New Techniques for Tracing and Designing HPC Storage Systems

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften”
am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

Marc-André Vef

geb. in Wiesbaden
Mainz, den 23. Januar 2023

1. Berichterstatter: Prof. Dr.-Ing. André Brinkmann
2. Berichterstatter: Prof. Dr. Felix Schuhknecht
Tag des Kolloquiums: 27.07.2023

D77

per aspera ad astra

Acknowledgement

First and foremost, I want to sincerely thank my advisor, Professor Dr.-Ing. André Brinkmann, for guiding and supporting me during my research studies since my master thesis. I am grateful that he introduced me to the field of storage and systems and indebted for the research opportunities and constant invaluable feedback in numerous discussions over the years. I would also like to acknowledge Professor Dr. Felix Schuhknecht as the second reviewer of this thesis, and I am thankful for his valuable comments.

[A significant section of the acknowledgements were removed from the online version of this thesis due to regulations of the Johannes Gutenberg University Mainz.]

I want to thank my friends and colleagues for their support, proofreading, constructive criticism, and wonderful discussions over various topics throughout the years. I also want to express my gratitude to JGU's data center for patiently answering my many technical questions related to HPC and for their help in setting up the necessary environments for my experiments.

Finally, I wish to express my profound gratitude to my family and my partner for providing me with unfailing support and encouragement throughout the years of my studies at the university. Without you, this work would not have been possible.

Thank you.

Abstract

In the domain of high-performance computing (HPC), large (distributed) parallel file systems form the storage backbone of HPC systems to solve complex computational problems. Such HPC systems can include hundreds of thousands of compute nodes backed by hundreds of petabytes of available storage. However, in recent years, file system access patterns have changed as increasingly more data-intensive applications have started using HPC systems to process massive amounts of experimental data. Their access patterns differ from traditional HPC workloads that mainly performed sequential I/O (input/output) operations on large files for which the parallel file systems were initially optimized. Since the file system is shared within the entire HPC system, access patterns of a single data-intensive application can interfere with other users of the HPC system, resulting in reduced I/O throughput, prolonged I/O latencies, and long waiting times. Various storage solutions were proposed to mitigate such issues, including dedicated hardware-based solutions or new software-based I/O interfaces. Nevertheless, these methods can be costly or, depending on the application, challenging to support, particularly if they involve adapting an application's I/O layer to new I/O APIs and semantics.

In this dissertation, we will discuss three topics in the realm of these new challenges, focusing on system analytics in parallel file systems, burst buffer file systems, and object store file systems. First, we will provide a detailed investigation and offer insights into the difficulties developers face when aiming to understand the behavior of the GPFS parallel file system. We will propose a new analysis framework, *FlexTrace*, alleviating various limitations and overheads of existing mechanisms. Next, we will design, implement, and evaluate two novel distributed file systems, *GekkoFS* and *DelveFS*, leveraging existing compute node-local storage devices that often remain unused. The GekkoFS distributed burst buffer file system offers a temporary and highly scalable I/O system that is optimized for the above-presented data-intensive HPC applications. GekkoFS can be started *ad hoc* in seconds, provides the standard I/O interface, and can be easily used exclusively by a single application. Therefore, challenging access patterns are handled by GekkoFS instead of the HPC system's parallel file system. By redefining file system protocols and semantics, the burst buffer file system can considerably outperform the capabilities of parallel file systems if given enough node-local storage resources while still supporting most

HPC applications. Lastly, the DelveFS event-driven semantic file system focuses on providing the standard I/O interface for data-intensive HPC applications that need to access scientific data which is only available via the custom APIs of object stores, e.g., OpenIO or Amazon's S3. DelveFS builds on new mechanisms of object stores to provide users with a new interactive way of accessing data, allowing them to define their own *views* on object store containers that sometimes contain millions of objects. By offering new techniques, DelveFS provides similar I/O throughput compared to an object store's native I/O interface while accelerating certain I/O operations that are generally challenging for object stores to perform efficiently.

German abstract

Im Bereich des Hochleistungsrechnen (engl. high-performance computing, HPC), das bei der Berechnung von komplexen numerischen Problemen zum Einsatz kommt, werden große (verteilte) parallele Dateisysteme eingesetzt, um den enormen Speicher-Bedarf von HPC-Systemen zu decken. Solche HPC-Systeme können Hunderttausende von Rechenknoten umfassen, die von Hunderten von Petabytes an Speicher gestützt werden. In den letzten Jahren hat sich der Zugriff auf parallele Dateisysteme jedoch stark verändert. Immer mehr daten-intensive Applikationen haben damit begonnen, HPC-Systeme zur Verarbeitung enormer Mengen von experimentellen Daten zu nutzen. Die entsprechenden Zugriffsmuster unterscheiden sich dabei stark von Zugriffsmustern traditioneller HPC-Applikationen, die vor allem durch sequenzielle Zugriffe auf große Dateien geprägt waren und dahingehend optimiert wurden. Da parallele Dateisysteme innerhalb des gesamten HPC-Systems gemeinsam verwendet werden, können diese neuartigen Zugriffsmuster der daten-intensiven Applikationen dazu führen, dass einzelne Anwendungen andere Nutzer des HPC-Systems negativ beeinflussen. Dies kann sich in einem verringerten Datendurchsatz und längeren Zugriffslatenzen äußern, was insgesamt zu längeren Wartezeiten beim Zugriff auf Daten im parallelen Dateisystem führen kann. Hierbei wurden bereits verschiedene Lösungen vorgeschlagen, um solche Probleme zu entschärfen. Darunter fallen sowohl dedizierte hardwarebasierte Lösungen als auch neuartige softwarebasierte Datenschnittstellen. Solche Methoden können jedoch kostspielig oder, je nach Anwendung, schwierig zu verwenden sein. Das gilt insbesondere dann, wenn Anwendungen modifiziert werden müssen, um nicht-standardisierte Datenschnittstellen oder Semantiken zu unterstützen. Dies ist aufgrund der Komplexität solcher Anwendungen oft nur unter hohem Aufwand möglich.

In dieser Dissertation werden wir drei Themen im Kontext dieser Herausforderungen erörtern und uns dabei auf die Systemanalyse in parallelen Dateisystemen, Burst-Buffer-Dateisystemen und Objektspeicher-Dateisystemen fokussieren. Zunächst werden wir Schwierigkeiten untersuchen, die auftreten, wenn das Verhalten von einem parallelen Dateisystem festgestellt werden soll. Hierbei werden wir außerdem die Perspektive der GPFS-Entwickler erläutern. Um diese Analyse zu vereinfachen, werden wir das neue *FlexTrace* Analyse-Framework beschreiben,

das verschiedene Einschränkungen und Overheads bestehender Mechanismen von GPFS entschärft. Als Nächstes werden wir die Architektur von zwei neuartigen verteilten Dateisystemen vorstellen und evaluieren: *GekkoFS* und *DelveFS*. Beide Dateisysteme nutzen dabei bereits existierende, lokale Flash-Speicher der Rechenknoten, die sonst oft ungenutzt bleiben. Bei GekkoFS handelt es sich um ein temporäres und hoch-skalierendes, verteiltes Burst-Buffer-Dateisystem, das im Hinblick auf die genannten Zugriffsmuster der HPC-Anwendungen optimiert wurde. Das Dateisystem stellt dabei die standardisierte Schnittstelle für Lese- und Schreib-Operationen zur Verfügung und kann (ohne administrative Unterstützung) exklusiv von einer einzigen Anwendung innerhalb eines Rechenjobs verwendet werden. Es kann dementsprechend *ad hoc* in wenigen Sekunden gestartet und wieder gestoppt werden. Anstelle des parallelen Dateisystems können die Zugriffsmuster somit direkt durch GekkoFS verarbeitet werden. Mit der Neudefinition von Dateisystemprotokollen und -Semantiken kann GekkoFS die Performanz von parallelen Dateisystemen weit übertreffen, sofern genügend lokale Speichermöglichkeiten verfügbar sind. Dabei kann GekkoFS die meisten HPC-Anwendungen unterstützen. Zuletzt werden wir das ereignis-gesteuerte semantische Dateisystem DelveFS vorstellen. Es stellt eine standardisierte Lese- und Schreib-Schnittstelle für HPC-Anwendungen bereit, die auf wissenschaftliche Daten zugreifen müssen, die aber nur über proprietäre Schnittstellen von Objektspeichern zugänglich sind. Als Beispiele für Objektspeicher können OpenIO oder S3 von Amazon genannt werden. Das Dateisystem nutzt dabei neuartige Mechanismen der Objektspeicher, um den Nutzern einen interaktiven Zugriff auf diese Objektspeicher zu bieten. Dies ermöglicht eine Nutzer-definierte Ansicht auf Objektspeicher-Container und erleichtert deren Handhabung erheblich, da solche Container mitunter Millionen von Objekten enthalten können. Durch neuartige Techniken bietet DelveFS einen ähnlichen Schreib- und Lese-Durchsatz, verglichen mit den proprietären Schnittstellen der Objektspeicher. Gleichzeitig beschleunigt es verschiedene Operation, die von den Objektspeichern häufig nur ineffizient durchgeführt werden können.

Contributions and funding

In this section, I will describe my own contributions to this thesis in detail, which are complete to the best of my knowledge. I personally performed the vast majority of the scientific and technical work that is presented in this thesis, and I authored the entirety of this dissertation. Nevertheless, I cannot claim sole authorship for every idea and design decision as they were subject to the usual scientific discussions with my doctoral advisor, supervisors, and external collaborators.

As listed in the following pages, this thesis will discuss various topics in the scope of distributed file systems in high-performance computing (HPC). The corresponding peer-reviewed publications [252, 253, 254, 255, 256] form the basis of this thesis, and I am the primary author of these workshop and conference papers as well as journal articles. I considerably extended these contributions by providing additional details, background information, experiments, and explanations. Further, if not otherwise clearly specified, I exclusively performed all experiments that were run in the context of these publications. I clearly marked any contributions that were partly made by me or exclusively by a third party.

I conducted the vast majority of this scientific work within the context of the ADA-FS project, for which I did not author its proposal. However, although the initial idea of using ad hoc storage systems in HPC was part of the ADA-FS project proposal, my design and implementation of the GekkoFS burst buffer file system [252, 253] significantly differ from it, drawing from insights that I achieved during my master's thesis [251]. This research has been conducted in collaboration with the Barcelona Supercomputing Center (BSC), with some experiments kindly executed on the MareNostrum 4 and NEXTGenIO supercomputers. Chapter 4 discusses GekkoFS, and its content is partly based on the conference paper "*GekkoFS – A temporary distributed file system for HPC applications*" by Vef et al. [253] and on the research articles "*GekkoFS – A temporary burst file system for HPC applications*" by Vef et al. [252] and "*Ad hoc file systems for High-Performance Computing*" (Section 4.2.5 – Interfacing ad hoc file systems) by Brinkmann et al. [32]. Moreover, Section 4.5.7, which presents GekkoFS in the context of deep learning, is partly based on the conference paper *"Streamlining distributed Deep Learning I/O with ad hoc file systems"* [218] by Schimmelpfennig and Vef et al. Note that although the author of this thesis

contributed to the work in [218], it should not be considered as a main contribution for this thesis.

I also designed and evaluated the DelveFS object store file system [254] within the ADA-FS project. The OpenIO object store used in this context was configured and set up by OpenIO at the MOGON II supercomputer at JGU. Its idea and design were not part of the project proposal as its benefits only became apparent later in the project. Chapter 5 discusses DelveFS, and its content is based on the conference paper "*DelveFS – An event-driven semantic file system for object stores*" by Vef et al. [254], which received a best paper award.

I investigated the challenges during tracing in the GPFS parallel file system and designed and implemented the FlexTrace tracing framework [255, 256] partly within the ADA-FS project. This topic was not initially part of the project proposal. I mostly worked on this topic during a research internship at IBM, who offered the initial idea and opportunity as it required access to the closed-source GPFS parallel file system. Because the corresponding publication [255] and its contents were cleared by IBM's legal department, I did not include further insights and experiments on this topic beyond textual and figure improvements, as well as additions to the background and related work of the general tracing topic. I clearly marked insights that were only possible due to private discussions with GPFS developers by using the citation [196], which are consistent with the corresponding publication [255]. Chapter 3 discusses this research direction, and its content is based on the research article "*Challenges and Solutions for Tracing Storage Systems: A Case Study with GPFS*" by Vef et al. [256]. Note that [255], and correspondingly Chapter 3, describes a research direction, and IBM does not have a commitment to integrate FlexTrace in its products.

This research was supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" and the ADA-FS project. Further, this work was partially funded by the European Union's Horizon 2020 under the "Adaptive multi-tier intelligent data manager for Exascale (ADMIRE)" project; Grant Agreement number: 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1. The vast majority of the research discussed in Chapter 3 was gratefully funded by IBM Research. Moreover, the scientific work presented in Chapters 4 and 5 was conducted using the supercomputer MOGON II and services offered by JGU. I gratefully acknowledge the computing time granted on MOGON II. Finally, this research was partially funded by JGU's Center for Data Processing under the direction of Prof. Dr.-Ing. André Brinkmann.

Publications

Most of the contributions presented in this thesis have been published and were peer-reviewed conference papers and journal articles. This thesis only discusses the contributions where I am the first author. Exceptions are clearly specified.

The following lists all publications including those I co-authored:

First author:

- **M. Vef**, V. Tarasov, D. Hildebrand, and A. Brinkmann. "Tracing of Complex Production Systems: Obstacles and Solutions". In: *Workshop on System Analytics and Characterization (SAC)* (Co-located with SIGMETRICS) (2016) [256].
- **M. Vef**, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, A. Brinkmann "GekkoFS - A Temporary Distributed File System for HPC Applications". In: *IEEE International Conference on Cluster Computing, CLUSTER* (2018) [253].
- **M. Vef**, V. Tarasov, D. Hildebrand, and A. Brinkmann. "Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale". In: *ACM Transactions on Storage 14.2* (2018) [255].
- **M. Vef**, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, A. Brinkmann. "GekkoFS – A Temporary Burst Buffer File System for HPC applications". In: *SPRINGER Journal of Computer Science Technology 35.1* (2020) [252].
- **M. Vef**, R. Steiner, R. Salkhordeh, J. Steinkamp, F. Vennetier, J. Smigielski, A. Brinkmann. "DelveFS - An Event-Driven Semantic File System for Object Stores". In: *IEEE International Conference on Cluster Computing, CLUSTER* (2020) [254].

Second author:

- S. Oeste, **M. Vef**, M. Soysal, W. E. Nagel, A. Brinkmann, A. Streit. "ADA-FS - Advanced Data Placement via Ad hoc File Systems at Extreme Scales". In: *SPRINGER Software for Exascale Computing – SPPEXA 2016–2019*. (2020) [174].

- F. Schimmelpfennig, **M. Vef**, R. Salkhordeh, A. Miranda, R. Nou, A. Brinkmann. "Streamlining distributed Deep Learning I/O with ad hoc file systems". In: *IEEE International Conference on Cluster Computing, CLUSTER* (2021) [218].

Co-author:

- T. Süß, L. Nagel, **M. Vef**, A. Brinkmann, D. Feld, T. Soddemann. "Pure Functions in C: A Small Keyword for Automatic Parallelization". In: *2017 IEEE International Conference on Cluster Computing, CLUSTER* (2017) [234].
- M. Soysal, M. Berghoff, T. Zirwes, **M. Vef**, S. Oeste, A. Brinkmann, W. E. Nagel, A. Streit. "Using On-Demand File Systems in HPC Environments". In: *17th International Conference on High Performance Computing & Simulation, HPCS* (2019) [229].
- A. Brinkmann, K. Mohror, W. Yu, P. H. Carns, T. Cortes, S. Klasky, A. Miranda, F. Pfreundt, R. B. Ross, **M. Vef**. "Ad Hoc File Systems for High-Performance Computing". In: *SPRINGER Journal of Computer Science and Technology 35.1* (2020) [32].
- T. Süß, L. Nagel, **M. Vef**, A. Brinkmann, D. Feld, T. Soddemann. "Pure Functions in C: A Small Keyword for Automatic Parallelization". In: *SPRINGER International Journal of Parallel Programming 49.1* (2021) [235].
- Y. Qian, W. Cheng, L. Zeng, **M. Vef**, O. Drokin, A. Dilger, S. Ihara, W. Zhang, Y. Wang, A. Brinkmann. "MetaWBC: POSIX-compliant metadata write-back caching for distributed file systems". In: *ACM/IEEE SC '22: The International Conference for High Performance Computing, Networking, Storage and Analysis* (2022) [197].

Abbreviations

ADMIRE	Adaptive Multi-tier Intelligent Data Manager for Exascale
AI	Artificial intelligence
API	Application programming interface
BSC	Barcelona Supercomputing Center
CLI	Command-line interface
CMA	Cross-memory attach
CPU	Central processing unit
DB	Database
DL	Deep learning
DNA	Deoxyribonucleic acid
DLM	Distributed lock manager
EXT	Extended file system
FAT32	File allocation table 32
FIDIUM	Federated Digital Infrastructures for Research on Universe and Matter
FS	File system
FUSE	File system in user space
GPFS	General parallel file system
GNU	GNU's not Unix
GPL	GNU General Public License
HA	High availability
HDD	Hard disk drive
HPC	High-performance computing
IBM	International Business Machines
I/O	Input/Output
IOPS	Input/Output operations per second
IP	Internet protocol
IPC	Inter-process communication
JGU	Johannes Gutenberg University Mainz
KV	Key-value

MDS	Metadata service/server
MDT	Metadata target
MMU	Memory management unit
Mutex	Mutual exclusion
MPI	Message Passing Interface
NAS	Network-attached storage
NFS	Network File System
NUMA	Non-uniform memory access
NSD	Network-shared disk
NVMM	Non-volatile main memory
OS	Operating system
OSS	Object storage service/server
OST	Object storage target
POSIX	Portable Operating System Interface
PTE	Page table entry
RAID	Redundant array of inexpensive disks
RAM	Random access memory
RDMA	Remote direct memory access
RPC	Remote procedure call
SAS	Serial-attached SCSI
SATA	Serial advanced technology attachment
SDK	Software development kit
SSD	Solid-state drive
SST	Static sorted table files
TLB	Translation lookaside buffer
UNIX	Uniplexed information computing system
USB	Universal Serial Bus
VFS	Virtual file system
YAML	YAML ain't markup language

Table of contents

1 Introduction

When consumer magazines publish articles about data storage, the conversation usually mainly revolves around a storage device's price and its performance, i.e., the write and read (I/O) speeds and the capacity [@31, @107, @120]. Flash-based storage technologies, such as *solid-state drives* (SSDs), have become the de-facto standard in many consumer products. Compared to traditional mechanical, magnetic disks, more commonly known as *hard disk drives* (HDDs), the superior read and write bandwidth, access latency, energy efficiency, and physical dimensions of SSDs are vital for premium laptops, phones, cameras, and other consumer devices. With manufacturers pushing for designing ever-thinner devices, *non-volatile memory express* (NVMe) SSDs are often used, offering even more write and read performance than SATA SSDs, and they reach up to an order of magnitude higher bandwidths than HDDs. Nevertheless, while SSDs, in general, outsold HDDs in unit sales by 3:2 in 2021, HDDs still outsold SSDs in terms of capacity by 4.5:1 [@224]. Therefore, with the much lower price per terabyte of HDDs compared to SSDs by up to 5 times in 2021, HDDs are still heavily used when capacity requirements are more important. This is especially the case when data is accessed sequentially, e.g., for backups, file storage, or for home *network-attached storage* (NAS) solutions.

While the type of storage device is the most driving factor behind its I/O performance, it cannot be solely used on its own to store data conveniently. For this task, *file systems* exist, which provide a software abstraction layer that organizes a user's data on a storage device. Based on the familiar idea of ordering physical documents (or *files*) into *folders* which can hold several files, file systems present a similar hierarchical structure to users which they can quickly grasp. In essence, a file in a file system represents a contextual piece of coherent data, such as a document or spreadsheet, accessed by its name. Files can then be organized into a digital folder (or a *directory*[1]) which itself can contain more directories. Therefore, although possible, users almost never use the storage device directly.

Overall, similar to how the placement and function of a car steering wheel and pedals are identical for the vast majority of automobiles, files and directories have

[1] Historically speaking, file systems at first only offered a flat namespace [104], i.e., a single directory, while hierarchical file systems were first introduced with the Multics operating system [49] and later became wide-spread with the FAT [62], MINIX [239], or HFS [33] file systems.

become an integral part of human-machine interaction for storage and, in their representation and function, are equivalent across the mainly used Windows, macOS, and Linux operating systems. As a result, consumers are accustomed to the basic functionalities of file systems due to their fundamental similarities in usage. That being said, users can still encounter incompatibilities between different file systems because of limitations, e.g., the 4 GiB maximum file limit of the FAT32 file system that comes pre-installed on most USB flash drives.

Nonetheless, file systems offer much more than organizing a user's data into files and directories. Over time, a large number of file systems were developed, each focusing on a specific use case, such as file systems for local machines [@140, 213], tapes [186], distributed environments [30, 219] and many more [6, 19, 21, 41, 54, 94, 100, 159, 211]. They have fostered many innovations and features; some have become so complex that they contain millions of lines of code [255]. Such features include increasing maximum allowed file sizes or the ability to use *symbolic* and *hard links*. More advanced features cover, e.g., *journaling* to prevent inconsistent file system states on crashes or *extents* to reduce metadata and fragmentation of data blocks [@140]. Even an entire set of I/O standards was specified (see the *Portable Operating System Interface* (POSIX) [262]) coupled with a well-defined interface of the Linux kernel for all file systems – the *virtual file system* (VFS) [113][2]. With the VFS, application developers can rely on an I/O interface supported by all Linux kernel file systems, and it further offers various caches, e.g., a cache for file metadata. However, as we will discuss later, the VFS and the POSIX I/O standards also have several drawbacks, especially when file systems operate in a distributed environment.

An example of such a distributed environment is a *supercomputer* or *compute cluster* that is used by *high-performance computing* (HPC) to solve complex computational problems. The size of HPC clusters can range from thousands of machines (or *compute nodes*) to hundreds of thousands of nodes with millions of processor cores in total [@247] that are connected via high-performance network fabrics. Because individually managing local file systems and their data on each node is highly impractical, HPC clusters also use one or several *parallel file systems* that pool together the resources of tens of thousands of storage devices. Examples of commercial and widespread parallel file systems are Lustre [30], GPFS [219], or BeeGFS [97]. Typically, these parallel file systems rely on a large number of HDDs for cost reasons. For instance, the *Frontier* supercomputer [@74] at *Oak Ridge Leadership Computing Facility* (OLCF) started user operations in 2022 and is one of the first exascale supercomputers in the world with a targeted 1.5 exaflops of peak performance. Frontier's

[2]Note that this thesis focuses on file systems for the Linux operating system.

Lustre parallel file system combines the performance and storage capacity of more than 45,000 HDDs providing more than 650 petabytes of storage capacity, and it uses 20 petabytes of NVMe SSD storage to handle file system metadata and to offer a fast storage tier.

Using HDDs for data and SSDs for metadata is common across many HPC systems and is often complemented by additional node-local SSDs for temporary storage [32, 265]. Nevertheless, while a single HDD can provide a bandwidth of more than 200 megabytes per second for sequential access, which has been the predominant use case for HPC systems in the past, newer *data-driven* applications, processing massive amounts of experimental data from various scientific fields, have different I/O behavior and larger data requirements [265, 276]. Their I/O operations include random and small I/O that can reduce disk throughput below one percent of an HDDs peak performance [32]. Further, limitations imposed by the POSIX standard can severely hinder the metadata performance of parallel file systems. This, and other challenges, e.g., cross-application interference due to I/O [139, 272], can lead to prolonged I/O latencies, reduced I/O throughput, and long wait times for users of the parallel file system. What is more, this does not only impact the user's application but can also heavily disrupt other applications that are using the same shared storage system [60, 243].

This thesis discusses three main subjects in this context, covering various challenges of parallel file systems in modern HPC systems and proposing possible solutions:

1. We will provide a detailed analysis of the challenges developers face when aiming to understand the behavior of the widely established GPFS [219] parallel file system. To examine system behavior, the *tracing* technique has become an essential tool in many use cases, including in development and production environments. However, tracing has various limitations that can cause high performance and storage overheads in certain situations and use cases, e.g., in metadata-intensive workloads. We will discuss and evaluate these tracing limitations and propose a new practical tracing interface – *FlexTrace* – to alleviate them.

2. We will discuss – *GekkoFS* – a novel ephemeral and highly scalable distributed file system that is optimized for I/O operations found in the above-introduced data-driven applications. It can be deployed *ad hoc* and uses a compute node's local storage capabilities as burst buffers to pool together the storage and performance of many nodes. As a result, GekkoFS can linearly scale data and metadata operations, substantially outperforming the capabilities of typical parallel file systems. By taking the design of existing distributed file systems

and previous studies on the behavior of HPC applications into account, we will present and evaluate a highly decoupled file system implementing relaxed file system semantics that any HPC user can deploy without administrative support.

3. We will present – *DelveFS* – an event-driven semantic file system for object stores that offers users the ability to define a custom file system, allowing multiple unique *views* onto the object store. Because of their high scalability and low maintenance overhead, object stores have become a popular option for many use cases ranging from archiving, media, and entertainment to large amounts of scientific data. However, object stores offer a custom I/O interface and cannot be utilized by existing applications that expect the standard file system interface. DelveFS provides such a file system interface, can run beside other object store interfaces and achieves similar performances as native object store interfaces. By exploiting object store events to keep file system views consistent, DelveFS significantly eases the usability of object store containers that contain huge numbers of objects and improves the performance of the corresponding file system operations considerably.

This dissertation is structured as follows: First, Chapter 2 will provide the theoretical background, covering all topics that are relevant to this thesis. Chapter 3 will present the challenges and solutions for tracing storage systems, discussing tracing techniques and their limitations, and will propose the FlexTrace tracing interface. Next, Chapter 4 will introduce the GekkoFS distributed burst buffer file system, covering its design and presenting a detailed evaluation for up to 512 nodes. Chapter 5 will discuss the DelveFS event-driven semantic file system for object stores, providing its design and evaluation. Chapter 6 will conclude this thesis.

2 Theoretical background

This chapter will provide essential background information for the context and topics covered in this thesis. This includes general concepts (e.g., file system basics and interface standards), file system paradigms (e.g., block storage and object storage), and the usage and challenges of distributed file systems in HPC environments. First, Section 2.1 will introduce the basic file system terminology and some of the core mechanisms. Section 2.2 will introduce parallel file systems and their architectural differences. Next, Section 2.3 will discuss scalability challenges in distributed file systems in the context of the POSIX standard and how burst buffer file systems can help. Lastly, Section 2.4 will briefly introduce object-based storage systems and their challenges for HPC applications.

2.1 File systems

File systems provide a way to organize data on storage devices in the form of files and directories. These can then be accessed by users and applications via the common file system interface. All objects in a Linux file system are generally considered *files* and are identified by their name. A file's type can refer to a *regular file*, a directory, a hardware device, a socket, and others [@195]. The regular file is an abstract data object identified by its name that people are used to.

The regular file, henceforward called a file, is the most common file type, and it is used for documents, spreadsheets, or program executables, for instance. A file contains both an ordered sequence of bytes (the corresponding *data*) and further information about the file's data (the file *metadata*). A folder or *directory* is another type of file and essentially contains a list of file names to refer to other file system objects within the directory. Therefore, the directory data is its list of files. As a result, nested directories allow the file system *namespace* (or directory tree) to form a hierarchical structure where files are the leaf nodes.

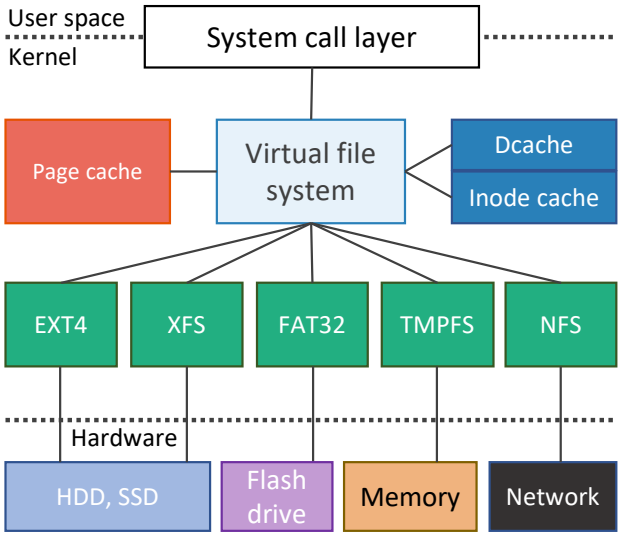


Figure 2.1.: Simplified kernel architecture for I/O and the VFS with its components.

2.1.1 The virtual file system

The *virtual file system* (VFS) is a software abstraction layer within UNIX kernels, handling all file system-related system calls and offering a unified interface for all UNIX file systems [28]. For instance, this makes it possible that a file can be copied between two file systems (on the same machine), e.g., an EXT4 Linux file system and a FAT32 file system which is often used on USB flash drives, by using the copy tool `cp`: `cp /home/vef/some_data /mnt/flash_drive/some_data`. In this example, `some_data` is transparently copied between the two file systems EXT4 and FAT32. This is only possible because both file systems support the unified VFS-defined interfaces, system calls, and data structures.

Figure 2.1 presents a simplified kernel architecture including the VFS. The VFS consists of four main components which will be discussed in the following in more detail: 1. the above-discussed abstraction layer; 2. the *inode cache* storing frequently accessed file metadata information; 3. the *dentry cache* (or *dcache*) storing frequently accessed directory information; and 4. the *page cache* representing the main kernel disk cache.

The inode object All file metadata information, such as ownership or file size, is kept in an *inode* (short for index node) structure on the disk. It uses the inode number for the lifetime of the file as a unique identifier. When an inode is in use, the VFS inode object is placed into the inode cache. Some of the inode data fields

```
~$ stat /home/vef/dummy_file
  File: /home/vef/dummy_file
  Size: 8089            Blocks: 32         IO Block: 4096   regular file
Device: 38h/56d Inode: 4995867     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   vef)   Gid: ( 1000/   vef)
Access: 2021-01-29 00:03:16.880647856 +0100
Modify: 2021-01-27 18:54:29.194008369 +0100
Change: 2021-01-27 18:54:29.202008398 +0100
 Birth: -
```

Figure 2.2.: Listing metadata of the `dummy_file` file with the `stat` command.

can be viewed by any user from the *command-line interface* (CLI) (see Figure 2.2), e.g., the file size, permissions, or the time of the last file write. Other inode fields include pointers to various file system structures or locking variables.

