and size. The *Result* object is then returned to the stage interface, that unmarshalls it, inspects it, and routes it to the original data path (❻). After this process, PAIO ensured that the request has met the objectives of the specified policy.

Optimizations. Depending on the policies and mechanisms to be employed, PAIO can enforce requests using only their I/O classifiers. While data transformations are directly applicable over the request's content, performance-driven mechanisms such as token-buckets and schedulers, only require specific request metadata to be enforced (*e.g.,* type, size, priority, storage path). As such, to avoid adding overhead to the system execution, PAIO allows for the request's content to be copied to the stage's execution path only when necessary.

3.4 Interfaces and Usage

We now detail how PAIO interacts with I/O layers and control planes, how to integrate PAIO in user-level layers, and how to build enforcement objects. Table 3.2 depicts the interface definitions for (1) I/O layers to interact with the data plane stage; (2) creating and using enforcement objects; and (3) SDS controllers orchestrate the data plane stage.

3.4.1 Interfaces

Stage interface. PAIO provides an API to establish the connection between an I/O layer and PAIO's internal mechanisms. As depicted in Table 3.2, it presents two functions: `paio_init` initializes a PAIO data plane stage, which connects to the control plane for internal stage management and defining how workflows should be handled; `enforce` intercepts requests from the layer and routes them, along the associated *Context* object, to the stage (§3.4.2 details how requests should be intercepted and submitted to PAIO). After enforcing the request, the stage outputs the enforcement result and the layer resumes the original execution path.

Control interface. Communication between data plane stages and the control plane is achieved through five calls, as depicted in Table 3.2. A `stage_info` call lists information about the stage, including the *stage identifier*, *PID*, *job identifier*, *hostname*, and *username*. Rule-based calls are used for managing and tuning the data plane stage. *Housekeeping rules* (`hsk_rule`) manage the stage lifecycle (*e.g.,* create channels and enforcement objects), *differentiation rules* (`dif_rule`) map requests to channels and enforcement objects, and *enforcement rules* (`enf_rule`) dynamically adjust the internal state (`s`) of a given enforcement object (`id`) upon workload and policy variations. The control plane also monitors stages though a `collect` call that gathers key performance metrics of all workflows (*e.g.,* IOPS, bandwidth) and can be used to tune the data plane stage.

This interface enables the control plane to define how PAIO stages handle I/O requests. Nonetheless, concerns related to the dependability of data plane stages, as well as the resolution of conflicting policies are responsibility of the control plane, and are thus orthogonal to this work.

3.4.2 Integrating PAIO in User-level Layers

Although the stage API is simple, porting I/O layers can require a few extra steps.

Using PAIO with context propagation. To integrate a stage within a layer, the system designer typically needs to:

1. Create the stage in the targeted layer, using `paio_init`.
2. Instrument the critical data path, where the layer-level information is accessible, and propagate it to the stage upon the *Context* object creation. This might entail creating additional data structures.
3. Create the *Context* object that will be submitted, alongside the request, to the stage. It can include the *workflow id*, *request type* and *size*, and the propagated information.
4. Add an `enforce` call to the I/O operations that need to be enforced at the stage before being submitted to the next layer. For example, to enforce the POSIX `read` operations of a given layer, all `read` calls need to be first routed to PAIO before being submitted to the file system.
5. Verify if the request was successfully enforced by inspecting the *Result* object, returned from `enforce`, and resume the execution path.

Using PAIO transparently. When context propagation is not required, PAIO stages can be used transparently between I/O layers, such as applications and file systems, without requiring any code changes.

As such, PAIO exposes layer-oriented interfaces (*e.g.,* POSIX) and uses `LD_PRELOAD` to replace the original interface calls at the top layer (*e.g.,* `read` and `write` calls invoked by applications) for ones that are first submitted to PAIO before being submitted to the bottom layer (*e.g.,* file system) [123]. Each supported call defines the logic to create the *Context* object, submits the request to the stage, verifies the *Result*, and invokes the original I/O call. This enables layers to use PAIO without changing any line of code. Moreover, this method is not exclusive to the POSIX interface, and can be used with other I/O layers and interfaces (*e.g.,* KVS interface, RPC interface) as long as the targeted libraries are dynamically loaded in the OS [64].

3.4.3 Building Enforcement Objects

PAIO exposes to system designers a simple API to build enforcement objects, as depicted in Table 3.2.

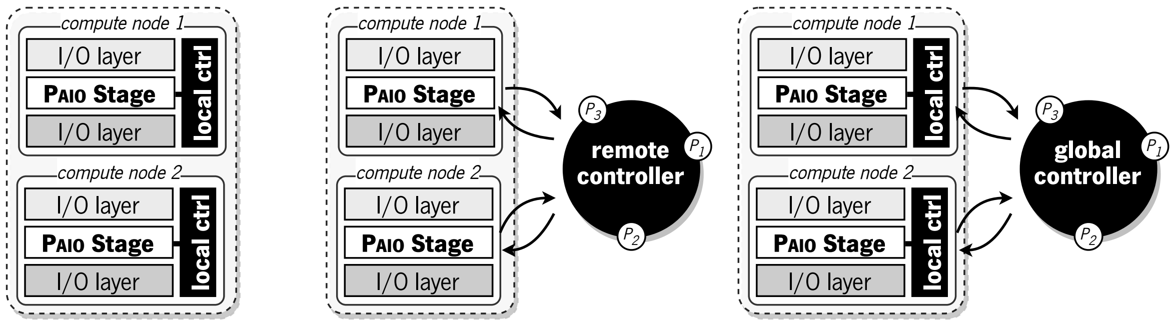
- **`obj_init`.** Create an enforcement object with initial state `s`, which includes its type and initial configuration.
- **`obj_config`.** Provides the tuning knobs to update the enforcement object's internal settings with a new state `s`. This enables the control plane to dynamically adapt the enforcement object to workload and variations (ensure that the current policy is met at all times) and to new policies.
- **`obj_enf`.** Defines the actual I/O logic to be applied over requests. It returns a *Result* that contains the updated version of the request (`r`), after applying its logic. It also receives a *Context* object (`ctx`) that is used to employ different actions over the I/O request.

By default, PAIO preserves the operation logic of the targeted system (*e.g.,* ordering, error handling), as both enforcement objects and operations submitted to PAIO follow a synchronous model. While developing asynchronous enforcement objects is feasible, one needs to ensure that both correctness and fault tolerance guarantees are preserved.

3.4.4 Combining PAIO Stages With a Control Plane

While addressing the research challenges of the SDS control plane (*e.g.,* scalability, dependability, distribution and partitioning control responsibilities) is out of the scope of this thesis, we now discuss how data plane stages built with PAIO can be combined with different control plane settings, in terms of controller distribution and usage. As depicted in Figure 3.4, stages can be orchestrated by SDS controllers in different settings, including *local* (a), *remote* (b), and *hierarchical* (c) controllers. For all scenarios, stages communicate with controllers through PAIO's control interface (§3.4.1).

Local controller. Local SDS controllers run in the same compute node (or server) as the data plane stage. This is appropriate when the defined policies are specific to the data plane stage (or stages) of a single compute node (*i.e.,* do not interfere with shared remote resources). For instance, this local



(a) Local controller setting. **(b) Remote controller setting.** **(c) Hierarchical controller setting.**

Figure 3.4: Combination of different type of SDS controllers orchestrating PAIO stages, namely (a) local, (b) remote, and (c) hierarchical controllers.

controller can be used for ensuring per-application local disk bandwidth guarantees (§5). Under this scenario, communication between data plane stages and the controller can be established through inter-process communication mechanisms such as named pipes, UNIX Domain Sockets, and shared memory.

Remote controller. Remote SDS controllers run in a dedicated compute node, and data plane stages connect to it through TCP sockets, RPC, or other remote communication mechanism. This setting is particularly useful when multiple stages, distributed throughout the infrastructure, enforce policies over shared resources, and thus require global visibility and control. For example, this remote controller can be used for ensuring QoS control of multiple applications, each running in a different compute node, accessing a shared file system.

Hierarchical controller. In a hierarchical setting, as discussed in §2.3, there are multiple SDS controllers: local controllers (*i.e.,* sub-controllers) are co-located with data plane stages, and a global controller (*i.e.,* core controllers) runs at a dedicated compute node. Stages communicate directly with local controllers, which in turn communicate with the global controller. This setting is useful when there are many data plane stages (deployed throughout the I/O infrastructure) operating concurrently over shared resources, and need to be continuously managed by the control plane. Creating a hierarchy of controllers minimizes the number of connections to the global controller (*i.e.,* given that there can be multiple data plane stages in the same compute node) and reduces the number of exchanged messages, thus reducing the load imposed over the global controller.

3.5 Implementation

We have implemented PAIO prototype with 9K lines of C++ code, being provided as an I/O library that targets layers at the user-level. It enables the construction of new stage implementations and straightforward integration with existing layers, requiring none or minor code changes.

Enforcement objects. We implemented two enforcement objects. `Noop` implements a passthrough mechanism that copies the request's content to the *Result* object, without additional data processing.

Dynamic Rate Limiter (DRL) implements a token-bucket to control the rate and burstiness of I/O workflows [33]. The bucket is configured with a maximum token capacity (*size*) and period to replenish the bucket (*refill period*). The rate at which the bucket serves requests is given in *tokens/s*. On `obj_init` the bucket is created with an initial *size* and *refill period*. On `obj_config`, a `rate(r)` routine changes the *size* according to a function between `r` and *refill period*. For each request, `obj_enf` verifies the *context's size* classifier and computes the number of tokens to be consumed. If not enough tokens are available, the request waits for the bucket to be refilled. To demonstrate the portability and maintainability of PAIO's I/O mechanisms, we apply the DRL object over multiple scenarios composed of different I/O layers and storage objectives.

I/O cost. We consider a constant cost for requests *e.g.,* each byte of a `read` or `write` request represents a token. Although the cost depends on several factors (*e.g.,* workload, operation type, cache hits, storage devices), we continuously calibrate the token-bucket so its rate converges to the policies' goal. Our experiments show that this approach works well in our scenarios (§4 and §5), as the bucket's rate converges within few interactions with the control plane. Nevertheless, determining the I/O cost is complementary to our work [87, 200]. Combining PAIO with these could be useful under scenarios where policies are sensitive to the I/O cost.

Statistics and I/O differentiation. PAIO implements per-workflow statistic counters at channels to record the bandwidth of intercepted requests, number of operations, and mean throughput between collection periods. To create unique identifiers that map requests to channels and enforcement objects, we used a computationally cheap hashing scheme [9] (*i.e.,* MurmurHash3) that hashes classifiers into a fixed-size token.

Context propagation. To propagate information from layers, we implemented a shared map, indexed by the *workflow identifier* (*e.g.,* thread-id), that stores the *context* of the requests being submitted, which is similar to those used in [144, 145].

Transparently intercepting I/O calls. PAIO uses `LD_PRELOAD` to intercept POSIX calls and route them either to the stage or to the kernel. `LD_PRELOAD` is a dynamic linking primitive that allows defining the order of linkage of shared libraries (*e.g.,* `libc.so`) at runtime. To enforce the policies demonstrated in §5, PAIO supports `read` and `write` calls, as well as their different variations such `pread`, `pwrite64`, and more. Further, §6 discusses the support of other POSIX calls by a data plane stage built with PAIO. We defer the support of other calls and interfaces (*e.g.,* KVS, object store) to future work.

Control plane. We built a simple but fully-functional control plane with 3.6K lines of C++ code that enforces policies as a *local controller* for two of the use cases presented in this thesis, namely §4 and §5. This controller has also served as basis for the development of an *hierarchical controller*, which is further discussed in §6. Communication between the local controller and stages is established through UNIX Domain Sockets. Policies were implemented as control algorithms. To calibrate enforcement objects it collects I/O metrics generated by the targeted layer from the `/proc` file system [166]. Specifically, it inspects the `read_bytes` and `write_bytes` I/O counters, which represent the number of bytes read/written from/to the block layer, and compares them with the stage statistics to converge to the targeted performance goal. This tuning was used for the use case presented in §5.

3.6 Evaluation

Our evaluation seeks to demonstrate the performance of the PAIO framework. The ability and feasibility of building data plane stages with PAIO, as well as their applicability and performance over different policies and storage scenarios, are evaluated in dedicated chapters of this thesis, namely §4, §5, and §6.

Experimental setting. Experiments were conducted under two hardware configurations. ***Configuration A*** respects to a compute node of the ABCI supercomputer with two 20-core Intel Xeon processors (80 cores), 4 NVidia Tesla V100 GPUs, 384 GiB of RAM, and a 1.6 TiB Intel SSD DC P4600, running CentOS 7.5 with Linux kernel 3.10 and `xfs` file system [4]. ***Configuration B*** respects to a server with two 18-core Intel Xeon processors (72 cores), 192 GiB of RAM, a 1.6 TiB Dell Express Flash PM1725b SSD (Non-Volatile Memory Express (NVMe)) and a 480 GiB Intel D3-S4610 SATA SSD, running Ubuntu Server 20.04 LTS with kernel 5.8.9 and `ext4` file system.

Methodology. We developed a benchmark that simulates an application that submits requests to a PAIO data plane stage. This benchmark aims to demonstrate the maximum performance achievable with PAIO by stress-testing it in a loop-back manner. It generates and submits multi-threaded requests in a closed loop through *Stage interface*'s `enforce` call, under a varying number of clients (*e.g.,* workflows) and request sizes. Request size and number of client threads range between 0 – 128 KiB and 1 – 128, respectively. Each client thread submits 100 million requests. A PAIO stage is configured with varying number of channels, which match the number of client threads, each containing a `Noop` enforcement object that copies the request's buffer to the *result* object. All reported results are the mean of at least ten runs and standard deviation is kept below 5%.

IOPS and bandwidth. Figure 3.5 depicts the cumulative IOPS ratio with respect to a single channel. 0B represents a *context*-only request, as described in §3.3. Results marked with $*$ and $+$ were conducted under configurations ***A*** and ***B***, respectively.

For configuration ***A***, under a 0B* request size, a single PAIO channel achieves a mean throughput of 3.05 MOps/s and a 327 ns latency. Since the workload is CPU-bound, the performance does not scale linearly, as client threads compete for processing time. Under 128 channels, it achieves a cumulative throughput of 97.4 MOps/s, corresponding to a 31× performance increase. As the request size increases so does the total bytes processed by PAIO. When configured with 128 channels, it processes 128 KiB* requests at 384 GiB/s. For a single channel, PAIO processes requests at 2.1 GiB/s and 11.7 GiB/s for 1 KiB* and 128 KiB* request sizes.

For configuration ***B***, PAIO achieves higher throughput results as it operates under a later kernel version. Since the machine is configured with 72 cores, PAIO's performance peaks at 64 client threads. Under a 0B$^+$ request size, PAIO achieves 3.43 MOps/s (1 channel) and 102.7 MOps/s (64 channels), representing

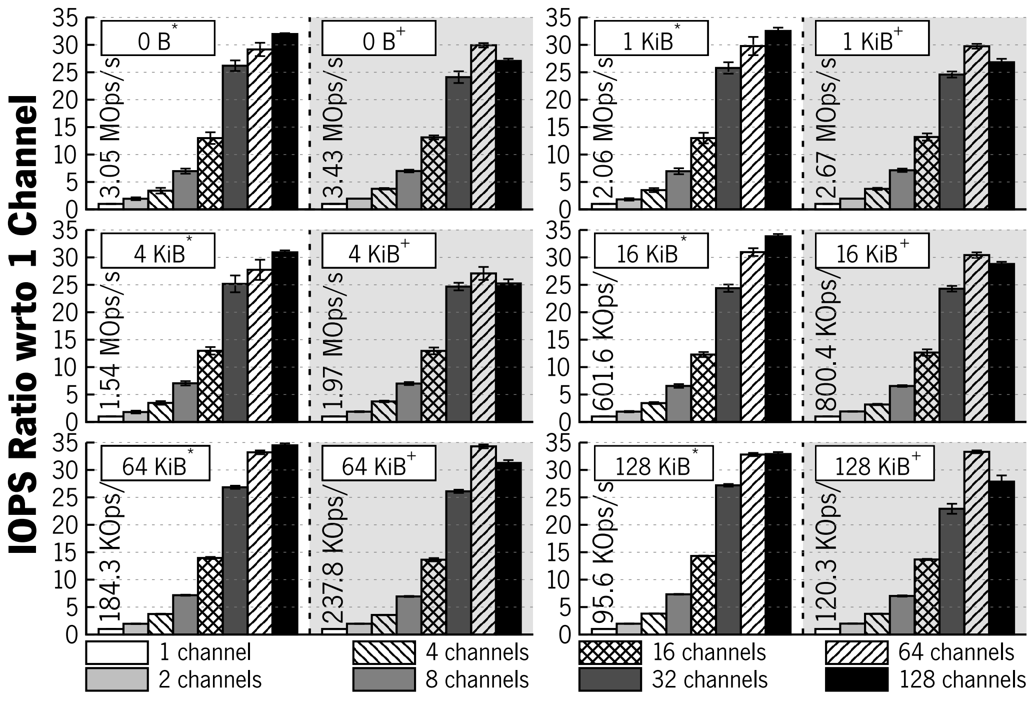


Figure 3.5: Cumulative IOPS of PAIO under varying number of channels (1 – 128) and request sizes (0 – 128 KiB). Absolute IOPS value is shown above the 1 channel bar.

a 30× performance increase. When configured with 64 channels, it is able to process 128 KiB^{+}-sized requests at 489 GiB/s. For a single channel, PAIO processes requests at 2.5 GiB/s and 14.7 GiB/s for 1 KiB* and 128 KiB* request sizes, respectively.

Profiling. We measured the execution time of each PAIO operation that appears in the main execution path. Depending on the hardware configuration, *Context* object creation takes between 17 – 19 ns, while the channel and enforcement object selection take 85 – 89 ns to complete (each). The duration of `obj_enf` ranges between 20 ns and 8.45 μs when configured with 0B and 128 KiB request sizes.

Summary. Results show that PAIO has low overhead, as it is provided as a user-space library, which does not require costly context-switching operations. In fact, PAIO can be used to implement data plane stages over I/O stacks that use more legacy kernel versions, such as kernel 3.10 from configuration ***A***, and even leverage from the optimizations implemented in recent versions, such as kernel 5.8.9 from configuration ***B***. We expect that the main source of overhead will always be dependent on the type of enforcement object (and policy) applied over I/O requests. For the enforcement objects used in this work (§4 – §6), we have not observed significant performance overhead.

3.7 Related Work

SDS systems. PAIO builds on a large body of work on SDS systems. IOFlow [211], sRoute [205], and PSLO [128] target the virtualization layer (*i.e.*, hypervisor, storage and network drivers) to enforce QoS policies. PM [248] enforces rate limiting services at the Network File System. Mesnier *et al.* [155] employ caching optimizations at the block layer. Pisces [201] and Libra [200] enforce bandwidth guarantees in multi-tenant KVS. Malacology [194] improves the programmability of Ceph to build custom applications on top of it. Retro [143] and Cake [222] implement resource management services at the Hadoop stack. SafeFS [175] stacks FUSE-based file systems on top of each other, each providing a different service. Crystal [84] extends OpenStack Swift to implement custom services to be enforced over object requests.

All systems are targeted for specific I/O layers, as their design is *tightly coupled to* and *driven by* the architecture and specificities of the software stacks they are applied to. In contrast, PAIO is disaggregated from a specific software stack, enabling developers to build custom-made data plane stages applicable over different user-level layers, while requiring none to minor code changes — we demonstrate this by integrating PAIO over different I/O layers (§4–§6). Previous works are also unable to enforce the policies demonstrated in §4, as they do not provide context propagation [200, 201], inhibiting request differentiation at a finer granularity (*i.e.,* foreground *vs* high-priority *vs* low-priority background tasks). Other solutions actuate at the kernel-level [155, 211], where the *context* is unreachable without significantly changing legacy APIs. Further, these are also unfit to achieve the policies demonstrated in §5 and §6, as solutions like [128, 205, 211] cannot be used under scenarios that require bare-metal access to resources, such as HPC infrastructures and bare-metal cloud servers.

Context propagation. Some works use context propagation techniques to tag data across kernel layers. Mesnier *et al.* [155] classifies and tags requests with classes to be differentiated at the block layer. IOFlow [211] tags requests to differentiate tenants that share the same hypervisor. Split-level scheduling [233] identifies the processes that caused a given I/O operation throughout the VFS, page cache, and block layer. PAIO acts at the user-level and enables the propagation of additional information from the targeted I/O layer to the stage (*e.g.,* propagate the *context* at which a given request was created, as in §4), allowing more fine-grained differentiation and control over requests. Enabling the intended granularity by PAIO at kernel-level approaches would require breaking standard user-to-kernel and kernel-internal interfaces, reducing portability and compatibility [5].

Our contributions are also applicable under kernel-bypass storage stacks (*e.g.,* SPDK, PMDK), which is not the case for previous work. In more detail, under a kernel-bypass storage stack where I/O operations are submitted directly to the device from user-space, in-kernel systems like IOFlow, Mesnier *et al.*, Split-level scheduling, PM, and mechanisms such *dm-crypt* [57], *dm-cache*, *blkio* [29], cannot be used. On the other hand, PAIO can be used and integrated with kernel-bypass stacks under two settings: (1) it can be used in use cases where applications consider the storage backend as black-box; (2) we could integrate PAIO as a new abstraction on top of SPDK — through SPDK's logical block device abstraction [36] — or PMDK — as done by existing libraries, namely *libpmemobj*, *libpmemblk*, and *libpmemkv* [173].

Storage QoS. Many works ensure QoS SLOs at specific storage layers, including the block layer [29, 88, 139, 154, 220, 242], hypervisor [86, 87, 89, 128, 211], and distributed storage [176, 222, 223, 228]. These works are targeted for a specific I/O layer and storage objective. In contrast, PAIO is more general, providing a framework for building custom data plane stages applicable over different layers. Also, most of these solutions only differentiate requests based on their type. PAIO provides differentiation at workflow, request type, and request context. Approaches like [89, 139, 154, 242] follow a decoupled design that separates the QoS algorithm from the mechanism that applies it. While complementary to our work, these could be incorporated into our framework as new enforcement objects.

3.8 Summary and Discussion

In this chapter, we show the design, implementation, and evaluation of PAIO, a framework that enables system designers to build custom-made SDS data plane stages applicable over different I/O layers. PAIO provides fine-grained differentiated treatment of requests and allows implementing storage mechanisms adaptable to different policies. By combining ideas from SDS and context propagation, we demonstrated that PAIO decouples system-specific optimizations to a more maintainable and programmable environment, while enabling similar I/O control and performance, and requiring minor to none code changes. Our evaluation shows that PAIO's performance scales with the number of channels, achieving high throughput and low latency. The next chapters show the feasibility of using PAIO by describing the data plane stages built with it to address the requirements of different storage stacks.

Tail latency control in KVSs (§4). We built a data plane stage that achieves tail latency control at the 99^{th} percentile in RocksDB, a LSM-based KVS. Results show that a PAIO-enabled RocksDB improves tail latency by 4× under different workloads, and enables similar performance and I/O control as system-specific optimizations (*i.e.,* SILK [17]).

Per-application bandwidth control (§5). We built a data plane stage that ensures dynamic per-application bandwidth guarantees under a shared storage scenario; specifically, where multiple TensorFlow [1] instances (which execute on the same compute node) compete for local disk bandwidth. Results show that all PAIO-enabled TensorFlow instances are provisioned with their bandwidth goals. This scenario was driven by the requirements of the ABCI supercomputer.

Metadata QoS control in PFSs (§6). We built PADLL, a SDS storage middleware that proactively controls and ensures QoS over metadata workflows in HPC storage systems. Results show that PADLL can dynamically control metadata-aggressive workloads, prevent I/O burstiness, and ensure I/O fairness and prioritization of jobs that compete for metadata resources over Lustre-like PFSs.

4

Tail Latency Control in Log-Structured Merge Key-Value Stores

Log-Structured Merge tree Key-Value Stores such as LevelDB [80], RocksDB [183], and PebblesDB [178], have become a crucial component of modern storage infrastructures, being used in distributed databases [191, 209, 214], file systems [224], stream processing and machine learning engines [2, 42, 161], and more. While great progress has been made to improve the throughput of KVSs, by mainly reducing the cost of internal operations [106, 138, 168, 178], recent studies have demonstrated that these systems suffer from tail latency spikes when foreground and background tasks compete for shared storage resources [17]. SILK addresses this problem by proposing an I/O scheduler that controls the interference between these tasks [17]. However, it follows an intrusive approach and applying its I/O scheduler over RocksDB required changing several core modules made of thousands of LoC [15]. This chapter describes a new data plane stage built with PAIO, that enables similar performance and control as SILK but without requiring profound refactoring to the original codebase. Results show that a PAIO-enabled RocksDB improves 99th percentile latency up to 4× under different workloads and testing scenarios.

4.1 Log-Structured Merge tree Key-Value Stores

We now discuss how LSM KVSs are organized and how their internal components fit together. For this purpose, we consider the design of RocksDB, a widely-used production-ready KVS from Facebook [183].

KVSs built on top of a LSM data structure are generally organized in three main components. A *memory component*, know as *memtable*, is a sorted data structure that resides in memory and is used to receive all write operations from KVS clients. Every client write is also written to a Write-Ahead Logging (WAL) component that is persisted on disk. Its purpose is to make every update (on a key-value pair) persistent, so that in the event of a failure, it can be used to completely recover the *memtable* data and restore the KVS to its original state. For crash consistency purposes, the WAL is flushed after every client write. The *disk component* is organized in multiple levels (L_0, L_1, ..., L_N), each containing multiple files, called Sorted Strings Table (SST), that hold sorted key-value pairs. Levels have a predetermined size and grow in hierarchical manner. Specifically, level L_{i+1} is $N\times$ larger than L_i where N is a configurable factor (usually 10). As such, lower levels of the LSM (*e.g.,* L_0, L_1) are usually sized between MiB and few GiB, while higher levels can hold several GiB, or even reach TiB in size as seen in production [17, 214].