When specific file systems are implemented, the VFS inode object is then extended appropriately, e.g., how the data of the corresponding file is found and read on the disk (see Section 2.1.3 later). The exact approach of how a file system treats inodes or accesses data is left to the file system developers as long as the corresponding interfaces are supported.

The dentry object and pathname lookup While the inode contains most file metadata, it does not contain the file's name by which users and applications refer to it. Instead, the filename is part of a *dentry* (short for directory entry) object, associating a file with its directory. More specifically, the dentry object contains the file's name, a pointer to the file's inode, a pointer to the dentry object of the parent directory, and others. Therefore, a dentry exists for each component in a path, connecting the root (`/`) directory to its subdirectories and final component. In the example of Figure 2.2, the path `/home/vef/dummy_file` involves four dentry objects: one for each directory (including `/`) and one for the `dummy_file` file. Note that the dentry object is not directly stored on disk and only resides in memory. This is because the file system can decide how directory contents, or more generally, how data is stored as long as a dentry object can be created from this information.

A dentry object is created during a *pathname lookup* operation, henceforth referred to as *lookup*, for each path component. Thus, each lookup operation loads the corresponding dentries, accessed via the directory's inode object. Next, the lookup operation looks for the name in the directory for the next path component. Overall, each iteration of this process retrieves the dentry object and the inode of the next path component. When the final path component is resolved, the loop stops. These

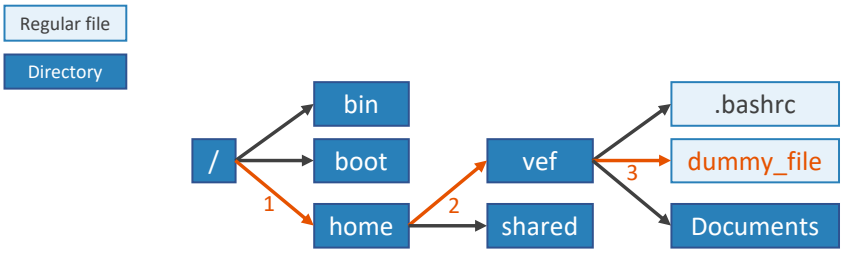


Figure 2.3.: An examplary pathname lookup procedure in three steps walking to the regular file: `dummy_file`.

consecutive lookup operations on the entire path are also known as a *path walk*, exemplarily shown in Figure 2.3.

Path walks are commonplace and constantly occur while the system is running. Accessing the disk for the same directory entries repeatedly would therefore be unnecessarily expensive and is avoided by caching dentry objects in the dcache. As a result, each lookup operation initially consults the dcache if a dentry object already exists for an inode-pathname pair. If this fast lookup fails, the corresponding directory information is accessed on the disk and, if successful, creates a new dentry object that is then placed into the dcache. If unsuccessful, the path does not exist, returning the error code `ENOENT` to the calling application.

File object and file descriptors Every time a file is opened, a corresponding *file* object is created as well, and it therefore counts as one of the important VFS data structures. This object remains in memory until the file is closed and stores information on how a process interacts with said file. This information includes a pointer to the file's dentry object and file permissions. Most notably, the file object stores the current offset (`f_pos`) in the file for the next I/O operation. For instance, a user might set this offset explicitly with the `lseek()` operation, after which a `read()` operation starts reading at the previously specified offset for the specified number of bytes. However, the `read()` operation also sets a new offset implicitly for future I/O calls based on the number of bytes read. The current offset is not stored within the file's inode since multiple processes can open the same file simultaneously, each with separate offsets.

Whereas the file object contains information on how a process interacts with a file, the *file descriptor* (fd) describes the connection between the opened file and the process. Programmatically, the user receives the file descriptor as an integer when the `open()` system call succeeds. Subsequent file operations then require the file

descriptor as an argument. The `close()` system call finally closes the file descriptor, freeing the file object.

2.1.2 Block devices

When retrieving and storing data on block-addressable storage (*block device*), such as HDDs, file systems access data by *blocks* that make up a file. A block is a file system abstraction of several *sectors*, that is, the smallest physically addressable unit, ranging from 512 bytes in older devices to 4096 adjacent bytes in modern devices. Sectors and blocks are the most basic units in block-addressable storage devices and file systems for transferring data. Nevertheless, since the continuous blocks of a file (*file blocks*) are not necessarily physically adjacent on disk, file systems further translate the file blocks to *logical blocks*. Therefore, logical block indices start from the beginning of the disk or partition, and file block indices start from the beginning of the file. File blocks form the contiguous sequence of bytes in the file, while the actual logical blocks on the storage device may not be adjacent at all (see Figure 2.4). The mapping between both block types is then accessible via the file's inode, discussed in the next section in detail.

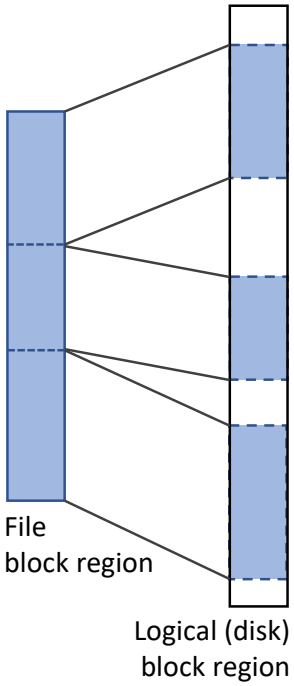


Figure 2.4.: Relationship between file blocks and logical blocks.

Before a process can read the data stored on disk, it must first be copied into main memory. Further, processes do not know the physical memory addresses where the data has been copied to and must use *virtual addresses*. Among others, virtual memory increases security through memory isolation so that each process can only access its own memory space, and it saves processes from managing the shared memory space themselves. Virtual addresses are translated by the *memory management unit* (MMU), which is part of the CPU, to the corresponding physical memory addresses. The MMU further divides the virtual address space into *pages*, a fixed-length continuous block, e.g., 4 KiB. The actual mapping of each virtual page to a physical memory page is then stored in the *page table*. However, because resolving each *page table entry* (PTE) for each memory access can be costly, the *translation lookaside buffer* (TLB) as part of the MMU caches the most recent virtual-to-physical address translations. Finally, the page's content is

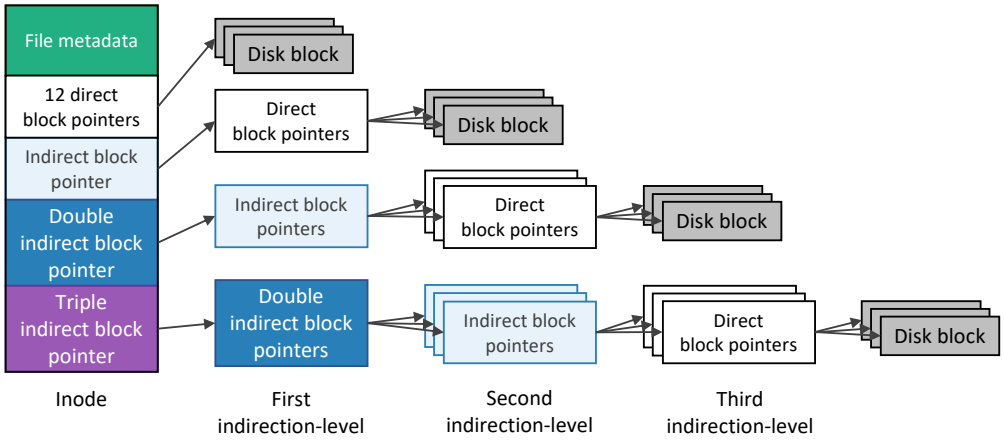


Figure 2.5.: EXT2 inode and levels of indirection for data blocks.

cached in the page cache to avoid the costly disk access[1]. The page cache typically utilizes the currently unused portions of main memory.

2.1.3 File system specifics

We exemplarily use the EXT2 file system (short for *the second extended file system*) [39] to briefly explain basic file system mechanisms with regards to how parts of the VFS interact with a specific file system. In particular, we will focus on inodes, data block addressing, directories, and how modern file systems extend these mechanisms. EXT2 was released in 1993, is native to the Linux kernel, and is still used in practice (e.g., in boot partitions).

Inodes and data block addressing As introduced earlier, each VFS inode object must contain several fields, e.g., file size or permissions. Nevertheless, the VFS does not dictate how a file system addresses a file's data. EXT2 extends the VFS inode by several fields in the `ext2_inode` structure, most notably, the `i_block` array of pointers to data blocks. These data blocks contain the mapping between the file block number (relative to the beginning of the file) and the logical block number (relative to the beginning of the disk or partition). By default, the `i_block` array holds 12 *direct blocks* and 3 *indirect blocks* visualized in Figure 2.5. Direct blocks map the first 12 file block numbers to their corresponding logical block numbers. The three indirect blocks, on the other hand, point to second, third, and fourth-order arrays before containing the mapping to the logical block number. These *levels of*

[1] Sometimes, it can be beneficial to by-pass the page cache via the `O_DIRECT` flag when opening a file.

indirection allow EXT2 to store around 4 TiB in total per regular file while the 12 direct blocks only allow 48 KiB (based on a 4 KiB block size). Depending on the file size and the corresponding levels of indirection required, this structure benefits smaller files since larger files have a higher number of memory accesses per data block.

Directories Directories use data blocks to store information about which files belong to the directory. Thus, directory data blocks are also called *directory blocks*, each containing many directory entries. The directory entries are not to be confused with the earlier-introduced in-memory dentry objects. The directory entry corresponds to a file within the directory and includes storing its filename and inode number. For instance, when a directory is read during a `lookup()` operation, the inode number in the directory entry allows the file system to traverse to the next component of the path. Therefore, a directory block on the disk is only read when the corresponding VFS dentry object is not already available in the dcache. Note that reading all directory entries of a directory allows the file system to list all filenames without actually accessing their inodes for further metadata.

Modern file systems Although EXT2 still finds usage, modern file systems have added significant improvements for the above-introduced basic mechanisms. For instance, the EXT4 file system [37, 148], released in 2008 and kernel version 2.6.19, is nowadays used in many Linux environments and extends EXT2 by many features.

These features include *extents* which describe ranges of contiguous physical blocks and replaces the logical block map, reducing the required metadata for the data blocks. Extents decrease data access times, especially for large files, and allow for up to 16 TiB file size and a maximum of 1 EiB overall file system size. Nevertheless, some file systems are still using the indirect block design, e.g., *GPFS* [219], introduced later in this chapter.

Journaling is another prominent feature and was introduced in EXT3 [38]). Journaling keeps track of changes that have not yet been persisted on disk in a circular log, called a *journal*. In the event of a system crash, not yet persisted changes can be replayed to repair the file system, reducing the likelihood for file system corruption and data loss[2].

[2]Typically journaling only applies to metadata for performance benefits at the cost of possible data corruption. EXT4 forces all data to disk before metadata is committed to the journal, although full data journaling can be enabled when mounting the file system.

Other features introduced persistent pre-allocation of disk space, delayed allocation, online defragmentation, support for significantly larger directories, and many more [@140].

2.2 General-purpose parallel file systems

A *general-purpose parallel file system* (in short parallel file system) is a file system that runs in a distributed environment, e.g., in supercomputers used for HPC. Therefore, it is a type of distributed file system that communicates over the network. A parallel file system uses a number of server and management compute nodes connected to a large number of storage devices. These devices are typically combined into several logical units with the *RAID* [184] (redundant array of inexpensive disks) storage virtualization technology. RAID offers multiple configurations (or levels) depending on the performance or redundancy requirements. The file system clients run on a separate set of compute nodes presenting a standard UNIX mount point and appear just as any other file system for users.

Throughout this thesis, we will use two of the most popular parallel file systems, GPFS and Lustre, in various experiments. From the user perspective, GPFS [219] and Lustre [29] are traditional parallel file systems. Typical use cases range from small (less than ten machines) to large (tens of thousands of nodes) compute clusters. Unlike some other distributed file systems [81, 225], GPFS and Lustre provide a complete POSIX interface with a strong consistency model (see the following section) to applications which contributes additional complexity to their architecture. Therefore, when an application on one file system client creates or updates a file in GPFS or Lustre, other nodes seamlessly see the updates. Both parallel file systems are used as the storage backbone in most HPC centres [@105, @247].

2.2.1 GPFS

The *General Parallel File System* (GPFS) started as a research project in the early 1990s [92] and was released in 1998. It is closed-source and developed, supported, and released by IBM. In 2016, GPFS's codebase already contained over 1 million lines of code added by over 73 thousand commits [255].

GPFS uses a shared block device model to store its data. In general, the block device of any compute node can be assigned as a *shared disk*. They can be shared naturally via a *storage area network* (SAN), or they can be exported through GPFS's proprietary

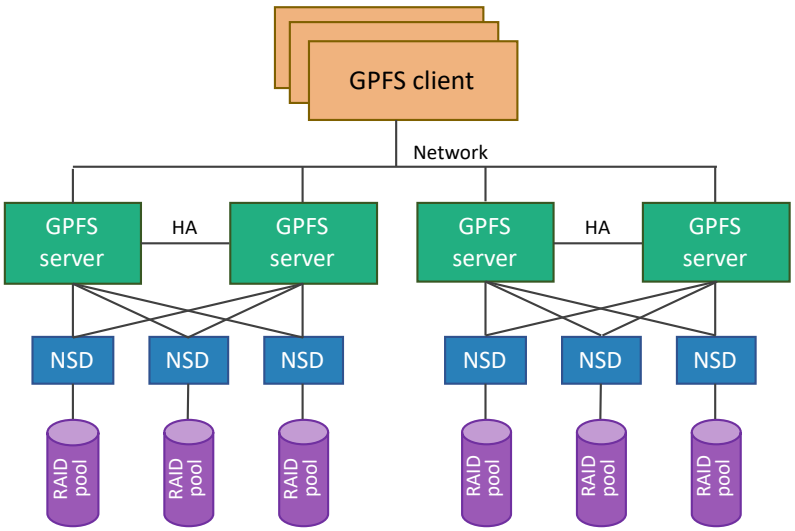


Figure 2.6.: A simplified example of the GPFS architecture in an NSD configuration.

block-level protocol. Since SAN configurations can become increasingly complex for large GPFS clusters, a *network shared disk* (NSD) configuration is commonly used. To achieve high performance in terms of throughput and latency when performing I/O, GPFS stripes its metadata and data across all shared disks, called *wide striping*. For coordinating the parallel updates to the shared disks, a configurable number of *token servers* control the distribution of per-object (e.g., per-block or per-file) *lock tokens* to GPFS clients. When a client owns a token for a file, it has exclusive access and can update the file and its metadata without any coordination with other nodes.

Figure 2.6 presents an example configuration where GPFS servers are set up in a *high availability* (HA) configuration such that access to all NSDs is ensured when a GPFS server in a pair fails. Each HA server pair can use multiple NSDs, each using either a single block device directly or a virtual device. The virtual device can consist of multiple block devices pooled together in a RAID configuration. Users then transparently access GPFS's namespace via the corresponding mount point at the compute node (GPFS client). More information about the GPFS architecture can be found in [98, 219].

To support the needs of new customers, many advanced features were added to GPFS over time [99]. These include snapshots (as well as writable per-file snapshots), integrated backups, user-defined tiering policies, file sets, caching over wide area network (WAN), rolling upgrades, online file system resizing, and *quality of service* (QoS) policies. For compatibility, GPFS also includes major file system protocol implementations, namely NFS (*network file system*), SMB (*server message block*), and

the *RESTful Object* protocol. GPFS runs on Linux, AIX, and Windows and supports x86, Power, and System/390 architectures.

2.2.2 Lustre

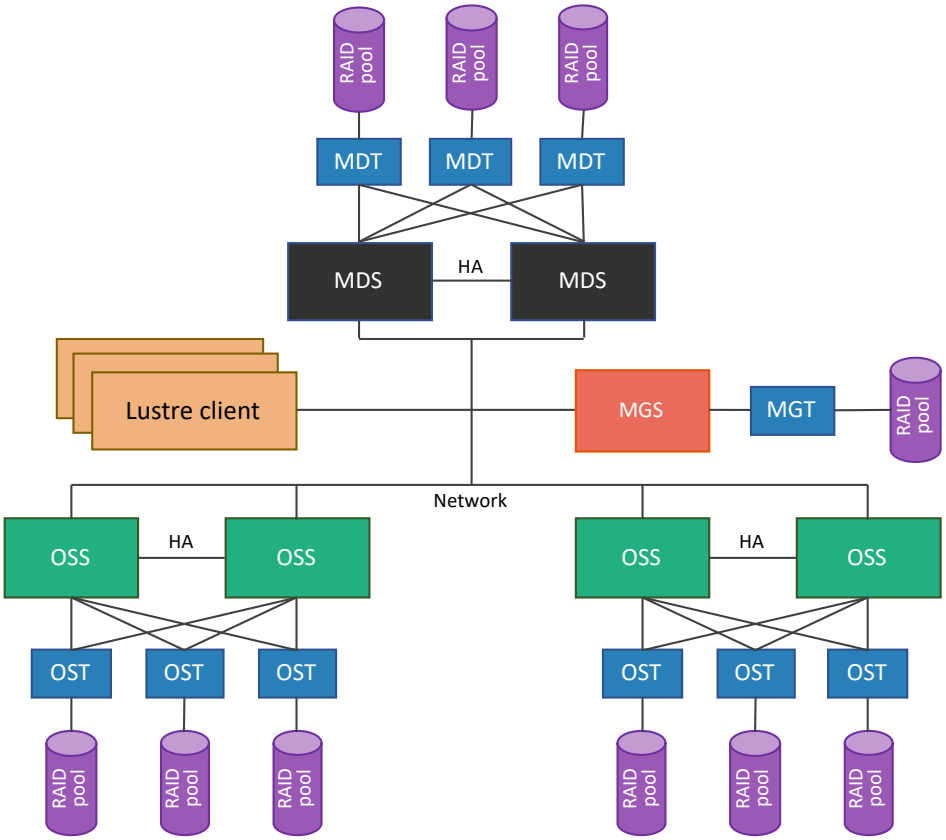


Figure 2.7.: A simplified example of a Lustre configuration.

The development of Lustre started in 1999 by Peter Braam at Carnegie Mellon University, PA, USA. In contrast to GPFS, Lustre is an open-source project licensed under the GPL 2.0 software license. Overall, Lustre's history is complex, with several companies acquiring its assets over the years. Since 2010, Lustre has been mainly developed by Whamcloud, but many other companies, e.g., Fujitsu, contributed to the codebase as well. At the time of writing, DDN's (DataDirect Networks) acquired Whamcloud division has developed, supported, and released the Lustre parallel file system since 2018.

Like GPFS, Lustre provides a POSIX-compliant parallel file system with strong consistency guarantees. On the other side, Lustre's fundamental architecture differs significantly from GPFS and is based on the object storage concept (see Section 2.4

later in this chapter). The object storage abstraction layer (the *object storage device*, OSD) is built on top of the underlying block storage devices formatted with the LDISKFS (based on EXT4) or ZFS local file system. For instance, an object can be considered a file on such a local file system [29]. Note that, depending on *stripe size* that can be set at directory granularity, a Lustre file can be distributed over multiple OSTs. Three OSD types are connected to one or more storage devices:

1. The *metadata target* (MDT) used by the *metadata service* (MDS) to manage Lustre's metadata and maintain its namespace.
2. The *object storage target* (OST) used by the *object storage service* (OSS) to handle all data objects that contain the contents of Lustre files.
3. The *management target* (MGT) used by the *management service* (MGS) to manage general Lustre configuration data.

Figure 2.7 shows an exemplary Lustre architecture with HA MDS and OSS pairs to ensure data is still accessible if one MDS or OSS fails. Each HA pair can use multiple OSTs or MDTs, whereas each target uses a single block device directly or a virtual device consisting of multiple block devices. For example, a single OST could represent 15 HDDs pooled together via ZFS in a RAID configuration. In such a configuration, OSSs can share the load by assigning them to distinct OST sets while, due to HA, a single one can manage all OSTs in case one OSS fails. While a Lustre file system can scale to hundreds of OSSs, a typical Lustre installation may only use few MDSs for handling metadata. Moreover, it is common that the MGS is co-located with an MDS since it only uses few computational resources. Users transparently access Lustre's namespace via the corresponding mount point at the compute node (Lustre client). More information about the Lustre architecture can be found in [29, @190].

2.3 Scalability challenges in distributed file systems

This section will discuss some of the scalability challenges that occur in distributed file systems and bring them into context with the POSIX standard and burst buffer file systems.

2.3.1 The POSIX standard

The *Portable Operation System Interface* (POSIX) [262] is a set of standards introduced in 1988 by the *Institute of Electrical and Electronics Engineers* (IEEE) and *The Open Group*. Their main goal has been to set out various operating systems guidelines to ease cross-platform development of systems and applications via specific definitions of OS interfaces or file system protocols. *POSIX.1-2017* is the most recent release discussing a multitude of topics like standard shell interfaces or definitions on portability and error handling for the C programming language.

In the context of file systems and this thesis, the following topics in the POSIX standard are particularly important. As part of the *System Interfaces Volume*, POSIX defines the interface of I/O operations, i.e., syntax, behavior, and error codes. Notice that it does not define its implementation. Therefore, I/O operations, such as `open()` or `write()`, implicitly hold assumptions about how data must be accessed and when data of a `write()` operation is readable by other processes. For instance, the UNIX man page of the `write()` page reads: *"POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data."* [@132]. Further, all write operations must appear atomic to any reader, regardless of location. In other words, a write operation must be immediately visible to any reader in a POSIX-compliant file system. Thus, if multiple processes write to overlapping data regions, the rules of the I/O system guarantee consistent results. This consistency is also called *strong consistency* (or strong POSIX semantics) and has become a major point for discussion in the systems community [19, @119, @193, 241, 263].

When POSIX was first ratified in 1988, local file systems were accessed by a single-core machine. This allowed file systems to implement a strong consistency model without complex mechanisms to control concurrent access to a file. With multi-core processors, file systems needed to control access to a shared resource, e.g., a storage disk, to avoid undefined results when multiple processes concurrently access the same file region. This is commonly done with synchronization primitives, e.g., locks or mutexes, to coordinate atomic access efficiently.

When a similar consistency model is required in a distributed environment, these synchronization protocols must extend to a large number of computing nodes. Maintaining strong consistency under these conditions with many file system servers and clients is therefore a communication-intensive and complex task. Two possible approaches to control access in a distributed setting are the usage of a central broker node (*centralized management*) or a *distributed locking manager* (DLM) [30, 121, 219]. In the former case, a dedicated broker node is used to act as a proxy server for

all clients. In other words, all I/O requests must pass through the broker node, which then essentially serializes access to a data region. In the DLM case, a dedicated node is used to hand out tokens to clients. Each token can refer to a specific shared resource, such as a data region, and gives a client shared or exclusive access to the resource.

Such locking mechanisms can scale reasonably well for large sequential I/O access patterns [178] in cases where each process accesses its own file since a corresponding lock does not need to be acquired repeatably. On the other hand, they can also cause significant scalability issues. This is especially the case when multiple processes access the same file concurrently, causing lock congestion at the locking manager and resulting in delays in performing an I/O operation. To mitigate such delays, distributed storage systems employ granular locking techniques in which locks only apply to file regions instead of the whole file; see *byte-range locking* in Lustre [30] and GPFS [219]. However, strong consistency semantics and poor parallel I/O support in file systems [123] combined with scientific applications' access patterns [168] are the main reasons for I/O challenges in today's HPC environments. Despite ever-increasing computing performance, data-intensive applications running on HPC systems face scalability challenges that prevent them from fully utilizing the offered computing power [46]. These challenges are particularly apparent during metadata workloads that include many *metadata operations*, which we will discuss next.

2.3.2 Metadata and small I/O operations

For shared accesses, parallel file systems typically rely on expensive distributed locking mechanisms to avoid conflicts, e.g., byte-range locking. This is generally a fundamental feature for parallel file systems because it allows them to represent a fully POSIX-compliant I/O interface with strong consistency semantics. As a result, users can access the (parallel) file system just as any other local file system. Nevertheless, while distributed locking mechanisms can work well in sequential workloads on large files, they can become challenging for metadata operations.

At its core, metadata in Unix-like operating systems can be categorized into three components: an object's (e.g., file or directory) metadata, a file's data, and a directory's contents [207]. A corresponding metadata operation involves accessing a file's inode or a directory's data blocks containing its directory entries. In contrast to large I/O operations, which mainly target a file's data blocks[3], metadata operations are typically only in the range of kilobytes. Handling such operations in a strongly

[3] A write operation must also update the inode's file size field.

consistent distributed file system is challenging because inodes and directory blocks were not designed for parallel accesses as only a single block can be accessed by one process at a time. Moreover, metadata operations can be significantly more complex than data operations, whereas a single metadata operation, e.g., listing a directory or even creating a file, results in write-accessing multiple inodes and directory blocks. In these cases, each metadata modification requires a write-lock to access the corresponding metadata structure. It is therefore not surprising that in concurrent metadata workloads, multiple processes compete to acquire the corresponding locks. This, in turn, frequently results in lock-thrashing, i.e., locks are continuously granted and revoked, causing even more overhead in the form of network communication between file system clients and distributed lock managers. The high complexity combined with a potentially high frequency of metadata operations (due to their small size) exacerbates the metadata challenges parallel file systems face, causing poor scalability and performance (see Section 4.5.3 later on Lustre's metadata performance).

Prominent example workloads with a huge number of metadata operations are the listing of large directories [182], bulk operations that create millions of files in a single directory in parallel [19, @179, 182, 255], or small I/O operations in the range of bytes to kilobytes [40]. To measure metadata performance, the evaluations in this thesis frequently include the concurrent file creation workload for two reasons: First, due to the complexity of the file create process, this workload stresses many file system components at the same time. In particular, we analyzed the complexity of the file create process and its implication for GPFS in an earlier work [251], showing the large impact of various synchronization and locking mechanisms on the file system's metadata performance. Other works showed how concurrent file creation from thousands of clients is one of the most difficult workloads for a parallel file system to perform efficiently [205]. Second, we used concurrent file creation because of its significance in many HPC applications. For instance, it is a use case in *checkpointing* to protect intermediate results from failures [72, 77, 182]. Here, each process often creates files in a single directory [19]. This workload is so common in HPC that official requirements for the CORAL supercomputer include a 50,000 file creates per second rate in a single directory [@179]. Other workloads that include concurrent file creation are data ingestion [151] and migration [199], especially from object stores that use flat namespaces and therefore files are often created in a single directory in a file system corresponding to a single object store container [@12, 161].

Other works discussed that metadata costs are a major factor in the application I/O throughput in supercomputer environments [40, 144]. Carns et al. [40], for instance,

showed the high metadata overhead of an application writing many small files of about 400 KiB on average. As a result, researchers presented various techniques throughout the last decade for several distributed storage systems, exploring new ideas to handle heavy metadata loads effectively [19, 73, 182, 183, 273, 274]. This challenge is still prevailing and is becoming an even bigger challenge for upcoming data science applications due to their I/O access patterns and used data volumes. One of the possible solutions for this challenge is to employ burst buffer file systems which we will discuss next.

2.3.3 Burst buffer file systems

Burst buffers are fast storage systems logically placed between the application and the parallel file system to soak potentially harmful I/O operations and bursts. Therefore, they aim to reduce an application's I/O overhead and load on the parallel file system [134]. Burst buffers can be categorized into two groups [265]: remote-shared and node-local. Remote-shared burst buffers are centralized, dedicated I/O nodes structured as a forwarding layer, e.g., DDN's IME [@55] and Cray's DataWarp [95].

Nowadays, supercomputers typically offer fast flash-based storage devices at the compute nodes that can be used temporarily by an application to accelerate its I/O performance. However, these *node-local SSDs* are often under-utilized by applications [32, 265], and therefore burst buffer file systems can leverage them to form an ephemeral distributed file system that combines the capacities and performance of all node-local SSDs. Thus, burst buffer file systems are often collocated with compute nodes and deployed *ad hoc*. Such file systems are also called *ad hoc file systems* and their ability to be quickly deployed and destroyed in a job context is particularly important not to waste precious computational resources. Moreover, burst buffers can also be dependent on the parallel file system (e.g., PLFS [19]) or managed directly by it [@179]. Because these ad hoc burst buffer file systems are often accessed by a single application, they can heavily optimize for an application's requirements. If possible, such a file system could even relax consistency guarantees to offer higher performance and scalability than the parallel file system can provide.

In this context, Chapter 4 will introduce the *GekkoFS* burst buffer file system, a temporary, highly scalable distributed file system that attempts to solve metadata challenges for HPC applications by using a decoupled file system architecture that does not require inter-node locking. It provides relaxed POSIX semantics targeting

file system operations that are mainly used by HPC applications. As a result, GekkoFS can provide scalable I/O performance and reaches millions of metadata operations already for a small number of nodes, significantly outperforming the capabilities of common parallel file systems. By flexibly deploying multiple burst buffer file systems, temporary file systems can work together with the parallel file system combining the performance of distributed burst buffers and the long-term data persistence of parallel file systems.

In summary, well-known central data structures, such as inodes and directory blocks, are one of the root causes of today's metadata challenges. Although POSIX only defines the I/O interface and therefore not the implementation of these data structures, maintaining a strongly consistent file system in a distributed context remains challenging for performance. In this thesis, we argue that a fully POSIX-compliant file system is not always necessary and can greatly benefit performance as well as reduce its complexity, exemplarily shown by GekkoFS.

2.4 Object storage systems

Object storage, or object-based storage, is another approach to store data [7, 153, 161, 268]. While file systems use files as distinct data units, object storage uses the concept of *objects*. Objects can be placed into *containers* (or *buckets*), but they cannot form a hierarchy, as is the case with directories in file systems. Thus, object stores use a *flat namespace* with sometimes millions of objects within a single container [@12]. Also, object stores are not implementing a strong consistency model and are not POSIX-compliant. They do not interact with UNIX kernel components, for example, and use an entirely different API to interact with objects that are often accessed via cloud services over the internet.