Foreground operations. The main operations submitted from KVS clients are writes (`put(key,value)`), reads (`get(key)`), and scans (`seek(key`$_1$`,key`$_2$`)`). The `put(key,value)` operation stores the mapping from `key` to `value` in the KVS; if `key` exists, its `value` is updated. Directly submitting each `put` operation to the file system would generate a workload made of *random writes*; to prevent this, all `put` operations are absorbed by the *memtable*, transforming a *random write* workload into a *sequential* one. If enabled, these are also *appended* to the WAL. The `get(key)` operation returns the latest value associated with `key`, while the `seek(key`$_1$`,key`$_2$`)` operation retrieves all key-value pairs comprehended within `key`$_1$ and `key`$_2$ range. Read-based operations are first submitted to the *memtable*; if `key` is not found, it traverses the LSM in hierarchical order. While traversing the tree, only a single SST file on each level is accessed since key-value pairs are sorted (except for L_0, due to the opposite reason).

Background operations. LSM KVSs perform two types of background operations for internal system management. When the *memtable* fills, a *flush* operation occurs, which writes the contents of the *memtable* directly to the first level of the LSM (L_0) as an SST file. Flushes are sequentially written and only proceed when there is enough space in L_0. Moreover, because flushes need to have high throughput to avoid blocking client's writes, the *memtable* is written to disk without additional processing (*i.e.,* sorting), which leads L_0 to have overlapping key ranges. When levels of the LSM fill (*i.e.,* exceed their maximum size), *compactions* are triggered, where SST files from level L_i are picked and merged with SST files from level L_{i+1}. As such, compactions induce high I/O overhead, as these generate several POSIX `read` and `write` calls. In fact, depending on the size of the KVS system, compactions may read/write many GiB of data and take several minutes to complete [17, 243].

All background operations are held in an internal FIFO queue. High level compactions ($L_i \rightarrow L_{i+1}$, where $i > 0$) can be executed in parallel. Flushes and low level compactions ($L_0 \rightarrow L_1$) are sequential. Compactions and flushes are handled by separate thread pools.

4.2 The Tail Latency Problem

A common problem of LSM KVS systems is the interference between foreground and background workflows, generating high latency spikes for clients. Fundamentally, latency spikes occur when flushes cannot proceed because $L_0 \rightarrow L_1$ compactions and flushes are slow or on hold, which can happen for two main reasons [17].

- **Flushes are slow.** When flushes are slow due to insufficient disk I/O bandwidth, the *memtable* fills up and cannot absorb any more client writes. Under this scenario, client writes are stalled until there is enough space on the memory component to handle them, causing latency spikes.

- **Low level compactions are slow.** A second cause for flushes to be halted occurs when $L_0 \rightarrow L_1$ compactions are slow, either (1) due to insufficient disk bandwidth because compactions from higher levels are executing and using a significant portion of the disk I/O bandwidth, or (2) because all workers from the dedicated compaction thread pool are in use (executing compactions from higher levels), which results in low level compactions to wait in the compaction queue. This leads to the accumulation of several SST files on L_0, which can block flushes when there is no more storage quota left at this level.

This means that not all background operations have an equal impact on the KVS performance. Internal operations that are closer to clients' requests (namely, flushes and low level compactions) are critical, as their slowdown negatively affects the overall tail latency perceived by clients.

Rate limiting internal operations. To address this problem, RocksDB can reserve more I/O bandwidth to foreground workflows by rate limiting the I/O of internal KVS operations. To do so, RocksDB implements a rate limiter that can either be statically fixed at a given bandwidth rate or be dynamically changed through a multiplicative-increase, multiplicative-decrease algorithm, which is termed as *auto-tuned rate limiter* [116]. Such an approach however suffers from two main limitations [17]. First, as foreground and background operations are intrinsically dependent, if background writes are rate limited indiscriminately, it can lead to low level compactions or flushes to be slow down. Second, the auto-tuned rate limiter allocates more bandwidth to background workflows when there is more backlog, regardless of the I/O rate of foreground workflows, leading to an immediate latency spike of clients' requests.

SILK. By following a distinct approach, SILK [17], a RocksDB-based KVS, prevents this problem through an I/O scheduler that:

1. **Allocates bandwidth for internal operations when client load is low.** Given that, in production environments, the load of KVS clients varies over time [17, 37], SILK reserves less I/O bandwidth to internal operations when client load is high (limiting the interference between both operation types), and increases this limit when client load is low, which prevents accumulating a large backlog of background operations.

2. **Prioritizes flushes and low level compactions.** Leveraging from the fact that flushes and low level compactions have higher impact on clients' tail latency, SILK enforces different priorities for each type of background tasks. Specifically, flushes have higher priority, since they need to ensure the *memtable* has enough space to absorb clients' writes; then, it gives second priority to low level compactions, to ensure that flushes do not get blocked; finally, high level compactions have the lowest priority, since even though they to eventually execute, their immediate completion does not impact clients' tail latency on the short term.

3. **Preempts high level compactions with low level ones.** Because high level compactions can take several minutes to execute, SILK implements a new compaction algorithm that allows to pause them, while giving priority for low level ones to execute.

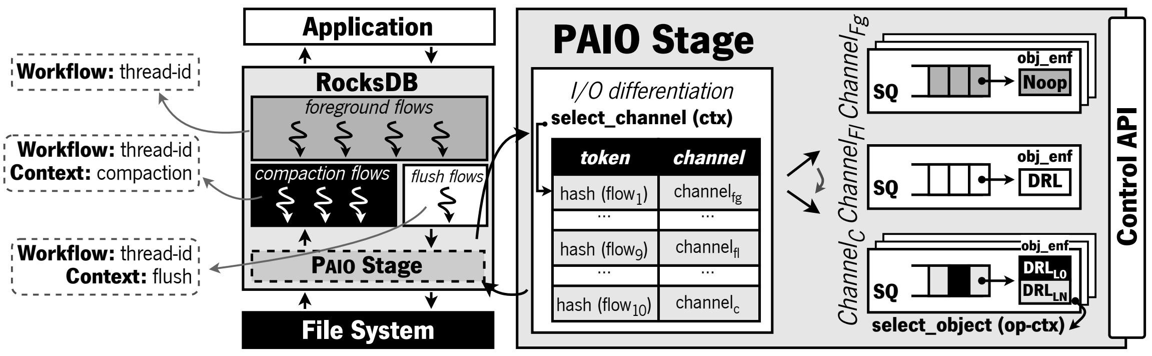


Figure 4.1: Organization of the PAIO data plane stage for achieving tail latency control in LSM-based KVS.

SILK employs these techniques through the following control algorithm. As these KVSs are typically embedded and co-located with other services running in the I/O stack, the KVS I/O bandwidth is bounded to a given rate (KVS_B). It continuosly monitors clients' bandwidth (Fg), and allocates leftover bandwidth ($left_B$) to internal operations (I_B), given by $I_B = KVS_B - Fg$. To enforce rate I_B, SILK uses RocksDB's I/O rate limiters [71]. Flushes and $L_0 \rightarrow L_1$ compactions have high priority and are provisioned with minimum I/O bandwidth (min_B). High level compactions have low priority and can be paused at any time. Because all compactions share the same thread pool, it is possible that, at some point, all threads are handling high level compactions. As such, SILK preempts one of them to execute low level compactions.

Applying these optimizations however, required reorganizing RocksDB's internal operation flow, changing core modules made of thousands of LoC including *background operation handlers*, *internal queuing logic*, and *thread pools allocated for internal work* [15]. Further, porting these optimizations to other KVS that would equally benefit from them, such as LevelDB [80] and PebblesDB [178], requires deep system knowledge and substantial re-implementation efforts.

4.3 Tail Latency Control with PAIO

Rather than modifying the RocksDB engine (core modules), we found that several of these optimizations could be achieved by orchestrating its I/O workflows. Thus, we applied SILK's design principles as follows: a PAIO *data plane stage provides the I/O mechanisms for prioritizing and rate limiting background flows*, while the *control plane re-implements SILK's I/O scheduling algorithm* to orchestrate the stage. Figure 4.1 depicts the organization of this data plane stage.

Data plane stage. The stage intercepts the POSIX operations submitted from all RocksDB workflows, including foreground, flush, and compaction flows. We consider each RocksDB thread that interacts with the file system as a workflow, and each of these is handled by a different PAIO channel, namely *Channels*$_{Fg}$ (foreground), *Channel*$_{Fl}$ (flush), and *Channels*$_{C}$ (compaction). Channel differentiation is made using the *workflow id*. Since at POSIX level only the *request type*, *size*, and *workflow id* (*thread-id*) are known, we instrumented RocksDB to propagate the *context* at which a given operation is created. Specifically,

Table 4.1: Lines of code added to RocksDB to integrate the PAIO stage.

	Lines added to RocksDB
Targeted codebase size	≈335K [72]
Initialize PAIO stage	10
Context propagation	47
Create *Context* object	7
Instrument I/O calls	17
Verify *Result* object	4
Total	**85**

RocksDB's *FlushJob* class was instrumented to propagate `flush` contexts [70], and the *CompactionJob* class to propagate `compaction` contexts [69], including the involved LSM levels at which the compaction occurs (*e.g.*, $\texttt{compaction\_L}_0\texttt{\_L}_1$, $\texttt{compaction\_L}_2\texttt{\_L}_3$).

Foreground flows are handled with `Noop` enforcement objects (which act as a *passthrough*), and are continuously monitored for collecting clients' bandwidth (Fg). Background flows are handled by channels made of DRL objects (used for rate limiting). Flushes flow through a dedicated channel (*Channel*$_{Fl}$). As compactions with different priorities can flow through the same channel (*Channel*$_C$), each of these channels contains two DRL objects configured at different rates, one for high priority compactions (DRL$_{L0}$) and another for low priority compactions (DRL$_{LN}$). Enforcement object differentiation is made through the *request context* classifier, and requests are enforced with the optimization described in §3.3. PAIO also collects the bandwidth rate of flushes (Fl), and low (L_0) and high level compactions (L_N). As listed in Table 4.1, integrating PAIO in RocksDB only required adding 85 LoC of which 47 LoC respect to the context propagation, while the remainder are used for initializing the stage (10), create the *Context* object (7), instrument read and write calls (17), and verify the *Result* object (4).

Control algorithm. The control plane implements the control portion of SILK's scheduling algorithm (Algorithm 4.1). It uses a feedback control loop that performs the following steps. First, it collects statistics from the stage (1) and computes leftover disk bandwidth (*left*$_B$) to assign to internal operations (2). To ensure that background operations keep flowing, it defines a minimum bandwidth threshold (3), and distributes *left*$_B$ according to workflow priorities (4–11). If high priority tasks, from both types (Fl and L_0), are executing it assigns them an equal share of *left*$_B$, while ensuring that high level compactions keep flowing (min_B), preventing low level ones from being blocked in the queue (5). If a single high priority task is being executed, *left*$_B$ is allocated to it and min_B to others (6–9). If no high priority task is executing, it reserves *left*$_B$ to low priority ones (11). It then generates and submits enforcement rules (`enf_rule`) to adjust the rate of each enforcement object (12). For low priority compactions, it splits B_{L_N} between all DRL objects that handle these. Since high priority compactions are executed sequentially [17, 183], it assigns B_{L_0} to the respective objects. Rate B_{Fl} is assigned to those responsible for flushes. For this use case, we used a local SDS controller (as described in §3.4.4).

Algorithm 4.1: Tail Latency Control Algorithm

```
   **Initialize:** $KVS_B = 200$; $min_B = 10$
1  $\{Fg, Fl, L_0, L_N\} \leftarrow$ collect ()
2  $left_B \leftarrow KVS_B - Fg$
3  $left_B \leftarrow max\ \{left_B \mid min_B\}$
4  if $Fl > 0 \wedge L_0 > 0$ then
5      $\{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B/2,\ left_B/2,\ min_B\}$
6  else if $Fl > 0 \wedge L_0 = 0$ then
7      $\{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B,\ min_B,\ min_B\}$
8  else if $Fl = 0 \wedge L_0 > 0$ then
9      $\{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ left_B,\ min_B\}$
10 else
11     $\{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ min_B,\ left_B\}$
12 enf_rule ($\{B_{Fl},\ B_{L_0},\ B_{L_N}\}$)
13 sleep (loop_interval)
```

4.4 Evaluation

We now demonstrate how the PAIO data plane stage achieves tail latency control under several workloads. We compare the performance of four KVS systems: (1) RocksDB; (2) Auto-tuned, a version of RocksDB with the auto-tuned rate limiter of background operations enabled [116]; (3) SILK; and (4) PAIO, *i.e.*, a PAIO-enabled RocksDB.

Testbed configuration. Experiments were conducted using a server with two 18-core Intel Xeon processors (72 cores), 192 GiB of RAM, a 1.6 TiB Dell Express Flash PM1725b SSD (NVMe) and a 480 GiB Intel D3-S4610 SATA SSD, running Ubuntu Server 20.04 LTS with kernel 5.8.9 and `ext4` file system. Unless stated otherwise, experiments were made using the available NVMe device. Moreover, as used in the SILK testbed, we limit memory usage to 1 GiB and I/O bandwidth to 200 MiB/s using `cgroups` [29, 153] (unless stated otherwise). Both memory and disk bandwidth limits are based on Nutanix production environments [17].

KVS configuration. All KVS systems are tuned as follows. The `memtable-size` is set to 128 MiB. We use 8 threads for client operations and 8 background threads for flush (1) and compactions (7). The minimum bandwidth threshold for internal operations is set to 10 MiB/s. To simplify results, compression and WAL are turned off; while enabling them impact the absolute performance of the system, they do not change the observations made in this evaluation. All experiments are conducted using the `db_bench` benchmark [68].

Workloads. We focus on workloads made of bursty clients to better simulate existing services in production [17, 37]. Client requests are issued in a closed loop through a combination of peaks and valleys. An initial valley of 300 seconds submits operations at 5 kops/s, and is used for executing the KVS internal backlog. Peaks are issued at a rate of 20 kops/s for 100 seconds, followed by 10 seconds valleys at

5 kops/s. All datastores were (pre)loaded with 100 million key-value pairs, using a uniform key distribution, with 8B-sized keys and 1024B-sized values.

We use three workloads with different read:write ratios: *mixture* (50:50), *read-heavy* (90:10), and *write-heavy* (10:90). *Mixture* represents a commonly used Yahoo! Cloud Serving Benchmark (YCSB) workload (workload A) and provides a similar ratio as Nutanix production workloads [17]. *Read-heavy* provides an operation ratio similar to those reported at Facebook [37]. Further, to present a comprehensive testbed, we include a *write-heavy* workload. For each system, workloads were executed three times over 1-hour with uniform key distribution. For figure clarity, we present the first 20 minutes of a single run. Similar performance curves were observed for the rest of the execution. Figure 4.2–4.11 depict throughput and 99^{th} percentile latency of all systems and workloads. Theoretical client load is presented as a red dashed line. Mean throughput is shown as an horizontal dashed line.

Mixture workload. Figures 4.2 and 4.3 depict the throughput and 99^{th} percentile latency results (respectively) of each system under the mixture workload. Due to accumulated backlog of the loading phase, the throughput achieved in all systems does not match the theoretical client load. RocksDB presents high tail latency spikes due to constant flushes and low level compactions. Auto-tuned presents less latency spikes but degrades overall throughput. This is due to the rate limiter being agnostic of background tasks' priority, and because it increases its rate when there is more backlog, contending for disk bandwidth. SILK achieves low tail latency but suffers periodic drops in throughput due to accumulated backlog. Compared to RocksDB (11.9 kops/s), PAIO provides similar mean throughput (12.4 kops/s). As for tail latency, while RocksDB experiences peaks that range between 3–20 ms, PAIO and SILK observe a 4× decrease in absolute tail latency, with values ranging between 2–6 ms.

Read-heavy workload. Figures 4.4 and 4.5 depict the throughput and 99^{th} percentile latency results (respectively) of each system under the read-heavy workload. Throughput-wise all systems perform identically. At different periods, all systems demonstrate a temporary throughput degradation due to accumulated backlog. As for tail latency, the analysis is twofold. RocksDB and Auto-tuned present high tail latency up to the 400 seconds mark. After that mark, RocksDB does not have more pending backlog and achieves sustained tail latency (1–3 ms), while on Auto-tuned, some compactions are still being performed due to rate limiting, increasing latency by 1–2 ms. SILK and PAIO have similar latency curves. During the initial valley both systems significantly improve tail latency when compared to RocksDB. After the 400 seconds mark, SILK pauses high level compactions and achieves a tail latency between 1–2 ms. By preempting high level compactions and serving low level ones through the same thread pool as flushes, it ensures that high priority tasks are rarely stalled. SILK achieves this by modifying the RocksDB's queuing mechanism. In PAIO, while sustained, its tail latency is 1 ms higher than SILK's in the same observation period. Since PAIO does not modify the RocksDB engine, it cannot preempt compactions (§4.2), resulting in a small increase on client's (tail) latency.

Write-heavy workload. Figures 4.6 and 4.7 depict the throughput and 99^{th} percentile latency results (respectively) of each system under the write-heavy workload. Write-intensive workloads generate a large

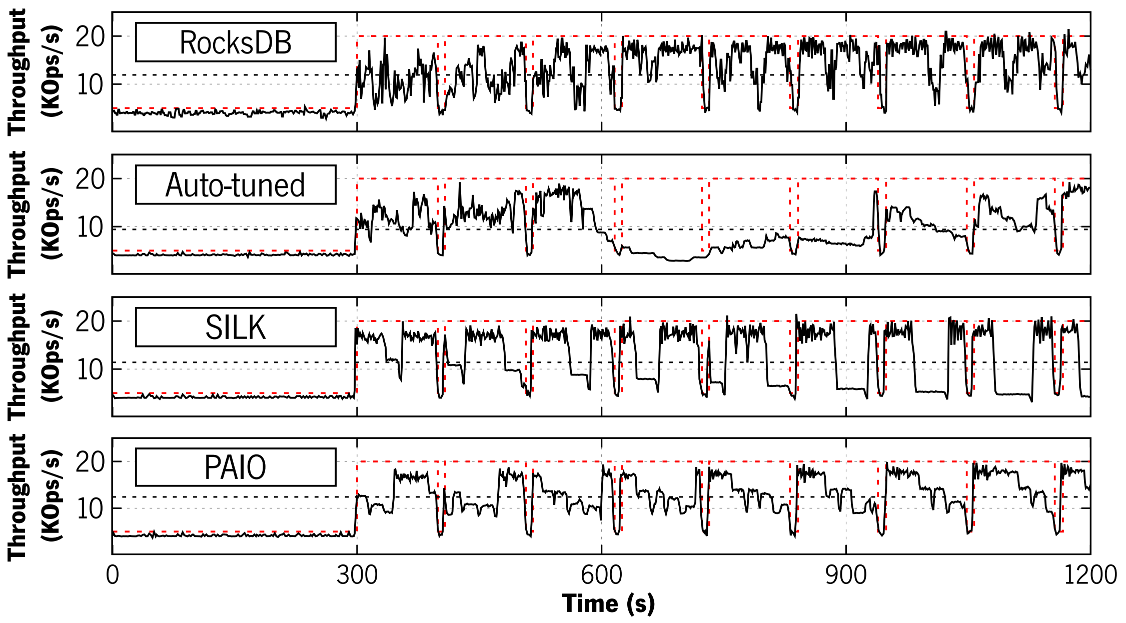


Figure 4.2: Throughput of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload.

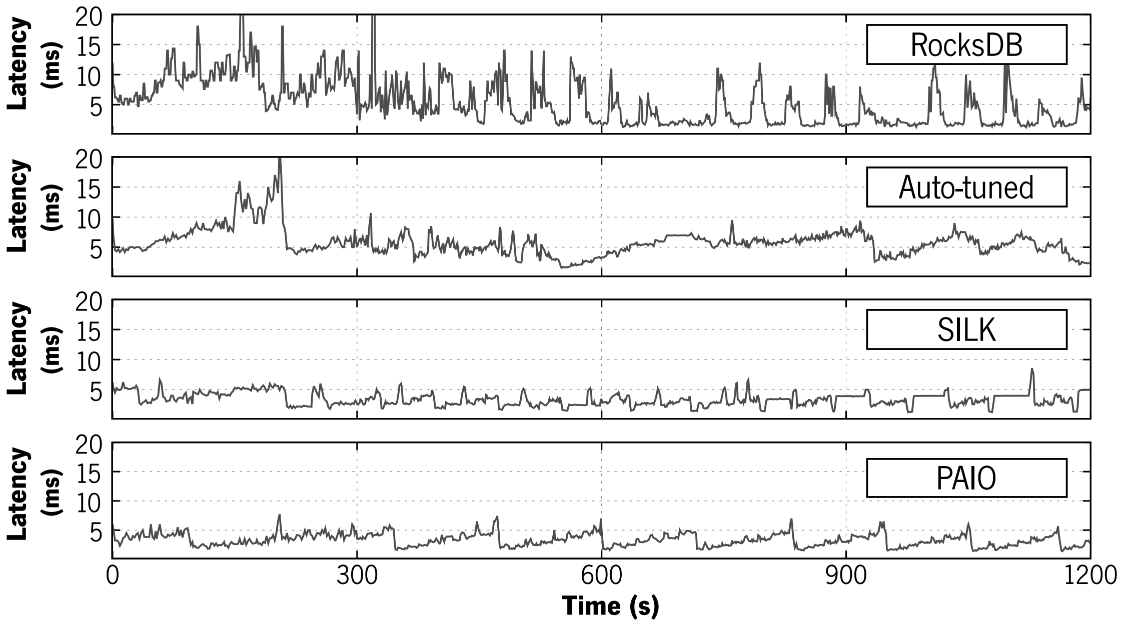


Figure 4.3: 99th percentile latency of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload.

backlog of (latency-critical) background tasks, leading RocksDB to experience high latency spikes. Auto-tuned limits all background writes, reducing tail latency but still exceeding the 5 ms mark over several periods. SILK pauses high level compactions and only serves high priority tasks, improving mean throughput and keeping latency spikes below 5 ms. In PAIO, since flushes occur more frequently, the control plane slows down high level compactions more aggressively, which leads to low level ones to be temporary halted at the compaction queue, waiting to be executed. Even though mean throughput is decreased, PAIO significantly reduces tail latency, never exceeding 6 ms. The throughput difference between PAIO and SILK is justified by the latter preempting high level compactions, as described in the read-heavy workload.

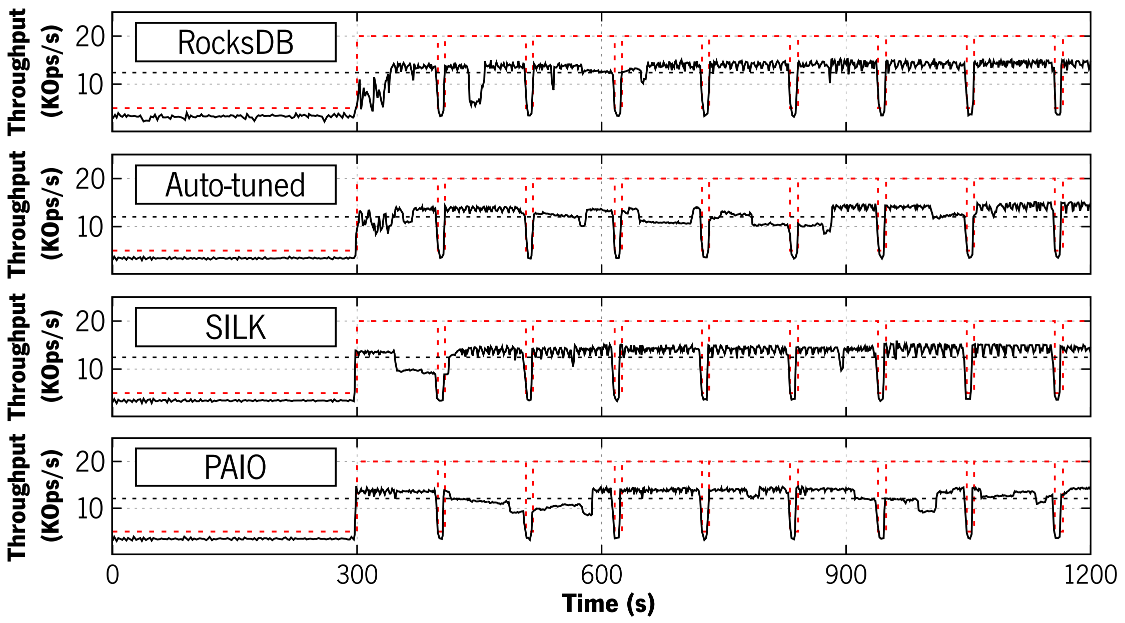


Figure 4.4: Throughput of RocksDB, Auto-tuned, SILK, and PAIO under *read-heavy* workload.

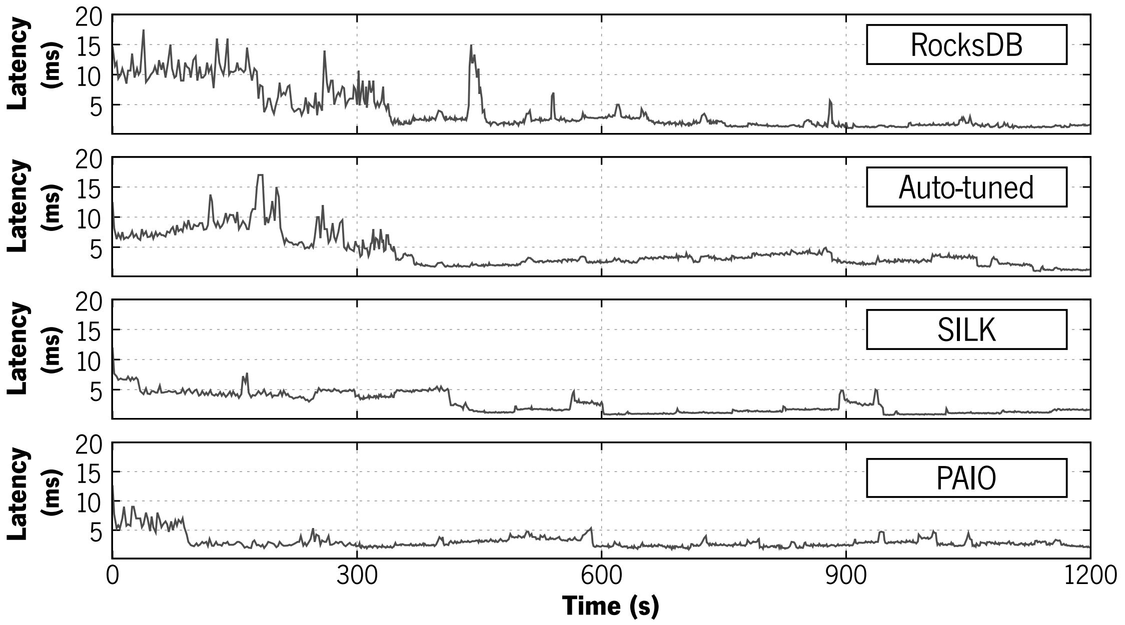


Figure 4.5: 99th percentile latency of RocksDB, Auto-tuned, SILK, and PAIO under *read-heavy* workload.

Mixture workload without rate limiting. We conducted an additional set of experiments to assess the impact of the tail latency control algorithm under a scenario where the KVS has access to the full bandwidth of the storage device. We compared the performance of RocksDB, SILK, and PAIO under both SSD (480 GiB Intel D3-S4610) and NVMe (1.6 TiB Dell Express Flash PM1725b) devices, without rate limiting, using the *mixture* workload. The KVS_B parameter was set with a value closer to the device's limit (reported by the manufacturer). Remainder system configurations were kept unchanged.

Figures 4.8 and 4.9 depict the throughput and 99th percentile latency (respectively) under the SSD device. Due to accumulated backlog all systems experience poor throughput performance, averaging at

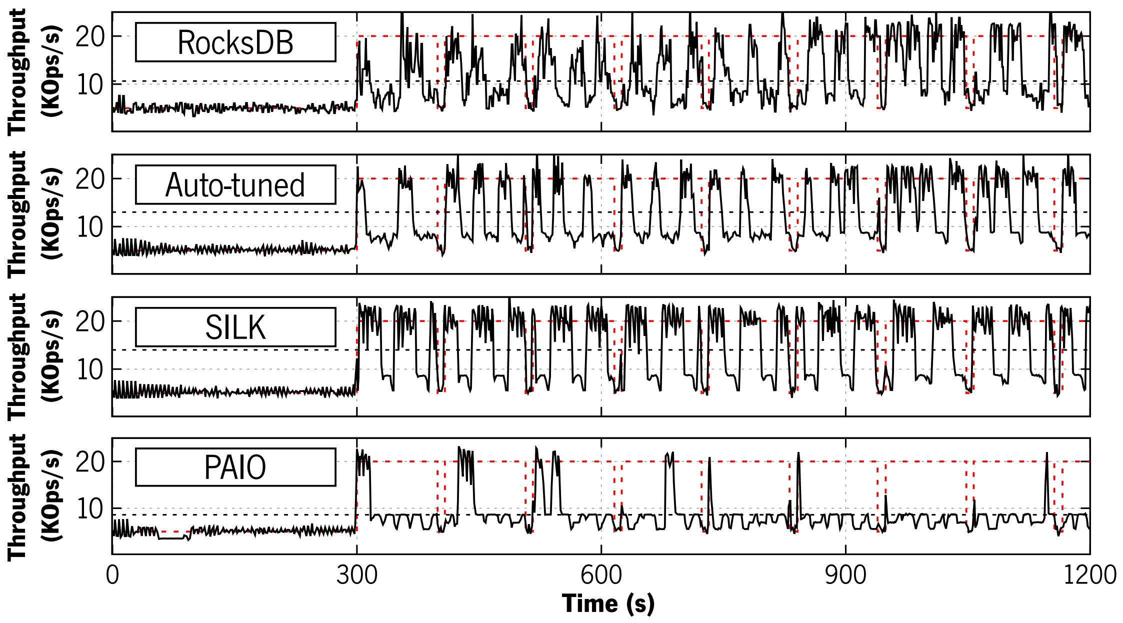


Figure 4.6: Throughput of RocksDB, Auto-tuned, SILK, and PAIO under *write-heavy* workload.

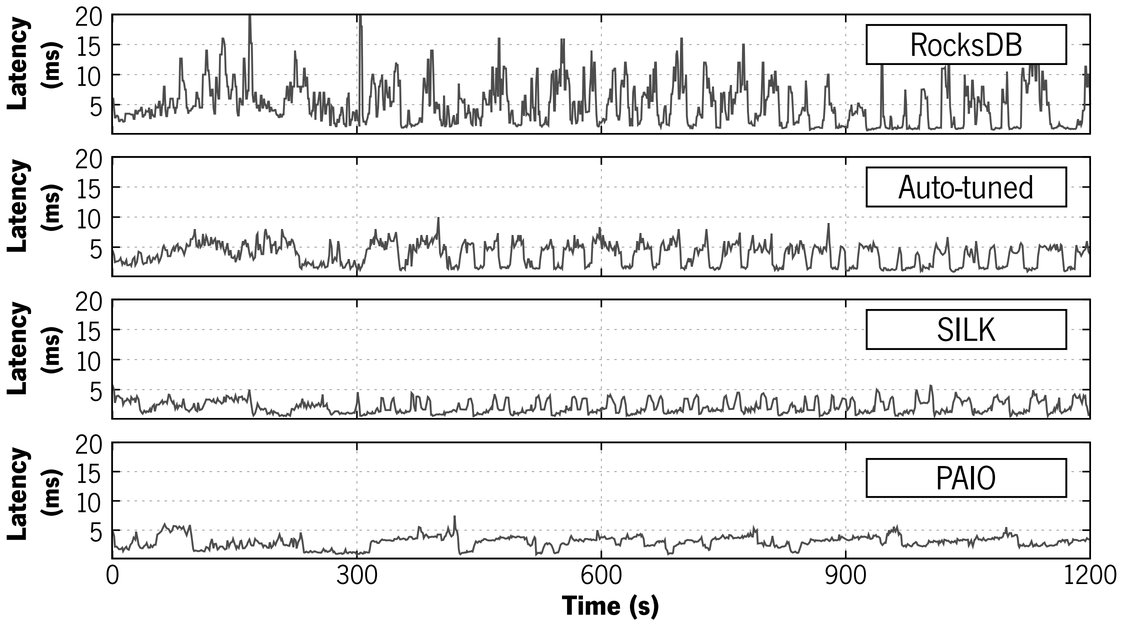


Figure 4.7: 99th percentile latency of RocksDB, Auto-tuned, SILK, and PAIO under *write-heavy* workload.

7.46 kops/s (RocksDB), 4.93 kops/s (Auto-tuned), 7.52 kops/s (SILK), and 8.88 kops/s (PAIO). During the loading phase, and until finishing the accumulated backlog (0–400 seconds), RocksDB experiences long periods of high tail latency, peaking at 111 ms. After that, it observes latency spikes due to constant flushes and low level compactions, with values ranging between 15–60 ms. Auto-tuned experiences low performance throughout the overall execution due to its auto-tuner algorihtm combined with the poor performance of the storage device. Specifically, because the device has low throughput and parallelism, the system cannot perform background tasks in a timely manner, which end up being enqueued in the internal queues, generating more backlog to perform. As a response, the auto-tuner algorithm reserves

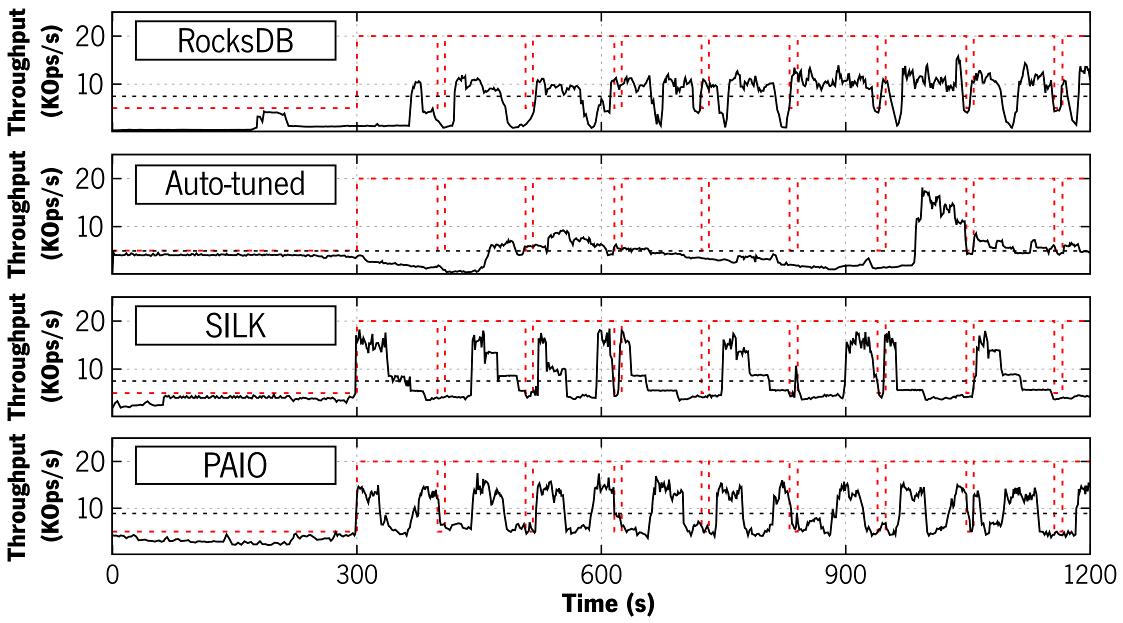


Figure 4.8: Throughput of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload without rate limiting (SATA SSD).

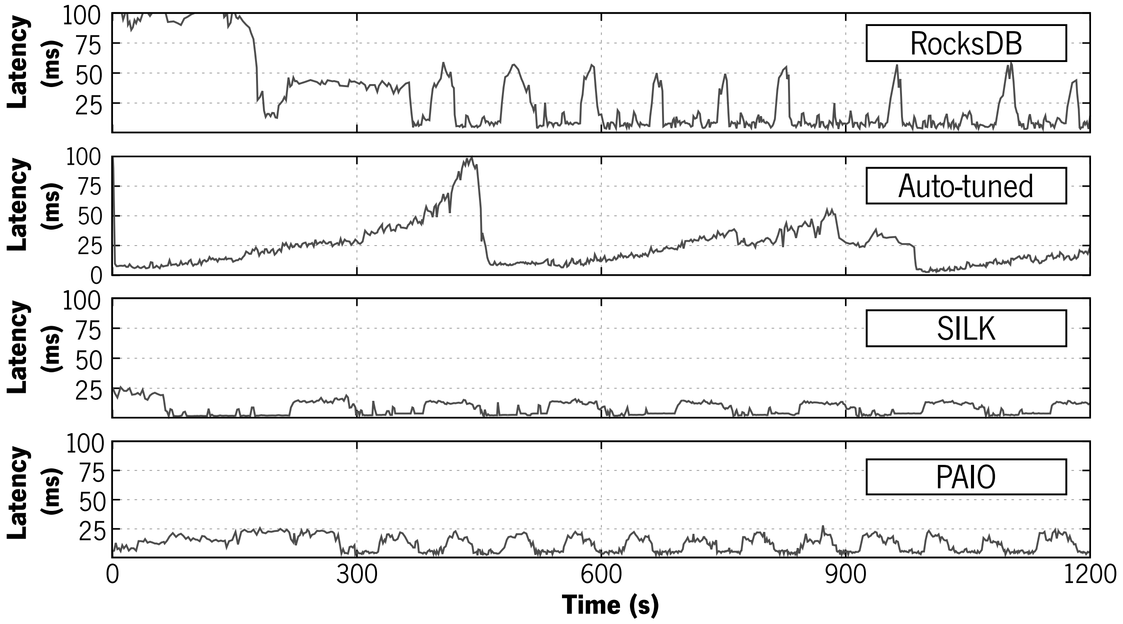


Figure 4.9: 99th percentile latency of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload without rate limiting (SATA SSD).

more bandwidth to background tasks, which consequently decreases the rate of foreground flows, and thus, causing high 99th percentile latency and low throughput. SILK and PAIO present a more sustained latency performance, never exceeding the 25 ms mark throughout the overall observation period. Specifically, while RocksDB and Auto-tuned experienced a variability of 21.1 ms and 12.2 ms, SILK and PAIO achieved 4.7 ms and 5.8 ms, respectively. The variability results correspond to the average of the absolute deviations of data points (*i.e.,* each tail latency measurement) from their mean. Throughput-wise, both

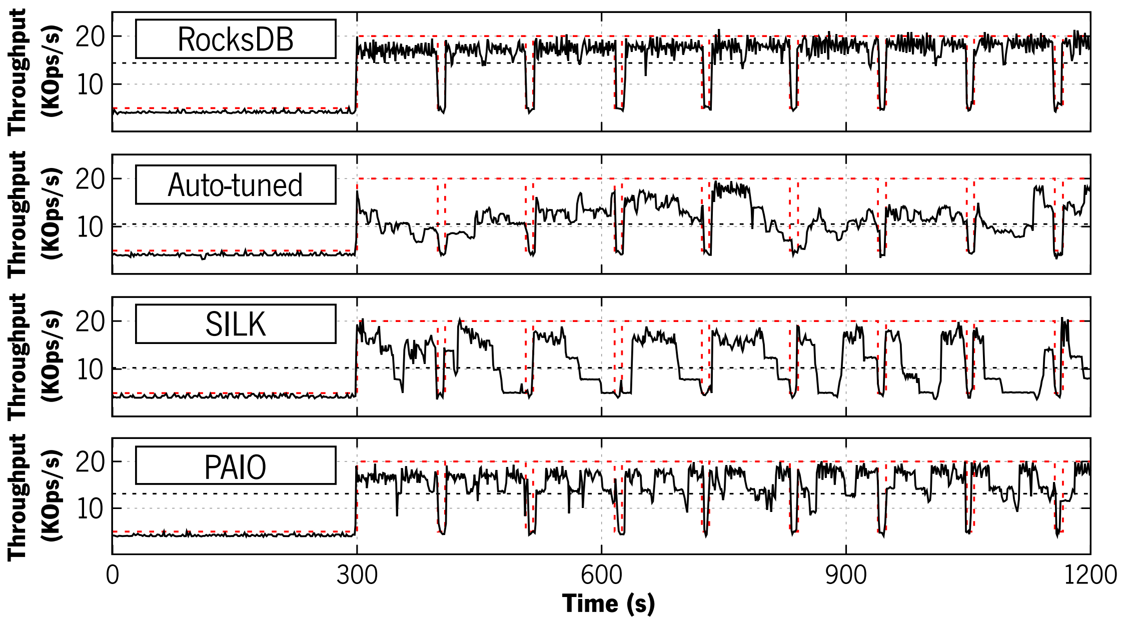


Figure 4.10: Throughput of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload without rate limiting (NVMe).

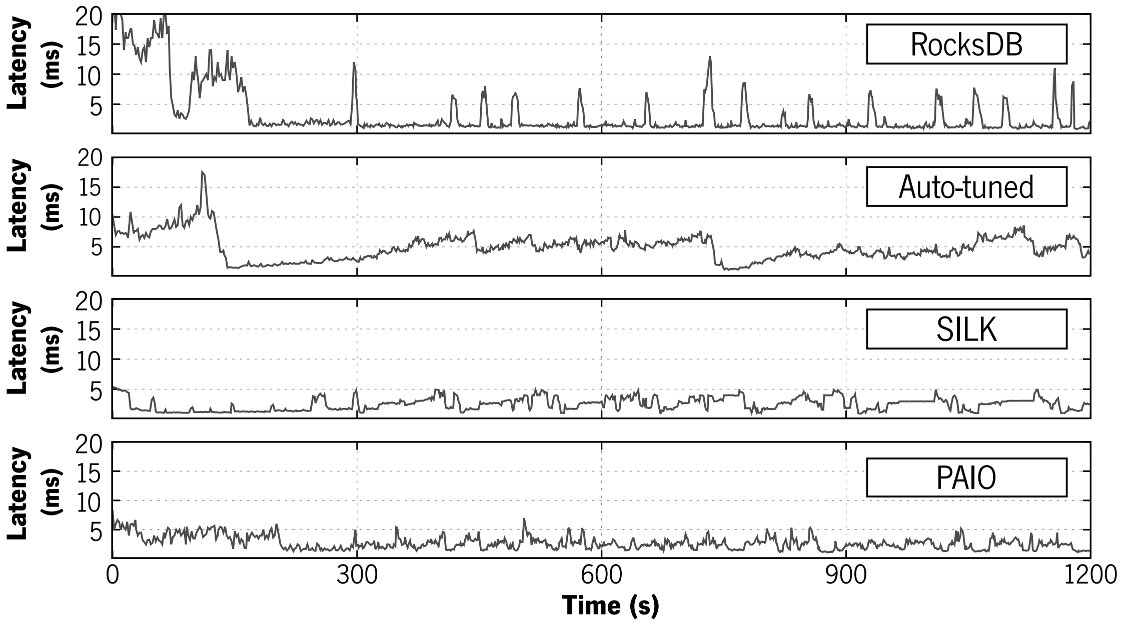


Figure 4.11: 99th percentile latency of RocksDB, Auto-tuned, SILK, and PAIO under *mixture* workload without rate limiting (NVMe).

systems observe periodic drops due to accumulated backlog. However, PAIO is able to recover faster than SILK. Because it cannot preempt compactions, PAIO reserves more bandwidth (than SILK) to low priority compactions, ensuring that high priority tasks do not wait to be executed. As such, PAIO follows a more *proactive* approach for assigning bandwidth to compactions, while SILK follows a more *reactive* approach.

Figures 4.10 and 4.11 depict the throughput and 99th percentile latency (respectively) under the NVMe device. All systems experienced higher throughput performance, averaging at 14.39 kops/s (RocksDB), 10.53 kops/s (Auto-tuned), 10.27 kops/s (SILK), and 13.11 kops/s (PAIO). RocksDB follows a similar performance curve as the theoretical client load. The reason behind this is twofold. First, it completes all accumulated backlog during the initial valley (at the cost of higher tail latency), which positively reflects in the remainder execution (*i.e,* no significant performance loss is observed). Second, since NVMe devices have higher throughput performance and parallelism than SSD devices (Figure 4.8), RocksDB achieves a more sustained performance. After the initial valley, RocksDB observes latency spikes that range between 7–15 ms due to frequent flushes and low level compactions. SILK and PAIO follow similar tail latency curves, never exceeding the 6 ms mark. In detail, throughout the overall observation period, RocksDB and Auto-tuned observed a variability of 2.5 ms and 1.5 ms (respectively), while SILK and PAIO only observed a variability of 0.8 ms. Similarly to previous results, both system experience periodic throughput drops.

4.5 Related Work

This section discusses and compares existing work with the PAIO data plane stage developed to achieve tail latency control in LSM-based KVS. For brevity, we refer to this data plane stage as PAIO–RocksDB.

System-specific optimizations. There is extensive prior work in building and optimizing KVS to improve client throughput and tail latency. Specifically, several systems achieve this by (1) reducing the overhead of background operations [16, 17, 130, 138, 181, 243], (2) implementing new compaction algorithms [3, 17, 168, 178], and (3) proposing new data structures [18, 82, 106, 178]. These systems however, have the same shortcoming as SILK, as they are tightly coupled with the targeted system, being directly implemented within the KVS. PAIO–RocksDB on the other hand, only required adding 85 LoC (of which 47 were used to perform context propagation), and does not entail any changes to RocksDB's internal mechanisms such as internal work queues, thread pools, or background operation handlers. As such, this approach demonstrates that by propagating application-level information to the data plane stage, PAIO–RocksDB can achieve similar performance and control as system-specific optimizations.

SDS systems. Current SDS systems are unable to enforce the storage policies demonstrated in this chapter (§4.3). Most systems are targeted for I/O layers that are not within this scope, being co-designed with hypervisors (IOFlow [211], sRoute [205], PSLO [128]), distributed file systems (Retro [143]), object stores (Malacology [194], Crystal[84]), and the block layer (Mesnier *et al.* [155]). PAIO–RocksDB however, is a data plane stage fined-tuned to handle the I/O workflows of RocksDB, being able to control tail latency performance under several workloads and testing scenarios.

Moreover, several SDS systems do not provide context propagation, inhibiting request differentiation at a finer granularity (*i.e.,* foreground *vs* high-priority *vs* low-priority background tasks) [175, 200, 201]. Some works use context propagation techniques to tag data across kernel-level layers, including the block layer and device drivers [155, 211]. However, because these actuate at the kernel, the *context* needed to enforce these policies requires breaking legacy APIs (user-to-kernel and kernel-internal interfaces), reducing portability and cross-compatibility. On the other hand, PAIO–RocksDB actuates at the user-level and enables propagating application-level information with minimal code changes and without modifying any interface of the involved layers of the I/O stack.

4.6 Summary and Discussion

In this chapter, we show that production-based LSM KVS experience high tail latency spikes when foreground and background I/O workflows compete for shared resources. To address this problem, we propose a new data plane stage built with PAIO that orchestrates the I/O rate at which foreground and background operations (including flushes, and low level and high level compactions) flow. By propagating application-level information (context at which a given POSIX operation is created, such as flushes and compactions), our PAIO stage outperforms RocksDB by at most 4× in 99^{th} percentile latency, under different workloads and testing scenarios. Moreover, we demonstrate that through minor code changes this data plane stage enables similar control and performance as SILK, which required profound refactoring to the original RocksDB codebase. These results allow us to conclude that *it is possible to implement complex I/O optimizations as user-level, general applicable storage data plane stages.*

Futhermore, considering a scenario where changing RocksDB would be prohibited, the proposed data plane stage would still be able to orchestrate its I/O workflows, but would not have the same degree of performance and effectiveness as demonstrated in this chapter. Under this scenario, the PAIO stage would intercept POSIX operations with `LD_PRELOAD` (*i.e.,* `open`, `read`, `write`, and `close`) and would be able to infer the type of each workflow. Specifically, flushes only submit `write` operations to SST files (whose file extension is given by `.sst`); foreground flows `read` from SST files and `write` to the WAL (whose file extension is given by `.log`); and compactions `read` and `write` from/to SSTs. However, because application-level information is not propagated, the data plane stage would not be able to differentiate between low level and high level compactions, which means that all compaction operations would be treated equally, ultimately leading to tail latency spikes.

5

Per-Application Bandwidth Control in Shared Storage Environments

HPC infrastructures are increasingly popular to support computational demanding workloads, including scientific simulations (*e.g.,* quantum chemistry [126], computational fluid dynamics [219]), neural network training [1, 169], and large scale visualizations [179]. Traditionally, to run these applications in a supercomputer, users allocate one or more compute nodes, having exclusive access to compute, memory, and storage resources. However, several research centers reported that many jobs that run on their infrastructures are not computational demanding and only require a subset of the available resources, leading to over-provisioning and waste of system resources.

The AI Bridging Cloud Infrastructure (ABCI) supercomputer [4], hosted by the National Institute of Advanced Industrial Science and Technology (AIST) research center, differs from traditional systems by enabling a cloud-like allocation model, where users can reserve a full compute node (or several) or a fraction of it, by having exclusive access to compute (CPU cores and GPU) resources, memory, and storage quota. However, the same is not ensured for local disk bandwidth, which leads to jobs competing for shared resources (creating I/O interference) or not being able to reserve different bandwidth priorities.

This chapter addresses this problem by proposing a data plane stage built with PAIO that ensures per-application bandwidth control under shared storage. The stage transparently intercepts I/O workflows of each job and dynamically rate limits them in holistic manner, according to a local controller that has system-wide visibility. Results demonstrate that all PAIO-enabled jobs that ran co-located in the same compute node were provisioned with their bandwidth goals, and their execution time improved when compared to a static rate limited setup.

5.1 AI Bridging Cloud Infrastructure Overview

ABCI is a TOP500 supercomputer that is designed upon the convergence between AI and HPC workloads. The current setup consists of over 1,088 compute nodes, each with 4 NVidia V100 GPU accelerators, 120 compute nodes set with 8 NVidia A100 GPUs each, and other computing resources. Jobs executed at ABCI are predominately AI and DL-oriented, being conducted with several AI frameworks such as TensorFlow [1], PyTorch [169], MXNet [44], and Chainer [212].

To execute these jobs users can reserve full compute nodes, following the traditional manner and having exclusive access to all system resources; or only fraction of the compute node, where jobs execute concurrently, similar to other shared infrastructures such as cloud ones. In the latter setting, compute nodes are partitioned into resource-isolated *instances* through *Linux control groups* (`cgroups`) [153]. Each instance has exclusive access to CPU cores, memory space, a GPU, and local storage quota. Depending on the selected configuration, *instances* can have more or less resources, being applied with the respective charging rate. However, the local disk bandwidth is still shared, and because each *instance* is unaware of the others (*i.e.,* jobs are executed in isolation), jobs compete for disk bandwidth leading to I/O interference and performance variation. Even if the block I/O scheduler is fair, all *instances* are provisioned with the same service level, preventing the assignment of different priorities and the enforcement of per-application bandwidth policies.