A low-level interface to object storage was already defined in 2004 by the ANSI T10 body [162]. This and similar interfaces successfully served as a foundation for parallel file system implementations [29, 270]. Today, most object storage interfaces provide two types of web-based *representational state transfer* (REST) interfaces: a custom interface tailored to the corresponding object store via a corresponding *software development kit* (SDK) and the Amazon S3 API [79]. Nonetheless, even when using the S3 API, there are still limitations, subtle differences, and S3 version incompatibilities that are challenging for applications to adapt to and that restrict the portability of applications closely tied to one API [188].

Object-based storage has several advantages over traditional storage methods, such as file systems. Generally, object stores tend to scale well [153] because object store and container complexity do not increase when objects are uploaded. Moreover, it is possible to add arbitrary metadata to objects, and the combination of data and metadata allows object stores to optimize the data layout or to provide QoS guarantees based on individual objects [269]. The flat structure of an object store also allows for a low maintenance overhead.

Especially for unstructured data, e.g., large media libraries or *data lakes* where information is stored in the raw format [89], object-based storage has become a popular choice for many internet-scale applications like Netflix, Spotify, or Dropbox. It is therefore not surprising that nearly all cloud providers provide or internally use object storage [35, 79, 231]. Over the past decade, many academic implementations have fostered the development of object stores [2, 149, 150, 170, 242, 246]. Further, open-source and commercial providers also enable on-premise hosting of object stores [7, 268].

For legacy applications or applications that require a POSIX file system interface and strong consistency, such an object-based API is not a usable option. As a result, numerous file system *connectors* have been developed that internally utilize object-based APIs to provide a UNIX file system [21, 130, @177, @214, 261, 267]. However, the flat namespace of object stores and huge container sizes can still cause significant metadata challenges, e.g., when listing such a container. Further challenges include usability because of the lack of basic semantic properties and structure, as in standard UNIX file systems.

Chapter 5 will discuss this topic and introduce *DelveFS*, an event-driven semantic object store file system proposing a solution for these challenges by offering the ability to compose a custom semantic file system that allows multiple unique *views* onto the object store. Through flexible filters, users can specify each view's content, tailored to their unique interests or an application's requirements. By processing object store events that describe changes in the object store, DelveFS is able to keep all views eventually consistent. DelveFS allows to operate concurrently through the object and file system interfaces on the same set of objects, delivering similar file system throughput compared to the native object store interfaces or other file system connectors.

3 Challenges and solutions for tracing storage systems

IBM Spectrum Scale's *General Parallel File System* (GPFS) has a 20-year development history with over 100 contributing developers. Its ability to support strict POSIX semantics across more than 10K clients leads to a complex design with intricate interactions between the compute nodes. Tracing has shown to be a vital tool to understand the behavior and the anomalies of such a complex software product [196]. In essence, the produced trace records are time-stamped journal entries containing selective variables at a specific location within a code path. However, the necessary trace information is often buried in hundreds of gigabytes of byproduct trace records. Further, the overhead of tracing can significantly impact running applications and file system performance, limiting the use of tracing in a production system.

This chapter will discuss the evolution of the complex and highly scalable GPFS tracing tool and demonstrate how the current approach can cause significant problems in long-living systems that are deployed in large production environments. We will provide detailed insights into the proprietary GPFS file system, explain how its tracing mechanisms work, and analyze the root causes for the above-mentioned challenges. Further, we will present an exploratory study of a possible new tracing interface for GPFS, *FlexTrace*, which allows developers and users to accurately specify what to trace. FlexTrace integrates seamlessly into GPFS's existing tracing framework and gives users and developers fine-grained control over each individual tracepoint minimizing the typical tracing volumes and overheads of GPFS's original tracing mechanisms. Coupled with user-defined tracing profiles, users can dynamically design and enable a specific set of trace points for a given use case.

FlexTrace's simplicity from the conceptual and implementation points of view is an essential design choice, showing how a simple (and yet not widely recognized) technique can have many advantages over other existing tracing techniques. We will evaluate our methodology and prototype and demonstrate that the proposed approach has negligible overhead even under intensive I/O workloads and with low-latency storage devices. Therefore, we will show how FlexTrace's approach is beneficial when developing systems that are expected to live long and that are developed by many engineers.

Section 3.1 will motivate and introduce the work in detail. In Section 3.2, we will provide background information and the related work of the general tracing topic and of the numerous tracing techniques and tools. Moreover, Section 3.3 will thoroughly describe GPFS's tracing tool and how its internal mechanisms function. We will explain the tracing challenges concerning tracepoint distribution in the source code, offer insights into developer decisions when placing tracepoints, and evaluate the performance and data volume overheads of tracing in Section 3.4. Section 3.5 will introduce FlexTrace, which we will evaluate in Section 3.6. Finally, Section 3.7 will conclude this chapter.

3.1 Motivation

The complexity of modern software tends to be extremely high. For example, Microsoft Windows contains over 50 million *lines of code* (LoC), the Linux kernel has almost 16 million LoC, and IBM's General Parallel File System (GPFS) [219] contains more than 1 million LoC. Complex code logic incorporates many non-trivial interactions between software components and is specific to particular operating systems or the underlying hardware. To ensure continued support for running applications, software projects rarely remove existing features, resulting in growing code complexity, often at an alarmingly high rate. For instance, the Linux kernel grows by approximately 4.5K LoC daily. Therefore, reducing the hurdle of understanding the code and the resulting system behavior becomes a grand challenge for many organizations.

Distributed storage systems, such as parallel file systems, are an excellent example of complex software due to their extensive network communication and the need to support various complex semantics, such as POSIX [262] or NFSv4.1 [94]. The code complexity affects developers and maintainers of such software because it hinders their ability to add new features and debug issues efficiently. The difficulties of understanding code are complemented by the fact that, in many cases, the original developers are no longer available or have moved to other projects or companies. Scanning scarce (sometimes outdated) documentation, commit logs, or reviewing millions of LoC is a cumbersome, time-consuming, and complicated process. Moreover, this process is inefficient and typically not sufficient either because of unexpected side effects, especially when working in heavily parallel environments.

Nevertheless, code complexity also impacts users of distributed storage systems who struggle to understand and reproduce various issues of running applications that access the storage system. Hence, both developers and users need better tools to debug and diagnose the underlying storage system to solve issues and ultimately improve application performance and storage system efficiency.

In the past, tracing has been shown to be an essential method for a wide range of use cases during development, testing, and in production environments [196]. These use cases include: 1. gaining a detailed understanding of storage systems' behavior and the applications that use them, 2. recently implemented code verification, 3. workload characterization, 4. bug analysis, and 5. performance diagnosis. At its core, a trace (record) is a time-stamped journal entry that captures a set number of variables at a specific location in an application's code path. In general, finalized traces are human-readable text files that contain records and allow an understanding of how a system works. This knowledge can then be helpful in the above-described use cases.

By default, trace records are not collected since collecting traces remains a challenge in production environments. This is because trace collection consumes CPU, memory, network, and storage resources and thus can significantly cause performance degradation, leading to a change in execution flow and system behavior that potentially hides production defects. This challenge becomes even harder for larger and more powerful systems in the future since tracing overheads (and the trace output size) increase with the number of operations collaboratively executed per second. Although many studies and engineering efforts have focused on improving tracing performance and efficiency to better understand a system as a whole, this problem still persists in production-grade storage systems and in distributed systems in general [20, 145, 223, 256].

This chapter will explain the tracing framework of GPFS, a widely deployed storage system with over 20 years of continuous development. As described in more detail in Section 2.2.1, GPFS is a POSIX-compliant parallel file system that uses distributed locking techniques for its data and metadata management. While it was initially developed for HPC, GPFS's functionality was greatly extended to support a wide variety of modern workloads [99]. Nowadays, it is used by thousands of customers and runs on hundreds of thousands of servers.

Section 3.3 will first discuss how GPFS's tracing works in modern deployment, describing rarely explored practical insights that shaped GPFS's tracing infrastructure. After that, we will present statistics on how over 100 developers used the tracepoints, given its fixed and rigid tracing interface. We will show that when a tracing

interface is based on a traditional *subsystem-and-priority* classification of tracepoints, developers tend to use only a limited number of subsystems concentrated around default priority levels.

We will also discuss other significant limitations of this commonly used approach to tracing, such as high overheads due to many *byproduct* trace records, and propose a new practical tracing interface for alleviating this issue called FlexTrace. At its core, FlexTrace offers developers and users the ability to enable and disable *individual* tracepoints rather than static pre-configured sets. FlexTrace's conceptual and implementation simplicity is one of its most essential properties as it allows its design to be easily adapted to nearly any storage system and other complex applications. Although FlexTrace is not an entirely novel concept [83, @249], we are, to the best of our knowledge, the first to investigate and evaluate the benefits of a flexible tracing facility in a real-world storage system.

We implemented FlexTrace in GPFS and investigated how everyday trace-based tasks, e.g., understanding system behavior or workload characterization, can become significantly less data-intensive and easier to perform. We note how this simple (and yet not widely recognized) method can substantially impact many use cases where the rigid subsystem-and-priority classification fell short and sometimes required custom GPFS builds for customers. We will evaluate FlexTrace's overhead in the worst case (see Section 3.6) and demonstrate that it was indistinguishable from running applications without tracing, even for workloads that stress the system. This is especially important in cases where a specific number of traces need to be collected for more extended time periods. Because of the GPFS's tracing facility limitations, this is rarely possible today.

3.2 Related work

Even relatively simple local file systems have become significantly more complex [5, 208]. Distributed file systems work on a drastically higher scalability level which exacerbates common storage challenges. These include high performance, near-perfect availability, fault tolerance, support for POSIX (or similar) semantics, and advanced functionalities at scale. As a result, distributed file systems count towards the most complex software systems today [29, 30, 81, 198, 219, 225, 267].

Tracing is an essential tool for developers to continuously maintain such complex systems efficiently. Consequently, any production-type and widely deployed distributed file system offers a corresponding mechanism to collect traces [@93, @143]. With

GPFS being on the market longer than other major production-type distributed file systems, e.g., Lustre or Ceph, GPFS allows for unique observations of its evolving tracing mechanism, its use by developers, and its limitations when facing real-world problems of a deployed system.

The following will discuss tracing as a generic concept and compare existing instrumentation techniques.

3.2.1 Tracing

Over the last two decades, developers have cultivated a wide variety of techniques and associated tools for examining system behavior [48, 223, @230]. The techniques range from analyzing system logs [@238], monitoring performance counters and profiling [43, @133], to complex system tracing that produces a log of time-stamped events and their attributes [63]. To log an event via a trace mechanism, a *tracepoint* needs to be either predefined during compile time or dynamically added at runtime at the desired code location. For instance, a tracepoint only contains a template string but can capture the content of several crucial variables. Further, tracing needs a mechanism that conditionally executes the tracepoints to produce a *trace record*, that is, a corresponding log entry capturing the data of a tracepoint. For the sake of efficiency, trace records are typically written to an in-memory buffer first before asynchronously being moved to persistent storage [6, 48].

The term *tracing* itself is rather broad and at times ambiguous as its meaning depends on the context and the environment. For instance, tracing can refer to tracking network routes [4, 126], collecting information on consecutive function calls [212], or also recording disk I/O activity [80, 154]. In the case of GPFS, traces can contain information about events from almost any part of the software system, e.g., token managers, locking mechanisms, and remote procedure calls.

Because tracing plays an important role within the storage community [6, 64, 65, 282], many tracing tools and techniques have been implemented [@50, 58, @111, @230, @249]. They can be utilized in many use cases: understanding system behavior, locating various performance bottlenecks, general debugging, user workload characterization, application behavior optimizations, and more. For instance, Wright et al. [271] achieved a 25x speedup in I/O performance on a large *Parallel Log-structured File System* (PLFS) installation via I/O trace analysis. Moreover, traces were shown to be an essential tool during development and testing [145], including using them to accurately replay entire user workloads [154]. They have

also been shown to be a vital source of information both in controlled environments and in production environments [196].

Because the accumulation of trace records (the *trace output*) grows over time [240], engineers and researchers made efforts to reduce the trace output size. Google, for example, optimized the Linux kernel tracing to remove constant strings from the trace buffers [@111]. Among others, this allowed them to increase the time frame where recent events could be stored in trace buffers from the initial 10 seconds to 45 seconds. Thus, more traces could be captured over a larger time frame.

3.2.2 Tracing techniques

Nowadays, there is a plethora of tracing tools available [48, 58, 63, @85, 212, @230], making it challenging to find the appropriate approach in a specific situation. They usually differ in their flexibility and performance overheads outlined below. Strace [@230], for example, is easy to use and widely available but only gathers data about an application's used system calls and has well-known performance overheads. The *extended Berkeley Paket Filter* (eBPF) allows to trace an arbitrary execution but also allows filters with dynamically placed tracepoints. Although many improvements were made focusing on its performance, it still causes measurable overhead [223]. Other end-to-end tracing techniques gather information about the call path and timings in a distributed system of individual requests, analyzing the system's behavior and request flows [71, 217, 245]. Another set of tools monitors specific parts of a distributed system. For instance, some track the inter-node communications between cluster components [34], their accurate trace replay [15], and their compression [169]. Others target I/O workload characterization which is either achieved through I/O profiling [43, 227, 248] or I/O trace analysis and trace replay [142, 154, 271].

These tools can be categorized into two groups based on their underlying instrumentation technique: *static* and *dynamic*.

Static instrumentation Nowadays, static instrumentation is the most commonly used tracing technique in distributed storage systems, e.g., Lustre [30, @143], Ceph [@93, 267], or BeeGFS [@17, 97]. In static instrumentation, the tracepoint location and its format, i.e., the data it captures, is defined a priori by a developer.

Several tracing tools that use static instrumentation also support the individual enabling of tracepoints [58, @249]. The `TRACE_EVENT` macro [@249] in the Linux

kernel allows adding a static tracepoint at the desired location in the source code. Interestingly, tracepoints defined by `TRACE_EVENT` must still be assigned to static subsystems, a set of tracepoints (usually describing its context). We argue that such static subsystems limit the usefulness of tracing and that this decision should be made by the user rather than at compilation time (which is not always possible). Later, Section 3.4.1 will debate the practicality of subsystems in the context of large storage systems.

Dynamic instrumentation With dynamic instrumentation, a tracepoint can be added at runtime after the software has been developed, compiled, and deployed by using binary code instrumentation [36]. *SystemTap* is one example in Linux that uses the *Kprobe* facility for binary instrumentation of running code at a desired location [63, 115]. At its core, a Kprobe replaces an in-memory instruction with a trampoline instruction or an interrupt, leading to the corresponding user-defined code.

Google's Xray framework attempts dynamic low-overhead tracepoints [20] by placing `NOOP` instructions at a function entry and exit during compilation. When tracing is enabled by the user, these `NOOP` instructions are then replaced with the actual tracing code. Ftrace provides a comparable mechanism for the Linux kernel [212]. Nevertheless, functional tracing is not always sufficient, especially when larger functions with many conditional branches need to be traced.

Fully dynamic instrumentation can be achieved with DTrace [48] and Kprobes [115] in Unix-like systems and Fay in Windows [66]. They allow adding tracepoints at arbitrary kernel locations. Pin [141] offers another technique for runtime binary instrumentation on Linux, using either probes similar to Dtrace or just-in-time (jit) compiler techniques to insert and instrument code. IOPin [112] uses Pin's probe-mode for runtime profiling of parallel I/O in HPC systems. While it can be helpful when analyzing the I/O path performance of workloads, its overhead is too high to run in production environments [227].

The authors of Pivot Tracing [145] argue that it is hard to decide where tracepoints should be placed during development and that this decision should be given to users at runtime. We share this idea, but we further argue in this chapter how practical constraints limit the adaptation of this approach in a software product with a long history of development.

Comparing static and dynamic instrumentation For static tracing, the data each tracepoint emits (due to its fixed format) cannot be changed without recompilation. On the other end of the spectrum, dynamic instrumentation allows deciding what data is collected while an application is running instead of during development. Moreover, dynamic tracepoints can be placed at almost any code location while the locations of static tracepoints are predefined.

Though there is no clear understanding of the dynamic instrumentation overhead, it is higher than static instrumentation when the tracepoint is active. This is because of the slow speed of the interrupt instruction and because of the CPU having to jump to a new, unpredictable location, slightly reducing the effectiveness of the CPU instruction prefetcher and CPU caches. Further, the jumping cost can increase in certain instruction cache issues. On the other hand, when tracing is disabled, dynamic tracepoints do not exist in the code and cannot incur any overhead. However, even when disabled, static tracing requires at least one comparison and branch to determine whether the tracepoint is active. This pollutes instruction caches with instructions that are never actually executed.

Dynamic instrumentation in production environments is also challenging from a maintenance perspective because many code versions and patches of the same code are deployed at various customer sites. Even minor code modifications then result in a unique list of dynamic tracepoints for each code version. This is because each version may change the offsets of functions, their names, arguments, or other variables.

Our discussions with GPFS's engineers showed that they prefer static tracing over dynamic tracing [196]. They argued that maintaining per-build lists of tracepoints is an overwhelming and time-consuming task. This problem is not so acute with static tracing since the tracepoints are part of the code itself. Further, the overhead of dynamic instrumentation is seen as too high. This preference towards static tracing seems to be shared among storage developers as other parallel file systems used in production environments, e.g., Lustre [30] and Ceph [267], are also using static tracepoints [@93, @143].

3.3 Tracing in GPFS

GPFS's tracing framework has evolved in its lifetime, incorporating a highly optimized tracing mechanism with over 20K tracepoints positioned at strategic code locations. All tracepoints are grouped into 62 sets, called *subsystems*, which describe

the tracepoint's context and software layer it is positioned in, such as locking operations or GPFS's virtual file system layer. To describe each tracepoint's importance, each is assigned a *priority level* (1 to 14). Smaller values indicate a higher importance for a tracepoint.

When users trace GPFS, they must first set the priority level for each subsystem they are interested in. The user then enables tracing per file system node, which collects the traces only for its own operations. Note that specifying a subsystem for a higher priority level value, that is, tracepoints of lower importance, also enables all tracepoints with a lower priority level value. Therefore, it is not possible to only enable tracepoints with a priority level `10`, for instance, without also enabling all tracepoints from level `1` to `9`.

Engineers set the assigned subsystem and priority level of each tracepoint during development and, hence, before compilation and shipment of the product. That is to say, a tracepoint and its context (subsystem) reflect an engineer's special interest in the particular location of the tracepoint and the information it emits. Such sets were introduced in the first place since CPU, memory, I/O overheads, let alone the space consumption of traces, are too high and greatly disrupt an application when over 20K tracepoints are enabled.

Today, GPFS offers two tracing modes: *blocking* and *overwrite*. In both modes, GPFS stores the trace records in an in-memory buffer before persisting them as a trace file (containing all trace records); one for each file system node. By default, GPFS uses the blocking mode where, with tracing being enabled, all trace records are written from the in-memory buffer (configurable between 1 MiB and 64 MiB) to the trace file. However, when the trace collection rate is higher than the throughput capabilities of the underlying storage device where the traces are written to, GPFS blocks until enough buffer space is flushed to accommodate incoming trace records. Moreover, if the maximum allowed trace file size (set by the user) has been exceeded, GPFS discards the oldest written trace records to keep new trace records.

Because of the known overhead of the blocking mode, overwrite mode was added to GPFS later on. In contrast to the blocking mode, the trace records are moved to persistent storage only when tracing is stopped. In case more traces are collected than can fit in the in-memory buffer while tracing is running, GPFS overwrites the oldest trace records in the *circular buffer*. Over the years, several optimizations were added to the overwrite mode, such as lock-free memory management, which makes it more efficient than the blocking mode. In summary, while the overwrite mode causes less performance degradation than the blocking mode, the trace file size in the latter is usually larger than the circular buffer in the overwrite mode, allowing

```
1 # TIMESTAMP  PID     SUBSYSTEM      TRACE_MESSAGE
2   0.000785   11281 TRACE_VNODE: gpfs_i_lookup enter: diP 0xFFFF880B28458000
          dentryP 0xFFFF880B60DC0840 name 'someFilename' d_count 1
```

Figure 3.1.: A human-readable trace example for entering the file lookup function, containing the trace collection timestamp (relative to when tracing was started), the thread ID, the tracepoint's subsystem, and the trace message, containing a constant template text and the event-specific parameters (in **bold**).

fewer traces to be recorded in total. However, increasing the circular buffer size to capture more traces makes a significant memory portion unusable for the running application.

For the in-memory buffer, the trace records are kept in a compressed binary form. In particular, each trace record contains a timestamp, the collecting thread id, an integer *tracepoint id*, and the *trace data*, containing only event-specific parameters (variable values) without the constant text added by the developer. In the blocking mode, GPFS constantly compresses and moves the binary trace records to the underlying storage while the overwrite mode keeps them compressed in the circular buffer. Overall, the compression reduces the trace data needed to be flushed to the disk and, thus, reduces the frequency the blocking mode blocks the application. In the overwrite mode, compression allows a higher number of trace records to be collected in the circular buffer. When tracing is stopped, the compressed binary records are converted into human-readable text files. For this, GPFS merges the trace data (containing the captured variable values) of each trace record with the constant text templates of each tracepoint, resulting in the final *trace message*. Figure 3.1 provides an example excerpt of a single trace record from the final human-readable trace file when entering the function to look up a file in GPFS. Here, the trace message (starting with `gpfs_i_lookup`) of a tracepoint contains the memory address to the inode of the current directory, the memory address to the directory entry of the file, and the name of the file itself as the event-specific parameters. The non-bold portions of the trace message are the tracepoints' constant template text.

In the code, all tracepoints are added via preprocessor directives to avoid the overhead of a function invocation in case the tracepoint is disabled. Before each trace is collected, GPFS first checks if tracing is enabled with a single global variable. This global check reduces the overhead of further conditional checks when tracing is disabled, which is the most common situation. When tracing is enabled, additional checks test if the tracepoint should be collected with regards to the tracing configuration and the tracepoint's priority level and subsystem. If the tracepoint is collected, the function generates the trace record that is then added to the buffer.

3.4 Challenges in tracing in GPFS

While GPFS's trace collection mechanism is highly optimized, the fundamental logic behind what tracepoints to collect and in which situations causes several serious challenges that reduce the efficiency of these optimizations. This section will explore these challenges by investigating the tracepoint distribution across subsystems and by evaluating the trace collection overhead.

3.4.1 Tracepoint distribution

As introduced earlier, GPFS categorizes its tracepoints into subsystems and priority levels. The silent assumptions behind this simple and commonly used categorization policy is that 1. some tracepoints are more important than others, 2. each tracepoint belongs to only one subsystem, and 3. developers make the right decisions when assigning the subsystem and its priority to a new tracepoint. Although seemingly uncontroversial, we found that these assumptions do not always hold for products that have been developed for a long time, like GPFS.

GPFS offers two *tracing layers*, `default` and `io`, which are presets of specific subsystem and priority level configurations. The `default` layer enables over 14K tracepoints across all subsystems at varying priority levels with the goal of collecting tracing data that will be useful in most situations. However, this kind of detailed tracing data comes at the cost of data volume and performance overhead (see the next section). The `io` layer, on the other hand, only enables around 900 tracepoints of selected subsystems with high priorities to gain a basic understanding of the system's I/O behavior. These two layers are often used in various situations, e.g., bug analysis, performance diagnosis, and workload characterization during development and in production environments.

In the experience of GPFS's developers [196], the above-described categorization and layers work well in use cases where diagnostic data has to be collected for previously unknown bugs, performance issues, or workloads. Nevertheless, depending on the situation, most of the collected trace records (enabled by the `default` layer, for instance) may not be necessary for a given issue. But, this can also not be avoided since it is impossible to know beforehand which trace records are actually required to solve an issue. In other use cases where the exact tracepoints are known, GPFS's tracing categorization can result in many unnecessary *byproduct* trace records. In these cases, it is already known a priori that the byproduct traces are irrelevant for a

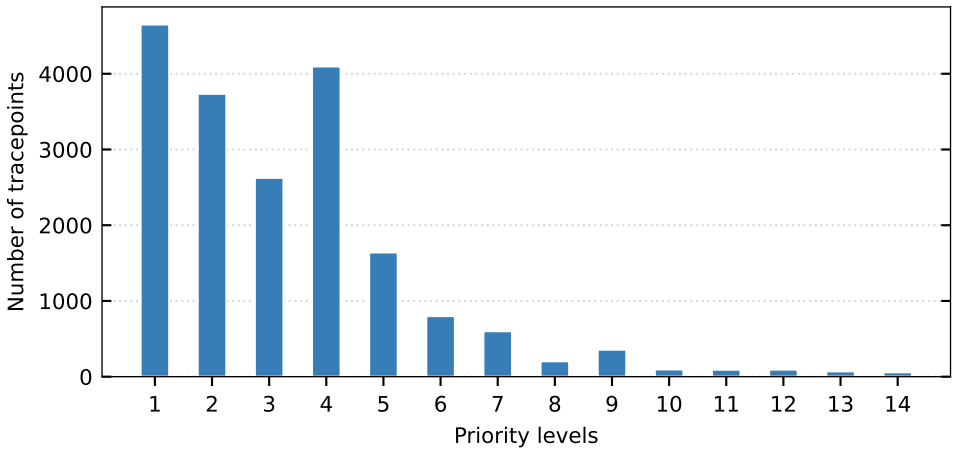


Figure 3.2.: The tracepoint distribution (y-axis) across priority levels 1 – 14 for all subsystems (x-axis). The smaller the priority level value of a tracepoint the more important it is.

given problem, but they can also not be disabled due to the static subsystem and priority interface.

Tracepoint distribution according to priority levels Figure 3.2 presents the tracepoint distribution across the priority levels in GPFS without taking their subsystems into account. As can be derived from the non-uniform distribution, developers tend to place many tracepoints in levels `1` to `4` and almost don't use levels `6` to `14` at all. This is because the `default` tracing layer sets the default priority to `4` for most subsystems. Developers are aware of the `default` layer's configuration and, therefore, usually assign new tracepoints a high priority level so that its trace record is collected in a typical deployment. In other words, for the developers, the specific priority level of a tracepoint is irrelevant, and their decision is mainly driven by two configurations: within the `default` layer or not. Naturally, most developers choose to have their tracepoints collected by default.

This behavior can be seen in Figure 3.2 as the `default` level of all subsystems covers almost 75% of all tracepoints. The corresponding overhead in all of these enabled tracepoints can reduce the system's performance in terms of runtime by up to 95% compared to an environment where all traces are disabled (see the following section). To slightly mitigate the performance overhead, the `default` levels are periodically re-evaluated. However, this is usually an inefficient process since it is not clear which priority level a tracepoint should be assigned long after creation, especially in highly complex code paths.

Whenever a new feature is developed, developers often add more tracepoints to the `default` level than required in the final version. While the code evolves during the software development cycle, the tracepoint's priority levels are re-evaluated and lowered in their importance. However, at the same time, developers must ensure not to lose valuable trace information when important tracepoints are no longer part of the `default` layer, which may affect the ability to analyze an issue later on. Deciding which tracepoints can be de-prioritized is difficult due to two reasons: First, each developer has a different interpretation and vision of a tracepoint's importance. Second, each tracepoint on its own only adds a relatively small incremental benefit in reducing total performance overhead and tracing volume compared to the risk of not being able to access important information during the later trace analysis. Thus, developers are cautious when de-prioritizing tracepoints out of the `default` layer due to the risk of picking the wrong tracepoints, resulting in more tracepoints in the `default` layer than may be required.

Tracepoint distribution across subsystems Further challenges arise from the tracepoint distribution across the subsystems. In general, each subsystem is associated with a specific mechanism (e.g., locking or *remote process communication*, RPCs) or a file system layer (e.g., GPFS's VFS layer). Over the years, some areas of GPFS's codebase became larger in scope and complexity, resulting in an uneven distribution of tracepoints. Nevertheless, the number of tracepoints per subsystem is also influenced by the developers, who may have varying strategies when adding tracepoints because it is often unclear to which subsystem a tracepoint should belong. Figure 3.3 shows how many tracepoints are part of each subsystem, visualized on a logarithmic scale. The distribution is grossly uneven, with some subsystems containing fewer than 10 tracepoints while others (e.g., `FS`) contain almost 3,000 tracepoints. Consequently, it is not clear to developers and users how many tracepoints are actually enabled by a subsystem, and hence it is impossible to predict its performance implications. Moreover, some code overlaps between different parts of GPFS, making it challenging to decide which subsystem a tracepoint should belong to. If the wrong subsystem is chosen, a whole subsystem might need to be enabled only to capture a few tracepoints in a specific code segment, unnecessarily increasing the trace output size further.

We conclude that statically assigning tracepoints to a subsystem does not work well in projects where many developers work on a large, ever-increasing code base. Further, each developer might have a different understanding of which tracepoints should belong to which subsystem and at which priority level. The following section will

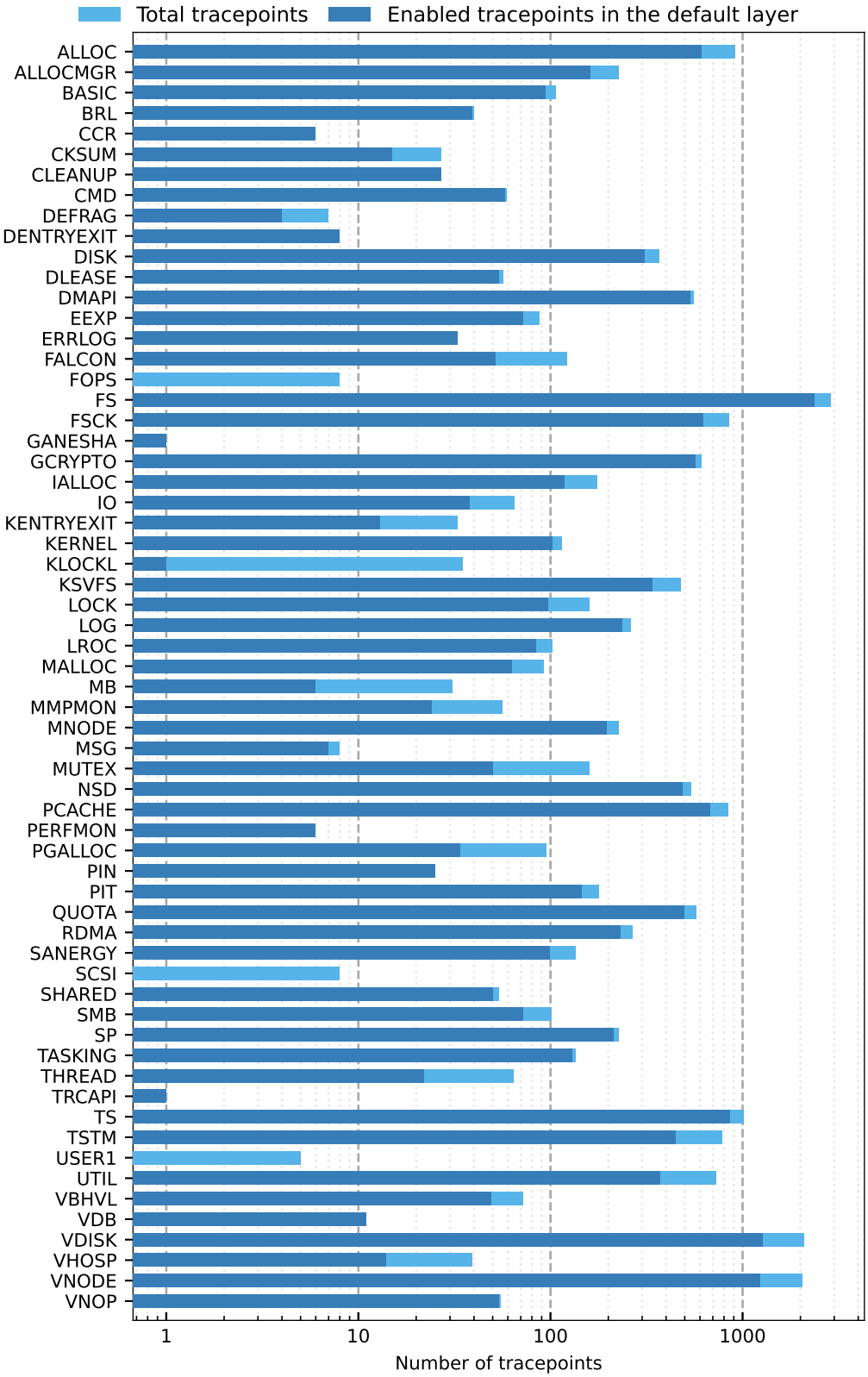


Figure 3.3.: The distribution of all and enabled-by-default tracepoints across subsystems (y-axis) on a logarithmic scale (x-axis).

discuss and evaluate the corresponding performance implications of static tracing in GPFS.