Static rate limiting. Using `cgroups`' Block I/O Controller (BLKIO) allows rate limiting `read` and `write` operations of each *instance* [29]. Specifically, BLKIO is an I/O subsystem that actuates at the OS block layer, and is responsible for controlling and monitoring the access on block devices. Currently, it supports two main policies:

- *I/O throttling:* this policy specifies the upper limit (*e.g.,* bytes per second, operations per second) of `read` and/or `write` operations that a given job (group) can submit to the block device. With this policy, I/O workflows are statically rate limited.
- *Proportional weight division:* this policy is implemented in the Completely Fair Queuing (CFQ) scheduler [41] and enables setting bandwidth weights to specific jobs (groups) in the system. This ensures that a given group has access to a reserved proportion of the overall disk bandwidth.

However, ABCI adopts the *I/O throttling* policy, which means that once the rate is set it cannot be dynamically changed at execution time, as it requires stopping the jobs, adjust the rate of all groups, and restart the jobs, being prohibitively expensive in terms of overall execution time. This creates a second problem where if no other job is executing in the node, the *instance* cannot use leftover disk bandwidth, leading to longer execution periods. Moreover, even if ABCI used the *proportional weight division* policy, it requires manual intervention from system administrators to adjust the bandwidth proportion assigned to each *instance*, to comply with the job's I/O requirements (*e.g.,* read-write proportion, weight of each job).

5.2 Per-Application Bandwidth Control with PAIO

To address this problem, we built a PAIO data plane stage that *implements the necessary mechanisms to dynamically rate limit I/O workflows at each instance*, while the *control plane, implements a proportional sharing algorithm to ensure all instances meet their policies*. Figure 5.1 depicts the organization of this

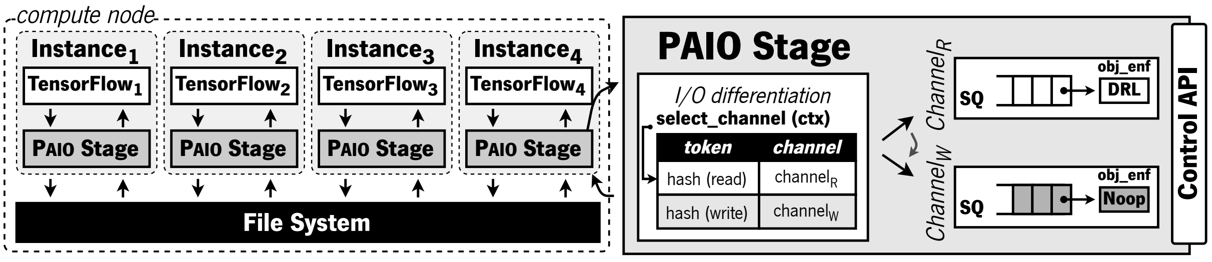


Figure 5.1: Organization of PAIO data plane stages that orchestrate TensorFlow instances for achieving per-application bandwidth control.

data plane stage. Our use case focuses on the DL model training phase. Each instance runs a TensorFlow job that uses a single I/O workflow to read dataset files from the local file system. Periodically, TensorFlow checkpoints the current state of the training model generating POSIX write operations. Depending on the complexity of the model, the resulting checkpoint state is typically within a few MiB to few GiB in size, and thus, it does not impose the main source of I/O contention and interference in this setting.

Data plane stage. The stage targets all POSIX `read` and `write` operations generated from TensorFlow's I/O workflows. We consider each TensorFlow thread that interacts with the file system, either for reading the dataset or persisting the checkpoint state, as a workflow. Each of these is handled by a PAIO channel, namely *Channel*$_R$ (read) and *Channel*$_W$ (write). Channel differentiation is made using the *operation type*. TensorFlow's `read` requests are handled with a DRL enforcement object, while `write` requests are submitted to a `Noop` enforcement object. Requests are enforced with the optimization described in §3.3. A PAIO stage is applied over each active instance in the system, as depicted in Figure 5.1.

Contrary to the use case presented in §4, this PAIO stage does not require context propagation, as policies can be met using the *request type* and *size* I/O classifiers. As such, instead of embedding this optimization within TensorFlow's codebase (which has approximately 2.3M LoC [83]), we integrated the stage without any code changes. To achieve this, we used `LD_PRELOAD` to reimplement the logic of the `read` and `write` routines of `libc` library, by intercepting and forwarding each operation to the PAIO stage (before being submitted to the local file system). All supported calls implement the logic necessary for the request to be enforced, including the creation of the *Context* object using the *request type* and *size* classifiers; stage enforcement; verification of the enforcement *Result*; and its submission to the original execution path (file system). Such an approach allows to transparently enforce I/O requests, regardless of the application that is submitting them, enabling general applicability, transparency, and portability.

Control algorithm. The control plane has global visibility of resources and implements a max-min fair share algorithm (in holistic manner) to ensure per-application bandwidth guarantees (Algorithm 5.1), which is typically used to achieve resource fairness policies [143, 211]. The overall disk bandwidth available (given by *Max*$_B$) and bandwidth demand of each instance (given by *demand*) are defined *a priori* by the system administrator or the mechanism responsible for managing resources of different job instances (*e.g.,* SLURM [238]). The algorithm uses a feedback control loop that performs the following

Algorithm 5.1: Max-min Fair Share Control Algorithm

```
   Initialize: $Max_B$ = N GiB; $Active > 0$; $demand_i > 0$
1  $\{I_0, ..., I_{Active-1}\}$ ← collect ()
2  $left_B$ ← $Max_B$
3  for $i = 0$ in [0, Active−1] do
4      if $demand_i \leq \frac{left_B}{Active-i}$ then
5          $rate_i$ ← $demand_i$
6      else
7          $rate_i$ ← $\frac{left_B}{Active-i}$
8      $left_B$ ← $left_B$ - $rate_i$
9  for $i = 0$ in [0, Active−1] do
10     $rate_i$ ← $rate_i$ + $\frac{left_B}{Active}$
11 enf_rule ($\{rate_0, I_0\}, ..., \{rate_{Active-1}, I_{Active-1}\}$)
12 sleep (loop_interval)
```

steps. First, it collects statistics from each active instance's stage, given by I_i (1), as well as the bandwidth generated by each TensorFlow job (collected at `/proc`). Then, it computes the rate of each active instance (3-10). If an instance's *demand* is less than its fair share, the control plane assigns its *demand* (4-5), assigning the fair share otherwise (7). It then distributes leftover bandwidth (*left*$_B$) across instances (9-10). Having computed all rates, because some operations may be absorbed by the OS page cache, the control plane calibrates the rate of each instance in a function of I_i, *rate*$_i$, and the bandwidth generated by each TensorFlow job, generating the enforcement rules (`enf_rule`) to be submitted to each stage (11). Finally, the control plane sleeps for *loop_interval* before beginning a new control cycle (12). For this use case, we used a local SDS controller (as described in §3.4.4), that ensures global visibility between all stages competing for shared local disk bandwidth.

5.3 Evaluation

We now demonstrate how the PAIO data plane stage achieves per-application bandwidth guarantees under a shared storage scenario. Our setup was driven by the requirements of the ABCI supercomputer.

Testbed configuration. Experiments were conducted in a compute node of the ABCI supercomputer with two 20-core Intel Xeon processors (80 cores), 4 NVidia Tesla V100 GPUs, 384 GiB of RAM, and a 1.6 TiB Intel SSD DC P4600, running CentOS 7.5 with Linux kernel 3.10 and `xfs` file system.

We used TensorFlow 2.1.0 [83] with the LeNet training model (I/O-bound) [124], configured with a batch size of 64 TFRecords [210]. We used the ImageNet dataset [186], that includes 1.28 million images (≈138 GiB) for training and 50,000 images (≈6 GiB) for validation. Each instance runs with a dedicated GPU and dataset, and its memory is limited to 32 GiB. Overall disk bandwidth is limited to 1 GiB/s (*Max*$_B$). All resources are isolated using Linux `cgroups`. At all times, the compute node executes at most four instances with equal resource shares in terms of CPU, GPU, and RAM. Each instance executes

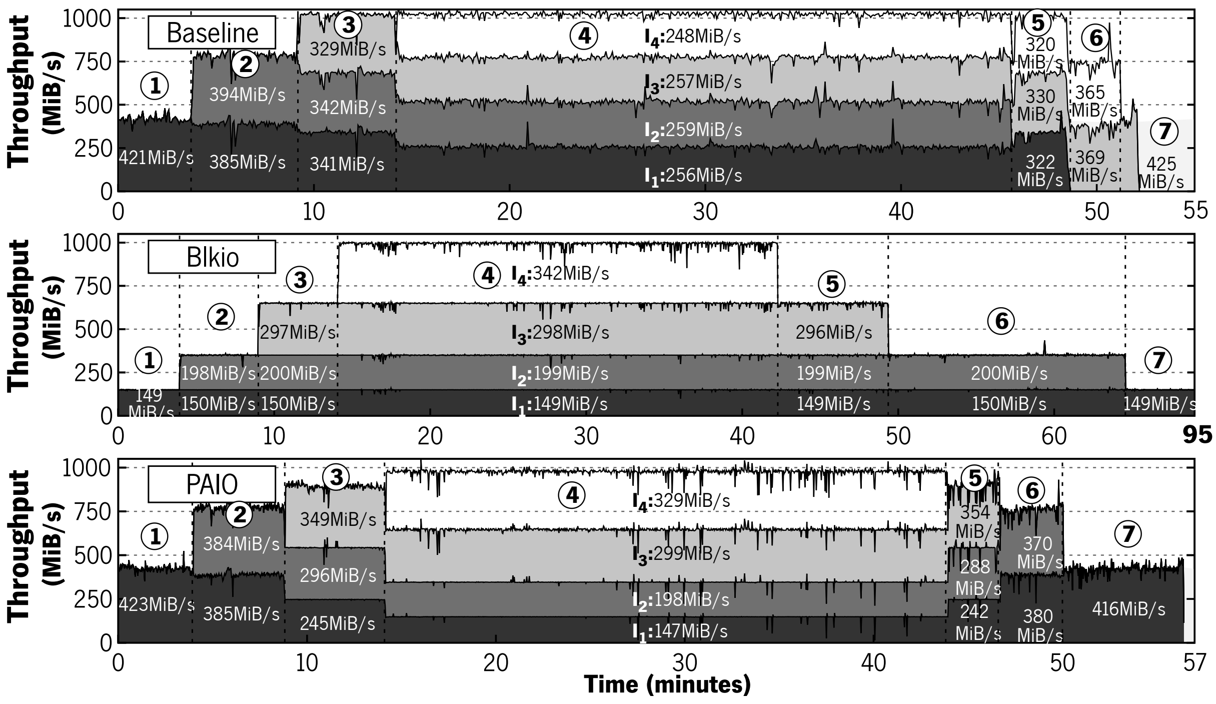


Figure 5.2: Per-application bandwidth under shared storage for Baseline, Blkio, and PAIO setups.

a TensorFlow job, is assigned with a bandwidth policy, and executes a given number of training epochs. Specifically, instances 1 to 4 are assigned with *minimum* bandwidth guarantees of 150, 200, 300, and 350 MiB/s, and execute 6, 5, 5, and 4 training epochs, respectively.

Setups. Experiments were conducted under three setups: *Baseline*, *Blkio*, and *PAIO*. *Baseline* represents the current setup supported at the ABCI supercomputer, where all instances execute without bandwidth guarantees (compete for local disk bandwidth). The *Blkio* setup represents a scenario where all instances are enforced with (static) disk bandwidth limits, using BLKIO. In the PAIO setup, each instance executes with a PAIO data plane stage that enforces the specified bandwidth goals dynamically. Across all setups, all instances have exclusive access to a share of compute and memory resources of the compute node. Figure 5.2 depicts, for each setup, the I/O bandwidth of all instances at 1-second intervals. Experiments include seven phases, each marking when an instance starts or completes its execution.

Baseline. Experiments were executed over 52 minutes. At ①, I_1 reads at an average rate of 421 MiB/s. Whenever a new instance is added, the I/O bandwidth is shared evenly (②). At ③, the aggregated instance throughput matches the disk limit. At ④, instance performance converges to ≈256 MiB/s, leading to all instances experiencing the same service level. However, since instances have different priorities, I_3 and I_4 miss their policies, as instances I_1 and I_2 have access to more I/O bandwidth than their fair share. After 46 minutes of execution (⑤), I_3 completes its executions, and leftover bandwidth is shared with the remainder instances. Again, I_4 cannot achieve its targeted goal. At ⑥ and ⑦, active instances have access to leftover bandwidth and finish their execution.

Blkio. Experiments were executed over 95 minutes. From ① to ⑦, whenever a new instance is added, it is provisioned with its exact bandwidth limit (ensuring that their bandwidth guarantees are met at all times). However, because the I/O rate of each instance is set using BLKIO, instances cannot use leftover bandwidth to speed up their execution. For example, while on *Baseline* I_1 executes under the 50-minutes mark, it takes 95 minutes to complete its execution on *Blkio*. To overcome this, a possible solution would require to stop and checkpoint the instance's execution, reconfigure BLKIO with a new rate, and resume from the latest checkpoint. However, doing this process every time a new instance joins or leaves the system is prohibitively expensive, as it would significantly delay the execution time of all running instances.

PAIO. Experiments were executed over 56 minutes. At ① and ②, instances are assigned with their proportional share, as the control plane first meets each instance demands and then distributes leftover bandwidth proportionally. At ③, contrary to *Baseline*, the control algorithm bounds the bandwidth of I_1 and I_2 to a mean throughput of 245 MiB/s and 296 MiB/s, respectively. As depicted in Algorithm 5.1, these bandwidth values respect to the fair share of each instance (4-5) combined with existing leftover bandwidth (9-10). At ④, all instances are set with their bandwidth limit. During this phase, *PAIO* provides the same properties as *Blkio*. From ⑤ to ⑦, as instances end their execution, active ones are provisioned as in ① to ③. Contrary to *Baseline*, *PAIO* ensures policies are met at all times, and shares leftover bandwidth across active instances whenever it is available. Compared to *Blkio*, *PAIO* finishes 39, 15, and 3 minutes faster for I_1, I_2, and I_3, and performs identically for I_4.

5.4 Related Work

This section discusses and compares existing work with the PAIO data plane stage developed to achieve per-application bandwidth guarantees under shared storage environments. For brevity, we refer to this data plane stage as PAIO–ABCI.

Storage optimizations. Many works ensure QoS SLOs at the hypervisor [27, 86, 87, 89] and block layer [29, 88, 154, 220, 233]. Virtualized environments, such as the case of hypervisor-level optimizations, are not supported over HPC infrastructures as jobs are executed with bare-metal access to resources. PAIO–ABCI intercepts the POSIX operations submitted from a given *instance* to the file system, thus being suitable for applications that are executed over either virtualized or bare-metal environments. Optimizations at the file system or block layer require changes to the kernel, decreasing portability and compatibility while increasing the risk of introducing bugs in production. PAIO–ABCI actuates at the user-level, intercepting POSIX system calls using `LD_PRELOAD`, thus not requiring changes to any layers of the I/O stack, including the application, file system, or block device.

Nevertheless, even though the aforementioned solutions are not directly applicable over this storage setting, the QoS algorithms proposed by these, such as mClock [87], pClock [88], SRP [89], could be incorporated with PAIO–ABCI as new control algorithms.

SDS systems. Existing systems that target the virtualization layer (*e.g.,* hypervisor, device drivers) such as IOFlow [211], sRoute [205], and PSLO [128], could be used for enforcing the policies presented in this chapter under traditional cloud-based infrastructures. However, given that this use case targets HPC infrastructures (ABCI), where jobs are executed with bare-metal access to resources, these solutions are unfit for ensuring such objectives. Other systems like Crystal [84], Cake [222], and Retro [143] enforce bandwidth SLOs over distributed file systems and object stores, making these approaches unsuitable for the storage setting demonstrated in this chapter. SafeFS [175] could be used to implement a FUSE-based file system that would provide the necessary mechanisms for rate limiting `read` and `write` system calls. However, due to the (costly) context switching between user-level and kernel-level attached to FUSE file systems, it would introduce significant performance overhead as demonstrated in existing literature [217]. On the other hand, the PAIO–ABCI data plane stage, alongside the described proportional sharing algorithm, are specifically designed for enforcing per-application bandwidth guarantees under shared local storage devices. Further, data plane stages built with PAIO incur low performance overhead, and can service thousands to millions of I/O requests per second, as demonstrated in in §3.6.

5.5 Summary and Discussion

In this chapter, we show the design, implementation, and evaluation of a data plane stage built with PAIO that enables per-application bandwidth control under shared storage environments at the ABCI supercomputer. We achieve this by (1) combining the PAIO data plane stage with `LD_PRELOAD`, which enables intercepting and handling POSIX operations (*i.e.,* `read` and `write` routines from the `libc` shared library) transparently, and enforcing the necessary I/O mechanisms (dynamic rate limiting) before being submitted to the shared storage device; and (2) connecting each PAIO stage to a local SDS controller that has global visibility over (local) shared resources and orchestrates the I/O rate of each stage holistically. Experiments demonstrate that contrarily to the current setup used on ABCI (*Baseline*), all PAIO-enabled instances are provisioned with their bandwidth goals. Moreover, whenever leftover bandwidth is available, PAIO distributes it across all active instances, decreasing the execution of TensorFlow instances by at most 39 minutes (41%), when compared to the *Blkio* setup. These results allow us to conclude that *it is possible to create generally applicable and transparent storage data plane stages that achieve storage objectives that require global visibility and control of resources.* We expect that the developed data plane stage is effective and applicable over multiple POSIX-compliant applications and storage scenarios that experience the same limitations as those described in this chapter.

6

Metadata Control in Parallel File Systems

Modern supercomputers are establishing a new era in HPC, providing unprecedented compute power that enables large-scale parallel applications to run at massive scale [61, 76]. However, contrary to long-lived assumptions about HPC workloads, where applications were predominately compute-bound and write-dominated, modern applications (*e.g.,* DL training) are data-intensive, read-dominated, and generate massive bursts of metadata operations [46, 56]. Indeed, several centers have already observed a surge of metadata operations in their clusters, and they expect this to become more severe over time [134, 170].

While these workloads demand scalable, high throughput, and low latency storage, most TOP500 supercomputers [213] rely on Lustre-like PFSs, which provide a centralized metadata management service [39, 47, 192]. In these data centers, having multiple concurrent jobs competing for shared I/O resources can lead to severe I/O contention and performance degradation [134, 171, 176]. For example, existing studies report that even a single user's I/O operations can saturate Lustre metadata resources, leading to unresponsiveness of the file system, reduced speed of computations for all running jobs, and even failures of metadata servers [97, 134, 176]. While there are numerous solutions to assess the bottlenecks generated from data workflows in HPC clusters [62, 107, 171, 176, 203, 244, 245], the metadata counterpart has not received the same level of attention, and existing approaches are suboptimal.

This chapter discusses the design and implementation of PADLL, a storage middleware built with PAIO that enables QoS control of metadata workflows in HPC storage systems. It allows system administrators to proactively and holistically control the rate at which POSIX requests are submitted to the PFS. It introduces a new proportional sharing algorithm that continuously readjusts reservations of metadata operations to prevent over-provisioning/under-provisioning across active jobs. Results demonstrate the performance and applicability of PADLL under different scenarios using both synthetic and realistic I/O workloads.

6.1 Parallel File Systems Overview

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 chapter, we focus on Lustre-like file systems (*e.g.,* Lustre [34, 192], BeeGFS [47], PVFS [39]), which are present in most

TOP500 supercomputers. A typical Lustre-like file system consists of several building blocks. Metadata Server (MDS) nodes 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 Target (MDT) nodes. Data operations are serviced by Object Storage Server (OSS) nodes, which are connected to compute nodes via high-speed interconnects, and store files on Object Storage Target (OST) servers. Files are typically distributed across multiple OSTs for parallelism and availability. File system *clients* reside 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 [142]. In some deployments, the namespace is persisted across multiple MDTs and a single MDS handles all metadata operations, having additional MDS nodes as standby replicas; in others, different MDSs/MDTs manage/persist different parts of the namespace, balancing the metadata load between them.

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

When used at scale, this centralized design comprises several limitations that can severely bottleneck the file system and impact the performance of all running jobs. First, different metadata operations carry different costs to the PFS. Depending on the file system implementation, read-only operations such as `getattr` only require acquiring read-locks, while operations like `open`, `close`, and `unlink` require more expensive locking, as they need to update the namespace state [34, 141]. 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 [54], OpenImages [119]), 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. For instance, the current configuration of the Frontier supercomputer holds 40 MDS nodes that serve more than 9,400 file system clients (one client per compute node), representing an approximate ratio of MDS nodes to file system clients of 1:235 [76].

6.1.1 Analyzing Metadata Operations in Production Clusters

To understand the impact of metadata operations in production, we analyze the logs of a Lustre file system from the ABCI supercomputer. The storage at ABCI is made of multiple PFSs. Of these, the `/group` area is managed by a *DataDirect Networks 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$_A$.

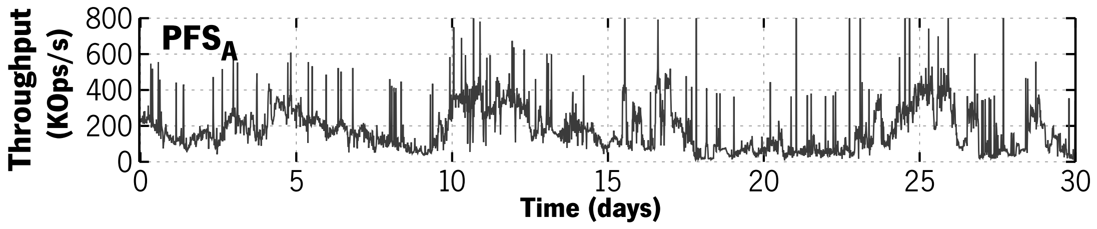


Figure 6.1: Throughput of metadata operations in PFS$_A$ throughout a 30-day period.

We monitored the I/O activity of the most frequent metadata operations at MDSs/MDTs, using *DataDirect Networks Storage's LustrePerfMon* [53]. We collected per-MDT performance statistics for `open`, `close`, `getattr`, `setattr`, `rename`, `mkdir`, `mknod`, `rmdir`, `statfs`, `sync`, and `unlink` operations. The logs report per-operation performance statistics captured with 1-minute samples over a 30-day observation period. Further, we also monitored the I/O bandwidth (`read` and `write`) observed by OSSs over the same observation period.

Overall metadata load. We first examine the throughput of metadata operations throughout the overall observation period. Figure 6.1 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 [170]. For example, a study from the National Energy Research Scientific Computing Center (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).

Moreover, existing reports indicate that the current limit for creating files in Lustre file systems is 50 kops/s and the known production usage is 15 kops/s [10, 78]. While we do not have exact values of `creat` operations, PFS$_A$ experienced several periods that lasted from few minutes to several days serving `open` operations with rates comprehended between 50 kops/s and 125 kops/s.

> **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 [170] and the results observed from PFS$_A$, it is expected that these values will continue to increase over time.*

Metadata throughput *vs.* I/O bandwidth. Figure 6.2 depicts the `read` (top) and `write` (middle) bandwidth of PFS$_A$'s OSSs. Given that most jobs conducted at ABCI are AI-oriented, write throughput is low, averaging at 0.6 GiB/s, while reads are served at an average rate of 48 GiB/s. However, as depicted

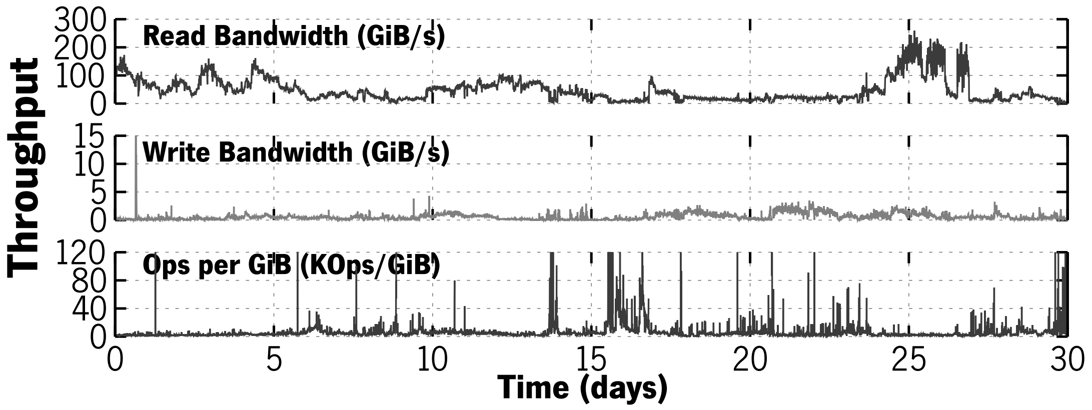


Figure 6.2: Throughput of read (top) and write (middle) operations from PFS$_A$'s OSS nodes, and ratio between metadata operations (kops/s) and I/O bandwidth (GiB/s) of PFS$_A$ (bottom).

in Figure 6.2 (bottom), we observe that in certain periods metadata operations have a significantly higher throughput than GiBs read/written from/to the PFS. For example, between days 13 and 20, in several time periods, metadata operations were submitted at a rate over 120 kops for each GiB (or 120 ops for each MiB) read/written from/to the PFS. This behavior occurs when there is a subset of metadata-intensive jobs running in the cluster.

Observation #2. *There are several time periods where the amount of submitted metadata operations far exceeds the GiBs of data read/written from/to the PFS. This means that only ensuring QoS for data workflows is insufficient for controlling the overall PFS performance, and metadata operations should be handled as well.*

Type and frequency of metadata operations. Figure 6.3 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 [34, 141]. As for `getattr` operations, while less costly than the others, PFS$_A$ received almost 250,000 million requests during the observation period, representing an average and continuous rate of 95.8 kops/s.

Observation #3. *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.*

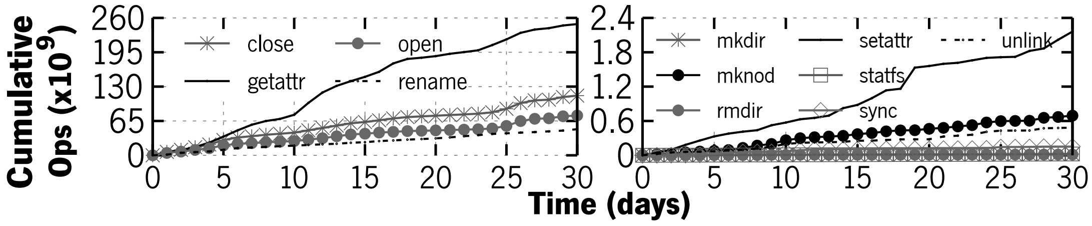


Figure 6.3: Cumulative metadata operations in PFS$_A$.

6.1.2 Current Approaches

Existing approaches for controlling metadata workflows in HPC storage systems experience the following shortcomings.

Manual intervention. In several HPC research facilities, system administrators stop jobs with aggressive metadata 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 [97, 134]. While this helps protecting 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 CALCioM [62], GIFT [171], and TBF [176] propose optimizations to mitigate I/O contention and performance variability, these are tightly coupled to the system implementation and require high intrusiveness to several I/O layers of the HPC software stack, including the shared PFS, job scheduler (*e.g.,* SLURM), and I/O libraries (*e.g.,* MPI-IO). 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.

In particular, solutions that actuate at the PFS level are especially challenging due to three main reasons. First, tightly coupled optimizations may be difficult to port between different PFSs (*e.g.,* Lustre *vs.* BeeGFS *vs.* PVFS) as even though they share a similar high-level design, the internal logic differs across implementations. Second, several clusters use storage appliances from *DataDirect Networks*, *Fujitsu*, *IBM*, or other storage vendors, being unable to implement optimizations over such systems. Finally, given that several HPC centers use and maintain *in-house* versions of open-source PFS implementations, fine-tuned for their I/O requirements and workloads, indiscriminately implementing these optimizations can introduce bugs or cause performance issues.

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 [96]. 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 [201].

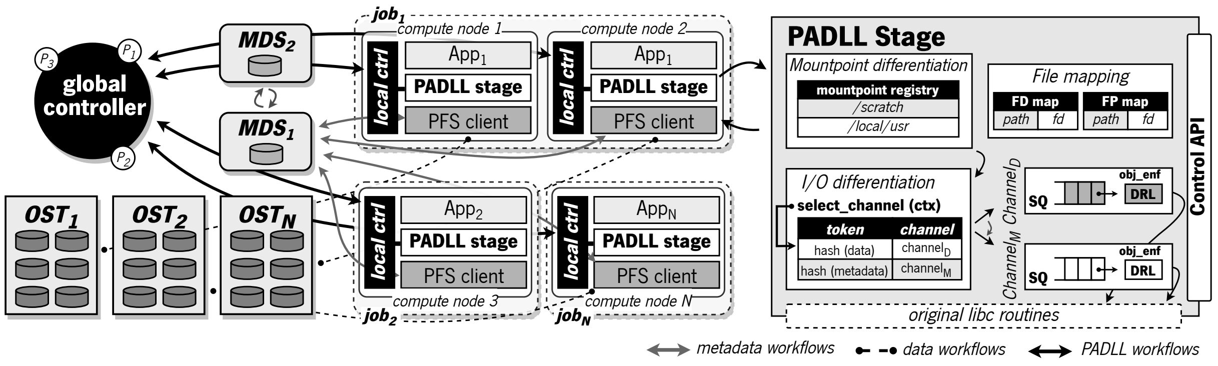


Figure 6.4: High-level architecture of the PADLL storage middleware and organization of data plane stages for achieving QoS control of metadata operations in PFSs.

6.2 PADLL Storage Middleware

We address these challenges with PADLL, an application and file system agnostic storage middleware that enables QoS control of metadata workflows in HPC storage systems.[1] 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. As depicted in Figure 6.4, PADLL follows a decoupled design. The data plane (§6.2.1), partially built with PAIO, is made of multiple stages that are distributed over compute nodes, each mediating the I/O requests between a given application and the shared file system. The control plane follows a hierarchical design where local controllers are placed in each compute node, while a global controller ensures cluster-wide visibility.

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 file system to which the requests are submitted to. Being separated from the PFS enables applying PADLL over multiple applications under different POSIX-compliant storage backends. This is important since (1) different HPC centers may use different file systems (*e.g.,* Lustre, BeeGFS, GPFS, PVFS) or storage appliances; (2) it is compliant with *in-house* file system implementations that have code hardened optimizations made through several years of practice; and (3) does not require users to change their applications.

6.2.1 Data Plane

The PADLL data plane was partially built using PAIO. Stages actuate at the compute node level, each of which sits between the application and the file system layer. PADLL transparently intercepts and reimplements multiple POSIX system calls from different operation classes before being submitted to the PFS, including data (*e.g.,* `read`, `fwrite`), metadata (*e.g.,* `open`, `rename`), extended attributes (*e.g.,* `getattr`, `setattr`), and directory management (*e.g.,* `mkdir`, `mknod`).

[1] PADLL stands for Programmable and Adaptable I/O workflows for Dynamically Loaded Libraries.

To control the rate of I/O workflows of a given job, multiple PADLL stages may be used. Under single node jobs, a single stage may handle all I/O workflows. As depicted in Figure 6.4, this is the case of *job*$_2$ where *App*$_2$ only executes at *compute node 3*. For distributed jobs, where application instances run on separate compute nodes, multiple stages need to be set (*i.e.,* one per instance). For example, as depicted in Figure 6.4 *job*$_1$, two stages are needed to effectively rate limit the I/O workflows of *App*$_1$, since it executes in *compute nodes 1* and *2*.

Mountpoint differentiation. Compute nodes can have access to multiple file systems, both local (*e.g.,* `xfs` or `ext4` for managing local storage devices, `tmpfs` for non-persistent file storage) and remote (*e.g.,* NFS client which submits requests to a remote NFS server; Lustre client which submits requests to a Lustre file system). Given that PADLL intercepts POSIX requests regardless of the destined file system, it needs to identify which requests are destined towards the PFS to be treated accordingly. PADLL performs this differentiation in three phases:

- *Registering mountpoints:* the system administrator defines which mountpoints should be managed with PADLL by registering their full path on a *mountpoint registry*. For example, as depicted in Figure 6.4, PADLL can handled all requests that are destined towards the file systems whose mountpoints are `/scratch` and `/local/usr`.
- *Handling path-based operations:* all system calls that define the *pathname* of the targeted file, such as `open`, `fopen`, `rename`, and `mkdir`, are intercepted and analyzed. Requests that are destined towards any of the registered mountpoints will follow the regular operation flow of a PAIO stage, namely I/O differentiation and enforcement; otherwise, requests are directly submitted to the corresponding file system without any changes.
- *Handling file descriptor and file pointer based operations:* to determine if a system call that accesses files through file descriptors (*e.g.,* `read`, `fgetattr`) or file pointers (*e.g.,* `fwrite`, `fclose`) is destined towards a mountpoint listed in the *mountpoint registry*, for each valid `open`-based call PADLL stores the resulting file descriptor (or file pointer) in an internal *file mapping* module. Then, whenever these file descriptor (or file pointer) based system calls are intercepted, PADLL verifies if the corresponding identifier is registered in the *file mapping*. If so, requests follow the regular operation flow of a PAIO stage; otherwise, these are directly submitted to the corresponding file system without additional processing. On `close`-based operations, the file descriptor is removed from the *file mapping*.

Context object creation. After differentiating which requests should be enforced (*i.e.,* rate limited), PADLL creates the *Context* object to be submitted to the PAIO data plane stage. For each request, the *Context* object is built with the following I/O classifiers: *workflow id*, set with the thread-ID; *operation type* (*e.g.,* `open`, `read`, `close`, `getattr`); *operation context*, which defines the operation class that a given operation respects to — `read` and `write`-based operations are defined as `data`, while the remainder such as `open`, `rename`, `mkdir`, and `getattr`, are defined as `metadata`; and *operation size*, which defines the cost in tokens of each operation, being set to 1 for *metadata* operations and the corresponding *buffer size* for `data` operations.

I/O differentiation and enforcement. The stage targets all POSIX operations studied in §6.1.1, given their prevalence in the I/O workloads observed in HPC storage systems. As depicted in Figure 6.4, operations may be handled by two PAIO channels, namely $Channel_D$ for `data` and $Channel_M$ for `metadata` operations. Unless stated otherwise, channel differentiation is made using the *operation context*. Due to the different evaluation scenarios presented in this chapter, further discussion on the exact POSIX operations handled by PADLL is made in §6.3. All channels are configured with DRL enforcement objects to ensure requests are rate limited before being submitted to the PFS.

Implementation. We implemented the PADLL's data plane with 16K lines of C++ code. The data plane exposes a POSIX interface that reimplements 42 calls from different operation classes. Moreover, it implements the necessary building block for differentiating requests based on their mountpoint, including the *mountpoint differentiation* and *file mapping* modules The logic for rate limiting requests, namely channel differentiation and enforcement, was built using PAIO.

6.2.2 Control Plane

The control plane follows a hierarchical design (as described in §3.4.4). The *global controller* has global system visibility and enforces cluster-wide policies over the overall SDS system, and is deployed in a dedicated compute node. *Local controllers* have local visibility and only orchestrate the data plane stages of a given compute node. Communication flows in a hierarchical manner; specifically, to enforce policies and collect I/O statistics, the *global controller* communicates with *local controllers*, which in turn communicate with their corresponding data plane stages.

Given that each compute node can have multiple data plane stages, either due to multiple concurrent jobs (as discussed in §5) or from multi-process applications, which create one data plane stage per process, having a *local controller* allows reducing the number of connections to the *global controller*. Currently, *local controllers* act as proxies that aggregate statistics from stages before being dispatched to the *global controller*, and forward enforcement rules to the respective stages. The actual *control decisions* are made by the *global controller*. We leave scalability and dependability improvements, as well as the delegation of control responsibility to *local controllers* as future work.

Orchestrating stages from the same job. Every time a job starts, its corresponding stages are initialized and connected to the *local controller*. Stages send to the controller information that characterizes the job and the compute node where it is running (*e.g.,* `job-ID`, PID, hostname, username). The *local controller* then aggregates this information and shares it with the *global controller*. Based on this, the control plane knows which job each stage respects to, orchestrating the stages that belong to the same `job-ID` as a single one.

Algorithm 6.1: Max-min Fair Share Without False Resource Allocation Control Algorithm

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

Control algorithm. Algorithm 5.1 is a common proportional sharing algorithm used to enforce I/O fairness policies when a given resource is bottlenecked (*i.e.,* if a resource is overloaded, the policy reduces its load by orchestrating the workflows that access it while ensuring max-min fairness), as demonstrated in §5 and in previous work [143, 211]. However, while this algorithm is well suited for workloads that have a sustained I/O behavior, it is suboptimal under volatile and bursty workloads. Specifically, the algorithm assigns to active instances either their *demand* or their *fair share*, while also proportionally distributing any leftover resources in the system; however, if a given instance exhibits a volatile workload, the algorithm may assign a resource share larger than that the instance needs, which results in over-provisioning. We refer to this behavior as ***false resource allocation***.

Since metadata workloads in HPC storage systems are volatile and bursty (§6.1.1), we propose a new max-min fair share algorithm that prevents false resource allocation to ensure QoS control over metadata workflows, which is depicted in Algorithm 6.1. Briefly, rather than assigning the instance's *demand* or *fair share* exclusively based on the number of active instances in the system, we consider the actual usage of each instance and redistribute resources in a max-min fair share manner based on those observations.

Given the context and problem discussed in this chapter, from this point forward, we consider the metadata throughput that a given file system can service as the main resource to be distributed among instances. The overall metadata rate available (given by Max_R) and metadata rate demand of each instance/job (given by *demand*) are defined *a priori* by the system administrator or the mechanism responsible for managing resources of different job instances (*e.g.,* SLURM). The algorithm is computed on the *global controller* and uses a feedback control loop that performs the following steps. First, it collects statistics from each active instance's stage to determine its actual metadata rate usage, given by $usage_i$

(1). For each active instance, the algorithm computes its *fair_share* (4) and verifies if the current rate (*usage*$_i$) is lower than its *demand* (5). Under this scenario, *instance*$_i$ can be serviced at a rate lower than its *demand*. As such, it assigns the minimum between *fair_share* and *usage*$_i$+*threshold*$_i$ (7). *Threshold*$_i$ is computed based on the product of a configurable ε value and the different between *demand*$_i$ and *usage*$_i$, and is used to absorb the rate of highly volatile workloads (6). If *usage*$_i$ is higher than *demand*$_i$, the controller assigns the minimum between *demand*$_i$ and the *fair_share* (8–9).

The algorithm then distributes leftover rate (*left*$_R$) across actives instances (11–14). Specifically, it computes the overall rate used by all instances (11), and assigns *left*$_R$ based on their usage proportion, given by *usage_proportion*$_i$ (13–14). Finally, the *global controller* generates the enforcement rules (`enf_rule`) to be submitted to each *local controller* (15), and sleeps for *loop_interval* before beginning a new control cycle (16).

Implementation. We have implemented PADLL's control plane with 6K lines of C++ code. The implementation of the control plane discussed in §3.5 served as basis for the development of this one. The *global controller* implements the necessary building blocks for specifying policies and control algorithms, and managing *local controllers* (*e.g.,* collect statistics, submit rules). *Local controllers* implement the logic for communicating with data plane stages, and preprocessing the statistics that will be shared with the *global controller*. Communication between *local controllers* and data plane stages is established using UNIX Domain Sockets, while communication between controllers is established through RPC, using the gRPC framework [85].

6.3 Evaluation

We now demonstrate how PADLL can enforce metadata QoS policies in PFSs. Specifically, our evaluation demonstrate that PADLL (1) can enforce policies at different granularities, (2) can control I/O burstiness, (3) enforces I/O prioritization and proportional sharing objectives over concurrent jobs in the system, and (4) has negligible overhead.

Testbed configuration. Experiments were conducted in compute nodes of the Frontera supercomputer [204]. Each compute node is equipped with two 28-core Intel Xeon processors (112 cores), 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, namely `read` and `write`, we used the *IOR* benchmark [197]. *IOR* is a synthetic I/O benchmark used to evaluate HPC storage systems. The `write`/`read` workload sequentially writes/reads a single file with 875 GiB using POSIX-compliant system calls.

To generate realistic metadata workloads, we implemented a *trace replayer* that submits (*"replays"*) metadata operations (namely, `open`, `close`, `rename`, `getattr`, `setattr`, `mkdir`, `mknod`, `rmdir`, `statfs`, and `unlink`) 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 metadata operation at a rate that follows the same performance curve as the original logs. The rate at which operations are submitted depends on how many compute nodes are used for a given experiment; this is further discussed in the following sections. The execution period was also accelerated, where each second of the *replayer* corresponds to a minute's worth of operations in the original log.

Methodology. For all experiments, the *global controller* runs at a dedicated compute node, and each job respects to the execution of *IOR* or the *trace replayer* under a specific workload. *IOR* experiments were conducted using the PFS. 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, at user-space, it can intercept and handle POSIX operations submitted to any in-kernel file system registered in the VFS, including local and remote file systems (*i.e.,* Lustre kernel client). We expect PADLL to achieve the same level of effectiveness when used over PFSs.

6.3.1 Functional Evaluation

This section demonstrates PADLL's capability of rate limiting I/O workflows at different levels of granularity, namely *per-operation type* and *per-operation class*. All experiments presented in this section were conducted under a single compute node.

Setups. Experiments were conducted under three setups:

- *Baseline*: represents the benchmark — *IOR* or *trace replayer* — without using PADLL.
- *Passthrough*: respects to a scenario where POSIX operations submitted by the benchmark are intercepted by PADLL but are not rate limited. This setup was used to measure the overhead introduced by PADLL.
- *PADLL*: represents a scenario where POSIX operations submitted by the benchmark are intercepted by PADLL and throttled at a given rate.

Workload configuration. Experiments using the *trace replayer* were configured as follows. The rate of each operation type was scaled-down to half to ensure that the file system could serve them without overloading. The trace used in the experiments corresponds to the metadata operations of a single MDT server of the PFS$_A$ file system.

Per-operation type rate limiting. 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 experiments, PADLL was configured to throttle operations with a static rate, whose value changes every *N* minutes upon instruction of the system administrator, namely 6 minutes for metadata and 1 minute for data operations.

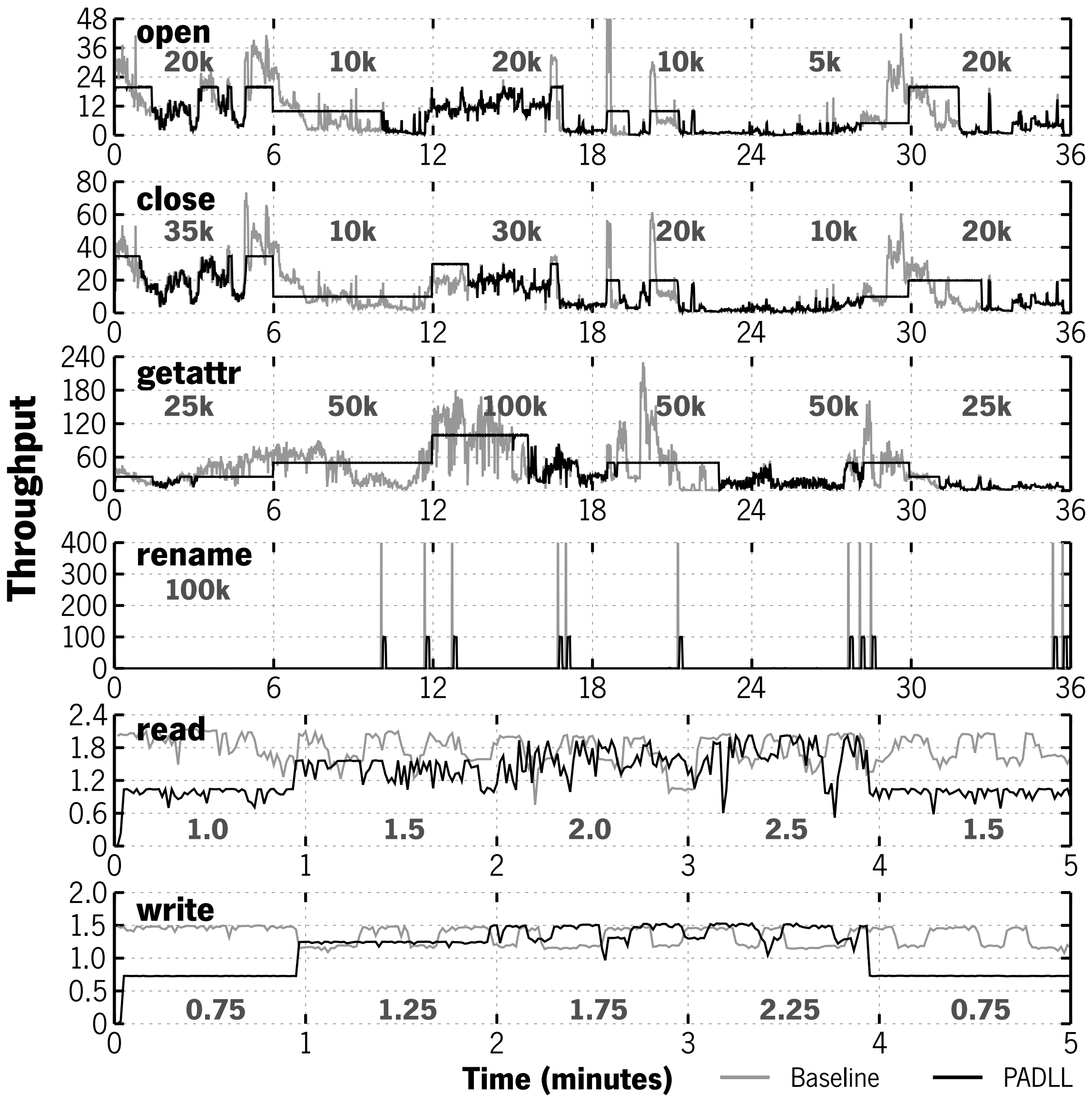


Figure 6.5: Per-operation type rate limiting. Experiments show that PADLL can enforce different rate limits over different POSIX operations, including `open`, `close`, `getattr`, `rename`, `read` and `write`.

Figure 6.5 depicts the throughput results of *Baseline* and *PADLL* setups under different operation types. Grey lines depict the *Baseline* setup, while black lines depict the PADLL setup.

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 the `open` experiment between 12 and 18 minutes (*i.e.,* periods where the black line is not flat). This is because, the limit set by the system administrator, for that interval, is higher than the operations submitted by the *replayer*. Analogously, we observe periods where the *PADLL* setup achieves higher throughput than *Baseline*, as observed in the `getattr` experiment between the 6 and 12 minutes interval. 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.

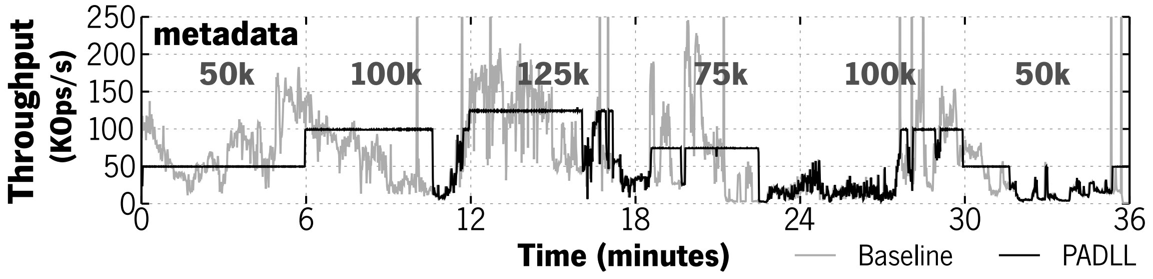


Figure 6.6: Metadata rate limiting with PADLL.

We observe similar results for data-oriented operations. However, since the targeted file system of the *IOR* experiments is the shared storage backend (*i.e.,* Lustre), we notice more variability.

Per-operation class rate limiting. We now demonstrate how PADLL controls the I/O workflows of a given operation class, namely `metadata`. We followed a similar testing methodology as the previous setup. The *trace replayer* spawns ten threads, one for each operation type. Figure 6.6 depicts the obtained results. The throughput corresponds to the accumulated rate of all *replayer* threads.

At all times, *PADLL* effectively controls the rate of all metadata operations throughout the overall experiment. In several periods, *PADLL* matches or achieves higher throughput performance than *Baseline*; we draw similar observations as in the previous setup.

Overhead. To further 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 Figures 6.5 and 6.6, as its performance line practically overlaps with the *Baseline* one.

6.3.2 Per-Job QoS Control

We now demonstrate how PADLL achieves metadata QoS control in HPC storage systems by orchestrating the I/O workflows of active jobs in the supercomputer. Under this scenario, metadata operations are seen as a finite and shared I/O resource, and for the PFS to provide sustained I/O performance, jobs need to meet specific metadata SLOs.

Testbed configuration. At all times, there are at most four jobs in the system, each running in a dedicated compute node and executing the *trace replayer*, which submits metadata operations at different rates. Jobs are incrementally added to the system every 3 minutes. Under this scenario, we consider that the system administrator defines a maximum rate of metadata operations (Max_R) that can be submitted to the targeted storage system, being set at 220 kops/s. Again, to prevent harming the production PFS, metadata operations are submitted to each compute node's local file system. As such, Max_R corresponds the maximum metadata rate available to all of these file systems.

Workload configuration. The trace used in the experiments corresponds to the metadata operations of all MDT servers of PFS$_A$; *i.e.,* the combined rate of all jobs follows the overall metadata load observed

Table 6.1: Per-Job QoS Control Testing Scenarios.

	Testing scenario #1	**Testing scenario #2**	**Testing scenario #3**
Job$_1$	{ 25% – 30 kops/s }	{ 15% – 30 kops/s }	{ 15% – 80 kops/s }
Job$_2$	{ 25% – 50 kops/s }	{ 20% – 50 kops/s }	{ 20% – 50 kops/s }
Job$_3$	{ 25% – 60 kops/s }	{ 20% – 60 kops/s }	{ 20% – 60 kops/s }
Job$_4$	{ 25% – 80 kops/s }	{ 45% – 80 kops/s }	{ 45% – 30 kops/s }

at the PFS$_A$ file system. Each job is assigned with a given load that varies across testing scenarios, as depicted in Table 6.1. For instance, in *testing scenario #1*, all jobs submit the same load (*i.e.,* 25%). The trace replayer was configured with 10 threads, each of which replayed a given operation type with the same rate as the original log.

Setups. Experiments were conducted under five setups: *Baseline* represents the current setup supported at most supercomputers, where all jobs execute without any throttling. The remainder setups are rate limited with PADLL, with a maximum combined rate of *Max*$_R$. *Uniform* represents a scenario where jobs are throttled with a fixed limit throughout their entire execution. Across all testing scenarios, each job is rate limited to 55 kops/s. *Priority* represents a setup where jobs are statically rate limited, similarly to *Uniform*, but are assigned with different rates. This enables defining different I/O priorities to all jobs in the system. The *Proportional sharing* setup represents a scenario where metadata workflows are controlled with Algorithm 5.1. Specifically, the control algorithm enforces per-job metadata rate reservations and proportionally distributes leftover metadata rate whenever it is available. Finally, *Proportional sharing without false allocation (PSFA)* represents a scenario where metadata workflows are controlled with Algorithm 6.1, which prevents *false resource allocation* when workloads are bursty and volatile. For *Priority*, *Proportional sharing* and *PSFA* setups, the rate limits of each job are depicted in Table 6.1, which vary across testing scenarios. For *Proportional sharing* these limits represent the per-job maximum rate when all jobs are active. For *PSFA* these limits represent the per-job maximum rate when (1) all jobs are active and (2) each job's *usage* is higher than its *demand*.