3.4.2 Trace collection overhead

In general, depending on the invocation rate, each tracepoint adds some amount of CPU, memory, and I/O overhead, while the latter only occurs in the blocking mode. In practice, users and developers often only need to enable a few tracepoints, but they are usually scattered among subsystems and priority levels. As discussed in the previous section, collecting a few tracepoints of interest leads to a large amount of byproduct traces that are then discarded during post-processing. As a result, gigabytes of potential byproduct trace records are collected, causing unnecessary overhead while getting increasingly worse as the number of subsystems increases. GPFS cluster administrators are aware of these overheads and are reluctant to enable tracing.

Test setup For our experiments, we used two IBM System x3650 M4 machines equipped with two Intel Xeon E5-2650 v2 CPUs each. Each CPU has 8 cores @ 2.6 GHz with two hyper-threads per core. Each system includes 96 GiB of main memory and one 10 Gb Ethernet Mellanox card connected to a 10 Gb Force 10 S2410 Ethernet switch. Further, each machine uses an internal ServeRAID M5110e SATA/SAS controller with a 15,000 RPM SAS Enterprise HDD of 146 GiB capacity and a single Intel SSD of 120 GiB capacity connected to it.

We will demonstrate the tracing-induced overhead by simulating a common HPC metadata-intensive workload using the well-known *mdtest* benchmark [@106]. In the experiments, mdtest performs concurrent file creation in a single directory, creating two million zero-byte files from two file system clients. We selected this workload for two reasons: First, concurrent file creation is a common workload in many HPC applications and, second, it is one of the most difficult workloads for a parallel file system to perform efficiently. Section 2.3.2 explained these two topics, elaborating on the metadata challenges and their corresponding operations, which are major factors in the application I/O throughput in supercomputer environments. Because of the workload's complexity and commonness, they are frequently traced in various production deployments [196]. Moreover, metadata operations are usually in the kilobyte range, requiring extensive synchronization and logging mechanisms. Therefore, the time is mainly spent within the file system, triggering significantly

more tracepoints compared to data-intensive workloads where the time is spent waiting for I/O operations to complete [251].

The experiments to investigate the tracing overhead were run on two nodes. We only used two nodes for two reasons: First, because each node only traces its own execution, there is little change in additional trace records when the number of nodes is increased. In essence, the number of triggered tracepoints scales almost linearly with the number of nodes, and there is no fixed amount of fewer tracepoints collected by using more nodes in the same workload. In fact, the opposite can be the case as the network communication increases with more file system nodes, resulting in more triggered tracepoints overall. Second, we require at least a two-node configuration as a single node cluster would not need to handle inter-node and node-local locking of file system resources.

The experiments were run in two configurations regarding the underlying storage: RAM disk and SATA SSD. We decided on this configuration as faster storage technologies are expected to become more common in storage systems in the future, potentially replaced by *persistent memory* technologies [24, 90, 160]. Already today, many parallel file system installations are using SSDs for storing metadata, while the newest supercomputing clusters are incorporating dedicated NVMe-based burst buffers [@179]. Further, these configurations increase the number of IOPS that the file system can perform, causing more tracepoints to be triggered in general. Therefore, we target these methods which especially require efficient tracing mechanisms.

We ran the experiments with tracing in the blocking mode, with tracing in the overwrite mode, and without tracing at all. Experiments with enabled tracing were evaluated in two settings, using 1. different priority levels for the `file system` (`FS`) subsystem, and 2. using the `default` tracing layer. We used the `FS` subsystem as it has the highest number of tracepoints.

Notice that the blocking mode writes its in-memory buffer periodically to the disk, whereas the overwrite mode keeps it in memory. Thus, the blocking mode's performance depends on the underlying storage device. For storing the trace files, a single enterprise-grade HDD per node was used. This is a common setup among GPFS users, who are often reluctant to invest in additional equipment just for trace collection [196].

Evaluation Figure 3.4 visualizes different levels of tracing impact on file creates per second. In all cases, the relative standard deviation was less than 5%. The

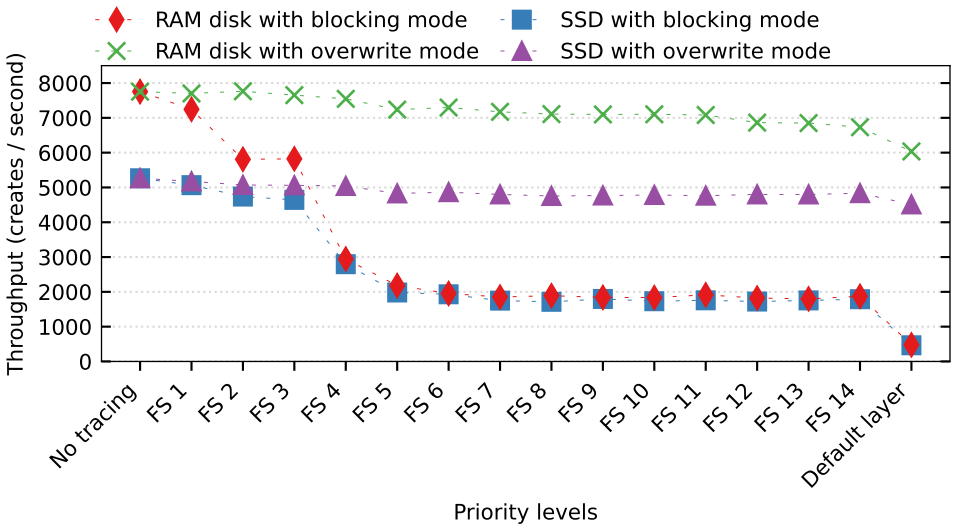


Figure 3.4.: The average performance of concurrently creating two million files in a single directory with various tracing settings (x-axis): `no tracing`, `FS` (file system) subsystem in all levels, and the `default layer`, in four tracing modes.

in-memory trace buffer was set to the default (4 MiB) in blocking mode and 4 GiB in overwrite mode. For both SSD and RAM disks, the blocking mode's overhead is severe, causing a performance drop by more than 60% when the priority level 4 was used compared with the no-tracing case. With higher priority levels (less importance), the performance remains similar because most tracepoints are located in levels 1 – 4. For the `default` layer, which enables the tracepoints of all subsystems, the performance drops by almost 95%. We also ran the experiments with all 20K+ tracepoints enabled. However, GPFS detected hanging nodes during our experiments rendering the results invalid, and they are therefore not included in Figure 3.4.

Due to the blocking modes overhead, traces can also be of limited use for performance analyses and diagnosing problems, e.g., caused by race conditions. This is because the trace collection overhead can change the system behavior, resulting in false conclusions. GPFS developers have encountered such situations numerous times over the past 20 years [196]. In the overwrite mode, on the other hand, the trace collection overhead is significantly less, with only 10% performance degradation in the RAM disk case. Nevertheless, the overwrite mode requires expensive main memory to be dedicated for the in-memory buffer, and thus a significantly smaller number of events can be traced in a single execution. For example, with 8 GiB used for tracing buffers in total on both nodes, only about 16% of the overall most recent traces could be kept when two million files were created with the `default` layer.

Subsystem	Compressed (GiB)	Uncompressed (GiB)	Trace records (millions)
FS 1	2.1	19.2	179.6
FS 2	4.4	40.4	395.4
FS 3	4.8	41.6	413.3
FS 4	8.5	80.1	747.2
FS 5	12.4	109.1	1,021.1
Default layer	88.1	458.1	4,032.7

Table 3.1.: The amount of data generated when four million files are concurrently created in a single directory on two nodes for different subsystem configurations.

Consequently, the large trace size leads to the next challenge. These experiments used the overwrite mode and concurrently created four million zero-byte files on two nodes which ran about ten minutes in all cases. Table 3.1 lists the human-readable trace file sizes for the compressed (by `gzip`) and uncompressed states, in addition to the overall number of trace records. The table shows the numbers for the first five `FS` subsystem priority levels and the `default` layer. Higher priority levels were omitted as fewer additional tracepoints were assigned from levels 5–14, and therefore the trace file sizes remained similar. For the `default` layer, 10 minutes produced about 460 GiB of uncompressed traces, corresponding to over 4 billion trace records. Note that because each node is only tracing its own execution, the amount of generated traces grows almost linearly with the number of file system nodes. Therefore, in a much larger setup (some users have more than 10K file system clients), the generated trace data becomes practically intractable.

We conclude that the blocking mode suffers from severe performance degradation, making many analyses, such as time-related bugs, hard to reproduce. However, other bugs may also be more likely to occur because artificial tracing delays can cause unexpected file system behavior. On the other hand, while the overwrite mode causes considerably less performance degradation than the blocking mode, there might not be enough data captured for trace analysis due to the large trace sizes. Yet, both modes are essential tools on a daily basis but suffer from the limitations of the subsystem-and-priority classification scheme in many use cases.

3.5 Design of FlexTrace

We present *FlexTrace*, a new tracing interface to overcome the challenges presented in the previous section. It is built on top of and extends the existing GPFS tracing interface, allowing a wider use of tracing for production systems and, as a result, better

insights for developers and users. Although GPFS and FlexTrace are closed-source implementations, FlexTrace's conceptual and implementation simplicity makes its design easily adaptable to other storage systems.

FlexTrace aims for the following use cases where GPFS's subsystem-and-priority classification has shown not to work well [196]: 1. diagnosing known performance issues, bugs, or workloads by reducing the performance degradation and the data volume caused by byproduct traces, 2. understanding complex code behavior, and 3. verifying recently implemented code during development. Moreover, FlexTrace should eliminate situations that required special tracing builds for customers to identify a specific problem. Finally, FlexTrace is combined with GPFS's overwrite mode, which already shows low trace collection overhead, to make always-on tracing possible in many more situations, allowing for a more straightforward diagnosis at customer sites.

The next sections will describe FlexTrace's goals, including its requirements to seamlessly integrate into GPFS's existing tracing framework, and its implementation.

3.5.1 Goals

We define the following goals for the tracing improvements to GPFS:

Functionality With FlexTrace, both GPFS developers and users should have fine-grained control over which tracepoints to enable to ensure minimal trace overhead and byproduct traces.

Backward compatibility GPFS users, which include developers, administrators, and testers, are accustomed to today's tracing API and CLI commands and, in many cases, have built upon it via management frameworks or other scripts. Breaking the API or semantics would require modifications of such software, and it could cause disruptions of deployment upon upgrade. Therefore, the existing trace-related APIs and CLIs should still be functional for backward compatibility. Further, the trace record format should not change so that analysis tools still work as expected with FlexTrace.

Minimal code changes With over 20K tracepoints in GPFS's code, it is hard to estimate the impact of even small modifications to the tracing framework concerning the entire system behavior. Changes could cause unexpectedly high performance degradation and increased resource utilization, especially when operating on a highly optimized tracing framework. Developers, therefore, would be reluctant and naturally alarmed if the tracing framework were to be heavily modified. Consequently, our initial goal is to limit the number of changes and avoid modifying each source file that contains a tracepoint.

Low overhead and increased system performance With FlexTrace's fine-grained control over which tracepoints to enable, this, in turn, reduces the amount of collected byproduct traces significantly (depending on the use case). Thus, decreasing the amount of byproduct traces would increase the overall system performance when tracing is enabled. However, the GPFS developers were concerned that giving users such fine-grained control would increase the overhead and decrease tracing performance, especially when most tracepoints are disabled. This is because, for each encountered tracepoint, the executing thread must check if the specific tracepoint is enabled. In addition to executing more instructions, GPFS and applications might encounter an increase in CPU cache misses with an increasing number of tracepoints. Thus, our goal is to keep the overhead and performance degradation as low as possible.

Improved usability Allowing complete fine-grained control over 20K+ tracepoints raises several usability challenges. First, a numeric (non-descriptive) identifier would make the tracing interface difficult to use for both users and developers. Second, and even more fundamental, it is not clear how users would be supposed to know which tracepoints to enable. Developers could consult the source code, but it can be time-consuming, especially when a developer is not familiar with the entire code base. This is not an option for users because they do not have access to the proprietary source code. Therefore, our goal is to make FlexTrace's interface easy to use for developers with various levels of experience as well as for users on a deployed system.

3.5.2 Implementation

In the remainder of this chapter, the unmodified GPFS tracing version is referred to as *vanilla*. To achieve the goal of allowing flexible and fine-grained control

over all tracepoints, FlexTrace builds on GPFS's vanilla tracing framework and extends it. Though we initially considered dynamic instrumentation for FlexTrace, its performance and management drawbacks (see Section 3.2.1) counter our goals. Also, because the GPFS source is not publicly available, dynamic instrumentation would be of limited use to users while also requiring extensive code modifications. As a result, FlexTrace builds on GPFS's static instrumentation mechanism. To not break existing software that analyzes trace records, we did not change the trace record format.

FlexTrace uses a bitmap to store the tracepoint state, that is, if it is enabled or disabled, for all 20K+ tracepoints. Vanilla GPFS already maintains a unique (integer) identifier for each tracepoint to efficiently transform a binary trace to its textual form (see Section 3.3). FlexTrace reuses these identifiers as bitmap indices to reduce the amount of newly added code. Although all tracepoints would ideally only need 2.4 KiB for the bitmap, the tracepoint IDs are actually sparsely distributed across 65K indices, making FlexTrace's bitmap 8 KiB in size. The bitmap size is important and should be as low as possible so that it still fits in the CPU cache while leaving enough space for other code and data. Generally, the L1 and L2 CPU caches per core can contain 64 KiB (32 KiB instruction cache and 32 KiB data cache) and 256 KiB in size, respectively.

To control which tracepoints are enabled, the existing GPFS trace command now also accepts a list of tracepoint IDs. However, because tracepoint IDs on their own are non-descriptive and meaningless for users, FlexTrace leverages the already existing preprocessor names each tracepoint is associated with. The preprocessor names are typically very descriptive, e.g., the `ACCESSDIRENTRY_EXIT` tracepoint captures an exit from the function that processes access to a directory entry. Therefore, users can use the preprocessor name in place of the ID[1]. Further, FlexTrace extends the trace command to allow listing all tracepoints with IDs and names.

When tracing is disabled, a single global variable is used to check if at least one tracepoint is enabled, which is identical to the vanilla tracing mechanism. As a result, FlexTrace has no performance impact when tracing is disabled because the bitmap is never accessed. Moreover, FlexTrace's individual tracepoints are independent of one another, which is in contrast to the vanilla subsystem-and-priority classification, where tracepoints of a lower importance level would also enable all tracepoints of the same subsystem with a higher importance level.

When at least one tracepoint is enabled, FlexTrace checks whether each of the corresponding bits for each encountered tracepoint is enabled. This could cause

[1]Note that a corresponding list of preprocessor names would need to be publicly available.

more overhead than in the vanilla case, which only checks the subsystem and priority levels for a given tracepoint. In the vanilla case, this is a lightweight operation as all priority levels fit into just 64 bytes (one byte per subsystem) and, hence, into a single cache line[2]. FlexTrace's bitmap might therefore decrease GPFS performance, but, as we will show in Section 3.6.3 later, this is not the case.

Because manually activating a large number of individual tracepoints is a tedious task, FlexTrace introduces a new type of tracepoint sets: *tracing profiles*. Profiles contain a list of tracepoints that should be enabled, and, contrary to static subsystems, they are flexible and easily modifiable after GPFS is built and deployed at the customer's site. In many ways, trace profiles offer the flexibility of dynamic tracing without its overhead when enabled. Therefore, the GPFS trace command also accepts tracing profiles next to tracepoint names and IDs.

In general, tracing profiles allow users to define their own tracepoint sets, tailored to their specific use case. Nevertheless, we do not expect users, that is, customers, to extensively use the tracing profiles directly. As GPFS's developers are already supporting customers today, tracing profiles can help developers create custom tracing profiles to analyze a particular issue they face rather than creating custom GPFS builds. Also, this allows developers to create tracing profiles to debug issues of applications' I/O workloads. For example, I/O workload profiles can help understand an I/O workload's used I/O sizes, randomness, or read vs. write balance, among others. In the past, GPFS had numerous customers who could rewrite their application to be more efficient after they understood that their I/O workload was not suitable for the storage system they were using [196]. In summary, flexible tracing profiles can help developers build more efficient applications, while GPFS developers can easily provide them with the appropriate tracing profiles to enable the necessary tracepoints. In addition, FlexTrace can help to debug problems application developers face by being able to achieve a more detailed understanding of an I/O workload through rich trace-like information.

To present an example of how tracing profiles can be used, we defined a special `create-analysis` profile to analyze concurrent file create performance in a single directory, a common workload and one of the most difficult for a parallel file system to handle efficiently (see Section 2.3.2 for details). Optimizing this complex workload demands a great understanding of many code paths, which would require the enablement of many trace subsystems in vanilla GPFS to catch all necessary information. Considering the performance degradation, which can hide bottlenecks

[2]The CPU reads and writes the entire cache line whenever any location is accessed within the 64 byte region.

when tracing is enabled, and the large trace file sizes of vanilla GPFS tracing, this is a suitable workload candidate to demonstrate FlexTrace's advantages. The create-analysis tracing profile and the corresponding analysis script not only help understand GPFS's behavior but also profiles the time GPFS spends in various stages of the file create process.

Further, for the sake of backward compatibility, additional tracing profiles were developed for the tracepoints of existing subsystems and their priority levels, as well as for the `default` and `io` layers. For instance, the `IO-5` profile is equivalent to the I/O subsystem with the priority level `5` (without FlexTrace). As in vanilla GPFS, this enables all tracepoints for the I/O subsystem with priorities `1` to `5`. In the long run, we expect that a library of profiles grows over time suited to trace a wide variety of issues or gain a deeper understanding of various workloads.

3.6 Evaluation of FlexTrace

This section will compare FlexTrace's impact on GPFS's subsystem-and-priority tracepoint classification system. First, we will present the experimental setup and used benchmarking tools. Then, Section 3.6.2 will evaluate the overhead of GPFS's static instrumentation when tracing is disabled and investigate if a single tracepoint can have a measurable impact on the overall file system performance. These experiments mainly aim to determine if small parts of a big system can be continuously traced in production environments. Finally, Section 3.6.3 will empirically check if and to what extent FlexTrace's bitmap (of 8 KiB) pollutes the CPU caches, potentially degrading the file system's and application's performance.

3.6.1 Test setup

We used the same workload and test setup for performing the experiments as presented earlier in Section 3.4.2, i.e., mdtest [@106], to concurrently create two million zero-byte files in a single directory of a GPFS file system on two file system nodes. If not mentioned otherwise, all experiments were run at least 10 times on 2 nodes with 16 used processes each. Both nodes used SSDs and RAM disks for GPFS's underlying storage. The performance is presented as the average throughput, i.e., file creates per second, over ten iterations.

Further, when indicated, the *cache* benchmark of the *lmbench* CPU benchmark suite [@135] polluted and stressed the CPU caches on the test system. To investigate

the effects of FlexTrace's large bitmap on the CPU, lmbench was run alongside mdtest, pinned to the same CPU cores. We monitored the overhead caused by the two benchmarks competing for L1 and L2 CPU caches with the commonly used *Linux perf* [@133] application. Each core has a 32 KiB data and a 32 KiB instruction cache for the L1 cache and 256 KiB for the L2 cache. The L3 cache with a size of 20 MiB is shared among all cores.

3.6.2 Static tracing and single tracepoints

Even if all tracepoints are disabled, static tracing requires at least one conditional check per tracepoint, causing some amount of overhead. This section will investigate the overhead of the 20K+ statically placed tracepoints when all of them are disabled in the vanilla GPFS case (not yet including FlexTrace). Therefore, the performance of two GPFS binaries is compared: One without any code modifications and one with completely removed (compiled-out) tracepoints, which decreased the binary size by ∼5%.

The results showed that statically placed tracepoints could decrease the create throughput by up to 3% in the two-node experiments for both SSD and RAM disk cases. However, it is important to note that the level of performance degradation also depends on the number of encountered tracepoints in the executed code paths. Thus, fewer tracepoints cause fewer conditional checks if a tracepoint is enabled and less performance degradation in general. For instance, the I/O code path contains few tracepoints and is, as a result, not noticeably affected by tracing. This is in addition to the delays when performing I/O in which the executing thread has to wait for the disk operation to complete. Analysis tools, e.g., *iostat* [@131], refer to this waiting time as `iowait`.

However, the impact of disabled tracepoints can also be higher with more tracepoints in the code path and with more IOPS being executed. If the experiments are run only on a single node, for example, global locking mechanisms and inter-node communication are not required. This results in a file create throughput increase of ∼30% and ∼50% for the RAM disk and SSD cases, respectively. In turn, the increased throughput rate then leads to more encountered tracepoints per second. In this scenario, adding static (and disabled) tracepoints decrease performance by up to 9% for the RAM disks and 5% for the SSDs (without generating a single trace record) compared with the GPFS binary with completely removed tracepoints.

Next, we evaluated the impact of a single enabled tracepoint that is hardcoded into the code without requiring a conditional check. Such *hardcoded tracepoints*

are always enabled and cannot be dynamically disabled through a data structure like a subsystem or a bitmap. This experiment aims at understanding the overall performance degradation once FlexTrace's bitmap is introduced later to check if a tracepoint is active. As a result, all tracepoints except a single one were removed, which is emitted only *once* for each created file.

After performing and measuring several file create experiments on one and two nodes with both SSD or RAM disk as storage, there is no statistical difference in the average file create throughput for a single hardcoded tracepoint in the overwrite or the blocking mode compared with disabled tracing. This shows the insignificance of collecting single tracepoints on the overall system performance, although such metadata workloads cause a high frequency of tracepoints even if just a single tracepoint is enabled. With this information in mind, all other measured overhead (if any) can be attributed to FlexTrace's bitmap when it is used to control the tracepoints.

3.6.3 Bitmap

This section will discuss the impact of FlexTrace's bitmap, which stores each tracepoint's state, on the performance of the file create workloads. Further, we will clarify the concerns raised by the developers (see Section 3.5.1) with regards to GPFS's performance when CPU-cache-intensive applications are running because parts of the bitmap are forced into the cache when at least one tracepoint is enabled. When tracing is globally disabled via a corresponding variable, the bitmap is not accessed and does not incur any overhead. Moreover, note that the number of tracepoints does not impact the bitmap performance because the bitmap size remains constant, and the same number of indexes needs to be accessed in all cases (disabled or enabled tracepoints).

Performance comparison Figure 3.5 shows the performance in create throughput for various tracing experiments on the x-axis. They are categorized into four groups depending on whether a RAM disk or SSD was used as underlying storage and differentiate each between the blocking and the overwrite tracing modes. Every experiment was run at least ten times. The relative standard deviation was less than 5% in all cases unless otherwise indicated. Each group describes the performance of the following four scenarios: (`VANILLA_NO_TRACE`) vanilla with disabled tracing, (`VANILLA_FS_4`) vanilla with enabled tracing of the `FS` subsystem at the `default` priority level `4`, (`VANILLA_DEFAULT`) vanilla with enabled tracing of all subsystems

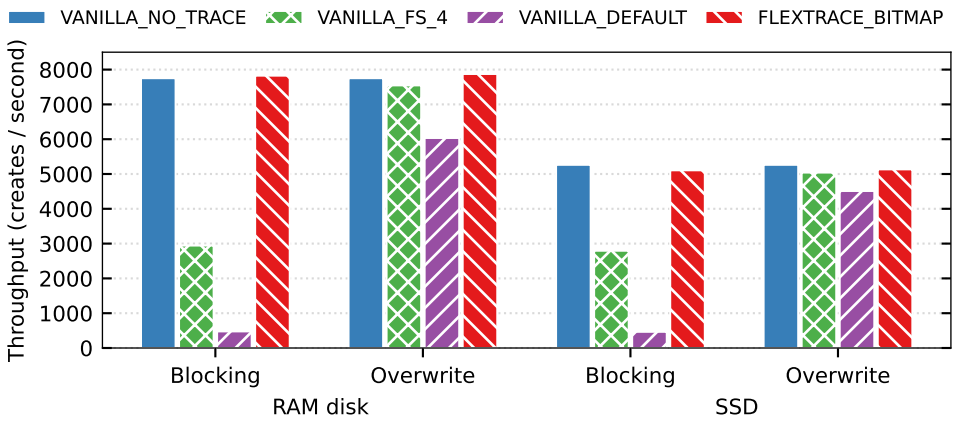


Figure 3.5.: A throughput comparison (y-axis) for concurrently creating files in four scenarios (x-axis).

with `default` priority levels (`default` layer), and (`FLEXTRACE_BITMAP`) FlexTrace with a bitmap and four enabled tracepoints.

In the vanilla case, the four specific enabled tracepoints in FlexTrace's bitmap are assigned to different subsystems with a low priority level (high importance level). Enabling the same four tracepoints via the subsystem-and-priority classification in vanilla GPFS would activate over 3K tracepoints, adding measurable overhead and a substantially increased trace data volume. We selected these tracepoints because our experimental workload triggers them for each file creation and because they were used in other projects involved with latency analysis of file create performance in GPFS [251].

As previously discussed in Section 3.4.2, the create throughput is greatly impaired by tracing in the blocking mode, especially if it involves the `default` layer, resulting in a slowdown of up to 95%. This type of overhead is also visualized in Figure 3.5 for both storage device cases (see the `VANILLA_FS_4` and `VANILLA_DEFAULT` scenarios). For the overwrite mode, the performance differences are not as serious but still up to 20% compared with disabled tracing (`VANILLA_NO_TRACE`). Nevertheless, the in-memory buffer of the overwrite mode can be exceeded in a matter of seconds, depending on the buffer size, the individual trace record size, and the tracepoint trigger frequency. In other words, the longer the trace collection period and the smaller the trace buffer, the less frequent trace information can be kept.

For the FlexTrace scenario using the bitmap data structure (`FLEXTRACE_BITMAP`) the performance remains similar within the margin of error in all cases. Moreover, there is no measurable overhead compared with the disabled tracing case (`VANILLA_NO_TRACE`). This shows that FlexTrace's bitmap does not add a measurable

overhead in our experiments, although the file system must access the bitmap index for each encountered tracepoint in the code path, regardless of its state. In fact, FlexTrace's results, in combination with the overwrite mode, are the most promising for cases where traces should be continuously collected. In this context, it is essential that trace collection does not slow down the application and does not alter its behavior.

Effects on the CPU cache When tracing with FlexTrace is enabled, the constant bitmap accesses for each encountered tracepoint force parts of the bitmap into the CPU caches, regardless of an individual tracepoint's state (enabled or disabled). Note that, in this context, disabling tracing is controlled via a single global variable, and the bitmap is not accessed at all when it is disabled. If tracing is enabled, the bitmap is accessed for each tracepoint in all cases. Applications that depend on and heavily use the caches would then compete with the bitmap. This affects its access latencies and can negatively influence the overall tracing performance. Thus, when an application and FlexTrace's bitmap compete for the caches, an increase in cache misses is to be expected. This scenario concerned GPFS developers and in-field experts and is addressed in the following paragraphs.

Several experiments evaluated the cache utilization with FlexTrace's bitmap. To pollute and stress the CPU caches on all available cores, the cache benchmark of the lmbench benchmarking suite was run, which continuously accesses and clears the caches. In addition, similar to previously presented experiments, two million zero-byte files were created in a single directory with mdtest. Both benchmarks have been pinned to the same CPU cores to ensure cache competition. The Linux perf tool monitored the cache miss percentages for all cache references encountered by mdtest when both benchmarks were executed when tracing was enabled and disabled. In addition, to not skew the caching results with executed tracing code beyond the bitmap, all tracepoints were modified so that no trace records are collected when the tracepoint is enabled. Therefore, each tracepoint results in accessing the corresponding index in the bitmap, no matter the tracepoint's state, leaving only the bitmap's potential overhead. In case the cache misses increase when tracing is enabled, the conclusion would be that the bitmap affects the caches and leads to a higher overall cache utilization.