Testing scenarios. To provide a comprehensive evaluation testbed, we consider three testing scenarios with varying load proportions and rate limits. Table 6.1 depicts the load proportion and the metadata rate limit for each combination of testing scenario and job. The load proportion varies across all setups, while the metadata rate limit is only enforced under *Priority*, *Proportional sharing*, and *PSFA*. The sum of the load proportion of all jobs corresponds to the metadata operations of all MDT servers of PFS$_A$, while the sum of the metadata rate limits corresponds to *Max*$_R$.

- *Testing scenario #1* (§6.3.2.1): all jobs follow the same workload but are assigned with different I/O priorities.
- *Testing scenario #2* (§6.3.2.2): jobs have different load proportions – for instance, *Job*$_1$ submits

15% of the overall metadata load, while *Job*$_4$ submits 45% – and rate limits are assigned proportionally to each job's load (*i.e.,* jobs with lower metadata load are assigned with lower priority).

- *Testing scenario #3* (§6.3.2.3): jobs follow the same load proportions as *testing scenario #2*, but the rate limits of *Job*$_1$ and *Job*$_4$ are switched (*i.e.,* the job with lower metadata load is assigned with higher priority, and vice-versa).

6.3.2.1 Testing Scenario #1

Figure 6.7 depicts, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #1*. Experiments include seven phases (①–⑦), each marking when a given job enters or leaves the system.

Baseline. Experiments were executed over 45 minutes. Each job executes over 36 minutes and leaves the system in the same order as it entered. Throughout the entire execution, we observe that the workload is extremely volatile and bursty, with peaks that reach 600 kops/s. When all jobs are executing, there are several periods where the file system continuously serves requests between 225 kops/s and 300 kops/s. Moreover, note that the performance curve (and maximum rate) observed in this workload does not match that in Figure 6.1, even though the logs used for replaying the metadata operations are the same. This is because jobs start their execution at different times, being separated by a 3-minutes gap.

Uniform. Experiments were executed over 45 minutes. Throughout the entire execution, whenever a new job is added, it is provisioned with its assigned rate, namely 55 kops/s. PADLL ensures that the throughput of all jobs is sustained and eliminates existing I/O burstiness. While rate limited, we observe that all jobs finish in the same time as their corresponding version in the *Baseline* setup. While this setup is useful to equally distribute metadata rate across jobs, it has three main limitations: first, it does not allow jobs to execute with different priorities; second, given that there are several periods where there is leftover metadata rate, jobs may be rate limited more aggressively than needed (for instance, in intervals ① to ③ and ⑤ to ⑦); finally, jobs can experience over-provisioning if they submit metadata operations at a rate lower that the defined limit, leading to waste of system resources.

Priority. Experiments were executed over 56 minutes. Similarly to the *Uniform* setup, 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 in *Job*$_1$, where its execution takes 20 minutes longer than in the previous setups. Moreover, while this setup enables defining metadata rate priorities, it still suffers from the other two limitations discussed in the *Uniform* setup.

Proportional sharing. Experiments were executed over 45 minutes. Whenever a new job enters (①–③) or leaves the system (⑤–⑦), it is assigned with its proportional metadata share. When all jobs are running (④), they are assigned with their demanded rate. During this phase, jobs are served with the same limits as in *Priority*. Compared to *Baseline*, this setup eliminates I/O burstiness and provides sustained metadata performance. Compared to *Uniform* and *Priority*, it ensures different priority rates while sharing leftover

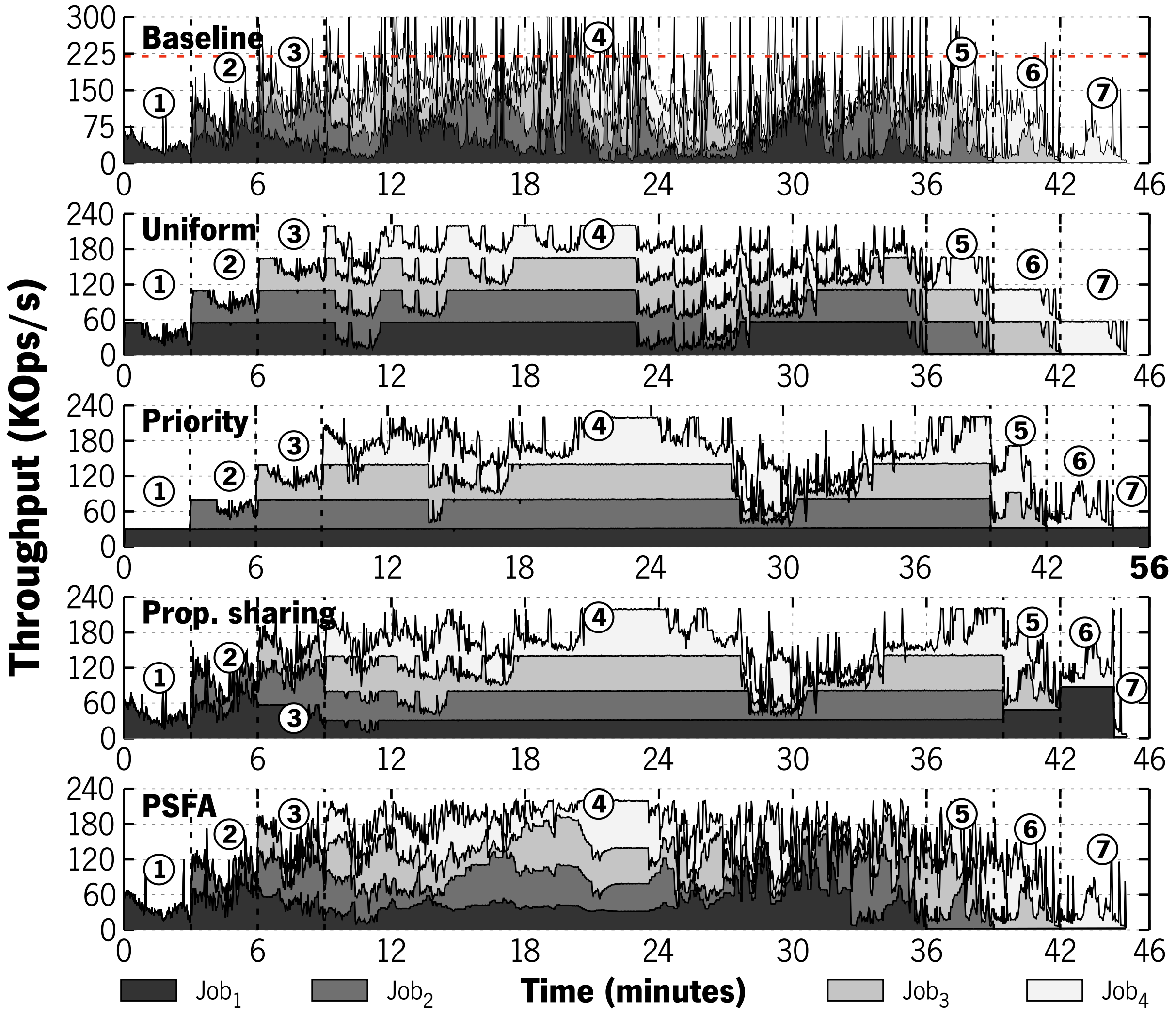


Figure 6.7: Per-job metadata control over Baseline, Uniform, Priority, Proportional Sharing, and PSFA setups under testing scenario #1.

metadata rate whenever it is available. However, given that *Job*$_1$ is aggressively rate limited during ④, its execution takes 8 minutes longer than the corresponding unthrottled version. Furthermore, because the workload is volatile and bursty, we observe several periods that demonstrate the *false resource allocation* problem (discussed in §6.2.2), being particularly noticeable in the 12–20 minutes and 24–38 minutes intervals. In these periods, one or more jobs are over-provisioned, and the exceeding resources could be used to improve the performance of the remainder jobs.

PSFA. Experiments were executed over 45 minutes. All jobs complete their execution in the same time as their corresponding unthrottled versions (*Baseline*). As observed throughout the overall execution, the *PSFA* algorithm continuously adjusts the limit of each job based on the actual rate it is using, removing the *false resource allocation* experienced in the *Proportional sharing* setup. Specifically, in the 12–20 minutes period, *PSFA* assigns unused metadata rate to *Job*$_1$, *Job*$_2$, and *Job*$_3$, temporarily having access to more rate than their *demand*. The same is observed for *Job*$_1$ and *Job*$_2$ in the 24–38 minutes period.

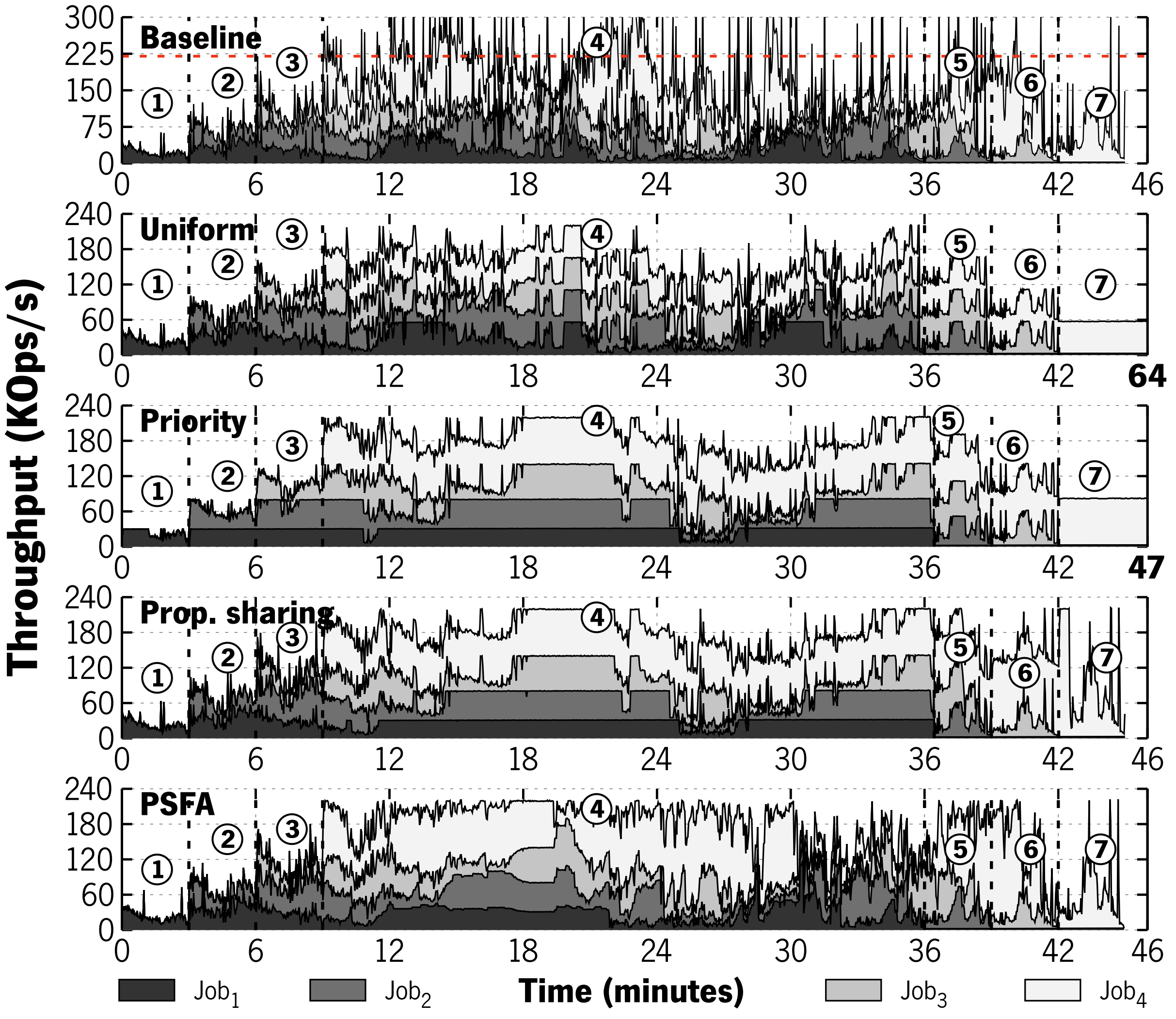


Figure 6.8: Per-job metadata control over Baseline, Uniform, Priority, Proportional Sharing, and PSFA setups under testing scenario #2.

As such, *PSFA* enables maximizing the use of available resources to accelerate the performance of more resource-hungry jobs, without degrading the performance of over-provisioned ones.

6.3.2.2 Testing Scenario #2

Figure 6.8 demonstrates, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #2*.

Baseline. Experiments executed over 45 minutes. Given that the overall metadata load is the same across all testing scenarios, we observe similar volatility and burstiness as in *testing scenario #1 Baseline* setup. The key difference is that now *Job*$_4$ generates a major part of the metadata load being noticeable throughout its entire execution, while *Job*$_1$ demonstrates significantly lower load.

Uniform. Experiments executed over 64 minutes. Similarly to *testing scenario #1*, throughout the entire execution, whenever new job is added, it is provisioned with its static rate, namely 55 kops/s. However, as

Job$_4$ now generates 45% of the overall metadata load, PADLL aggressively rate limits it, resulting in a longer execution period (namely, it takes 19 minutes longer to complete). On the other hand, *Job*$_1$ experiences over-provisioning for most of its execution, given that it only generates 15% of the overall metadata load. This demonstrates that a static rate limit policy is unsuited when jobs have different workload proportions.

Priority. Experiments executed over 47 minutes. Contrarily to *testing scenario #1*, due to the decreased metadata load, *Job*$_1$ is now able to finish in the same time as in *Baseline* setup. On the other hand, *Job*$_4$ takes 2 minutes longer to complete its execution. Specifically, due to its large metadata load, PADLL aggressively rate limits *Job*$_4$ throughout the overall execution period, resulting in a large backlog of metadata operations to be performed, as observed in ⑦. In addition, this also occurs because the specified policy does not enable jobs to leverage from leftover metadata rate available in the system.

Proportional sharing. Experiments executed over 45 minutes. Contrarily to *Uniform* and *Priority*, each job ius able to complete its execution in 36 minutes, since Algorithm 5.1 distributes leftover metadata rate whenever it is available. However, similarly to the observations made in *testing scenario #1*, *Proportion sharing* experiences periods under *false resource allocation*, being especially noticeable in the 12–18 minutes, 23–35 minutes, and 36–42 minutes intervals.

PSFA. Experiments executed over 45 minutes. Again, *PSFA* maximizes the use of metadata resources by reallocating unused rate from over-provisioned jobs. For instance, during the 23–30 minutes period, Job$_4$ improves its performance by leveraging from unused rate of the other jobs. Interestingly, during the 31–36 minutes interval, *PSFA* demonstrates lower usage of resources compared to *Priority* and *Proportional sharing*. This occurs because up to the 31-minutes mark the algorithm allocated enough rate to active jobs that allowed them to conduct any accumulated backlog of metadata operations. Thus, after that mark (and up to 36 minutes), all jobs flow with a rate closer to that observed in *Baseline*.

6.3.2.3 Testing Scenario #3

Figure 6.9 depicts, for each setup, the metadata rate of all jobs at 1-second intervals under *testing scenario #3*. Experiments conducted for *Baseline* and *Uniform* setups are the same as in *testing scenario #2*, as both metadata load and rate limits remain unchanged. We draw identical observations.

Priority. Experiments executed over 77 minutes. In this scenario, *Job*$_4$ submits 45% of the overall metadata load while being assigned with the lowest priority. As a result, the job takes 32 minutes longer to complete its execution. Of particular interest, during ⑦, PADLL aggressively rate limits metadata operations, being submitted at a constant rate of 30 kops/s, not leveraging from the remainder 190 kops/s available in the system.

Proportional sharing. Experiments executed over 49 minutes. Since *Job*$_4$ is the last to enter the system (④), it only leverages from leftover metadata rate when the other jobs complete their execution (⑤–⑦). During ⑦, due to accumulated backlog, the job submits metadata operations at a constant rate of *Max*$_R$ (over 7 minutes), achieving better performance than *Priority* (being 28 minutes faster), but still requiring

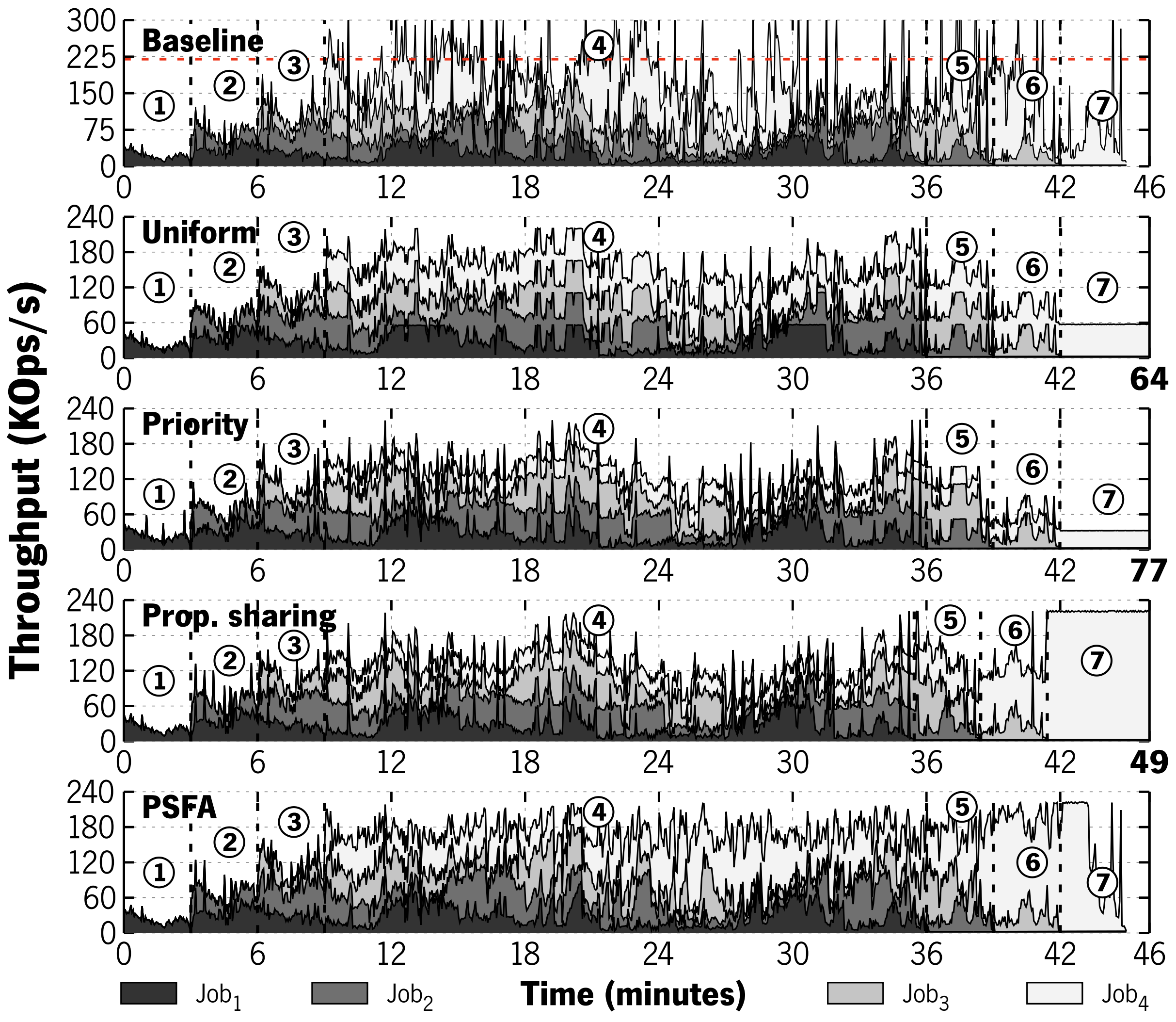


Figure 6.9: Per-job metadata control over Baseline, Uniform, Priority, Proportional Sharing, and PSFA setups under testing scenario #3.

an additional 4 minutes to complete when compared to *Baseline*. Furthermore, given that Job_1 only performs 15% of the overall metadata load but is assigned with the highest priority, it is over-provisioned for most of its execution.

PSFA. Experiments executed over 45 minutes. Since *PSFA* prevents *false resource sharing*, during the 12–42 minutes interval a large share of unused metadata rate is assigned to Job_4. As a result, in ⑦, Job_4 executes all accumulated backlog under 1.5 minutes. Note that *PSFA* is able to reassign unused resources without compromising other policies; for instance, Job_1 demonstrates the same performance curve as in setups with more strict policies, namely *Priority* and *Proportional sharing*. Finally, *PSFA* is able to execute jobs faster than *Uniform*, *Priority*, and *Proportion sharing*, while also eliminating existing burstiness in the system, bounding the overall metadata rate to Max_R.

6.4 Related Work

This section discusses and compares existing work on metadata QoS control with PADLL.

HPC storage QoS. Many works are designed to mitigate I/O contention in HPC storage stacks, including GIFT [171], CALCioM [62], IOrchestrator [244], UShape [245], and *Gainaru et al.* [77]. These however, ignore the impact that metadata workflows have over the overall system performance. PADLL is able to control the rate of both data and metadata workflows. Other systems are directly implemented within core layers of the HPC I/O stack, including the PFS [95, 107, 176, 244, 245], scheduler [77], and I/O libraries [40, 62]. 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 (through `LD_PRELOAD`) and rate limits POSIX requests at compute nodes, being transparent and portable to both POSIX-compliant file systems and applications [96]. However, because it does not provide global visibility over the system, OOOPS can only enforce static policies such as *Static* and *Priority*, discussed in §6.3.2. On the other hand, PADLL can enforce dynamic and cluster-wide policies that require system-wide visibility.

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 [128, 155, 205, 211]. Others, such as Retro and Crystal, implement resource management policies over distributed storage systems [84, 143, 222], but are directly implemented within the storage system itself, offering limited maintainability and portability. In particular, SIREN also enforces QoS policies over HPC storage systems, but is directly implemented within I/O forwarding and OSS nodes of the OrangeFS file system [39, 107]. PADLL is a bare-metal solution that actuates at the compute node level, and transparently intercepts and enforces POSIX requests, both data and metadata, before being submitted to the PFS. This makes it applicable over different applications and compatible with POSIX-compliant storage systems.

I/O optimizations. Many works propose I/O optimizations to reduce the amount of operations submitted to the PFS by resorting to storage tiering [52, 113], data reduction techniques [152, 172], and optimized data formats [75, 137]. While these can help protect the PFS to some extent, they still expose it to I/O burstiness and metadata-aggressive jobs, since metadata workflows are not rate limited. Nevertheless, these can be combined with PADLL to further enhance metadata workflows in HPC clusters.

6.5 Summary and Discussion

In this chapter, we study the impact in performance that metadata operations impose over large-scale HPC storage systems. We analyze traces from a production Lustre file system of the ABCI supercomputer, hosted by AIST, and reveal new insights about metadata operations at scale. First, we observe that modern I/O workloads are volatile and bursty, and generate massive amounts of metadata operations. Then, we notice that the amount of metadata operations far exceed the `read` and `write` bandwidth of the PFS. Finally, we observe that the most predominant metadata operations entail high costs to the PFS due to namespace housekeeping and locking.

Based on these insights, we show the design, implementation, and evaluation of PADLL, an application and file system agnostic storage middleware that enables enforcing QoS policies over metadata workflows in HPC clusters. The data plane, built with PAIO, 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. The control plane follows a hierarchical distribution, made of *global* and *local controllers*, and acts as a global coordinator that continuously monitors and manages all running jobs by adjusting the I/O rate of each data plane stage. Further, we introduce a new max-min fair share algorithm (PSFA) that prevents *false resource allocation* under bursty workloads. With PADLL, system administrators can proactively and holistically control the rate of I/O workflows of all running jobs, and thus, prevent metadata-aggressive ones from harming the PFS, as well as other jobs in execution.

Our experiments demonstrate that PADLL can (1) effectively control the rate of I/O workflows at different granularities; (2) prevent I/O burstiness and control metadata-aggressive workloads through static and dynamic policies; (3) coordinate the rate of multiple concurrent jobs holistically; and (4) when configured with PSFA, it maximizes the use of metadata resource to accelerate the performance of resource-hungry jobs, without degrading the performance of over-provisioned ones. These results allow us to conclude that it is possible to *build transparent storage data plane stages, distributed over the I/O infrastructure, that provide coordinated control of shared and performance-critical I/O resources*.

Furthermore, while we demonstrate how PADLL can be used to ensure per-job QoS control, it can be used in other scenarios. For example, AIST reported that PFS$_A$'s MDTs experience high load imbalance due to the scheduling policy for assigning MDT servers upon file creation. This not only causes clients to experience different levels of performance, but can also lead to overall performance degradation of the metadata service. Thus, PADLL could be used to prevent MDT servers from overloading.

7 Conclusion

Modern infrastructures feature long and complex data storage paths made of several I/O layers, including operating systems, hypervisors, file systems, databases, and device drivers. For each of these layers, good performance often means implementing multiple I/O optimizations, such as I/O scheduling, caching, and replication. These optimizations however, are implemented in a sub-optimal manner, as these are tightly coupled to the system implementation, and can interfere with each other due to lack of global context. This thesis addresses these challenges through a novel SDS system that enables system designers to implement complex I/O optimizations that are simultaneously (1) decoupled from the targeted system, (2) perform coordinated control decisions over I/O resources, and (3) are programmable and adaptable, to ensure the I/O requirements and storage objectives imposed by the targeted layer are met.

In recent years, the research on SDS increased at an accelerated pace, leading to a broad spectrum of proposals to address the shortcomings of conventional storage infrastructures. Despite this momentum, many aspects of the paradigm are still unclear, undefined, and unexplored. This challenge motivated the work described in §2, where we surveyed current SDS systems, explaining and clarifying fundamental aspects of the field. Specifically, we provided background concepts on SDS and outlined the distinctive characteristics of an SDS-enabled infrastructure. Then, we distilled a number of key design features for each plane of functionality, regarding their internal organization and distribution. Further, we proposed a taxonomy and classification of existing systems to organize the manifold approaches according to their storage infrastructure (*i.e.,* cloud, HPC, and application-specific), control strategy (*e.g.,* feedback control, performance modeling), and enforcement strategy (*e.g.,* scheduling, priority queues, logic injection).

Leveraging on the insights of our survey, as regards to the SDS data plane, we noticed that existing systems experience the same limitations as traditionally implemented I/O optimizations. Specifically, current SDS systems are targeted for specific I/O layers, and their designs are tightly coupled and driven by the architecture and specificities of the software stacks that they are applied to. As the core contribution of this thesis, we addressed these limitations in §3 with PAIO, a SDS data plane framework that enables building user-level, portable, and generally applicable storage optimizations. Contrary to existing solutions, PAIO implements optimizations *outside* the targeted systems as data plane stages, by intercepting and handling the I/O performed by these. These optimizations are then managed by a logically centralized controller that has the global context necessary to ensure holistic I/O control and performance. Building PAIO required addressing multiple challenges that are not supported by current solutions. To perform complex I/O optimizations outside applications, PAIO combines ideas from *context propagation*, enabling application-level information to be propagated to stages with minor code changes. Further, PAIO required designing new abstractions that allow differentiating and mediating I/O requests between user-space I/O layers, including context objects, channels, enforcement objects, and rules. These abstractions promote the implementation and portability of a variety of storage optimizations.

To demonstrate the performance and effectiveness of I/O optimizations built with PAIO, we developed three data plane stages, all driven by real use cases that exist in today's production clusters. First, we showed how to achieve tail latency control in LSM-based KVSs. Despite the advantages and wide adoption of LSM KVSs, such as RocksDB [183], LevelDB [80], and PebblesDB [178], a common problem of these systems is the interference between foreground and background workflows, generating high tail latency spikes for clients. SILK addresses this problem by proposing an I/O scheduler that controls the interference between these tasks [17]. However, it follows an intrusive approach, and applying its I/O scheduler over RocksDB required changing several core modules. As such, we addressed these problems in §4, with a data plane stage built with PAIO that implements SILK's design principles while following an SDS approach — the *data plane stage provides the I/O mechanisms for prioritizing and rate limiting background flows*, while the *control plane re-implements SILK's I/O scheduling algorithm* to orchestrate the stage. By propagating application-level information (specifically, the context at which a given POSIX operation is created such as flushes and compactions), our PAIO stage outperformed RocksDB up to 4× in 99^{th} percentile latency, under different workloads and testing scenarios, and enabled similar control and performance as system-specific optimization that required profound refactoring to the original codebase.

We then demonstrate how to ensure per-application bandwidth guarantees under a shared storage environment. The ABCI supercomputer enables a cloud-like resource allocation model, where users can reserve a full compute node or a fraction of it to execute their jobs, by having exclusive access to compute (namely, CPU cores and GPU) resources, memory, and storage quota [4]. While these resources are effectively isolated using Linux `cgroups`, the same is not ensured for local disk bandwidth. Under this scenario, jobs end up either being executed without bandwidth restrictions, which compete for shared resources and create I/O interference, or with static bandwidth limits, being unable to reserve different I/O priorities. We addressed these problems in §5 with a data plane stage built with PAIO that dynamically rate limits I/O workflows at each job, while the control plane implements a proportional sharing algorithm to ensure that all jobs meet their policies. Our approach presents two key differences when compared with other solutions: first, the data plane stage transparently intercepts POSIX calls using `LD_PRELOAD`, thus not requiring any changes to the original codebase of the targeted job's application; second, due to the global visibility of the control plane, we implemented a max-min fair share control algorithm that distributes leftover bandwidth that is not in use among active jobs in the system. Experiments conducted with TensorFlow jobs demonstrated that contrarily to the current setup used on ABCI, all PAIO-enabled TensorFlow instances were provisioned with their bandwidth goals. Further, whenever leftover bandwidth was available, PAIO distributed it across all active instances, decreasing their overall execution when compared to a setup that enforced static bandwidth limits.

Finally, we showed how to ensure QoS control of metadata workflows in HPC storage systems. In recent years, HPC workloads have become data-intensive, read-dominated, and generate massive bursts of metadata operations. While these workloads demand scalable, high throughput, and low latency storage, most supercomputers rely on Lustre-like PFSs, which provide a centralized metadata management service. Under this scenario, having multiple concurrent jobs competing for shared storage resources can lead to severe I/O contention and performance degradation. We addressed this problem in §6 with PADLL, an application and file system agnostic storage middleware built with PAIO. 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 differs from existing solutions in two main axis. First, the data plane actuates at the compute node level and uses `LD_PRELOAD` to mediate the I/O requests between a given application and the shared file system. This way, PADLL does not require changing any core layers of the HPC I/O stack, being agnostic of the applications it is controlling as well the file system to which the requests are submitted to. Second, besides following a hierarchical distribution and having global visibility over resources, the control plane introduces a new proportional sharing algorithm (PSFA) that continuously readjusts reservations of metadata operations to prevent over-provisioning/under-provisioning across all active jobs. Experiments demonstrated that PADLL can effectively control metadata-aggressive workloads and prevent I/O burstiness, can holistically coordinate the metadata rate of multiple jobs in the system, and when configured with PSFA, it can maximize the use of metadata resource to accelerate the performance of resource-hungry jobs.

With the contributions presented in this thesis, we demonstrated that it is possible to build complex I/O optimizations as user-level, general applicable storage data plane stages, that are decoupled from the targeted system and have global system visibility to perform holistic control decisions over I/O resources.

7.1 Future Work

Building on the contributions presented in this thesis, our work opens immediate research paths that can be pursued. There is a large scope of I/O optimizations that would benefit from being (re)implemented with PAIO, including multi-layer caches with global visibility, workload-aware storage tiering, load balancing of metadata operations in PFSs, and data reduction and privacy for user-level storage stacks. To achieve this, it would be interesting to expand the enforcement objects supported by PAIO, including caches, encryption and compression schemes, data placement mechanisms, and more. With these building blocks, system designers would only require to specify their policies to programme and adapt the data plane stage to meet the requirements of the targeted storage scenario (*i.e.,* workload, I/O layer, infrastructure).

Modern I/O stacks already include several frameworks that enable users to build storage layers fine-tuned for their application's requirements, using either more traditional frameworks like FUSE and *Network*

Block Device, or more recent ones such as SPDK and PMDK, which enable building kernel-bypass storage stacks. As such, to achieve wider applicability and provide finer control over I/O requests, it would be interesting to integrate PAIO as a new abstraction on top of these frameworks. For example, PAIO could be integrated as a new logical block device for the SPDK framework or be provided as a new PMDK-enabled library (similar to `libpmemobj`, `libpmemblk`, and `libpmemkv` [173]).

Large-scale I/O infrastructures, such as those demonstrated in §6, are made of hundreds to thousands of compute nodes. To ensure global control of these resources, it may require employing data plane stages at a similar or even larger scale. While we have presented a preliminary version of a hierarchical control plane, it is important to further explore its scalability and dependability. Given the amount of control points (stages) that will emerge when applied at a large scale, it will be fundamental to ensure scalable control. For instance, an immediate bottleneck that appears is the synchronization between controllers to ensure a global and correct state of the infrastructure, as well as the execution of centralized control algorithms. As such, it would be interesting to explore not only how to physically scale the control plane, but also how to delegate control power between different types of controllers. Furthermore, to ensure that the control plane has high-availability and provides a sustained service, it is important to explore techniques that improve the dependability of the control plane, such as primary-backup and state machine replication, while accounting with the performance penalty that it may impose in the overall system.

Finally, regarding to other research challenges of the SDS paradigm, in §2.6 we discuss in detail possible future directions of the field, grouped by storage infrastructure, planes of functionality, and more.

Bibliography

[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 Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283. url: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[2] Ian Ackerman and Saurabh Kataria. *Homepage feed multi-task learning using TensorFlow*. https://engineering.linkedin.com/blog/2021/homepage-feed-multi-task-learning-using-tensorflow.

[3] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. "ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys". In: *IEEE Transactions on Computers* 65.3 (2016), pp. 902–915. doi: 10.1109/TC.2015.2435779.

[4] *AI Bridging Cloud Infrastructure*. https://abci.ai/.

[5] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Beyond Storage APIs: Provable Semantics for Storage Stacks". In: *15th Workshop on Hot Topics in Operating Systems*. USENIX Association, 2015. url: https://www.usenix.org/conference/hotos15/workshop-program/presentation/alagappan.

[6] George Amvrosiadis, Ali R. Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, Irfan Ahmad, Remzi H. Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, Yue Cheng, Vijay Chidambaram, Dilma Da Silva, Angela Demke-Brown, Peter Desnoyers, Jason Flinn, Xubin He, Song Jiang, Geoff Kuenning, Min Li, Carlos Maltzahn, Ethan L. Miller, Kathryn Mohror, Raju Rangaswami, Narasimha Reddy, David Rosenthal, Ali Saman Tosun, Nisha Talagala, Peter Varman, Sudharshan Vazhkudai, Avani Waldani, Xiaodong Zhang, Yiying Zhang, and Mai Zheng. *Data Storage Research Vision 2025:*

Report on NSF Visioning Workshop Held May 30–June 1, 2018. Tech. rep. 2018. doi: 10.5555/3316807.

[7] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. "End-to-end Performance Isolation Through Virtual Datacenters". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 233–248. url: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/angel.

[8] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. "MOS: Workload-aware Elasticity for Cloud Object Stores". In: *25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 177–188. doi: 10.1145/2907294.2907304.

[9] Austin Appleby. *appleby/smhasher: SMHasher test suite for MurmurHash family of hash functions.* https://github.com/aappleby/smhasher. 2010.

[10] *Architecting a High Performance Storage System*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/architecting-lustre-storage-white-paper.pdf. Jan. 2014.

[11] Joe Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, Inc., 2014.

[12] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: *Computer Networks* 54.15 (2010), pp. 2787–2805. doi: 10.1016/j.comnet.2010.05.010.

[13] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. doi: 10.1109/TDSC.2004.2.

[14] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. "Persistent Memory: A Survey of Programming Support and Implementations". In: *ACM Computing Surveys* 54.7 (July 2021). doi: 10.1145/3465402.

[15] Oana Balmau. *theoanab/SILK-USENIXATC2019: Prototype of the SILK key-value store*. https://github.com/theoanab/SILK-USENIXATC2019. 2019.

[16] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores". In: *2017 USENIX Annual Technical Conference*. USENIX Association, 2017, pp. 363–375. url: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau.

[17] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. “SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores”. In: *2019 USENIX Annual Technical Conference*. USENIX Association, 2019, pp. 753–766. url: https://www.usenix.org/conference/atc19/presentation/balmau.

[18] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. “FloDB: Unlocking Memory in Persistent Key-Value Stores”. In: *Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 80–94. doi: 10.1145/3064176.3064193.

[19] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. “Distributed SDN Control: Survey, Taxonomy, and Challenges”. In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 333–354. doi: 10.1109/COMST.2017.2782482.

[20] Pete Beckman, Jack Dongarra, Nicola Ferrier, Geoffrey Fox, Terry Moore, Dan Reed, and Micah Beck. “Harnessing the computing continuum for programming our world”. In: *Fog Computing: Theory and Practice* (2020), pp. 215–230. doi: 10.1002/9781119551713.ch7.

[21] Nalini M. Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. “PADS: A Policy Architecture for Distributed Storage Systems”. In: *6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2009, pp. 59–73. url: https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/belaramani/belaramani_html/.

[22] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. “IX: A Protected Dataplane Operating System for High Throughput and Low Latency”. In: *11th USENIX Symposium on Operating System Design and Implementation*. USENIX Association. 2014, pp. 49–65. url: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[23] Samaresh Bera, Sudip Misra, and Athanasios V. Vasilakos. “Software-Defined Networking for Internet of Things: A Survey”. In: *IEEE Internet of Things Journal* 4.6 (2017), pp. 1994–2008. doi: 10.1109/JIOT.2017.2746186.

[24] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O’Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. “ONOS: Towards an Open, Distributed SDN OS”. In: *3rd Workshop on Hot Topics in Software Defined Networking*. ACM, 2014, pp. 1–6. doi: 10.1145/2620728.2620744.

[25] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. “The CacheLib Caching Engine: Design and Experiences at Scale”. In: *14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2020, pp. 753–768. url: https://www.usenix.org/conference/osdi20/presentation/berg.

[26] Dimitri P. Bertsekas and Robert G. Gallager. *Data Networks*. Prentice Hall, 1992.

[27] Jean-Pascal Billaud and Ajay Gulati. "hClock: Hierarchical QoS for Packet Scheduling in a Hypervisor". In: *8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 309–322. doi: 10.1145/2465351.2465382.

[28] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs". In: *2021 USENIX Annual Technical Conference*. USENIX Association, 2021, pp. 689–703. url: https://www.usenix.org/conference/atc21/presentation/bjorling.

[29] *BLKIO: Cgroup's Block I/O Controller*. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[30] Francieli Z. Boito, Eduardo C. Inacio, Jean L. Bez, Philippe O.A. Navaux, Mario A.R. Dantas, and Yves Denneulin. "A Checkpoint of Research on Parallel I/O for High-Performance Computing". In: *ACM Computing Surveys* 51.2 (2018), 23:1–23:35. doi: 10.1145/3152891.

[31] Fábio Botelho, Alysson Bessani, Fernando M.V. Ramos, and Paulo Ferreira. "On the Design of Practical Fault-Tolerant SDN Controllers". In: *2014 3rd European Workshop on Software Defined Networks*. IEEE, 2014, pp. 73–78. doi: 10.1109/EWSDN.2014.25.

[32] Fábio Botelho, Tulio A. Ribeiro, Paulo Ferreira, Fernando M.V. Ramos, and Alysson Bessani. "Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane". In: *2016 12th European Dependable Computing Conference*. IEEE, 2016, pp. 169–180. doi: 10.1109/EDCC.2016.12.

[33] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Vol. 2050. Springer Science & Business Media, 2001. doi: 10.1007/3-540-45318-0.

[34] Peter Braam. *The Lustre Storage Architecture*. 2019. doi: 10.48550/ARXIV.1903.01955.

[35] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. 1984.

[36] *Build Ultra High-Performance Storage Applications with the Storage Performance Development Kit*. https://spdk.io/.

[37] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Du. "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook". In: *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 209–223. url: https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[38] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., 2013. isbn: 1617290858.

[39] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. "PVFS: A Parallel File System for Linux Clusters". In: *4th Annual Linux Showcase & Conference*. USENIX Association, 2000. url: https://www.usenix.org/conference/als-2000/pvfs-parallel-file-system-linux-clusters.

[40] Jesus Carretero, Emmanuel Jeannot, Guillaume Pallez, David E. Singh, and Nicolas Vidal. "Mapping and Scheduling HPC Applications for Optimizing I/O". In: *34th ACM International Conference on Supercomputing*. ACM, 2020. doi: 10.1145/3392717.3392764.

[41] *CFQ: Complete Fairness Queueing*. https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[42] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. "Realtime Data Processing at Facebook". In: *2016 International Conference on Management of Data*. ACM, 2016, pp. 1087–1098. doi: 10.1145/2882903.2904441.

[43] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. "SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage". In: *19th USENIX Conference on File and Storage Technologies*. USENIX Association, 2021, pp. 17–32. url: https://www.usenix.org/conference/fast21/presentation/chen-hao.

[44] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. 2015. doi: 10.48550/ARXIV.1512.01274.

[45] Yu Chen, Wei Tong, Dan Feng, and Zike Wang. "Mass: Workload-Aware Storage Policy for OpenStack Swift". In: *49th International Conference on Parallel Processing*. ACM, 2020, 78:1–78:11. doi: 10.1145/3404397.3404427.

[46] Steven WD Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. "Characterizing Deep-Learning I/O workloads in TensorFlow". In: *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. IEEE, 2018, pp. 54–63. doi: 10.1109/PDSW-DISCS.2018.00011.

[47] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. "I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning". In: *48th International Conference on Parallel Processing*. ACM, 2019. doi: 10.1145/3337821.3337902.

[48] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. "SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores". In: *2020 USENIX Annual Technical Conference*. USENIX Association, 2020, pp. 49–63. url: https://www.usenix.org/conference/atc20/presentation/conway.

[49] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *Cryptology ePrint Archive* (2016). url: https://eprint.iacr.org/2016/086.

[50] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. “Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory”. In: *12th USENIX Conference on File and Storage Technologies*. USENIX Association, 2014, pp. 17–31. url: https://www.usenix.org/conference/fast14/technical-sessions/presentation/cully.

[51] Marco Dantas, Diogo Leitão, Cláudia Correia, Ricardo Macedo, Weijia Xu, and João Paulo. “Monarch: Hierarchical Storage Management for Deep Learning Frameworks”. In: *2021 IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 657–663. doi: 10.1109/Cluster48925.2021.00097.

[52] Marco Dantas, Diogo Leitão, Peter Cui, Ricardo Macedo, Xinlian Liu, Weijia Xu, and João Paulo. “Accelerating Deep Learning Training Through Transparent Storage Tiering”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2022, pp. 21–30. doi: 10.1109/CCGrid54584.2022.00011.

[53] *DDNStorage/LustrePerfMon: Lustre Monitoring System*. https://github.com/DDNStorage/LustrePerfMon.

[54] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. *FMA: A Dataset For Music Analysis*. 2016. doi: 10.48550/ARXIV.1612.01840.

[55] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware Scheduling for Heterogeneous Datacenters”. In: *18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013, pp. 77–88. doi: 10.1145/2451116.2451125.

[56] Hariharan Devarajan, Huihuo Zheng, Anthony Kougkas, Xian-He Sun, and Venkatram Vishwanath. “DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications”. In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 2021, pp. 81–91. doi: 10.1109/CCGrid51090.2021.00018.

[57] *device mapper: Linux kernel framework for mapping physical block devices onto higher-level virtual block devices*. https://docs.kernel.org/admin-guide/device-mapper/index.html.

[58] Sarah M. Diesburg and An-I Andy Wang. “A Survey of Confidential Data Storage and Deletion Methods”. In: *ACM Computing Surveys* 43.1 (2010), 2:1–2:37. doi: 10.1145/1824795.1824797.

[59] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. “Towards an Elastic Distributed SDN Controller”. In: *2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, pp. 7–12. doi: 10.1145/2491185.2491193.

[60] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. "Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience". In: *19th USENIX Conference on File and Storage Technologies*. USENIX Association, 2021, pp. 33–49. url: https://www.usenix.org/conference/fast21/presentation/dong.

[61] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. "The International Exascale Software Project Roadmap". In: *The International Journal of High Performance Computing Applications* 25.1 (2011), pp. 3–60. doi: 10.1177/1094342010391989.

[62] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 155–164. doi: 10.1109/IPDPS.2014.27.

[63] Ulrich Drepper. *ELF Handling for Thread-Local Storage*. Tech. rep. 2005.

[64] *Dynamically Loaded Libraries*. https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html.

[65] Peter van Emde Boas, R. Kaas, and E. Zijlstra. "Design and Implementation of an Efficient Priority Queue". In: *Mathematical Systems Theory* 10 (1977), pp. 99–127. doi: 10.1007/BF01683268.

[66] Tânia Esteves, Ricardo Macedo, Alberto Faria, Bernardo Portela, João Paulo, José Pereira, and Danny Harnik. "TrustFS: An SGX-Enabled Stackable File System Framework". In: *2019 38th International Symposium on Reliable Distributed Systems Workshops*. IEEE, 2019, pp. 25–30. doi: 10.1109/SRDSW49218.2019.00012.

[67] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. "CAT: Content-Aware Tracing and Analysis for Distributed Systems". In: *22nd International Middleware Conference*. ACM, 2021, pp. 223–235. doi: 10.1145/3464298.3493396.

[68] *facebook/rocksdb: Benchmarking tools (db_bench)*. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[69] *facebook/rocksdb: Compaction Job*. https://github.com/facebook/rocksdb/blob/v5.17.2/db/compaction_job.cc.

[70] *facebook/rocksdb: Flush Job*. https://github.com/facebook/rocksdb/blob/v5.17.2/db/flush_job.cc.

[71] *facebook/rocksdb: Rate Limiter*. https://github.com/facebook/rocksdb/wiki/Rate-Limiter.

[72] *facebook/rocksdb: RocksDB v5.17.2*. https://github.com/facebook/rocksdb/tree/v5.17.2.

[73] Alberto Faria, Ricardo Macedo, and João Paulo. "Pods-as-Volumes: Effortlessly Integrating Storage Systems and Middleware into Kubernetes". In: *7th International Workshop on Container Technologies and Container Clouds*. ACM, 2021, pp. 1–6. doi: 10.1145/3493649.3493653.

[74] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. "BDUS: Implementing Block Devices in User Space". In: *14th ACM International Conference on Systems and Storage*. ACM, 2021, 8:1–8:11. doi: 10.1145/3456727.3463768.

[75] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. "An Overview of the HDF5 Technology Suite and Its Applications". In: *EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47. doi: 10.1145/1966895.1966900.

[76] *Frontier: ORNL's exascale supercomputer is delivering world-leading performance in 2022 and beyond*. https://www.olcf.ornl.gov/frontier/.

[77] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. "Scheduling the I/O of HPC Applications Under Congestion". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1013–1022. doi: 10.1109/IPDPS.2015.116.

[78] Anjus George, Rick Mohr, James Simmons, and Sarp Oral. *Understanding Lustre Internals Second Edition*. Sept. 2021. doi: 10.2172/1824954.

[79] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011. isbn: 978-1-449-39610-7.

[80] Sanjay Ghemawat and Jeff Dean. *google/leveldb: LevelDB, A Fast Key-Value Storage Library*. https://github.com/google/leveldb.

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

[82] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. "Scaling Concurrent Log-Structured Data Stores". In: *Tenth European Conference on Computer Systems*. ACM, 2015. doi: 10.1145/2741948.2741973.

[83] *google/tensorflow: TensorFlow*. https://github.com/tensorflow/tensorflow/tree/v2.1.0.

[84] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. "Crystal: Software-Defined Storage for Multi-Tenant Object Stores". In: *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 243–256. url: https://www.usenix.org/conference/fast17/technical-sessions/presentation/gracia-tinedo.

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

[86] Ajay Gulati, Irfan Ahmad, and Carl A Waldspurger. "PARDA: Proportional Allocation of Resources for Distributed Storage Access". In: *7th USENIX Conference on File and Storage Technologies*. USENIX Association, 2009, pp. 85–98. url: https://www.usenix.org/legacy/events/fast09/tech/full_papers/gulati/gulati_html/index.html.

[87] Ajay Gulati, Arif Merchant, and Peter Varman. "mClock: Handling Throughput Variability for Hypervisor IO Scheduling". In: *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 437–450.

[88] Ajay Gulati, Arif Merchant, and Peter J. Varman. "pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems". In: *2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 2007, pp. 13–24. doi: 10.1145/1254882.1254885.

[89] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. "Demand Based Hierarchical QoS Using Storage Resource Pools". In: *2012 USENIX Annual Technical Conference*. USENIX Association, 2012, pp. 1–13. url: https://www.usenix.org/conference/atc12/technical-sessions/presentation/gulati.

[90] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. "Performance Characterization of NVMe-over-Fabrics Storage Disaggregation". In: *ACM Transactions on Storage* 14.4 (Dec. 2018). doi: 10.1145/3239563.

[91] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. "SDNRacer: Concurrency Analysis for Software-defined Networks". In: *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 402–415. doi: 10.1145/2908080.2908124.

[92] Red Hat. *What is software-defined storage?* https://www.redhat.com/en/topics/data-storage/software-defined-storage. 2018.

[93] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004. doi: 10.1002/047166880X.

[94] Chin-Jung Hsu, Rajesh K. Panta, Moo-Ryong Ra, and Vincent W. Freeh. “Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud”. In: *2016 IEEE 35th Symposium on Reliable Distributed Systems*. IEEE. 2016, pp. 127–136. doi: 10.1109/SRDS.2016.025.

[95] Yusheng Hua, Xuanhua Shi, Hai Jin, Wei Liu, Yan Jiang, Yong Chen, and Ligang He. “Software-defined QoS for I/O in exascale computing”. In: *CCF Transactions on High Performance Computing* 1.1 (2019), pp. 49–59. doi: 10.1007/s42514-019-00005-9.

[96] Lei Huang and Si Liu. “OOOPS: An Innovative Tool for IO Workload Management on Supercomputers”. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems*. IEEE, 2020, pp. 486–493. doi: 10.1109/ICPADS51040.2020.00069.

[97] Lei Huang, Yinzhi Wang, Chun-Yaung Lu, and Si Liu. “Best Practice of IO Workload Management in Containerized Environments on Supercomputers”. In: *Practice and Experience in Advanced Research Computing*. ACM, 2021, 13:1–13:7. doi: 10.1145/3437359.3465561.

[98] Markus C. Huebscher and Julie A. McCann. “A Survey of Autonomic Computing — Degrees, Models, and Applications”. In: *ACM Computing Surveys* 40.3 (2008), 7:1–7:28. doi: 10.1145/1380584.1380585.

[99] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *2010 USENIX Annual Technical Conference*. USENIX Association, 2010.

[100] IBM. *IBM: SDS solutions overview*. https://www.ibm.com/storage/software-defined-storage. 2020.

[101] Florin Isaila, Jesus Carretero, and Rob Ross. “Clarisse: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms”. In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2016, pp. 346–355. doi: 10.1109/CCGrid.2016.24.

[102] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. “B4: Experience with a Globally-deployed Software Defined Wan”. In: *2013 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2013, pp. 3–14. doi: 10.1145/2486001.2486019.

[103] Yaser Jararweh, Mahmoud Al-Ayyoub, Elhadj Benkhelifa, Mladen Vouk, and Andy Rindos. “SDIoT: a software defined based internet of things framework”. In: *Journal of Ambient Intelligence and Humanized Computing* 6.4 (2015), pp. 453–461. doi: 10.1007/s12652-015-0290-y.