Figure 3.6 shows the percentage of cache misses of all cache references (y-axis) in four scenarios for each storage subsystem (x-axis). The relative standard deviation was less than 10% in all cases. The baseline case is without cache pollution (`NO_CACHE_P`) paired with enabled tracing (`TRACE`) and with disabled tracing

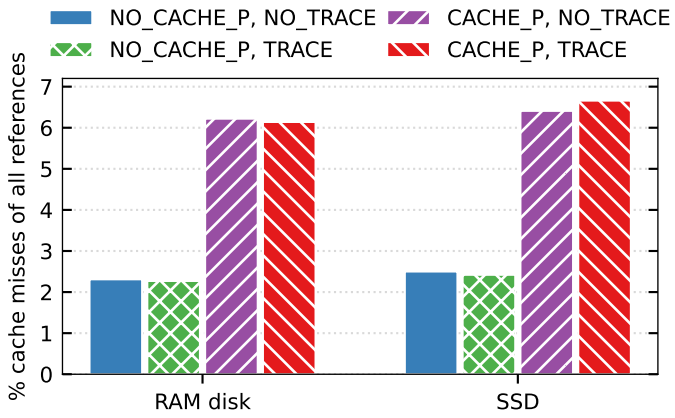


Figure 3.6.: A comparison of the bitmap's effects on the CPU cache by investigating the percentage of cache misses of all cache references (y-axis) when concurrently creating files with and without polluting the CPU caches (x-axis).

(`NO_TRACE`). In both cases, ∼2.5% of all cache references are accounted to cache misses on average. Note that accessing the bitmap does not impact the average creates throughput (see Figure 3.5). When the same workload is run for the disabled tracing case and cache pollution, a slowdown of almost three times could be observed because GPFS now needs to share CPU cycles with the computationally-intensive lmbench. At the same time, the cache misses increase to over 6% on average (`CACHE_P, NO_TRACE`). However, when tracing is additionally enabled, there was no measurable impact outside the margin for error on the overall create throughput or the number of cache misses (`CACHE_P, TRACE`). We therefore conclude that, under the conditions of our experiments, a cache-intensive application does not cause negative effects on the overall cache misses and the performance of FlexTrace's used bitmap.

In this regard, notice that not the entire bitmap is accessed (and therefore pulled into the cache) during this workload. This is because the number of encountered tracepoints depends on how densely they are populated in the executed code path. For instance, the presented workload comes across fewer than 900 different tracepoints per file create. Consequently, this results in the same amount of unique accesses to the corresponding bitmap indices, although the bitmap can contain up to 65K tracepoints. Furthermore, there was no measurable increase in CPU utilization caused by the bitmap in the presented experiments.

3.7 Summary

Tracing is a versatile and powerful tool that users and developers utilize in many use cases on a daily basis. Yet, the rapidly growing trace output makes it difficult to analyze, while enabling tracing can severely impair an application's performance and behavior. This chapter has examined today's tracing challenges from the point of view of GPFS's users and developers. Moreover, it has pointed out various difficulties that occurred during file system development because of its high complexity, long development history, large codebase, and a high number of developers.

This chapter has included several observations, for instance, that the typical approach of splitting tracepoints into static sets (e.g., by their scope and priority) can limit trace scalability and flexibility. Further, we showed that developers tend to put tracepoints around few "default" levels that can lead to high and unpredictable overhead when tracing is enabled. For moving towards a better tracing infrastructure, we have proposed, designed, implemented, and evaluated the FlexTrace research prototype – a new tracing interface that allows users fine-grained control over individual tracepoint activation with little overhead. Based on this interface, FlexTrace introduces the concept of tracing profiles that users can define, tailored for a specific analysis. By decoupling tracepoints from their static subsystems, users have full control over which information to collect, significantly reducing common tracing overheads in real use cases. Finally, because of the low overhead, FlexTrace even makes the continuous collection of few crucial tracepoints a possibility.

The future work covers two directions. First, we would like to conduct a user study to create useful tracing profiles. And second, we would like to explore additional use cases for FlexTrace, including always-on tracing that 1. is able to adjust the number of collected tracepoints automatically based on the amount of currently measured file system overhead, and 2. uses individual tracepoints for end-to-end tracing, allowing developers to understand call paths and timings of various file system operations.

4

GekkoFS: A temporary burst buffer file system for HPC applications

Nowadays, many scientific fields have started to use HPC to process and analyze massive amounts of experimental data. As a result, the HPC system's parallel file system has to cope with new access patterns that were not commonplace in the past. Such access patterns include large numbers of metadata operations, small I/O requests, or random file I/O.

Burst buffer file systems are a way to reduce pressure from the backend parallel file system and to increase an application's I/O performance (see Section 2.3.3 for details). At its core, a burst buffer file system is a separate distributed file system that stores temporary data. They combine the available node-local storage of compute nodes or use dedicated SSD clusters to offer a single global namespace. Therefore, given enough storage capabilities, they can offer a higher peak bandwidth than the backend parallel file system without interfering with it. However, current burst buffer file systems often provide many features that are not necessary for a scientific application that runs in an isolated environment.

This chapter will present *GekkoFS*, a temporary, highly scalable distributed file system that is specifically optimized for the aforementioned use cases. The burst buffer file system can be deployed *ad hoc* in a matter of seconds and provides relaxed POSIX semantics supporting only features that are actually required by most HPC applications. As a result, GekkoFS can provide scalable data and metadata performance, reaching millions of metadata operations per second already at small node numbers, thus substantially outperforming the capabilities of typical parallel file systems.

First, Section 4.1 will motivate and introduce the work in detail. In Section 4.2, we will provide background information and the related work of parallel file systems, burst buffers, and the challenge of metadata scalability in distributed environments. Further, we will discuss how users and applications can interface with file systems and present the benefits and drawbacks of existing approaches in the context of ad

hoc burst buffer file systems. Next, the overall design of GekkoFS will be described in Section 4.3. Section 4.4 will introduce some of the main use cases of such a burst buffer file system and how they can be beneficial. Section 4.5 will evaluate GekkoFS on several supercomputers and various workloads ranging from metadata-intensive and data-intensive workloads to real-world workloads found in distributed deep learning applications. Finally, Section 4.6 will present the future work, and Section 4.7 will conclude this chapter.

4.1 Motivation

Today, new types of applications are used in HPC. Where HPC applications have been primarily compute-bound, large-scale simulations in the past, increasingly more applications are generating, processing, and analyzing massive amounts of data. This trend, also known as *data-driven science*, affects various kinds of scientific fields, with several of them now being able to address many difficult challenges due to newly developed techniques [209, 244].

With the data-driven workloads came new I/O patterns, which imposed new requirements on the HPC file systems [44, 168, 181, 264]. These new I/O patterns differ significantly from past HPC workloads that mainly performed sequential I/O operations on large files. Compared to traditional HPC workloads, data-driven workloads are based on new algorithms and data structures, such as graph databases, including large numbers of metadata operations, data synchronization, non-contiguous and random access patterns, and small I/O requests [52, 168]. Consequently, these I/O patterns can not only slow down data-driven applications but can also heavily disrupt other applications that access the same shared storage system concurrently [60, 243]. This is because parallel file systems were optimized for traditional HPC use cases and workloads, and they are unable to handle these new kinds of workloads efficiently. Data-driven applications accessing their parallel file system can therefore suffer from reduced throughput, prolonged I/O latencies, and long waiting times. Other data-intensive use cases involve checkpoint-restart [19, 68, 191, 192], bulk-synchronous [32], and machine learning [44, 45, 283] workloads.

Software-based approaches aim to support data-driven applications and their access patterns by aligning them more to the abilities of the underlying storage system. This can be done by directly modifying the I/O layer of the application or by using middleware software and high-level libraries (e.g., ADIOS [137], or HDF5 [70]). Nevertheless, adapting software is typically a time-consuming process, hard to

couple with machine learning or big data libraries, or at times, depending on the used data structures and algorithms, not possible.

Hardware-based approaches employ NAND-based solid-state drives (SSDs) instead of magnetic disks (HDDs), which are still used as the primary storage device technology in parallel file systems. Today, many supercomputers use SSDs to achieve a much higher sequential and random access performance [@9, @51, @75, @147, @157, @226, @232]. SSDs can be used to improve metadata performance of the parallel file system as dedicated burst buffers [134] or as *node-local* burst buffers, whereas the node-local burst buffers use SSDs that are directly installed at the compute nodes. To improve file system performance, burst buffers can then be deployed in combination with a dynamic burst buffer file system [19, 265].

In general, burst buffer file systems can increase I/O throughput and latency compared to a parallel file system without modifying the application itself. Existing burst buffer file systems typically support the POSIX I/O and offer the standard semantics and consistencies accepted by most application developers. However, as discussed earlier in Sections 2.3.1 and 2.3.2, enforcing POSIX can severely reduce a file system's peak performance [257], particularly for metadata operations. Moreover, most scientific applications do not need all POSIX features [128, 263], especially when they can exclusively access a dedicated file system. Comparable argumentations hold for other advanced features, such as security or fault tolerance.

This chapter will present GekkoFS – a highly scalable distributed burst buffer file system for HPC applications. GekkoFS combines node-local storage resources, e.g., fast SSDs, into a shared namespace that all file system nodes can access. It is specifically designed for temporary scenarios, e.g., during an HPC job, or for longer-term use cases, such as campaigns where many nodes access data in parallel and in short bursts. To allow such an *ad hoc* deployment, the file system runs in user space and can be easily started in less than 10 seconds on a 512 node cluster by any user. GekkoFS relaxes POSIX and removes some of its semantics that impairs file system performance in a distributed setting. In addition, for GekkoFS, we considered earlier studies that investigated the behavior of HPC applications [128] to optimize for the most used file system operations.

For scalability and load-balancing, GekkoFS distributes all data and metadata across all file system nodes. Further, its modular I/O layer supports multiple data and metadata distribution schemes that can benefit various I/O patterns, e.g., when metadata needs to be globally available but data only locally. For moving the data between the nodes, GekkoFS uses the HPC RPC framework *Mercury* [228] via *Margo* [210] in combination with the HPC threading framework *Argobots* [42, 221].

This allows GekkoFS to natively use enterprise network technologies used in HPC systems, such as InfiniBand and Omni-Path.

This chapter will demonstrate the details of GekkoFS's lightweight design, allowing it to achieve scalable data and metadata performance with more than 45 million metadata operations per second on a 512 node supercomputer. Moreover, we will show that GekkoFS can run complex scientific applications, e.g., OpenFOAM solvers [109] or deep learning workloads with TensorFlow [1]. We achieve these features while GekkoFS operates synchronously and with strong consistency for all file system operations which target a specific file or directory.

4.2 Related work

This section will discuss existing HPC file systems and outline their differences from GekkoFS. In addition, we will describe possible options to develop and run user space file systems.

4.2.1 General-purpose parallel file systems

Earlier in this thesis, Section 2.2 introduced parallel file systems, which most HPC systems are using as a backend storage system to provide a global shared namespace for all users. Examples for parallel file systems are GPFS [219], Lustre [30, 198], BeeGFS [97], or PVFS [211]. These file systems focus on data consistency and long-term storage and offer a POSIX-like interface so that applications can use them as any other local file system. However, due to the nature of the file system being globally accessible, a single application can be enough to severely disrupt the I/O performance of others applications. Moreover, parallel file systems do not work well for small file access, specifically when operating on shared files, often used in scientific applications [168].

GekkoFS does not focus on a similar long-term storage solution. Instead, GekkoFS provides a separate namespace that is only accessible to nodes that are part of the file system within the context of an HPC job or other temporary use cases. Then, after the HPC job finishes, all data is deleted. In addition, GekkoFS offers a relaxed POSIX environment that allows for a significant increase in metadata performance and reduces the impact on other applications running on the same HPC system.

4.2.2 Burst buffers

Section 2.3.3 introduced node-local burst buffer file systems, which are logically placed between the parallel file system and an application. Their goal is to use the available node-local storage resources as a temporary distributed file system offering fast and scalable I/O performance for an application and reducing the load on the parallel file system. Burst buffer file systems are sometimes used ad hoc as part of a single compute job or multi-job workflows. For example, in a single job use case, they are started when a compute job starts and shut down when it ends.

Hermes [114], an I/O middleware library, provides a distributed I/O buffering system transparently combining multi-tiered storage and memory hierarchies of supercomputer environments. It considers both local and shared resources as potential burst buffers and places data on all available storage layers.

UnifyFS [@119] is a node-local burst buffer file system and offers *lamination semantics*, allowing a process to mark a file as no longer being modified. After lamination, the file becomes permanently read-only and can be accessed by other compute nodes without further synchronization, which can be useful for checkpoint-restart workloads. UnifyFS does not include a POSIX I/O interface and must be linked directly to the application.

BurstFS [265] is similar to GekkoFS regarding them being standalone burst buffers and not requiring a centralized instance. Nevertheless, BurstFS is limited to writing data only locally, whereas GekkoFS allows various distribution schemes, e.g., local write/reads coupled in a global namespace or user-defined distributions linked to a directory or file. By default, GekkoFS distributes all data and metadata across all file system nodes, balancing data workloads for write and read operations without sacrificing scalability.

The Hercules in-memory burst buffer storage system [78] focuses on in-memory use cases and further offers a custom put/get interface. GekkoFS, on the other hand, targets node-local SSDs and POSIX relaxation while both file systems further offer several different data distribution strategies.

BeeOND [@16] (BeeGFS on demand) is a special deployment mode of BeeGFS [97] which allows it to be deployed ad hoc, similar to GekkoFS. However, contrary to GekkoFS, BeeGFS is a POSIX-compliant kernel-based file system, and our measurements show GekkoFS having a much higher metadata throughput than offered by BeeOND (see Section 4.5.3 later).

4.2.3 POSIX

When parallel file systems provide POSIX semantics in a distributed environment, supporting a strong consistency model coupled with shared access, they typically rely on expensive distributed locking mechanisms, e.g., byte-range locking in Lustre [30] and GPFS [219]. Nevertheless, strong consistency guarantees and poor parallel I/O support in file systems [123] combined with access patterns of scientific applications [168] are the main reasons for I/O challenges in HPC. With the continuous increase in computing performance, data-intensive applications encounter scalability challenges, preventing them from fully using the computing power offered by HPC systems [46].

One solution to reduce locking-induced overheads is to give the responsibilities to applications and external libraries so that no shared file conflicts occur. PVFS, for instance, does not implement a locking sub-system for this reason [159, 211]. With GekkoFS, we support this argumentation and do not use a locking mechanism to avoid the discussed overheads. In fact, avoiding locking and therefore inter-node communication and synchronization between servers is one of the fundamental pillars GekkoFS is built on.

4.2.4 Metadata scalability

Earlier in this thesis, Section 2.3.2 discussed the metadata challenges in the context of parallel file systems. One of the large contributors to this issue are inodes and directory blocks, which were not designed for parallel access. Over the years, the file system community has presented several techniques for how to approach metadata scalability [19, 73, 182, 183, 273, 274], but this challenge is still acute and becoming even more important in upcoming data science applications.

For example, IndexFS is one such attempt to significantly improve metadata performance [205] by using it as a middleware software between the application and the parallel file system. Like GekkoFS, IndexFS stores all metadata information in a key-value database and distributes metadata across multiple IndexFS servers. IndexFS uses various client caches and assigns directories to IndexFS servers to further increase RPC efficiency. GekkoFS, on the other hand, does not use any client caching and uses a loosely coupled architecture, avoiding communication between GekkoFS servers or linking data structures to a set of components. Moreover, it uses wide-striping by default to aggressively distribute all metadata across all servers. Contrary to GekkoFS, IndexFS is still bound by POSIX file I/O semantics, requiring

it to use all file system protocols and interfaces of the used underlying file system. GekkoFS implements relaxed POSIX semantics and only offers strong consistency for requests that focus on a specific file or file data. As such, it weakens the concept of directories, no longer using directory blocks and removing the connection between inodes and directory entries. In fact, directory blocks are entirely removed and replaced by objects for each file that are stored in a strongly consistent key-value database. Among other things, this design allows GekkoFS to achieve linear scaling with tens of millions of metadata operations for 512 nodes and billions of files.

4.2.5 Interfacing ad hoc file systems

When applications interact with a standard file system, they expect a familiar and portable interface that follows a set of rules in the POSIX family of standards that define the syntax and semantics of the I/O interface. Therefore, the POSIX I/O interfaces are deeply embedded into the operating system and familiar to users and applications that, as a general rule, do not directly use the corresponding I/O system calls but use the *GNU C standard library* (glibc), which wraps the Linux system calls and offers additional I/O functions, e.g., `fwrite()`. Notice that these application-oriented interfaces should not be confused with other interfaces, such as block I/O, object-oriented I/O, or file-based I/O, which are defined for lower-level storage media access.

In general, a file system and its interfaces can be implemented at the kernel level or user level (*user space*). Figure 4.1 presents three possible approaches. Kernel-based file systems, e.g., EXT4 [37, @140], are the first option and run entirely in the kernel. The second option adds an additional user space component to which I/O requests are redirected, such as FUSE [@76, 250], which will be discussed later in this section. Therefore, developers can write a file system that is run in user space relying on the corresponding FUSE driver in the kernel. Such kernel-based file systems that aim to support the standard file system interface must register their interfaces to Linux's VFS that the glibc library then accesses via system calls. The third option is file systems that run entirely in user space (without any kernel component). Because they are not connected to the VFS, which is part of the kernel, the file system provides their implementation by *preloading* a library through the `LD_PRELOAD` environment variable in which it intercepts defined I/O calls and redirects them to the user space file system. The latter option is becoming increasingly popular [136, 160, 201, 265, 281] (including in ad hoc file systems) because of several benefits for code development, portability, debugging, maintenance, and, often, performance.

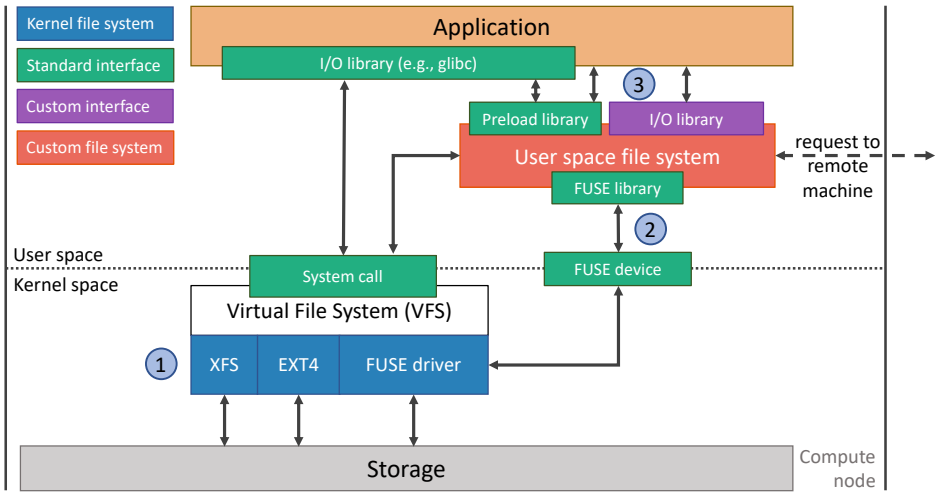


Figure 4.1.: Techniques to interface file systems with their components in three configurations: a kernel file system, a custom FUSE file system running in user space but relying on the FUSE driver in the kernel, and a custom user space file system without a kernel component.

An alternative to file systems using standard interfaces is dedicated *I/O libraries* or *I/O wrappers*, providing a thinner set of API functions that may ease deployment and maintenance. I/O libraries are not necessarily backed by user space file systems and are used to align access patterns with the capabilities of the underlying parallel file system (e.g., ADIOS [137]). They can also be used as an API for another user space file system, e.g., OrangeFS [159]. Other works, such as Aerie [260], use a flexible file system architecture using a key-value interface. Yet, custom interfaces that do not follow standards and usually require a modification of involved applications are not suitable in many cases.

FUSE file systems The FUSE (Filesystem in Userspace) [202, 250] framework is a popular approach to developing user space file systems due to its low development overhead. At its core, FUSE uses two components: the FUSE kernel module (FUSE driver) and the FUSE library (*libfuse*) in user space, supporting kernel-based and user-level redirection for I/O calls, respectively. The libfuse library is executed as part of a process that manages the entire file system at the user level. The library communicates with the kernel through the FUSE device and the FUSE kernel module that is connected to the VFS. Internally, all I/O calls are implemented as callbacks through the libfuse library, which supports the communication between the FUSE kernel module and the user space file systems [202]. As a result, a FUSE file system can be used as any other kernel-based Linux file system, providing the traditional file system interface. Due to FUSE's API, developers can quickly implement a file system

while avoiding much of the complexity of the kernel. Several file systems use the FUSE library, e.g., ChunkFS [96], SSHFS [100], FusionFS [280], or GlusterFS [54].

Although convenient, FUSE-based file systems also face several drawbacks due to their architecture which requires constant communication between libfuse and the kernel module. This is because libfuse is run as a separate user process in which the round trip between an application and the FUSE process can result in severe performance overhead [250]. Among other things, this is caused by a large number of context switches between the user level and kernel level, requiring two context switches for each file system request between the FUSE process and its kernel module. Note that a single file system operation can involve several such communication steps. DAOS, for instance, explored ways to mitigate this issue by coupling a FUSE file system with library preloading in which only read and write requests are intercepted by the interception library [136]. However, Vangoor et al. [250] showed that specifically metadata operations are most affected by FUSE's performance overheads. Moreover, mounting a FUSE file system requires extended privileges for their users and the availability of the FUSE kernel module, which can pose a challenge in some HPC environments.

We conclude that for ad hoc file systems utilizing fast node-local storage devices as burst buffers, the performance overheads can become too high for efficient usage of the underlying storage and outweigh the benefits of convenient development. Nevertheless, FUSE-based file systems can still be useful for file systems that do not require strong consistency semantics, as is the case with object storage, for example. As a result, many file system connectors for object stores are using FUSE. Chapter 5 will discuss this topic in detail and will present *DelveFS* that is also based on FUSE. In this context, DelveFS explores ways to mitigate various FUSE performance overheads, specifically when operating on large directories, which are commonplace in object storage systems.

User space file systems Because pure user space file systems cannot be registered to the VFS of an operating system due to a missing kernel component, user space file systems must use other methods to offer a file system interface to applications. Generally, there are two options: The file system is either directly linked to the application at compile time or dynamically loaded at runtime, which can be achieved by using a *preloading library* that intercepts defined file system-related function calls. UnifyFS [@119] and DAOS [136] are examples of distributed file systems using the former linking approach. UnifyFS supports the standard file system interface but still requires some file system setup modifications to the application source code.

DAOS, on the other hand, offers an entirely new I/O API for file system operations. Nevertheless, such approaches require access to the application's source code and potentially extensive code modifications to accommodate custom APIs. An alternative is preloading libraries that do not require application modification or recompilation. Example file systems using this option are CRUISE [201], BurstFS [265], and DeltaFS [281]. They all intercept the application I/O via a set of wrapper functions that are implemented in the form of user-level preloading libraries.

Using such libraries can ease application development and maintenance and can be beneficial concerning I/O throughput and latency because the kernel can be avoided. Although the existing kernel-based components for file systems are well suited for spinning disk-based storage, Volos et al. [260] discuss that they unnecessarily limit the design and implementation of file systems that utilize faster storage, e.g., persistent memory devices that can offer I/O performances close to DRAM. Their Aerie framework has shown that a POSIX-like file system in user space can perform similarly or better than kernel-based implementations.

The *Persistent Memory Development Kit* [@187] (PMDK) is another approach to efficiently use *non-volatile main memory* (NVMM) devices, also sometimes called *persistent memory* devices. It avoids the kernel and builds on Linux's DAX feature so that applications can utilize persistent memory as memory-mapped files. Therefore, approaches that bypass the kernel, e.g., by library preloading via the `LD_PRELOAD` environment variable, are an attractive consideration when designing an ad hoc file system, especially when it uses fast flash-based burst buffers. Reducing the time within the kernel is likely becoming increasingly important for future storage technologies, e.g., NVMM, so that applications can fully exploit their storage capabilities and achieve high I/O performance. Although not using PMDK, Simurgh [160] is an example user space NVMM file system in shared memory that uses library preloading to avoid kernel communication.

When using `LD_PRELOAD`, the corresponding library can, in principle, intercept any function. Because of this, all I/O functions need to be reinterpreted by the user space file system for the application to function correctly. Thus, the more I/O-related operations an application uses, the more functions must be intercepted as well. As a result, a large part of a given I/O library, such as those provided by glibc, would need to be reimplemented. The *system call intercepting library* [@237] (syscall intercept) aims to solve this challenge by offering a low-level interface for hooking Linux system calls while still using the `LD_PRELOAD` method. This is done by patching the machine code of the standard C library in the memory of the process, allowing the library user only to intercept system calls and leaving the functionality of a given I/O library

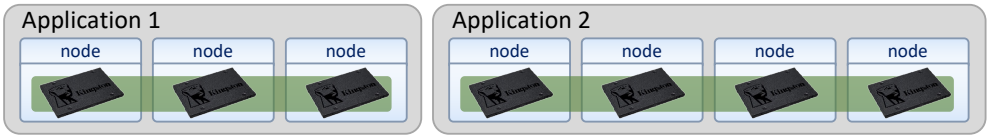


Figure 4.2.: GekkoFS combines the node-local SSDs of multiple nodes (highlighted in green) in two cases for different node numbers.

untouched. For instance, when an application uses the commonly-used `fwrite()` function of glibc, the interception library would need to reimplement its functionality while also providing possibly different implementations of the numerous glibc versions. Syscall intercept, on the other hand, limits the implementation to the `write()` system call while glibc still provides the `fwrite()` functionality. Therefore, syscall intercept dramatically decreases the number of functions that the user space file system needs to implement as it is sufficient to only intercept system calls. In this context, *GOTCHA* [@118, 194] is a wrapper library similar to `LD_PRELOAD`, but it operates via a programmable API. For instance, UnifyFS is using GOTCHA as an interception mechanism. Nevertheless, GOTCHA still requires some modifications to the application source code.

GekkoFS incorporates the syscall intercept approach for the above-stated benefits.

4.3 Design

This section will explain GekkoFS's goals and design considerations. First, a short overview will outline how GekkoFS works and how users interact with it. Next, GekkoFS's goals will be defined, followed by a brief introduction to GekkoFS's architecture and components. Finally, we will discuss each file system component in detail.

4.3.1 Overview

GekkoFS's goal is to provide a distributed file system in user space for use cases where temporary storage access is available, such as within the context of an HPC job that can use the node-local SSDs. It uses these node-local SSDs and combines their storage capacity into a single global shared namespace, illustrated in Figure 4.2. To achieve scalability with an arbitrary number of nodes, all data and metadata are distributed across all nodes that run a GekkoFS server.

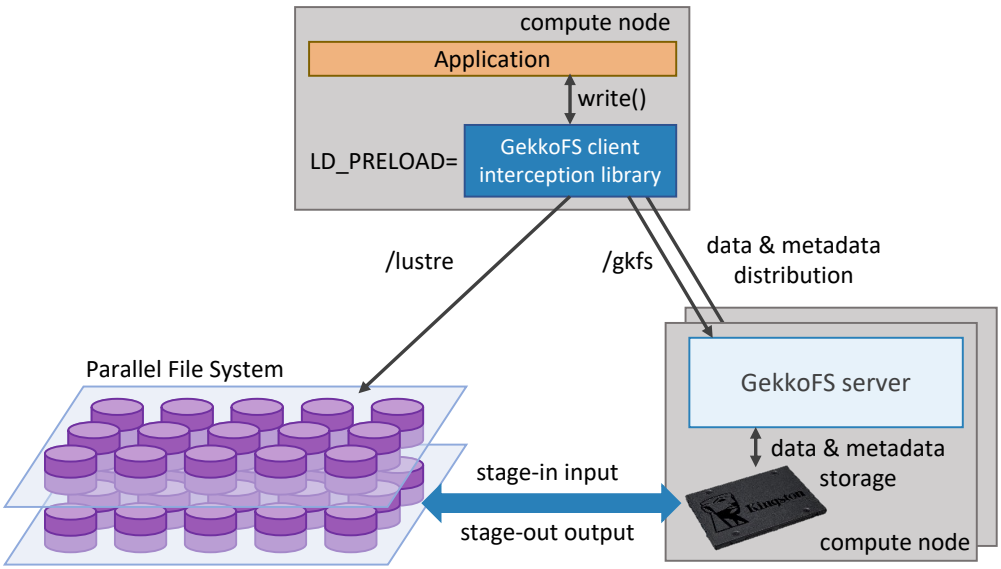


Figure 4.3.: Simplified overview of how an application interacts with GekkoFS.

Figure 4.3 presents a simplified overview of how an application interacts with GekkoFS. As the first step, a file system server must be started on each node that should use its local storage device for the file system. One of the server parameters includes a path to a shared *hosts file* which each server uses to register itself. Therefore, it contains all addresses of all servers used in the file system and it is supposed to be placed on the HPC system's parallel file system. The hosts file's path is then made available through an environment variable so that the clients know which servers are part of their particular file system instance. An application then *preloads* the GekkoFS client interception library, which intercepts all file system operations and checks if they operate within the GekkoFS namespace, i.e., path (see Figure 4.3). The client library can also be used to copy input data from the parallel file system (*stage-in*) into GekkoFS or copy output data from GekkoFS to the parallel file system (*stage-out*), if necessary. For instance, to stage-in data, a user can use the `cp` command on the *command line interface* (CLI) while having the client library preloaded. After a file system operation is intercepted and operates within GekkoFS, the client forwards it to the responsible server, determined by hashing the file's path, where it is independently executed. To avoid an unbalanced data distribution for large files, each file is additionally split into equally sized chunks by the client before being distributed to the servers. Due to this communication scheme, there is no communication required between the server instances.

4.3.2 Goals

We define the following goals for GekkoFS's design:

Functionality Any user should be able to deploy the file system on an arbitrary number of nodes without administrative assistance. The paths to the mount point (that is, the path where clients access the file system) and the data directory (that is, the path where file system data is stored) are passed to the server executable when it is started. The mount point should then present the user with a single global namespace, consisting of the aggregated node-local storage of each server node.

Scalability GekkoFS should scale well with an arbitrary number of nodes and efficiently use available hardware to benefit from current and future storage and network technologies.

Consistency model GekkoFS should provide the same consistency as POSIX for any file system operation that accesses a specific file or data (region) within a file. This includes read and write operations as well as any metadata operations that target a single file, e.g., file creation. However, the consistency of directory operations, e.g., an `ls` operation to list a directory, can be relaxed.

Fast deployment Compute time in HPC environments is valuable and expensive and should not be wasted for the purpose of file system deployment. Therefore, GekkoFS's startup should be finished within one minute to be used immediately by applications once the startup succeeds.