[104] Yaser Jararweh, Mahmoud Al-Ayyoub, Ala' Darabseh, Elhadj Benkhelifa, Mladen Vouk, and Andy Rindos. "Software defined cloud: Survey, system and evaluation". In: *Future Generation Computer Systems* 58 (2016), pp. 56–74. doi: 10.1016/j.future.2015.10.015.

[105] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, Xiyang Wang, Nosayba El-Sayed, Jidong Zhai, Weiguo Liu, and Wei Xue. "Automatic, Application-Aware I/O Forwarding Resource Allocation". In: *17th USENIX Conference on File and Storage Technologies*. USENIX Association, 2019, pp. 265–279. url: https://www.usenix.org/conference/fast19/presentation/ji.

[106] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "Redesigning LSMs for Nonvolatile Memory with NoveLSM". In: *2018 USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 993–1005. url: https://www.usenix.org/conference/atc18/presentation/kannan.

[107] Suman Karki, Bao Nguyen, and Xuechen Zhang. "QoS Support for Scientific Workflows Using Software-Defined Storage Resource Enclaves". In: *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2018, pp. 95–104. doi: 10.1109/IPDPS.2018.00020.

[108] Magnus Karlsson, Christos T. Karamanolis, and Xiaoyun Zhu. "Triage: Performance Differentiation for Storage Systems Using Adaptive Control". In: *ACM Transactions on Storage* 1.4 (2005), pp. 457–480. doi: 10.1145/1111609.1111612.

[109] Ian A. Kash, Greg O'Shea, and Stavros Volos. "DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale". In: *9th ACM Symposium on Cloud Computing*. ACM, 2018, pp. 374–385. doi: 10.1145/3267809.3267848.

[110] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. "Enlightening the I/O Path: A Holistic Approach for Application Performance". In: *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 345–358. url: https://www.usenix.org/conference/fast17/technical-sessions/presentation/kim-sangwook.

[111] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. "Flash Storage Disaggregation". In: *11th European Conference on Computer Systems*. ACM, 2016, 29:1–29:15. doi: 10.1145/2901318.2901337.

[112] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. "Onix: A Distributed Control Platform for Large-Scale Production Networks". In: *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010, 25:1–25:14.

[113] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System". In: *27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 219–230. doi: 10.1145/3208040.3208059.

[114] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. “Pesos: Policy Enhanced Secure Object Store”. In: *12th European Conference on Computer Systems*. ACM. 2018, 25:1–25:17. doi: 10.1145/3190508.3190518.

[115] Diego Kreutz, Fernando Ramos, Paulo Verissimo, Christian Rothenberg, Siamak Azodolmolky, and Steve Uhlig. “Software-Defined Networking: A Comprehensive Survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. doi: 10.1109/JPROC.2014.2371999.

[116] Andrew Kryczka. *RocksDB Blog: Auto-tuned Rate Limiter*. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[117] *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io/.

[118] Abhishek Vijaya Kumar and Muthian Sivathanu. “Quiver: An Informed Storage Cache for Deep Learning”. In: *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 283–296. url: https://www.usenix.org/conference/fast20/presentation/kumar.

[119] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. “The Open Images Dataset v4”. In: *International Journal of Computer Vision* 128.7 (2020), pp. 1956–1981. doi: 10.1007/s11263-020-01316-z.

[120] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40. doi: 10.1145/1773912.1773922.

[121] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. “Data Replication Strategies in Grid Environments”. In: *Fifth International Conference on Algorithms and Architectures for Parallel Processing*. IEEE, 2002, pp. 378–383. doi: 10.1109/ICAPP.2002.1173605.

[122] Leslie Lamport. “The Part-time Parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. doi: 10.1145/279227.279229.

[123] *ld.so: LD_PRELOAD*. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[124] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. doi: 10.1109/5.726791.

[125] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. “Logically Centralized?: State Distribution Trade-offs in Software Defined Networks”. In: *1st Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 1–6. doi: 10.1145/2342441.2342443.

[126] Ira N Levine, Daryle H Busch, and Harrison Shull. *Quantum Chemistry*. Vol. 6. Pearson Prentice Hall, 2009. isbn: 978-0-136-85511-8.

[127] Mu Li, David G. Andersen, Jun W. Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugune J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 583–598. url: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[128] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. “PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage”. In: *11th European Conference on Computer Systems*. ACM, 2016, 28:1–28:14. doi: 10.1145/2901318.2901330.

[129] Tong Li, Dan P. Baumberger, and Scott Hahn. “Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin”. In: *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 65–74. doi: 10.1145/1504176.1504188.

[130] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. “Differentiated Key-Value Storage Management for Balanced I/O Performance”. In: *2021 USENIX Annual Technical Conference*. USENIX Association, 2021, pp. 673–687. url: https://www.usenix.org/conference/atc21/presentation/li-yongkun.

[131] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. “Elastic Machine Learning Algorithms in Amazon SageMaker”. In: *2020 ACM SIGMOD International Conference on Management of Data*. ACM, 2020, pp. 731–737. doi: 10.1145/3318464.3386126.

[132] libfuse. *libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface*. https://github.com/libfuse/libfuse. 2001.

[133] Ke Liu, Xuechen Zhang, Kei Davis, and Song Jiang. “Synergistic coupling of SSD and hard disk for QoS-aware virtual memory”. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2013, pp. 24–33. doi: 10.1109/ISPASS.2013.6557143.

[134] Si Liu, Lei Huang, Hang Liu, Amit Ruhela, Virginia Trueheart, Susan Lindsey, and Quan Yuan. “Practice Guideline for Heavy I/O Workloads with Lustre File Systems on TACC Supercomputers”. In: *Practice and Experience in Advanced Research Computing*. ACM, 2021, 5:1–5:8. doi: 10.1145/3437359.3465570.

[135] Lennart Ljung and Torsten Söderström. *Theory and Practice of Recursive Identification*. MIT press, 1983.

[136] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. "Managing Variability in the IO Performance of Petascale Storage Systems". In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–12. doi: 10.1109/SC.2010.32.

[137] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)". In: *6th International Workshop on Challenges of Large Applications in Distributed Environments*. 2008, pp. 15–24. doi: 10.1145/1383529.1383533.

[138] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "WiscKey: Separating Keys from Values in SSD-conscious Storage". In: *14th USENIX Conference on File and Storage Technologies*. USENIX Association, 2016, pp. 133–148. url: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[139] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. "Façade: Virtual Storage Devices with Performance Guarantees". In: *2nd USENIX Conference on File and Storage Technologies*. USENIX Association, 2003, pp. 131–144. url: https://www.usenix.org/conference/fast-03/fa%C3%A7ade-virtual-storage-devices-performance-guarantees.

[140] Chen Luo and Michael J Carey. "LSM-based storage techniques: a survey". In: *The VLDB Journal* 29.1 (2020), pp. 393–418. doi: 10.1007/s00778-019-00555-y.

[141] *Lustre MDC: mdc_reint.c*. https://github.com/lustre/lustre-release/blob/master/lustre/mdc/mdc_reint.c. 2022.

[142] *Lustre Metadata Service (MDS)*. https://wiki.lustre.org/Lustre_Metadata_Service_(MDS).

[143] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. "Retro: Targeted Resource Management in Multi-tenant Distributed Systems". In: *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2015, pp. 589–603. url: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.

[144] Jonathan Mace and Rodrigo Fonseca. "Universal Context Propagation for Distributed System Instrumentation". In: *13th European Conference on Computer Systems*. ACM, 2018, 8:1–8:18. doi: 10.1145/3190508.3190526.

[145] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *ACM Transactions on Computer Systems* 35.4 (Dec. 2018). doi: 10.1145/3208104.

[146] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, and João Paulo. "The Case for Storage Optimization Decoupling in Deep Learning Frameworks". In: *2021 IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 649–656. doi: 10.1109/Cluster48925.2021.00096.

[147] Ricardo Macedo, Alberto Faria, João Paulo, and José Pereira. "A Case for Dynamically Programmable Storage Background Tasks". In: *2019 38th International Symposium on Reliable Distributed Systems Workshops*. IEEE, 2019, pp. 7–12. doi: 10.1109/SRDSW49218.2019.00009.

[148] Ricardo Macedo, Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, Richard Todd Evans, and João Paulo. "Protecting Metadata Servers From Harm Through Application-level I/O Control". In: *2022 IEEE International Conference on Cluster Computing*. IEEE, 2022, pp. 573–580. doi: 10.1109/CLUSTER51413.2022.00075.

[149] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. "A Survey and Classification of Software-Defined Storage Systems". In: *ACM Computing Surveys* 53.3 (2020). doi: 10.1145/3385896.

[150] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. "PAIO: General, Portable I/O Optimizations With Minor Application Modifications". In: *20th USENIX Conference on File and Storage Technologies*. USENIX Association, 2022, pp. 413–428. url: https://www.usenix.org/conference/fast22/presentation/macedo.

[151] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. doi: 10.1145/1355734.1355746.

[152] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. "A Study on Data Deduplication in HPC Storage Systems". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012. doi: 10.1109/SC.2012.14.

[153] Paul Menage. *Linux Control Groups*. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[154] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. "Maestro: Quality-of-Service in Large Disk Arrays". In: *8th ACM International Conference on Autonomic Computing*. ACM, 2011, pp. 245–254. doi: 10.1145/1998582.1998638.

[155] Michael Mesnier, Feng Chen, Tian Luo, and Jason Akers. "Differentiated Storage Services". In: *23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 57–70. doi: 10.1145/2043556.2043563.

[156] Microsoft. *Microsoft Windows Server: Windows Server Storage Documentation.* https://docs.microsoft.com/en-us/windows-server/storage/storage.

[157] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. "High Velocity Kernel File Systems with Bento". In: *19th USENIX Conference on File and Storage Technologies*. USENIX Association, 2021, pp. 65–79. url: https://www.usenix.org/conference/fast21/presentation/miller.

[158] Muthukumar Murugan, Krishna Kant, Ajaykrishna Raghavan, and David H.C. Du. "flexStore: A Software Defined, Energy Adaptive Distributed Storage Framework". In: *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE. 2014, pp. 81–90. doi: 10.1109/MASCOTS.2014.18.

[159] Marco A.S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L.F. Cunha, and Rajkumar Buyya. "HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges". In: *ACM Computing Surveys* 51.1 (2018), 8:1–8:29. doi: 10.1145/3150224.

[160] Francisco Neves, Nuno Machado, and José Pereira. "Falcon: A Practical Log-Based Analysis Tool for Distributed Systems". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2018, pp. 534–541. doi: 10.1109/DSN.2018.00061.

[161] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Samza: Stateful Scalable Stream Processing at LinkedIn". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1634–1645. doi: 10.14778/3137765.3137770.

[162] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. "Wiera: Towards Flexible Multi-Tiered Geo-Distributed Cloud Storage Instances". In: *25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 165–176. doi: 10.1145/2907294.2907322.

[163] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 305–319. url: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[164] OpenStack. *OpenStack Documentation: Storlets.* https://docs.openstack.org/storlets/latest/.

[165] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. "SDF: Software-defined Flash for Web-scale Internet Storage Systems". In: *19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 471–484. doi: 10.1145/2541940.2541959.

[166] Linux Man Page. *proc - Process Information Pseudo-File Sytem*. 1994. url: https://linux.die.net/man/5/proc.

[167] Linux Man Page. *dstat - versatile tool for generating system resource statistics*. url: https://linux.die.net/man/1/dstat.

[168] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. "Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store". In: *2016 USENIX Annual Technical Conference*. USENIX Association, 2016, pp. 537–550. url: https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis.

[169] 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 Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[170] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. "Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019. doi: 10.1145/3295500.3356183.

[171] Tirthak Patel, Rohan Garg, and Devesh Tiwari. "GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems". In: *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 103–119. url: https://www.usenix.org/conference/fast20/presentation/patel-gift.

[172] João Paulo and José Pereira. "A Survey and Classification of Storage Deduplication Systems". In: *ACM Computing Surveys* 47.1 (2014), 11:1–11:30. doi: 10.1145/2611778.

[173] *Persistent Memory Development Kit*. https://pmem.io/pmdk/.

[174] Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. "Arrakis: The Operating System is the Control Plane". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 1–16. url: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter.

[175] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. "SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All". In: *10th ACM International Systems and Storage Conference*. ACM, 2017, 9:1–9:12. doi: 10.1145/3078468.3078480.

[176] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. "A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, 6:1–6:12. doi: 10.1145/3126908.3126932.

[177] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B. Weissman. “Tiera: Towards Flexible Multi-tiered Cloud Storage Instances”. In: *15th International Middleware Conference*. ACM, 2014, pp. 1–12. doi: 10.1145/2663165.2663333.

[178] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. “PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees”. In: *26th ACM Symposium on Operating Systems Principles*. ACM, 2017, pp. 497–514. doi: 10.1145/3132747.3132765.

[179] Sabine Rathmayer and Michael Lenke. “A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications”. In: *11th International Parallel Processing Symposium*. IEEE Computer Society, 1997, pp. 181–186. doi: 10.1109/IPPS.1997.580882.

[180] David Reinsel, John Rydning, and John F. Gantz. “Worldwide global datasphere forecast, 2021 - 2025: The world keeps creating more data—now, what do we do with it all”. In: *IDC Corporate USA* (2021).

[181] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. “SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data”. In: *Proceedings of the VLDB Endowment* 10.13 (Sept. 2017), pp. 2037–2048. doi: 10.14778/3151106.3151108.

[182] Erik Riedel, Garth Gibson, and Christos Faloutsos. “Active Storage for Large-Scale Data Mining and Multimedia Applications”. In: *24th Conference on Very Large Databases*. Citeseer. 1998, pp. 62–73.

[183] *RocksDB: A persistent key-value store for fast storage environments.* https://rocksdb.org/.

[184] Eric W.D. Rozier, Pin Zhou, and Dwight Divine. “Building Intelligence for Software Defined Data Centers: Modeling Usage Patterns”. In: *6th International Systems and Storage Conference*. ACM. 2013, 20:1–20:10. doi: 10.1145/2485732.2485752.

[185] Zhenyuan Ruan, Tong He, and Jason Cong. “INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019, pp. 379–394. url: https://www.usenix.org/conference/atc19/presentation/ruan.

[186] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. “Imagenet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. doi: 10.1007/s11263-015-0816-y.

[187] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. “In-Band Synchronization for Distributed SDN Control Planes”. In: *SIGCOMM Computer Communication Review* 46.1 (2016), pp. 37–43. doi: 10.1145/2875951.2875957.

[188] Eric Jonas andJohann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019.

[189] Frank B. Schmuck and Roger L. Haskin. “GPFS: A Shared-Disk File System for Large Computing Clusters”. In: *1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002, pp. 231–244. url: https://www.usenix.org/legacy/events/fast02/full_papers/schmuck/schmuck_html/.

[190] Bianca Schroeder and Garth A. Gibson. “Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?” In: *5th USENIX Conference of File and Storage Technologies*. USENIX Association, 2007, pp. 1–16. url: https://www.usenix.org/legacy/events/fast07/tech/schroeder.html.

[191] William Schultz, Tess Avitabile, and Alyson Cabral. “Tunable Consistency in MongoDB”. In: *Proceedings of the VLDB Endowment* 12.12 (Aug. 2019), pp. 2071–2081. doi: 10.14778/3352063.3352125.

[192] Philip Schwan. “Lustre: Building a File System for 1000-node Clusters”. In: *Proceedings of the 2003 Linux Symposium*. Vol. 2003. 2003, pp. 380–386.

[193] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. “Willow: A User-Programmable SSD”. In: *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 67–80. url: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri.

[194] Michael Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. “Malacology: A Programmable Storage System”. In: *12th European Conference on Computer Systems*. ACM, 2017, pp. 175–190. doi: 10.1145/3064176.3064208.

[195] Michael A. Sevilla, Carlos Maltzahn, Peter Alvaro, Reza Nasirigerdeh, Bradley W. Settlemyer, Danny Perez, David Rich, and Galen M. Shipman. “Programmable Caches with a Data Management Language and Policy Engine”. In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2018, pp. 203–212. doi: 10.1109/CCGRID.2018.00035.

[196] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. “Mantle: A Programmable Metadata Load Balancer for the Ceph File System”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12. doi: 10.1145/2807591.2807607.

[197] Hongzhang Shan, Katie Antypas, and John Shalf. "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008. doi: 10.1109/SC.2008.5222721.

[198] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2018, pp. 69–87. url: https://www.usenix.org/conference/osdi18/presentation/shan.

[199] M. Shreedhar and George Varghese. "Efficient Fair Queueing Using Deficit Round Robin". In: *ACM SIGCOMM 1995 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. ACM, 1995, pp. 231–242. doi: 10.1145/217382.217453.

[200] David Shue and Michael Freedman. "From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra". In: *9th European Conference on Computer Systems*. ACM, 2014, 17:1–17:14. doi: 10.1145/2592798.2592823.

[201] David Shue, Michael Freedman, and Anees Shaikh. "Performance Isolation and Fairness for Multi-Tenant Cloud Storage". In: *10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 349–362. url: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/shue.

[202] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*. IEEE, 2010. doi: 10.1109/MSST.2010.5496972.

[203] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. "Server-side I/O Coordination for Parallel File Systems". In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, 17:1–17:11. doi: 10.1145/2063384.2063407.

[204] Dan Stanzione, John West, R. Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. "Frontera: The Evolution of Leadership Computing at the National Science Foundation". In: *Practice and Experience in Advanced Research Computing*. ACM, 2020, pp. 106–111. doi: 10.1145/3311790.3396656.

[205] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. "sRoute: Treating the Storage Stack Like a Network". In: *14th USENIX Conference on File and Storage Technologies*. USENIX Association, 2016, pp. 197–212. url: https://www.usenix.org/conference/fast16/technical-sessions/presentation/stefanovici.

[206] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. "Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers". In: *6th ACM Symposium on Cloud Computing*. ACM, 2015, pp. 174–181. doi: 10.1145/2806777.2806933.

[207] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. "Preserving I/O Prioritization in Virtualized OSes". In: *8th ACM Symposium on Cloud Computing*. ACM, 2017, pp. 269–281. doi: 10.1145/3127479.3127484.

[208] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. "Distributed Resource Management Across Process Boundaries". In: *10th ACM Symposium on Cloud Computing*. ACM, 2017, pp. 611–623. doi: 10.1145/3127479.3132020.

[209] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *2020 ACM SIGMOD International Conference on Management of Data*. ACM, 2020, pp. 1493–1509. doi: 10.1145/3318464.3386134.

[210] *TensorFlow: TFRecord and tf.train.Example*. https://www.tensorflow.org/tutorials/load_data/tfrecord.

[211] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. "IOFlow: A Software-Defined Storage Architecture". In: *24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196. doi: 10.1145/2517349.2522723.

[212] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. "Chainer: A Next-Generation Open Source Framework for Deep Learning". In: *Proceedings of Workshop on Machine Learning Systems in the 29th Annual Conference on Neural Information Processing Systems*. Vol. 5. 2015, pp. 1–6.

[213] *Top500: The List*. https://www.top500.org/.

[214] Raghav Tulshibagwale. *RocksDB at Nutanix*. https://www.nutanix.dev/2021/03/10/rocksdb-at-nutanix/.

[215] Keitaro Uehara, Yu Xiang, Yih-Farn R. Chen, Matti Hiltunen, Kaustubh Joshi, and Richard Schlichting. "SuperCell: Adaptive Software-Defined Storage for Cloud Storage Workloads". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2018, pp. 103–112. doi: 10.1109/CCGRID.2018.00025.

[216] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. “Guardat: Enforcing data policies at the storage layer”. In: *10th European Conference on Computer Systems*. ACM. 2015, 13:16–13:16. doi: 10.1145/2741948.2741958.

[217] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems”. In: *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 59–72. url: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

[218] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *4th Annual Symposium on Cloud Computing*. ACM, 2013. doi: 10.1145/2523616.2523633.

[219] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics - the finite volume method*. Addison-Wesley-Longman, 1995. isbn: 978-0-582-21884-0.

[220] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. “Argon: Performance Insulation for Shared Storage Servers”. In: *5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007, pp. 61–76. url: https://www.usenix.org/legacy/events/fast07/tech/wachs.html.

[221] Carl A. Waldspurger and William E. Weihl. “Lottery Scheduling: Flexible Proportional-Share Resource Management”. In: *1st USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 1994, pp. 1–11. url: https://www.usenix.org/legacy/publications/library/proceedings/osdi/waldspurger.html.

[222] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. “Cake: Enabling High-Level SLOs on Shared Storage Systems”. In: *3rd ACM Symposium on Cloud Computing*. ACM, 2012. doi: 10.1145/2391229.2391243.

[223] Yin Wang and Arif Merchant. “Proportional-Share Scheduling for Distributed Storage Systems”. In: *5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007, pp. 47–60. url: https://www.usenix.org/legacy/events/fast07/tech/wang.html.

[224] Sage Weil, Scott Brandt, Ethan Miller, Darrell Long, and Carlos Maltzahn. “Ceph: A scalable, high-performance distributed file system”. In: *7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 307–320. url: https://www.usenix.org/legacy/events/osdi06/tech/full_papers/weil/weil_html/.

[225] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. "JoiNS: Meeting Latency SLO with Integrated Control for Networked Storage". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE. 2018, pp. 194–200. doi: 10.1109/MASCOTS.2018.00027.

[226] Jake Wires and Andrew Warfield. "Mirador: An Active Control Plane for Datacenter Storage". In: *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 213–228. url: https://www.usenix.org/conference/fast17/technical-sessions/presentation/wires.

[227] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. "AC-Key: Adaptive Caching for LSM-based Key-Value Stores". In: *2020 USENIX Annual Technical Conference*. USENIX Association, 2020, pp. 603–615. url: https://www.usenix.org/conference/atc20/presentation/wu-fenggang.

[228] Joel C. Wu and Scott A. Brandt. "Providing Quality of Service Support in Object-Based File System". In: *24th IEEE Conference on Mass Storage Systems and Technologies*. IEEE, 2007, pp. 157–170. doi: 10.1109/MSST.2007.28.

[229] Miguel Xavier, Israel De Oliveira, Fabio Rossi, Robson Dos Passos, Kassiano Matteussi, and Cesar De Rose. "A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds". In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 253–260. doi: 10.1109/PDP.2015.67.

[230] Yiqi Xu, Dulcardo Arteaga, Ming Zhao, Yonggang Liu, Renato Figueiredo, and Seetharami Seelam. "vPFS: Bandwidth Virtualization of Parallel Storage Systems". In: *IEEE 28th Symposium on Mass Storage Systems and Technologies*. IEEE. 2012, pp. 1–12. doi: 10.1109/MSST.2012.6232370.

[231] 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. "End-to-end I/O Monitoring on a Leading Supercomputer". In: *16th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2019, pp. 379–394. url: https://www.usenix.org/conference/nsdi19/presentation/yang.

[232] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory". In: *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 169–182. url: https://www.usenix.org/conference/fast20/presentation/yang.

[233] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini Kaushik, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "Split-Level I/O Scheduling". In: *25th ACM Symposium on Operating Systems Principles*. ACM, 2015, pp. 474–489. doi: 10.1145/2815400.2815421.

[234] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. “SPDK: A Development Kit to Build High Performance Storage Applications”. In: *IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2017, pp. 154–161. doi: 10.1109/CloudCom.2017.14.

[235] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. “Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering”. In: *2017 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 432–445. doi: 10.1145/3098822.3098854.

[236] Soheil H. Yeganeh and Yashar Ganjali. “Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications”. In: *1st Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 19–24. doi: 10.1145/2342441.2342446.

[237] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. “On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems”. In: *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2016, pp. 750–759. doi: 10.1109/IPDPS.2016.50.

[238] Andy Yoo, Morris Jette, and Mark Grondona. “Slurm: Simple Linux Utility for Resource Management”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60. doi: 10.1007/10968987_3.

[239] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. “Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores”. In: *9th ACM Symposium on Cloud Computing*. ACM, 2018, pp. 162–173. doi: 10.1145/3267809.3267846.

[240] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. “The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems”. In: *28th Symposium on Operating Systems Principles*. ACM, 2021, pp. 195–211. doi: 10.1145/3477132.3483569.

[241] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. “Storage Performance Virtualization via Throughput and Latency Control”. In: *13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2005, pp. 135–142. doi: 10.1145/1168910.1168913.

[242] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. "Storage Performance Virtualization via Throughput and Latency Control". In: *ACM Transactions on Storage* 2.3 (2006), pp. 283–308. doi: 10.1145/1168910.1168913.

[243] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. "FPGA-Accelerated Compactions for LSM-based Key-Value Store". In: *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 225–237. url: https://www.usenix.org/conference/fast20/presentation/zhang-teng.

[244] Xuechen Zhang, Kei Davis, and Song Jiang. "IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination". In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11. doi: 10.1109/SC.2010.30.

[245] Xuechen Zhang, Kei Davis, and Song Jiang. "QoS Support for End Users of I/O-intensive Applications Using Shared Storage Systems". In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, 18:1–18:12. doi: 10.1145/2063384.2063408.

[246] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks". In: *17th USENIX Conference on File and Storage Technologies*. USENIX Association, 2019, pp. 207–219. url: https://www.usenix.org/conference/fast19/presentation/zheng.

[247] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. "WorkloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees". In: *8th ACM Symposium on Cloud Computing*. ACM, 2017, pp. 598–610. doi: 10.1145/3127479.3132245.

[248] Timothy Zhu, Alexey Tumanov, Michael Kozuch, Mor Harchol-Balter, and Gregory Ganger. "PriorityMeister: Tail Latency QoS for Shared Networked Storage". In: *5th ACM Symposium on Cloud Computing*. ACM, 2014, 29:1–29:14. doi: 10.1145/2670979.2671008.