Hardware independence GekkoFS should utilize networking hardware commonly used in HPC clusters, such as Infiniband or Omni-Path, and support the native protocols of these fabrics to move data between file system nodes efficiently. In cases where this is not possible, TCP/IP should be available for Ethernet, Infiniband, and Omni-Path. Moreover, GekkoFS should work with any modern and future storage subsystem that is (or will be) attached to compute nodes with the condition that the node-local storage is accessible at a local file system path permitted to the user.

4.3.3 Relaxed POSIX semantics

I/O interfaces and their definitions implicitly set requirements and expectations for how file systems retrieve data for the user. The POSIX model inherently results in a consistency model which requires atomicity and locking mechanisms in distributed environments. Adhering to the POSIX consistency model is especially challenging for the scalability of a parallel file system in the following cases:

1. Atomic operations which require exclusive access to a central data structure or central file system component within a distributed environment: Acquiring a (global) lock allows such atomicity requirements but can impair concurrent work. For instance, a basic creation of a file involves exclusive access to a number of central data structures, e.g., modifying the parent directory or allocating a new inode in the inode table. Another example for such atomic operations is listing a directory with the `ls -l` command, which creates a snapshot of the current directory state.

2. Cache coherency protocols: While there are several advantages of various forms of caching in a parallel file system, it is generally not clear whether an application with particular semantics can benefit from such general caching protocols. Moreover, distributed cache coherency protocols often require a large amount of network communication to maintain synchronization, which is commonly impractical at larger scales.

Consequently, GekkoFS does not implement a global locking mechanism, similar to PVFS [41] and OrangeFS [159]. Therefore, applications should be responsible for avoiding conflicts that would require complex locking within the file system, e.g., concurrent writing to overlapping file regions. Nonetheless, avoiding global locking mechanisms also affects certain file system operations that query an arbitrary number of files regarding their consistency guarantees. We call these *indirect file system operations* because the number of affected file system objects is not known a priori, e.g., when requesting the current state of a directory. In other words, `readdir()` operations which are called by the `ls -l` or `rm -rf /*` commands, for instance, follow an *eventual consistency model* [259]. In essence, eventual consistency states that the file system guarantees that all accesses will return the last updated value of an object *eventually* if it receives no additional updates [259]. Specifically, for GekkoFS, this means that a `readdir()` operation guarantees to return the current state of the directory at the time a server receives the corresponding request. However, if another process is simultaneously modifying the directory and creating files, a `readdir()` operation may return a directory state that has already changed once the operation finishes. Hence, such a directory operation does not influence

other directory operations via inter-node locking mechanisms, or similar, as it is the case in file systems that use a strong consistency model. Also, for the above-stated reasons, GekkoFS operations are synchronous and without any form of caching. For example, this is beneficial for the above-mentioned `readdir()` case concerning eventual consistency so that users are not stuck in a cached (outdated) view of the directory, even if repeatably executing the `readdir()` operation. Further, this reduces file system complexity and allows for an evaluation of GekkoFS's raw performance capabilities.

Moreover, studies on HPC application behavior have revealed that many common file system operations, such as move/rename operations, are rarely used or not used at all [128, 263]. This is owing to HPC applications running in a controlled batch script environment, whereas above-stated operations are only called from a console after the actual simulation ends and when parallel access is no longer required. With these observations in mind, GekkoFS does not optimize move or rename operations and linking functionality, although supporting them rudimentarily.

Finally, security management in the form of access permissions is not maintained by GekkoFS directly. This is because a user in a compute job environment must already follow the security protocols of the supercomputer and its batch systems as well as the permissions of the node-local file system where GekkoFS's data and metadata are stored. Thus, data can only be accessed in GekkoFS if the user has sufficient access permissions within the assigned environment.

4.3.4 Architecture

The GekkoFS architecture (see Figure 4.4) consists of two core components: the GekkoFS server or *daemon* and the GekkoFS client library. To use GekkoFS, an application must preload the client interception library through the `LD_PRELOAD` environment variable, allowing the library to intercept file system operations. When an I/O operation is intercepted, the client library processes it and determines if it is within the file system's namespace. If not, the operation is passed to the kernel without modification. Otherwise, the GekkoFS client processes the operation and forwards it to the corresponding GekkoFS daemon (or multiple daemons for some operations), sending a response when finished. The receiving daemon is then handling the operation accordingly without further communication to other file system nodes. The daemons are therefore effectively unaware of one another.

For communication between the client and the daemon, we leverage on the Mercury RPC framework [228]. A *remote procedure call* (RPC) [23] is essentially a transparent

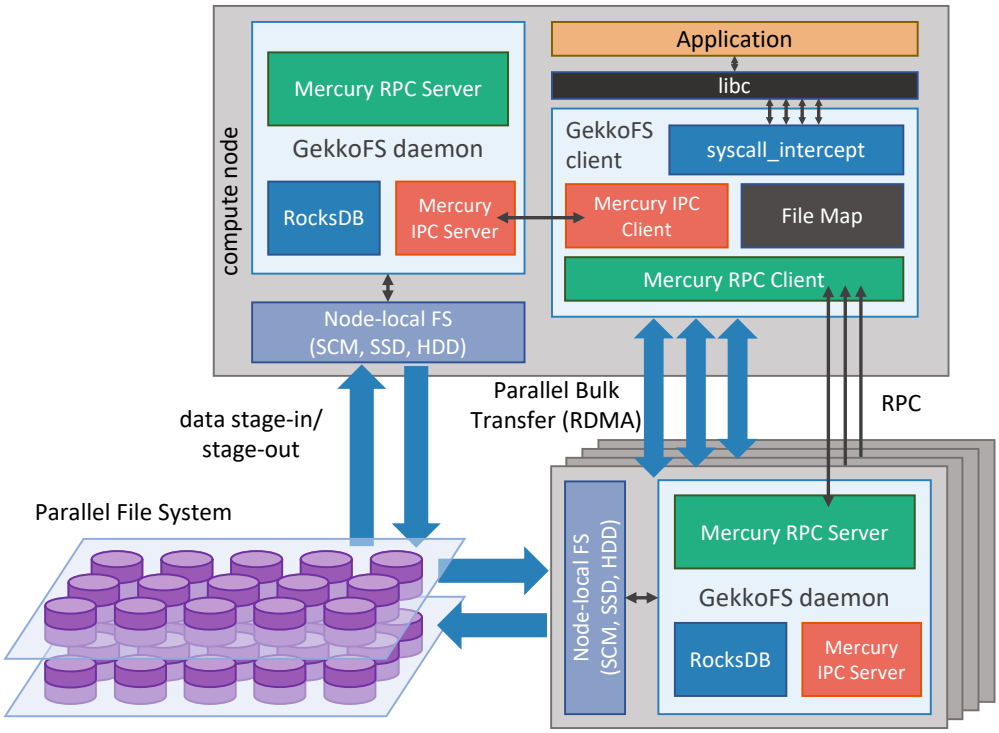


Figure 4.4.: GekkoFS architecture and its components.

way to call a remote function via a client-server model, which also handles serialization and deserialization of the function arguments. Mercury is developed by the *Argonne National Laboratories* (ANL) and focuses on HPC environments. It allows GekkoFS to be network-independent, achieving one of our design goals. GekkoFS can therefore use the native network transport layer to handle large data transfers efficiently, unlike other more generic RPC frameworks that focus only on TCP/IP communication. Mercury's *Network Abstraction Layer* provides a high-level interface on top of the lower-level network fabrics. Its modular approach offers a wide variety of network plugins to natively support common fabric protocols, e.g., InfiniBand or Omni-Path. It further provides local communication plugins to efficiently move data between a client and server running on the same machine. When allowed by the underlying network protocol, large data transfers are processed via *remote direct memory access* (RDMA) or *cross-memory attach* (CMA) in remote and local communications, respectively. This allows GekkoFS to transfer data within the file system with high throughput and low latency.

Within GekkoFS, Mercury is interfaced indirectly through the *Margo* library [210] which uses Argobots-aware wrappers to Mercury's API, allowing a simple multi-threaded execution model [42]. Argobots is a lightweight low-level threading and tasking framework developed to support massive on-node concurrency in mod-

ern HPC environments [221]. Mercury, Argobots, and Margo are all part of the Mochi project [@117] with the goal to design methodologies and tools that allow for the rapid development of distributed data services. Using the Mochi tools allows GekkoFS daemons to minimize resource consumption of Margo's threads and handlers that accept and handle RPC requests [42].

GekkoFS client The client consists of three components: 1. An interception interface that catches relevant calls to GekkoFS and forwards unrelated calls to the kernel; 2. a file map that manages the file descriptors of opened files and directories, independently of the kernel; and 3. an RPC-based communication layer that forwards file system requests to local/remote GekkoFS daemons.

An application uses the client library via the `LD_PRELOAD` environment variable, intercepting all I/O calls that target the GekkoFS namespace. As discussed earlier in Section 4.2.5, GekkoFS uses syscall intercept instead of the basic `LD_PRELOAD` method. When the GekkoFS client library intercepts an I/O call it has a hook defined for, it must determine if the operation is within GekkoFS's namespace. There are two ways how a file is referred to an operation: via its path or via its *file descriptor* (fd). Although GekkoFS can easily check if a path is within the file system's namespace, a file descriptor is more complicated because it is created in the kernel after a file has been opened. Since GekkoFS does not use a kernel module, it must reimplement some of the file descriptor logic in user space as part of the client library, called the *file map* client component. It allows GekkoFS to associate a file descriptor with its file name and keeps track of the `seek()` position when calling consecutive `write()` operations, for example. Therefore, a corresponding file map entry is created when a file is opened and it is removed upon closing the file.

After an application initially invokes the client library, it first initializes the interception layer and then looks for the hosts file containing all GekkoFS daemons' addresses that are part of the file system. It then proceeds to retrieve the current file system configuration from one of the daemons (preferably the local one if available), which includes the path of the mounting dir, defining the GekkoFS namespace for the interception layer.

While a file is open, the client uses the file path p to determine the GekkoFS daemon node that should process it. Specifically, the path p is hashed using a hash function h to resolve the responsible daemon for an operation by calculating:

$$nodeID = h(p) \pmod n$$

where n is the number of GekkoFS nodes. The corresponding operation is then processed and forwarded to the daemon identified via a unique $nodeID$. That is to say, the GekkoFS client uses a pseudo-random distribution to wide-stripe all data and metadata across all nodes. Using a distribution algorithm that allows all clients to independently resolve the responsible node for a file system operation is essential for the file system's scalability. This is because it enables GekkoFS to function without any central data structures that keep track of where metadata or data is located. We will further discuss the implications of wide-striping later in this section.

Note that, although wide-striping is the default setting, the distribution layer is modular and allows other algorithms as long as the client can resolve the target daemon on its own. For instance, another algorithm only wide-stripes metadata while storing data locally. This is useful for some scientific applications which only repeatably access data that has been written on the same compute node but still require knowledge of the whole namespace (e.g., OpenFoam solvers [109]). Moreover, to achieve a balanced data distribution for large files, data requests are split into equally sized chunks before they are distributed. During the data transfers, the client exposes the relevant chunk memory region to the daemon, which accesses it via RDMA, if supported by the underlying network fabric protocol.

GekkoFS daemon A GekkoFS daemon's main task is to process file system operations forwarded by the clients to store and retrieve data and metadata information. To achieve this goal, GekkoFS daemons consist of three parts: 1. A key-value store (KV store) used for handling metadata operations, 2. an I/O persistence layer that reads/writes data from/to the underlying node-local storage system, and 3. an RPC-based communication layer that accepts local and remote connections to handle file system operations.

For handling metadata operations, each daemon operates a single RocksDB key-value store (KV store), which provides a high-performance embedded database for key-value data, based on a *log-structured merge-tree* (LSM) [59]. The KV store is optimized for NAND storage technologies with low latencies and is therefore suitable for our requirements since SSDs are primarily used as node-local storage in HPC supercomputers.

For processing data chunks, the daemon leverages directly on the accessible node-local storage device and its file system. As a result, there is no need to prepare or reformat the backend storage. This is essential so that any user can run the GekkoFS daemon without administrative privileges. With the help of Argobots's *user-level threads* (ULTs) and *tasklets* [221], the daemon's data backend moves data chunks

from the client via RDMA over the network to node-local storage. In I/O operations involving multiple chunks to be processed at a single daemon, network transfer and disk access are done in parallel to use the available resources as efficiently as possible.

Modularity and configurations Overall, GekkoFS's architecture is modular, allowing various file system configurations and the replacement of the above-stated individual file system components in both client and daemon. Because GekkoFS can be run within a compute job, the file system can be configured to fit the application's requirements, unlike general-purpose parallel file systems, which have to optimize for the general case. This includes configurations targeting 1. the metadata and data storage, e.g., which metadata fields are stored or which chunk size is used, 2. the metadata and data distribution, and 3. modifications to the overall file system behavior, e.g., enabling the check whether a file exists before it is created or if the corresponding directory must be created before files can be created. We will further elaborate on some of these configurations concerning points 1 and 3 in the following sections.

4.3.5 Metadata management

Parallel file systems utilize several strategies to distribute data and metadata across all backend storage devices [30, 97, 219]. While these techniques typically work well for data, they do not achieve the same efficiency and performance (throughput or operations per second) for metadata operations because of expensive distributed locking mechanisms that are required due to legacy data structures (see Section 2.3.2). Hence, our goal is to design a file system that performs well for any metadata operation and allows them to scale with an arbitrary number of nodes.

With the first step being the research and acknowledgment of the discussed metadata challenges that today's distributed file systems face, we present our modifications to how GekkoFS handles metadata in the following paragraphs. The changes can be categorized into three topics: 1. Removing the concept of directory blocks, decouple directory entries from inodes, and aggressively distribute the metadata load, 2. reassessing the actually required metadata fields, and 3. simplifying file system protocols to avoid unnecessary file system complexity and overhead.

Decoupled wide-striping Instead of using the concept of directory blocks and inodes, we store the metadata in a daemon's KV store. Each file and directory is treated as a single entry in the KV store where the file path is used as the key and the value contains the file's metadata. As a result, each file becomes individually accessible by its path, resulting in a flat namespace where paths that share the same prefix are considered the children of a directory. Therefore, a directory's content and a file's hierarchical placement in the file system are implicitly given by its path without requiring a separate data structure, e.g., a directory block. A flat namespace also allows GekkoFS to directly look up a file without the need to perform a path walk, for instance, which would entail several RPC lookup calls. Moreover, treating each file as an independent KV store entry makes it possible to share the total metadata load by simply distributing all metadata across all nodes with the previously described distribution algorithm. This allows GekkoFS to avoid storing a large directory or a subset of the directory tree at a single file system node, which is common in other file systems [29, 205], potentially causing an grossly uneven metadata distribution and load.

However, using the path of a file system object as an index within a flat namespace does not come without cost. For example, when a directory is moved to a different file system path, the paths of all its contents would have to be recursively modified as well. Depending on the size of the directory and due to the distribution of metadata, this can be a time-consuming process because metadata might need to be migrated if the hashes for the new paths lead to different file system nodes. Nevertheless, as described earlier, GekkoFS does not support move and rename operations as they are not used in HPC applications [128, 263].

Metadata content In a particular use case, e.g., in the HPC job context, not all metadata may need to be stored, especially if a file is stored only for a limited time period. Therefore, when less metadata is stored and processed in the file system, less data must be maintained in the KV store and sent over the network. More importantly, it reduces the amount of code that needs to be executed for a file system operation. A typical example is the *last access time* (atime) metadata field which is updated each time a file is accessed. Updating this metadata field in a highly parallel environment with a huge number of files and file system operations is expensive and can cause significant performance degradation. As a result, many file systems [29, 97, @140, 219] allow to disable atime or to only update it once a day. Similar argumentations hold for other metadata fields, such as *last modified time* (mtime).

We categorize metadata fields into three categories: redundant, rarely used, and mandatory. The first category includes permission bits, user ID, and group ID information. These are used for security, which we consider as *redundant* because GekkoFS already has to follow the security protocols of the batch system and node-local file system. The second category contains metadata fields that are *rarely used* by HPC applications, including timestamps, the inode number, and block size information. We disable these metadata fields by default for the above-stated reasons, but they can still be enabled if needed[1].

In the third category, we place *mandatory* metadata fields which cannot be turned off: the file type and the file size. For the file type, we mainly differentiate between files and directories. Although GekkoFS does not use traditional directories and directory blocks, it still supports directory types because applications often check for a directory's existence before it is populated. The second mandatory metadata field, the file size, is used to track a file's data boundaries. The file size is necessary because it is required in each `write()` and `read()` operation, e.g., to check if an application is accessing a file outside its boundaries. Moreover, it allows GekkoFS to work with *sparse files* in which file systems generally do not write regions to disk that contain no information.

Simplifying file system protocols Because GekkoFS, as a pure user space file system, does not use a kernel component and is not connected to the VFS, GekkoFS does not need to implement the same protocols and call paths as kernel-based file systems. As a result, GekkoFS can simplify some of the file system protocols that could impact file system performance in distributed file systems. For example, in earlier work, we discussed the protocol to create a simple file in GPFS [251]. When a file is created with the `O_CREAT` flag via the `open()` system call, the file system must ensure that the file does not exist before it is created[2]. Because GPFS must follow the VFS-defined interfaces, a file creation is split into the VFS lookup function, followed by the create function. However, a file could have been created by another process after the lookup function, and, therefore, the create function must lookup the file again to ensure it does not exist. Such situations can cause complex and redundant protocols, but they are also problematic from a performance perspective. In the GPFS case, the lookup function contributed up to ∼75% of the overall time spent in the create process for four GPFS clients [251].

[1] One exception is the inode number which cannot be enabled as GekkoFS does not utilize inodes.

[2] When the `open()` system call is used with the `O_CREAT` flag and the file already exists, the file system opens the existing file quietly. If the `O_CREAT` and `O_EXCL` flags are both used on an existing file, `open()` returns an error and the error code `EEXIST`.

For this example, GekkoFS's file create protocol has been significantly simplified, requiring only a single communication step to a GekkoFS daemon to create the file which can handle the request independently, regardless of the number of nodes involved in the GekkoFS file system. Several protocol simplifications can be found in other GekkoFS parts with the goal of increasing the overall throughput of metadata and data operations. Another example involves GekkoFS allowing file placement into directories that do not technically exist if the user enables this feature. Other changes to file system protocols involve most metadata operations, such as *file stat* or *file removal* operations or the listing of all files at a file system path.

4.3.6 Data management

For storing the data in GekkoFS, the file system uses a node-local storage device and leverages its already existing local file system. In the following paragraphs, we will describe GekkoFS's I/O protocol and explain how shared file access is managed in an environment where global inter-node locks are not used.

I/O protocol Similar to metadata, data is evenly distributed across all file system nodes by splitting it into equally sized *chunks*. On the node-local file system level, each chunk is a regular file and is named after the numeric identifier describing the chunk's data offset. For instance, in a 512 KiB chunk-sized file system, the chunk number `4` identifies a file's offset beginning at 2 MiB[3]. All of a file's chunks are placed within the same directory on the node-local file system. In other words, one directory on the node-local file system contains the chunks in the form of regular files of one *GekkoFS file* with which a GekkoFS user interacts. Note however that the directory only contains the chunks of the GekkoFS file.

Figure 4.5 shows a write operation from the client's point of view. In this case, the write buffer is split into six chunks. GekkoFS computes the target node with the help of the file's path and chunk identifier for each chunk. The client then sends an I/O RPC message to each target daemon node, independently handling the write request for a group of chunks. Each GekkoFS daemon accesses the client's memory via RDMA and writes the corresponding chunks to its node-local file system. If the target daemon refers to the local machine, data is moved via cross-memory attach (CMA). Intra-node communication via CMA only involves a single copy and therefore allows to copy data between the address spaces of two processes without moving the data through kernel space. When multiple chunks hash to the same daemon, the

[3]The first index is chunk number `0`.

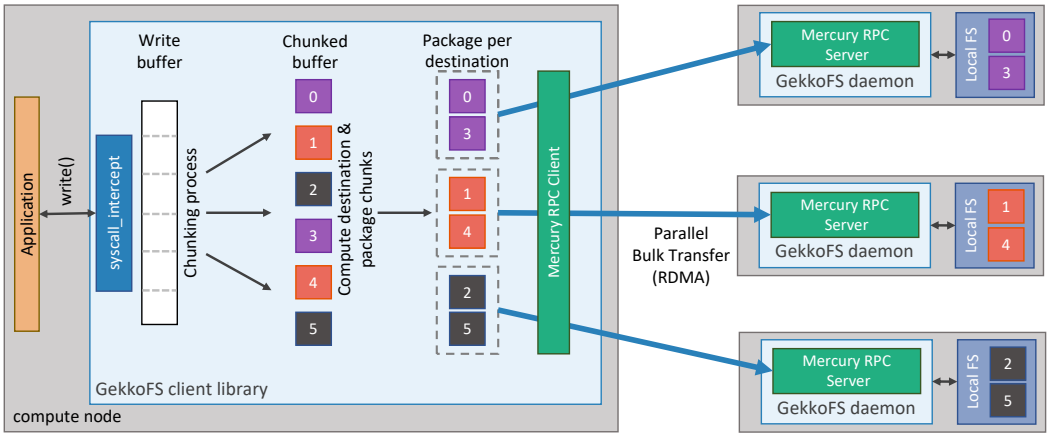


Figure 4.5.: GekkoFS's write operation where a write buffer is split into six chunks and then distributed among three daemons. Each daemon stores its chunk in a node-local file system.

GekkoFS I/O layer parallelizes the data transfer and disk I/O so that the network and storage device resources are used as efficiently as possible. This process works analogously for a read operation but in reverse. Notice however that the daemon must access the client's memory in both cases and not vice versa. This is because, regardless of the operation, the client must prepare the memory region for RDMA access. Therefore, the I/O RPC to the daemons also serves as confirmation and sends the necessary information to access the client's memory. Finally, as explained in Section 4.3.3, all I/O operations are synchronous without any caching mechanisms on clients or daemons. Depending on the configuration of the node-local file system, GekkoFS transparently uses the local file system's internal caching mechanisms.

Shared write conflicts As shown in Section 4.3.3, GekkoFS does not implement a global locking manager, which can impose challenges when working with shared files. For example, a shared write conflict could be caused when multiple processes write to the same file region in parallel, resulting in an undefined behavior concerning the data written to the underlying storage. However, due to GekkoFS's decoupled design, such locking conflicts can, in fact, be easily handled by any file system daemon independently. This is because GekkoFS builds on the POSIX-compliant node-local file system to store the data chunks. Thus, the local file system automatically sequentializes access to the same chunk file in case of a shared write conflict. Notice also that a GekkoFS file may be distributed across many chunk files and nodes, reducing potential shared write conflicts since it only affects one chunk file at a time. As a result, other chunks are unaffected by a potential shared write conflict.

However, even when conflicting I/O accesses do not occur in a shared file, there are atomicity concerns on the GekkoFS daemon that manages the file's file size since the file's metadata is represented by a single KV store entry. This is owing to each write operation also requiring a corresponding size update. Therefore, for each file size update, the daemon would have to read (`Get()`) the current size value, then update it locally, and write it back to RocksDB via a `Put()` operation. Since the daemon runs multiple processes in parallel to handle RPCs and the file size is updated simultaneously, this is a typical *read-modify-write race condition* because interleaving processes could issue consecutive `Put()` operations, effectively overwriting each other and resulting in a corrupted file size. GekkoFS solves this challenge by leveraging RocksDB's *merge operator* offering the possibilty for user's to define and implement how merge operations behave. In essence, the read-modify-write operation is encapsulated into a RocksDB interface which further allows some additional (unspecified) internal RocksDB optimizations, avoiding the extra cost of repeatedly issuing `Get()` operations.

4.4 Use cases

Burst buffer file systems, such as GekkoFS, can combine the capabilities of node-local storage of many nodes into a single global namespace. Coupled with the ability for an ad hoc deployment, GekkoFS can be a powerful option for creating a single distributed namespace within seconds. What is more, burst buffer file systems are able to not only absorb challenging access patterns but they can also transform random access patterns, for example, into larger sequential access patterns that work best for a parallel file system. That is to say, input data is read sequentially on the parallel file system side and copied into the burst buffer file system's distributed namespace (called *stage-in* process). After the application finishes, the output data is sequentially copied back (called *stage-out* process). Besides sequential access, GekkoFS aims to perform especially well in use cases with lots of random access patterns, small I/O sizes, and high metadata loads. The following paragraphs will outline some of today's most prominent workloads where these access patterns are relevant and where GekkoFS can be beneficial.

Bulk-synchronous workloads Bulk-synchronous workloads represent one of the most dominant workloads in modern HPC systems [32]. Bulk-synchronous applications are typically characterized by independently running processes synchronizing

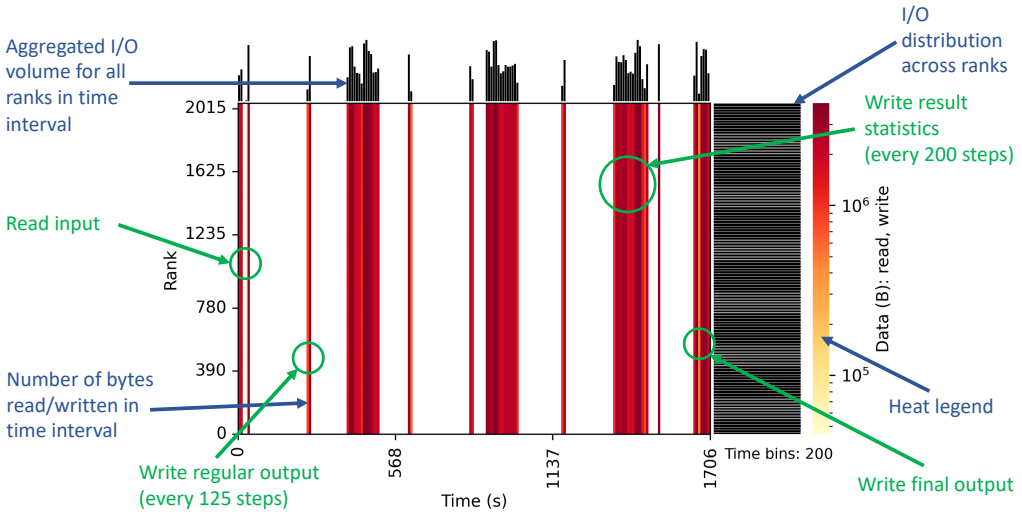


Figure 4.6.: An annotated I/O analysis for a workload run by the NEK5000 computational fluid dynamics application kindly executed at the Dardel HPC system at KTH Sweden [@53], showing the I/O in bytes over time broken down by MPI rank.

in bulk at specific step boundaries. These synchronization steps can include collective inter-process communication and I/O to the underlying storage, among others. Moreover, application processes do not impede themselves, and they either use a dedicated set of files per process or operate at non-conflicting offsets within a large shared file.

Depending on the amount of I/O and the state of the overall storage system, burst buffer file systems can often be advantageous in general as they can provide higher I/O performances than a parallel file system. Further, because processes are non-impeding concerning their I/O operations, there is no need for globally locking parts of a shared file, as is the case in parallel file systems, resulting in locking communication overhead and increased latencies. Finally, individual data distributions can be introduced on a per-application basis to optimize for its access pattern, as will be shown later in Section 4.5.6 with GekkoFS and the OpenFOAM application [109].

The NEK5000 [69] application for computational fluid dynamics is another example of a bulk-synchronous HPC application. Figure 4.6 presents the I/O footprint of a NEK5000 workload, created by the Darshan I/O profiling tool [227], which ran on 16 nodes with 128 processes per node. The experiment was kindly executed at the Dardel HPC system [@53] by the SimEx/FLOW Dept. Engineering Mechanics at the KTH Royal Institute of Technology in Sweden. In this case, all MPI ranks read input data initially, whereas the current state of the simulation (time and three velocity components) is written every 125 simulation steps. In addition, based on these four components, a large number of statistical quantities are written every 200 steps.

Overall, this specific workload produced close to 300 GiB of data in its 28 minutes of runtime, while the I/O bursts averaged around 650 MiB/s of write throughput (within the I/O burst time frame).

For such a type of application, it is useful to have as much data available as possible for post-processing. Nevertheless, because of I/O bottlenecks at the parallel file system, writing the resulting output more frequently is often not possible. And so, simulation output is essentially discarded, although it could be beneficial for further analysis.

Depending on the workload and I/O capabilities, burst buffer file systems could offer the ability to discard significantly fewer data due to their higher I/O performance. Further, they could gradually stage-out the output generated by these I/O bursts to the parallel file system and spread write operations over a larger time frame. In other NEK5000 workloads, GekkoFS has shown to improve simulation runtimes by up to 3 times compared to running the same workload on Lustre on JGU's MOGON II supercomputer. However, it is important to remember that the performance improvement depends on many variables, e.g., the number of GekkoFS daemons, the size and current usage of the parallel file system, and the simulation workload itself. Therefore, in bulk-synchronous workloads where burst buffer file systems are advantageous, they could allow even higher performance improvements.

Checkpoint-restart workloads In the event of software or hardware failures, applications can protect their progress by storing the application's state at set intervals to the parallel file system [32, 182]. This process is called *checkpointing* and allows applications to *restart* the application at a later time at the most recent checkpoint [72, 77, 182]. During checkpointing, an application writes each process's state into one large file at specified offsets or assigns one file per process in a single directory [19, 182]. Therefore, the larger systems and applications become, the higher is the load on the storage systems to accommodate checkpoint-restart workloads where many files are created.

Checkpointing has been a common I/O access pattern for decades [68, 191, 192] and remains one of the most day-to-day relevant workloads in HPC [32, 167, 263]. Among other metadata-heavy workloads, checkpointing has also been a contributor to specific requirements in HPC environments concerning file creates per second rate (see Section 2.3.2). Over the years, the storage community presented many approaches ranging from integrating node-local storage into the MPI-IO protocol [47], providing checkpointing libraries [158], using specific checkpointing file

systems, such as PLFS [19], or proposing an adaptive asynchronous checkpointing strategy [167] and more [@103].

Burst buffer file systems offer an alternative as they are not dependent on specific libraries or MPI. As a result, they can be used to write checkpoints on node-local storage or in memory, which can even be extended by asynchronous mechanisms [167, 201]. This way, checkpoints can be written to the burst buffer file system at a high frequency at memory speeds and asynchronously pushed to the parallel file system. In the context of GekkoFS, Section 4.5.3 will evaluate the performance for those I/O patterns that are typically found in checkpointing.

Deep learning workloads In general, *deep learning* (DL) aims to find an optimal way to connect a set of parameters of a deep layered model to predict a particular output and increase its accuracy [218]. Simplified speaking, to obtain an accurate DL model, many training samples are repeatedly passed through the model. Because the used optimizers are typically based on *stochastic gradient descend* (SGD) methods [27], DL access patterns on the input training data tend to be highly random [152].

In recent years, deep learning input data sizes have increased rapidly to produce more adequate models to detect real-world variations in input [32], exceeding the capacities of dedicated solutions of single machines [218], such as NVIDIA DGX systems [@173]. Consequently, HPC has become a cornerstone for these distributed DL use cases. For instance, using highly distributed setups and high-speed network fabrics of HPC systems, the required DL training runtime for the widespread *ImageNet* [57] was reduced from hours [87, 278] to below two minutes [155, 233].

Because of the nature of DL applications to randomly access training data, input data is first copied from the parallel file system to node-local storage. If the overall training performance then suffers from this copying process, the workload has encountered an I/O bottleneck [283]. However, copying a training set is only possible if it can fit into the storage capacity of a single node. In this case, such training sets are copied to each node, duplicating them across all nodes. This can minimize training biases (e.g., selection bias) since each node has access to the entire training set without it being partitioned. Contrary, this is not possible if the input data does not fit into a single node. A common approach for solving this challenge for both cases is to partition the data set into partitions [108, 116, 180] or *data shards* before there are distributed to the compute nodes. While this reduces data volume that needs to be copied per node, it is not trivial to partition a data set in a non-biased way since biases are not always immediately detectable [218].

In this regard, burst buffer file systems can be beneficial since they aggregate multiple nodes' local storage capacities and performances, creating a single global namespace. Therefore, it is no longer necessary to split input training data into several partitions because the data set must only be copied once into the burst buffer file system. Section 4.5.7 will discuss this topic in more detail, showing that GekkoFS can provide an easy-to-use transparent DL workflow and scales linearly for common DL workloads.

4.5 Evaluation

In this section, we will evaluate the performance of GekkoFS based on various microbenchmarks that catch access patterns common in HPC applications. First, we will introduce the experimental setup and explain the workloads we simulated with microbenchmark applications. Next, we will compare GekkoFS's metadata performance with a Lustre parallel file system. While GekkoFS and Lustre have different goals, we will highlight the performance benefits when GekkoFS is used as a burst buffer file system. Moreover, we will compare GekkoFS's metadata performance with BeeGFS's BeeOND, in which both use node-local storage as burst buffers. Then, we will evaluate GekkoFS's SSD usage efficiency and compare its data performance with BeeGFS's BeeOND. We will investigate GekkoFS's I/O variability compared to GPFS and explore GekkoFS's effects on the network when the OpenFOAM HPC application [109] is used compared to Lustre. Finally, we will discuss the benefits of GekkoFS in distributed deep learning workloads.

4.5.1 Experimental setup

We used three supercomputers in our experiments: *MOGON II* at the Johannes Gutenberg University Mainz in Germany [@157], *MareNostrum IV* at the Barcelona Supercomputing Center in Spain [@147], and the NEXTGenIO prototype at the EPCC supercomputing center in the UK [@165].

MOGON II The MOGON II supercomputer, located at the Johannes Gutenberg University in Mainz, Germany, consists of 1,876 nodes in total, with 822 nodes using Intel 2630v4 Intel Broadwell processors (two sockets each) and 1046 nodes using Xeon Gold 6130 Intel Skylake processors (four sockets each). Intel Broadwell

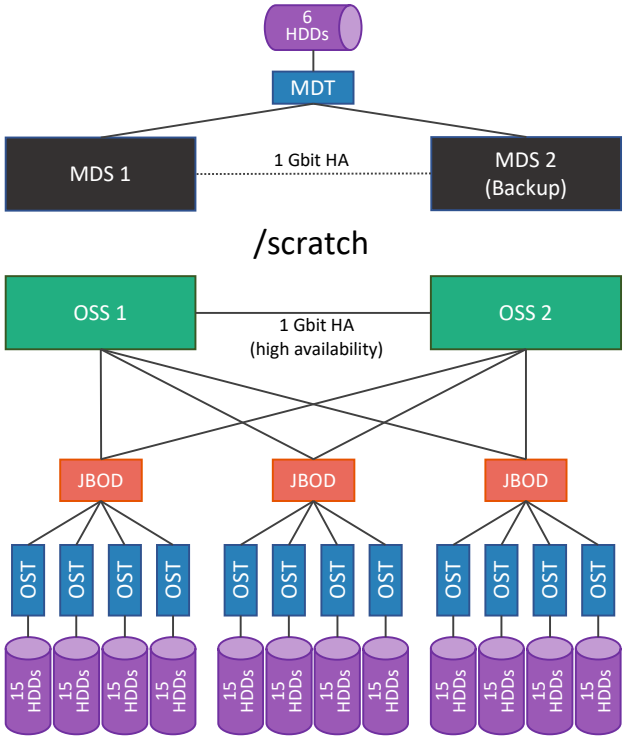


Figure 4.7.: MOGON II's Lustre scratch file system configuration.

processors were used in all experiments in this section if not otherwise noted. The main memory of the nodes ranges from 64 GiB up to 512 GiB.

MOGON II uses a 100 Gbit/s Intel Omni-Path interconnect to establish a fat tree between all compute nodes and offers a 7.5 PiB storage backend managed by several Lustre parallel file systems. Each node uses an Intel SATA SSD DC S3700 Series with 200 GiB or 400 GiB of storage space as a scratch environment (*XFS* formatted) which can be used entirely within a compute job. In our experiments with GekkoFS and BeeGFS, we use these SSDs for storing data and metadata. Both GekkoFS and BeeGFS use internal chunk sizes of 512 KiB.

All Lustre experiments were performed on a Lustre scratch file system with its configuration visualized in Figure 4.7[4]. Each OST consists of 15 HDDs with 10 TiB, each pooled together by a *zpool*, providing a ZFS file system built on top of a virtual storage pool. Within this zpool, 2 HDDs are used for parities and 1 HDD is reserved as a hot-spare disk. The MDT, on the other hand, uses 6 HDDs with 1.3 TiB each. For GekkoFS and BeeGFS experiments and before each experiment iteration, all SSD data is removed, and all kernel caches, i.e., buffer, inode, and dentry caches, are

[4]MOGON II offers two additional and significantly larger partitions for data storage, although all configurations utilize a single dedicated MDT with varying numbers of HDDs.

flushed. We also restarted all GekkoFS daemons before each iteration. The GekkoFS and BeeGFS server services and the application under test are pinned to separate processor sockets to ensure that the file system and application do not interfere with each other.

For deep learning experiments, we used a separate partition of MOGON II which includes 30 nodes with Intel Xeon CPU E5-2650 v4 processors and 6 NVIDIA Geforce 1080 Ti GPUs each. They used *NVIDIA CUDA* [@172] (v11.2.2) and the *NVIDIA collective communications library* [@171] (NCCL) (v2.8.4). Each node provides 128 GiB of memory and is connected via a 50 Gbit/s (4X FDR) Infiniband network. Similar to the other MOGON II nodes, a Micron 5100 Pro SSD (XFS formatted) is available at each node for the user.

MareNostrum IV The MareNostrum IV supercomputer, located at the Barcelona Supercomputing Center in Spain, consists of 3,456 Lenovo ThinkSystem SD530 compute nodes on 48 racks. Each node uses 2 Intel Xeon Platinum 8160 24C chips with 24 processors each at 2.1 GHz and 390 TB of total main memory. In addition, each node uses a 240 GiB Intel SSD DC S3520 Series as scratch space. All GekkoFS experiments use these SSDs as their underlying storage for data and metadata.

MareNostrum IV uses a 100 Gb Intel Omni-Path fat tree for the intercommunication network with a 14 PiB parallel file system offered by IBM's GPFS. All GPFS experiments were run on the `projects` file system, offering 4.4 PiB of storage with an 8 MiB file system block size.

NEXTGenIO prototype The NEXTGenIO prototype uses 34 compute nodes and is located at the EPCC supercomputing center at the University of Edinburgh in the UK. Each node utilizes a dual Intel Xeon Platinum 8260M CPU @ 2.40 GHz (48 cores per node), 192 GiB of main memory, and 3 TiB of node-local *Intel Optane DC Persistent Memory (DCPMM)*. All GekkoFS experiments use these DCPMMs as their underlying storage for data and metadata.

The compute nodes are connected via an Omni-Path fabric with two network devices per node, called `ib0` and `ib1`. They use a 56 Gbps InfiniBand to communicate with a Lustre server using 6 OSTs. In the corresponding experiments, GekkoFS uses TCP/IP over Omni-Path (emulated TCP), while Lustre uses the Omni-Path fabric with Infiniband emulation.

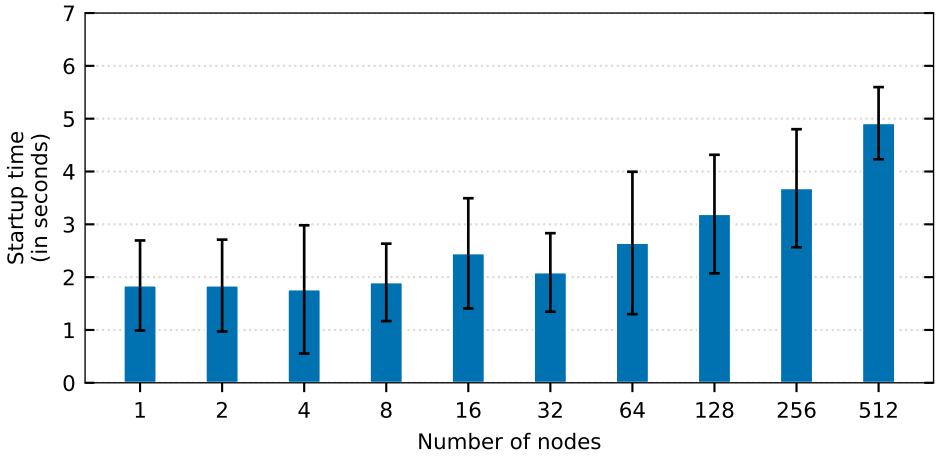


Figure 4.8.: GekkoFS daemon startup time for an increasing number of nodes.

4.5.2 Startup

When a file system is deployed ad hoc as part of a compute job, the additional time required to start the file system must be considered. Therefore, the required wall-time is either accounted directly to the user's quota or is part of the general job setup time, which includes preparing storage space on node-local for the user and the current job. Prolonged startup times reduce the efficiency of the ad hoc file system since the necessary time for the startup, data stage-in, and data stage-out must result in an overall decreased job wall-clock time compared to the application using the parallel file system instead. For instance, Soysal et al. [229] showed that starting the BeeGFS BeeOND ad hoc file system on 256 nodes can take more than 220 seconds. As a result, we defined a *fast deployment* as part of GekkoFS's goals (see Section 4.3.2) which we evaluate next.

On MOGON II, we started GekkoFS for up to 512 nodes, and for each node configuration, we computed the mean of at least five startups. To achieve a fast parallel launch, we used SLURM's `srun` command so that GekkoFS runs as part of a SLURM job step. SLURM [277] is a resource manager used in many HPC centers and offers the ability for users to submit *batch jobs* in which their applications run for a given wall-time.

Figure 4.8 presents the results showing that GekkoFS required at most 4.9 seconds on average for 512 nodes to provide a usable file system environment to the user. We note that, although the startup times remained mostly stable over several job and node configurations, we observed three significant outliers, which we omitted in the presented results, taking up to 30 seconds. Further investigation showed that a small number of daemons were unable to immediately register themselves

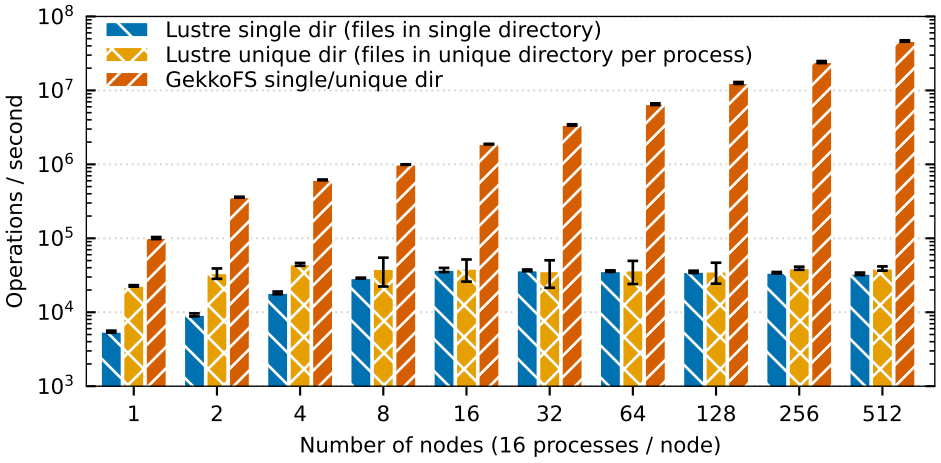
in the shared hosts file (see Section 4.3) that was placed on the parallel Lustre file system. In general, such minor delays when accessing a parallel file system are not uncommon and expected in storage systems that use distributed locking managers. Nevertheless, we aim to remove this hosts file dependency on the parallel file system as part of GekkoFS's future work.

4.5.3 Metadata performance

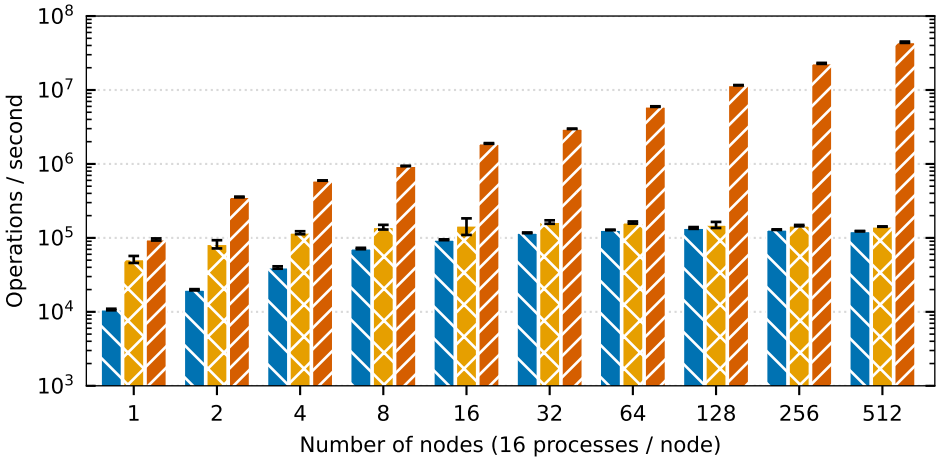
We simulated common metadata-intensive HPC workloads on MOGON II using the *mdtest* microbenchmark [@106] to assess GekkoFS's metadata performance. In our performed experiments, mdtest consecutively runs batches of `create`, `stat`, and `remove` operations in parallel in a single directory. For the GekkoFS experiments, each batch consists of 100,000 zero-byte files per process with 16 used processes per node. We ran the benchmark with this high number of files so that RocksDB flushes its in-memory tables to the underlying SSD, showing RocksDB's consistent performance. As a result, RocksDB created several *static sorted table files* (sst files) on the backend storage during the experiments. Concurrent metadata operations in a single directory are an especially important workload in many HPC applications, but they are also one of the most difficult workloads for a parallel file system to perform efficiently [255] (see Section 2.3.2).

While all files are created in a single directory (from the user's and application's perspective), there is conceptually no difference for GekkoFS if files are created within a single directory or spread across multiple directories. Notice also that traditional parallel file systems, on the other hand, may perform better when a workload is spread across many directories. In this case, inserting files only into a single directory mostly sequentializes the file creation process since multiple processes cannot access the same file system block in parallel. As a result, application developers are often asked to create files in a separate directory per process, even if this does not fit their natural ordering. This effect will be demonstrated in this section as well when comparing GekkoFS to Lustre and BeeGFS.

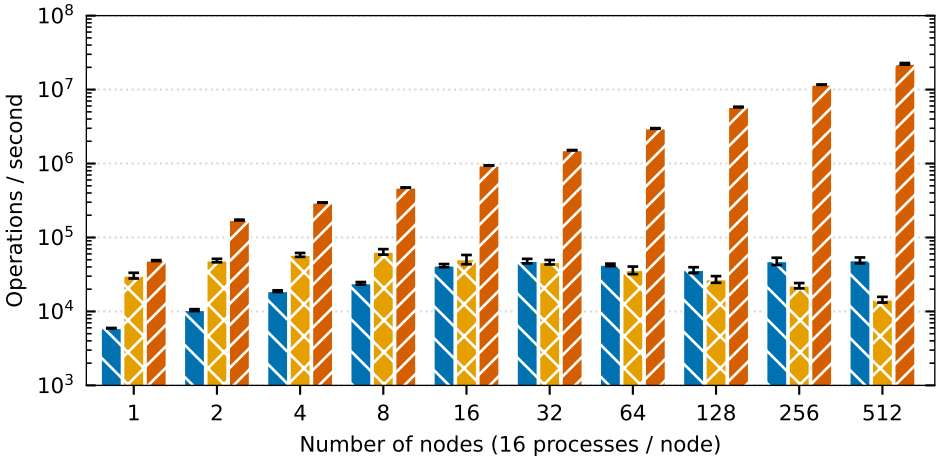
Figure 4.9 compares GekkoFS with Lustre with the above-described workload with up to 512 nodes (x-axis). The y-axis depicts the corresponding operations per second achieved for a particular workload on a logarithmic scale. For the experiment's runtime, GekkoFS was running exclusively on the compute cluster without other jobs interfering. Each data point represents the mean of at least five iterations of each experiment. GekkoFS's workload was weakly scaled, with 100,000 files per process doubling the workload when the node count is doubled as well. On the



(a) Create throughput



(b) Stat throughput



(c) Remove throughput

Figure 4.9.: GekkoFS's file create, stat, and remove throughput for an increasing number of nodes compared to a Lustre parallel file system.

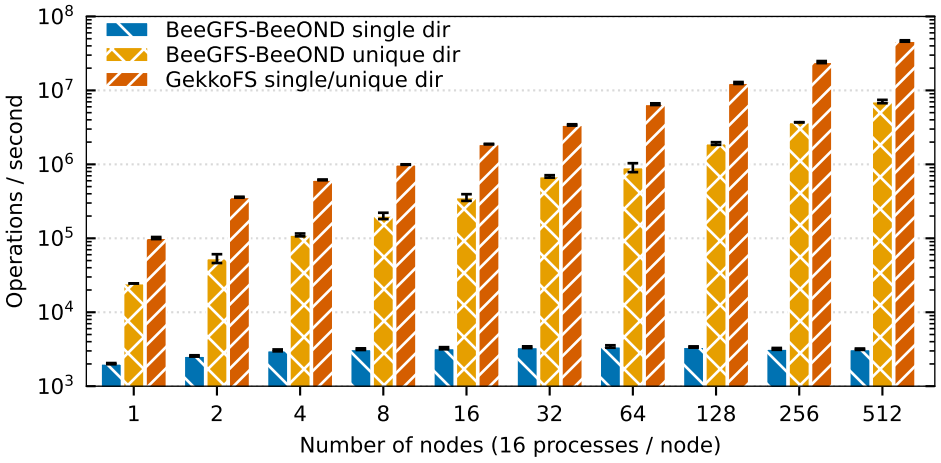
other hand, Lustre's workload was fixed to four million files for all experiments. We fixed the number of files for Lustre's metadata experiments because Lustre detected hanging nodes when operating on too many files and could therefore not complete its workload.

To examine the common advice to use multiple directories when operating on a large directory, we ran the Lustre experiments in two configurations: All processes operated in a single directory (`single dir`) or each process worked in its own directory (`unique dir`). Lustre's metadata performance was evaluated while the system was accessible by other applications as well.

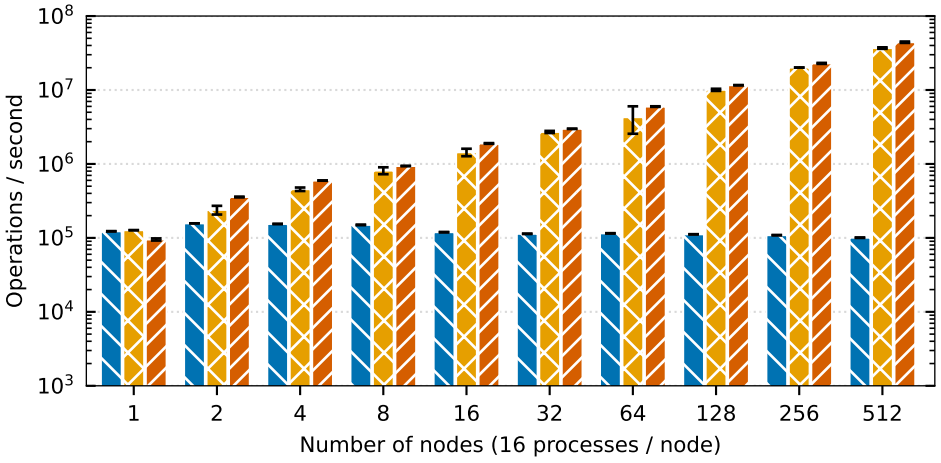
As seen in Figure 4.9, GekkoFS outperforms Lustre by a large margin in all scenarios, regardless of whether Lustre processes operated in a single or in an isolated directory. Compared to Lustre, GekkoFS achieved around 46 million creates per second (∼1,405x), 44 million stats per second (∼359x), and 22 million removes per second (∼453x) at 512 nodes. The relative standard deviation was less than 3.5% in all GekkoFS cases. For GekkoFS, each operation was performed synchronously without any caching mechanisms in place, showing close to linear scaling. Therefore, we achieve our scalability goal and have demonstrated the performance benefits of wide-striping metadata while decoupling directory entries from non-scalable directory blocks (see Section 4.3.5).

We also reran all experiments while MOGON II was used by other users in a production environment. In this situation, we were unable to measure a difference in metadata operations per second outside the margin of error with up to 128 nodes, compared to the above-shown experiments in an undisturbed environment. For 256 and 512 nodes, on the other hand, we measured a reduced metadata operation throughput between 10% and 20% for create and stat operations, respectively, revealing network interference within the cluster.

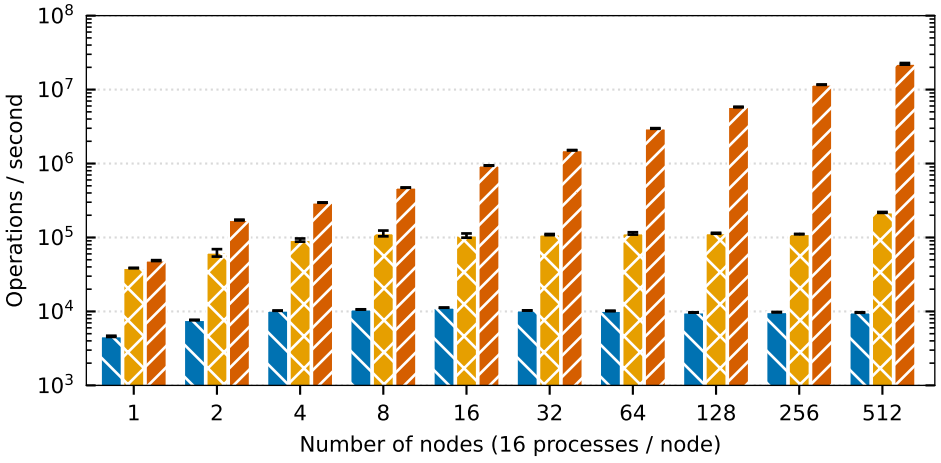
Lustre's metadata performance did not scale beyond approximately 32 nodes, demonstrating the aforementioned metadata scalability challenges of a general-purpose parallel file system. Note that if processes operated in their own directories, Lustre achieved a higher throughput in most cases. An exception is the remove case where Lustre's `unique dir` remove throughput is reduced by over 70% at 512 nodes compared with Lustre's `single dir` throughput. The reason is that the time required to remove the directory of each process (in which it creates its workload) is included in the remove throughput while the amount of `unique` directories increases as well with the number of the used process in an experiment. This is similar to the create operation, which also includes the creation of process directories, but does not show the same behavior, indicating optimizations towards create operations.



(a) Create throughput



(b) Stat throughput



(c) Remove throughput

Figure 4.10.: GekkoFS's file create, stat, and remove throughput for an increasing number of nodes compared to a BeeGFS's BeeOND.

In Figure 4.10, we compare GekkoFS with BeeGFS BeeOND, which is a POSIX-compliant burst buffer file system. We configured BeeGFS to match GekkoFS's properties as close as possible, i.e., distributing all data and metadata across all file system nodes and using the same file system block size (or chunk size in GekkoFS's case). Further, BeeGFS is deployed in an ad hoc fashion for the lifetime of a compute job and destroyed afterward while all metadata is stored on the node-local SSD. It is therefore only accessible by the benchmark application. Similar to the Lustre experiments, we ran BeeGFS experiments in two configurations: All processes operated in a single directory (`single dir`) or each process worked in its own directory (`unique dir`). GekkoFS's workload and all BeeGFS `unique dir` experiments were weakly scaled with 100,000 files per process. On the other hand, the workload for BeeGFS `single dir` experiments was fixed to four million files. This is because BeeGFS's distribution to multiple metadata servers is coupled with the number of directories. Therefore, single directory experiments only utilize a single metadata server, which inherently congests the responsible node, significantly impacting scalability.

Figure 4.10 presents the corresponding metadata experiments for both GekkoFS and BeeGFS. In both cases, the `unique dir` experiments show close to linear scalability with the number of nodes for the create and stat operations. However, BeeGFS's remove performance stagnated after four nodes, although its throughput doubled from 256 to 512 nodes. GekkoFS's unique dir and single dir performance are equivalent as both scenarios are internally treated indifferently. In summary, at 512 nodes, GekkoFS achieved a $\sim$6.5x higher create throughput, a $\sim$1.2x higher stat throughput, and a $\sim$102x higher remove throughput, compared to BeeGFS `unique dir` scenario. All BeeGFS `single dir` experiments did not scale beyond four nodes due to the above-described limitations.

4.5.4 Data performance

On the MOGON II supercomputer, we evaluated GekkoFS's I/O performance and designed our experiments to reflect some of the most difficult I/O patterns that scientific applications request from a parallel file system, e.g., small I/O requests or random access patterns. For this task, we ran the commonly used *IOR* microbenchmark [@106], offering a rich configuration interface to evaluate a file system's I/O performance in several scenarios. We performed our experiments for sequential and random access patterns in two configurations: Each process accesses its own file (*file-per-process*), and all processes access a single file (*shared-file*). We used four different write and read sizes, 8 KiB, 64 KiB, 1 MiB, and 64 MiB, in each

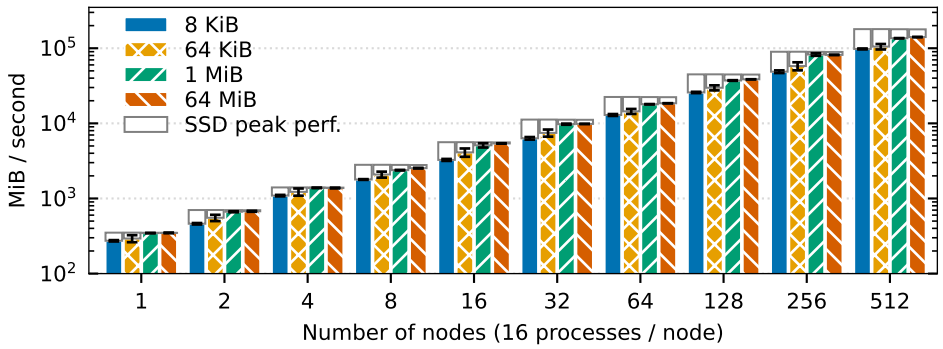
configuration, which we refer to as *transfer sizes* henceforward. Therefore, these configurations assess the file system's performance for many small I/O accesses and few large I/O requests. We ran 16 processes per client, each writing and reading 4 GiB in total.

We did not compare GekkoFS's data performance with the Lustre scratch file system because its peak performance for the assigned hardware, around 12 GiB per second, was already reached for ∼10 nodes for sequential I/O patterns. Moreover, Lustre has shown to scale linearly with the number of nodes for sequential access patterns in much larger deployments that employ more OSSs and OSTs [178]. Instead, we first focus on GekkoFS's behavior for various transfer sizes and I/O patterns to achieve close to linear scalability while evaluating its efficiency when using the node-local SSDs. Afterward, we compare GekkoFS's I/O performance with BeeGFS's BeeOND in similar file system configurations.

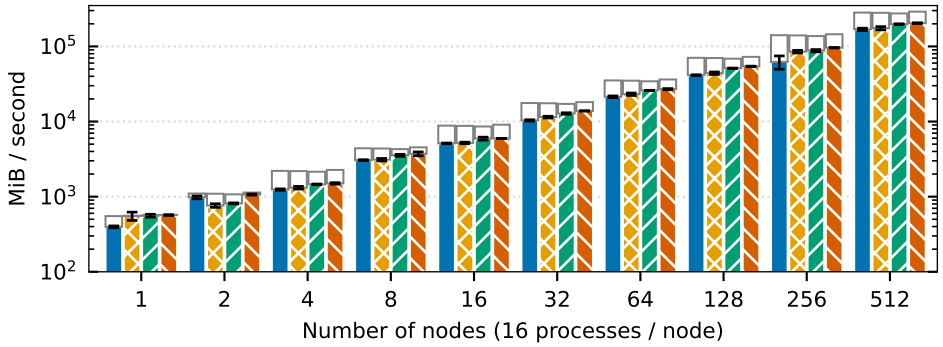
First, we will discuss the file-per-process throughput results, the *write and read operations per second* (IOPS), and the latencies. After, we will evaluate the shared-file performance and its challenges.

File-per-process and sequential access patterns GekkoFS's sequential read and write throughput is presented in Figure 4.11 for an increasing number of nodes (x-axis) and four transfer sizes (bars). Each data point represents the mean of at least five iterations. We calculated the relative standard deviation which was smaller than 5% in all cases except for 64 KiB writes varying up to 13%. We also compared each data point to the peak performance that all aggregated SSDs could deliver for a given node configuration if used locally (SSD usage efficiency), visualized as a white rectangle above each bar. In other words, the throughput disparity between a data point and SSD peak performance shows the cost of operating in a distributed namespace using GekkoFS. The results demonstrate GekkoFS's close to linear scalability with the number of nodes. For 512 nodes, it achieves about 141 GiB/s (∼80% SSD usage efficiency) and 204 GiB/s (∼70% SSD usage efficiency) in the corresponding write and read operations for 64 MiB transfer sizes, respectively.

Notice that any I/O operation larger than the used chunk size of 512 KiB is internally split into equally sized chunk files on the node-local file system. GekkoFS further packages all chunks assigned to the same node into a single RPC request, allowing the file system to perform parallel RDMA accesses to the client's memory from multiple nodes. Therefore, chunks of 64 MiB I/O requests can be served in parallel,



(a) Write throughput

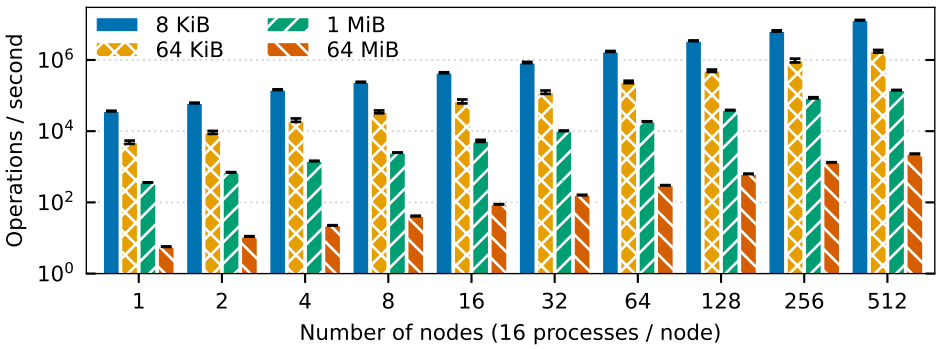


(b) Read throughput

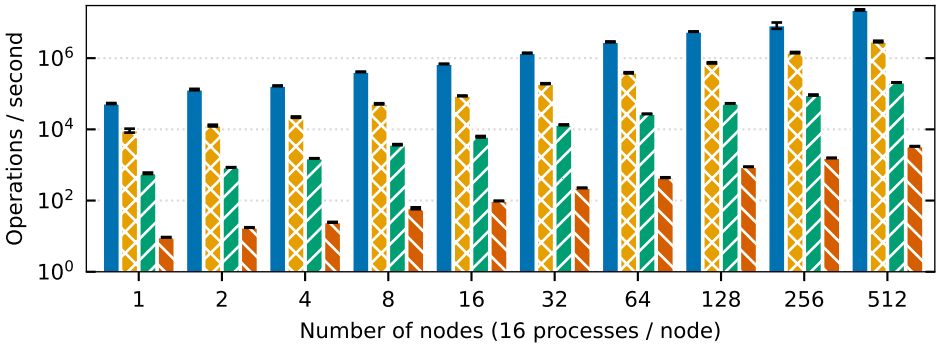
Figure 4.11.: GekkoFS's sequential write and read throughput for various transport sizes with each process operating on its own file compared with the plain SSD peak throughput.

achieving a higher throughput than 1 MiB I/O requests. I/O requests smaller than the chunk size always target a single chunk and thus a single node.

Figure 4.12 shows the *I/O operations per second* (IOPS) for an increasing number of nodes (x-axis). IOPS is another representation of the throughput evaluation of Figure 4.11 which only focuses on the raw operation rate without taking the transfer size into account. The metric is typically used as a performance measure for storage devices, but it is also sometimes used as a measurement in file systems, particularly when discussing metadata operations or small I/O operations. Because the transfer sizes are not taken into account, IOPS tend to be higher for smaller I/O requests since the amount of time the target node spends during an I/O operation is lower than for larger I/O requests. Therefore, the latency of each I/O request becomes the predominant factor in a small I/O operation, causing a decrease in throughput (MiB/s) and an increase in IOPS. GekkoFS confirms this expectation, achieving more than 13 million write IOPS and more than 22 million read IOPS at 512 nodes with an 8 KiB transfer size.



(a) Write IOPS



(b) Read IOPS

Figure 4.12.: GekkoFS's sequential write and read operations per second (IOPS) for various transport sizes with each process operating on its own file.

In Figure 4.13, we investigate the I/O latencies for such small I/O requests, exemplarily shown for 8 KiB transfer sizes. Each data point represents the mean latency for all write or read requests over five iterations. Also, the experiments were weakly scaled (4 GiB in total per process), and therefore more nodes translated into more IOPS. For instance, at 512 nodes, the I/O latency of over 167 million operations was considered. In all the presented cases, the I/O latencies were well below a modern hard disk drive's capability, which has access times of several milliseconds, showing GekkoFS's efficient utilization of the SSD storage backend. Note that the latencies of read operations are lower than of write operations since the latter involves an additional communication step to update the size of the file.

Further, we noticed rare but severe outliers, which resulted in a high relative standard deviation that needs to be further investigated. Despite these outliers, the throughput and IOPS in various scenarios, as shown above, were stable with a low relative standard deviation.

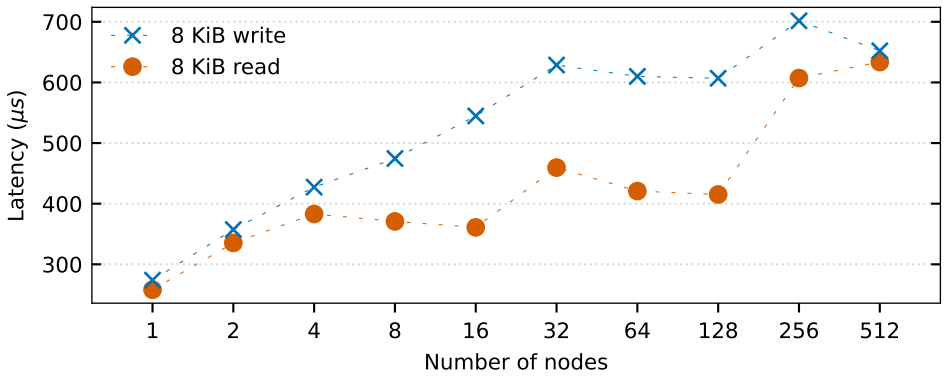
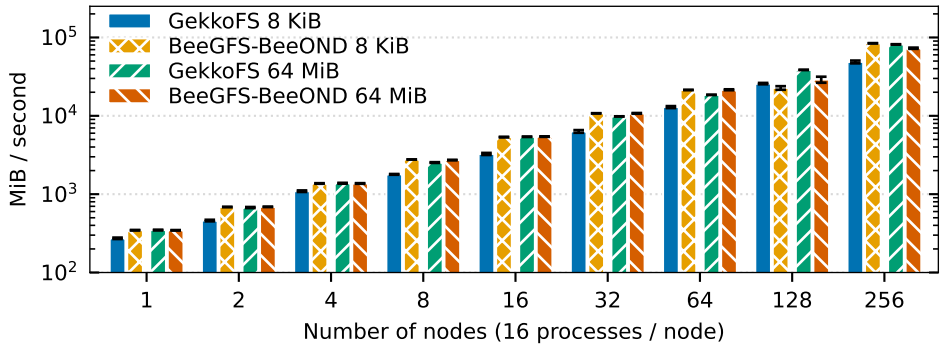
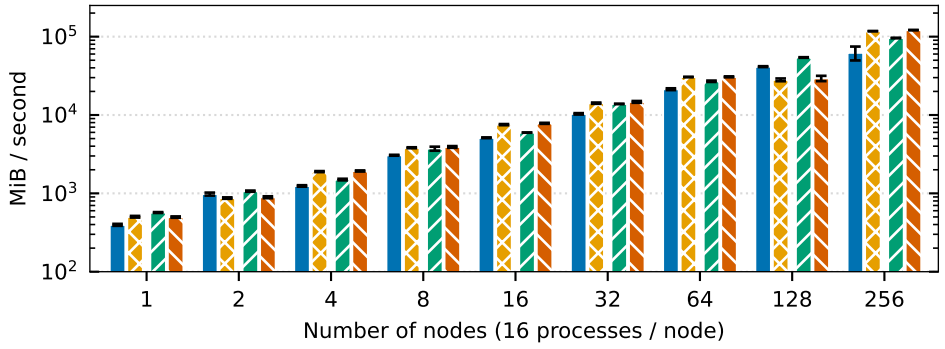


Figure 4.13.: GekkoFS's I/O latencies for 8 KiB transfer sizes.



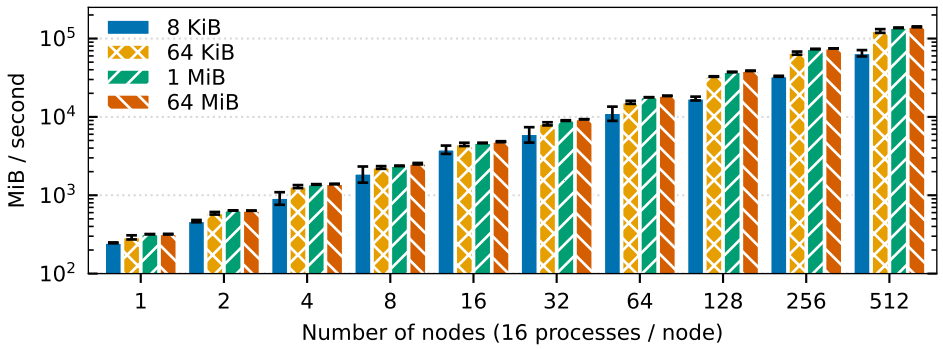
(a) Write throughput



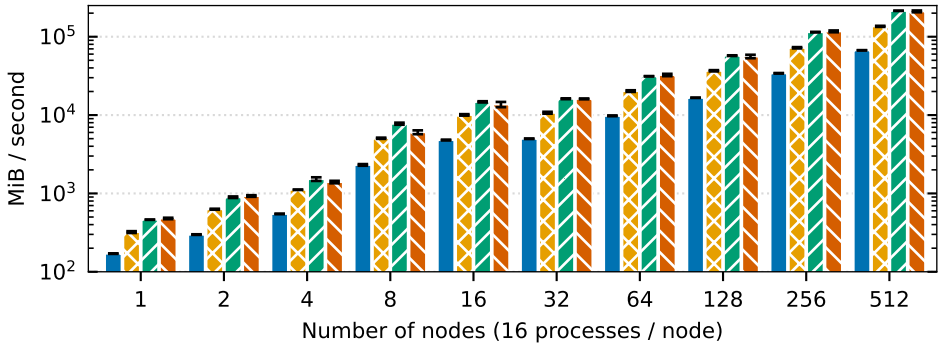
(b) Read throughput

Figure 4.14.: GekkoFS's sequential write and read throughput for various transport sizes with each process operating on its own file compared with BeeGFS's BeeOND.

In Figure 4.14, we compare GekkoFS with BeeGFS's BeeOND up to 256 nodes in two configurations, 64 MiB and 8 KiB transfer sizes, to evaluate large and small transfer sizes alike. Both burst buffer file systems scale almost linearly, exhibiting a relative standard deviation smaller than 9% in all cases. As GekkoFS, BeeGFS uses node-local SSDs for data distributed across all file system nodes. Notice that BeeGFS's



(a) Write throughput

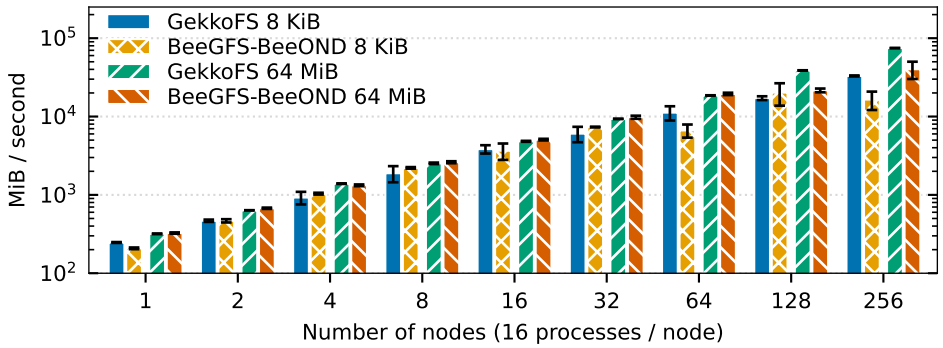


(b) Read throughput

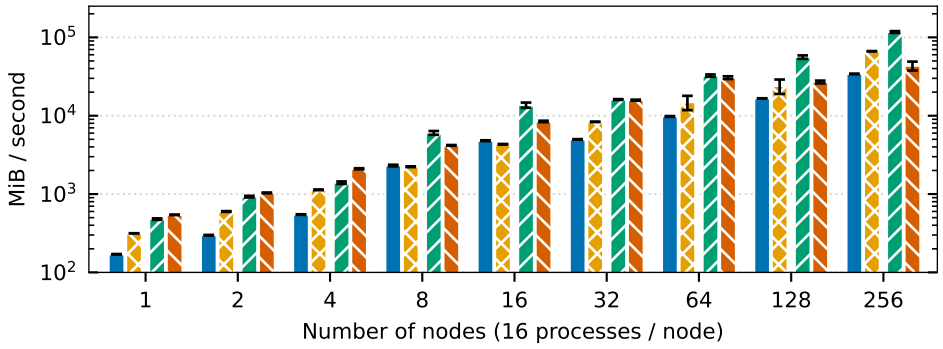
Figure 4.15.: GekkoFS's random write and read throughput for various transport sizes with each process operating on its own file.

distribution setting, i.e., to how many nodes data is spread, can be controlled via the *stripe* setting that can be set on a directory and file granularity. We set the test directory's stripe size to -1 to achieve a wide-striping configuration. At 256 nodes with 64 MiB transfer sizes, GekkoFS's write throughput is ~1.12x higher than BeeGFS while, on the other hand, BeeGFS's read throughput is ~1.26x higher than of GekkoFS. For 8 KiB transfer sizes, BeeGFS's write and read throughput is ~1.73x and ~1.86x higher than GekkoFS, respectively. For BeeGFS, we performed additional experiments and used several transfer sizes ranging from 1 MiB to 8 KiB, showing a similar write and read throughput, suggesting caching mechanisms for such transfer sizes. Because GekkoFS is not utilizing any caching mechanisms, a reduction in I/O throughput for smaller transfer sizes is expected since each I/O request must be sent individually, thereby becoming increasingly more latency-dependent in the process. We plan to target these small I/O use cases as part of the future work.

File per process and random access patterns Figure 4.15 shows GekkoFS's throughput (y-axis) for random accesses for an increasing number of nodes (x-



(a) Write throughput

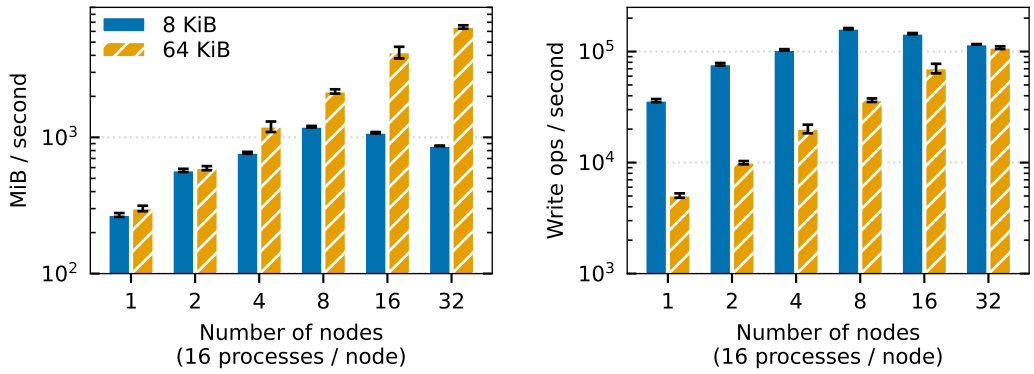


(b) Read throughput

Figure 4.16.: GekkoFS's random write and read throughput for various transport sizes with each process operating on its own file compared with to BeeGFS's BeeOND.

axis), exhibiting close to linear scalability in all cases. The file system achieved up to 141 GiB/s write throughput and up to 204 GiB/s read throughput for 64 MiB transfer sizes at 512 nodes.

For GekkoFS's behavior of the used transfer sizes, we expect that the throughput for random access patterns and transfer sizes larger than the chunk size (512 KiB in our case) are similar to the results presented earlier in Figure 4.11. This is because I/O operations of such cases are internally handled indifferently. Regardless of the access pattern (sequential or random), whole chunk files are written to and read from GekkoFS's data backend while distributed across several nodes. The results confirm this expectation, showing that the throughput for 64 MiB and 1 MiB transfer sizes are comparable to the sequential results. For transfer sizes smaller than the chunk size, we expected reduced performance compared to the sequential results as the chunk files in GekkoFS's data backend are randomly accessed as well. We observed a decreased write and read throughput by approximately 33% and 60%, respectively, for 512 nodes and a transfer size of 8 KiB. Hence, applications may benefit from choosing smaller chunk sizes if their transfer sizes are small.



(a) Write throughput

(b) Write IOPS

Figure 4.17.: GekkoFS's sequential write throughput and write IOPS for each process operating on a single shared file.

Figure 4.16 compares GekkoFS's throughput for random accesses with BeeGFS for up to 256 nodes and two transfer sizes: 64 MiB and 8 KiB. For 256 nodes with 64 MiB transfer sizes, GekkoFS's write and read throughput is ~1.86x and ~2.7x higher than of BeeGFS, respectively, possibly showing the above-described benefits of transfer sizes larger than the chunk size. For small 8 KiB transfer sizes, GekkoFS's write throughput is ~2x higher than BeeGFS while BeeGFS's read throughput is ~1.95x higher than of GekkoFS.

Single shared file Because of GekkoFS's design, shared file operations have many similarities to the previously presented experiments, where each process operated on its own file. For example, if the transfer size is larger than the chunk size, each chunk file is only accessed by a single process, regardless of whether the whole file is shared among many processes or accessed by just a single process. In cases where multiple processes access the same offset in the same chunk file, local locking mechanisms of the node-local file system serialize the access to this file (see Section 4.3.6).

In Figure 4.17, we present the write throughput and write IOPS for 64 KiB and 8 KiB transfer sizes when sequentially operating on a single shared file for up to 32 nodes. However, we can observe a drawback of GekkoFS's synchronous and cache-less design in these cases as no more than ~150K write operations per second were achieved. We omit experiments beyond 32 nodes since the throughput and the IOPS stagnate. We pinpointed the reason for this behavior at the file size metadata field because the single shared file only uses a single metadata entry maintained by only one node. Thus, it needs to be constantly updated by all nodes in a mutually exclusive way. This results in a large number of small RPC messages, each updating

the file size, causing network contention on the metadata-maintaining node. This observation is supported by the fact that read operations did not exhibit the same bottleneck because no metadata updates are necessary and that increasing the number of RPC handler threads did not result in a higher number of write IOPS.

The bottleneck becomes worse the more processes participate in an experiment. Eventually, this reaches a plateau, visible in the number of write IOPS stagnating. If we compare each GekkoFS daemon node's metadata load to the file-per-process experiments, each node only handled at most 25K write operations per second on average. As a result, we did not observe similar behavior in the previous experiments.

To combat this bottleneck, we added a rudimentary client cache to locally buffer size updates during consecutive write operations before sending them to the GekkoFS daemon that manages the file size. This, in turn, drastically reduces the number of RPC messages being sent to a single node. As a result, the sequential throughput on a single shared file for up 512 is showing close to linear scalability. The corresponding throughput and IOPS results are available in Figures A.1 and A.2 in the appendix. Again, due to the internal similarities in file system logic with regards to the data distribution, shared file performances are now similar to the above-shown file-per-process results. While such a cache may not be ideal for all applications, it significantly mitigates node contention.

4.5.5 I/O variability and worst-case

We ran several experiments to highlight GekkoFS's benefits in terms of I/O variability and throughput predictability. When HPC users ask for compute time in the context of an HPC job, they have to decide on the compute job duration to run their application. This duration estimate is important because the job allocation is revoked when the allocated time ends, terminating the application. Yet, the parallel file system I/O performance can differ significantly, making this estimate difficult, especially for data-driven applications. As a result, users tend to grossly overestimate the time they actually require for their application to finish [67, 266]. This, in turn, can cause the batch job scheduler to make inefficient decisions when placing a job into its queue.

On MareNostrum IV, we used the IOR benchmark to measure the I/O variability and worst-case I/O performance that applications experience both for GekkoFS and for the HPC system's GPFS parallel file system. To get an accurate real-world picture,

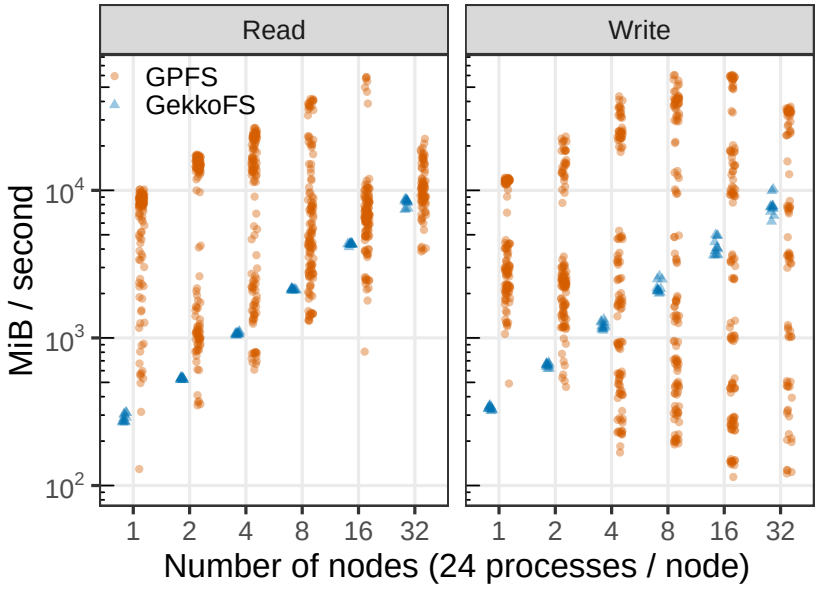


Figure 4.18.: I/O variability of GPFS and GekkoFS of multiple IOR runs on different times and node allocations throughout one week.

we ran all experiments during production, co-located with ordinary HPC workloads. We ran 25 independent repetitions of the same benchmark set for each file system, each using different node allocations at different times throughout one week. Each set of benchmarks used varying I/O request sizes ranging between 512 bytes and 64 MiB to evaluate a more realistic representation of real-world applications that use different I/O sizes throughout their runtime. In addition, we varied GekkoFS's chunk size as well, ranging between 128 KiB and 64 MiB. GPFS's block size is not dynamically changeable during production and remained 8 MiB. For the benchmarks, we used 24 out of the 48 available cores on each node. We ensured that the I/O size was large enough to fill the node's main memory to avoid cache effects. The IOR processes operated each on their own file, which was initially created at the beginning of the job.

Figure 4.18 presents the I/O variability of GPFS and GekkoFS as squares and triangles, respectively, up to 32 nodes (x-axis) for each IOR run. For GPFS, we observed significant variability in its throughput performance (y-axis), often scattered across orders of magnitude, with writes showing a higher variability than reads. In reality, this I/O behavior is commonplace at many HPC sites [114, 139, 272], known as *cross-application interference*. It is caused by an I/O bottleneck where a shared resource, e.g., a parallel file system, is accessed by multiple, uncoordinated applications. Nowadays, these kinds of I/O bottlenecks are already a great challenge at many

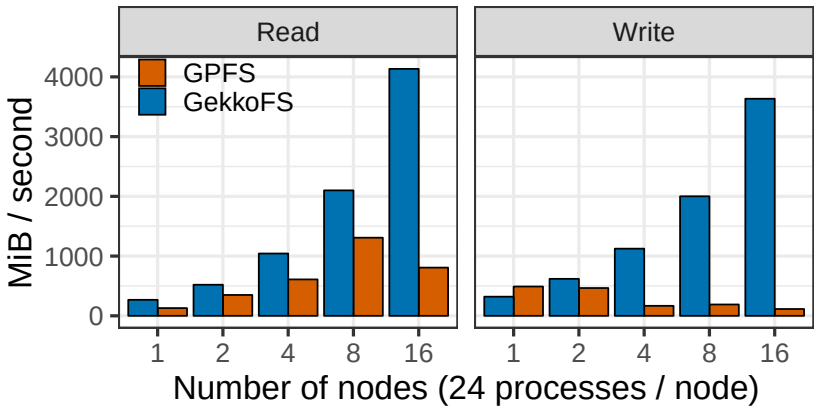


Figure 4.19.: I/O worst-case of GPFS and GekkoFS of multiple IOR runs on different times and node allocations throughout one week.

HPC sites and could become one of the core challenges in exascale supercomputers in the future [91, 138, 203].

GekkoFS, on the other hand, shows steady and predictable performance with low cross-application interference. This shows the advantages of using a node-local burst buffer file system accessed by a single application. In both read and write cases, GekkoFS scales almost linearly with the number of nodes as the available bandwidths and storage capabilities increase, which eventually closes the gap to GFPS's best-case performance. If we observe the worse-cast performance of both file systems at a specific node number (x-axis) in Figure 4.19, GekkoFS considerably excels GPFS's throughput in most cases due to the low impact of cross-application interference on GekkoFS. This shows that a burst buffer file system like GekkoFS can also help users to more easily predict the achieved I/O performance for a node configuration and, therefore, be able to give more precise job time estimates, allowing for more precise job scheduler decisions.

4.5.6 Effects on the network

To investigate GekkoFS's effects on the network, we ran the *OpenFOAM* application on the NEXTGenIO prototype. OpenFOAM [109] is a C++ library for developing user-customized numerical solvers for the solution of *continuum mechanics* problems, including *computational fluid dynamics*. The application's solvers often run multiple stages in a single execution and involve large amounts of I/O operations. OpenFOAM specifically benefits from distributing its workload on multiple nodes while

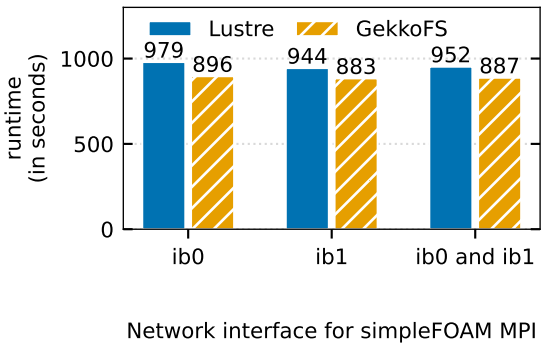


Figure 4.20.: Runtime of the simpleFOAM application when its MPI communication is run on different network interfaces (`ib0` and `ib1`).

parallelization is achieved via MPI. OpenFOAM solvers are used in both academia and industry in large-scale computations [@26, @102, @204].

In our experiments, we used the *simpleFOAM* solver, a steady-state solver for incompressible, turbulent flow, using the *semi-implicit method for pressure-linked equations* (SIMPLE) algorithm. We ran the application with 100 iterations, whereas the first time step generated around 25 GiB of data spread over 170K files. For the experiments, we used 4 nodes (24 processes per node) with the application pinned to socket `0`. GekkoFS and Lustre both used the `ib0` Omni-Path adapter. To investigate the file systems' effects on the network, SimpleFOAM's MPI communication was run in three configurations: MPI using the `ib0`, `ib1`, or `ib0` and `ib1` Omni-Path adapters. Therefore, in the `ib1` case, MPI communication is separated from Lustre's and GekkoFS's internal file system network traffic.

Figure 4.20 presents the runtime (y-axis) of the experiments when run with both file systems. Each of the three bar groups represents one of the above-described configurations. Although simpleFOAM in this configuration does not generate significant I/O, GekkoFS is still ∼7% to ∼9% faster in all cases compared with Lustre. Further investigation and profiling revealed that the runtime improvements were caused by the `MPI_Waitall` and `MPI_Allreduce` functions and not from direct file system operations. Hence, both MPI functions could benefit from less generated network pollution of GekkoFS as compared with Lustre. This is because we used the local data distributor where data is locally written but metadata is still distributed (see Section 4.3.6). As a result, less file system traffic is being put onto the network in general compared with Lustre, where all file system communication is remote. Therefore, in addition to GekkoFS's linear scaling for metadata and data operations, applications and their inter-node communication can also benefit from GekkoFS to reduce file system network pollution.

4.5.7 Distributed deep learning

In the following, we will present some of the results by Schimmelpfennig and Vef et al. [218] within the context of GekkoFS and deep learning (DL). Specifically, we will discuss the training performance of various compute-bound and I/O-bound training models when run with Lustre, GekkoFS, and node-local storage. Therefore, [218] includes a much more detailed discussion and evaluation of this topic. Further, the work discussed the impact of data shards (see Section 4.4) on training accuracy of various popular data sets, which are not included below.

To run the DL workloads, we used TensorFlow [1] (v2.4.1) and Horovod [222] (v0.21.3). TensorFlow is a widespread end-to-end open source machine learning platform that helps researchers and developers to train machine learning models. In a distributed setting, however, the model synchronization of gradients via *all-reduce* operations before each training step is limited. Frameworks, such as Uber's Horovod which can use TensorFlow, have therefore become an attractive option to run DL workloads on multiple nodes with multiple GPUs on each node. Further, Horovod supports NVIDIA's NCCL [@171] library which allows inter-node communication to use RDMA with Infiniband fabrics.

For measuring the training times, we used the popular ILSVRC2010 ImageNet data set [57] which consists of 1.26 million images with an average size of 109 KiB. Figure 4.21 presents the step time (y-axis) analysis for running a distributed training with Horovod on eight nodes when the data set is loaded from Lustre, locally, or via GekkoFS. In the latter two, we also differentiate between using memory and SSD. Each data point represents the mean of the average step times of at least five *training epochs* (lower is better). Within one epoch, each training set sample is applied once to the model. Before each training step, the model gradients are synchronized, representing an all-reduce operation. The data set is run with different training models and batch sizes (bs, see x-axis) ranging from compute-bound to I/O-bound. A batch size dictates how many training samples are part of a single training step.

For instance, Res50 is highly compute-bound since its step times did not change regardless of the used storage backend. Alex (`bs128`), on the other hand, is highly I/O-bound because the performance discrepancies were highest between the Lustre, GekkoFS, and local storage backends. The figure clearly shows that a parallel file system, such as Lustre, is unsuitable for I/O-bound deep learning workloads. Nevertheless, it is important to remember that parallel file systems are typically not used directly for training, and therefore it represents a performance baseline for the local and GekkoFS cases. In practice, node-local storage is generally used for

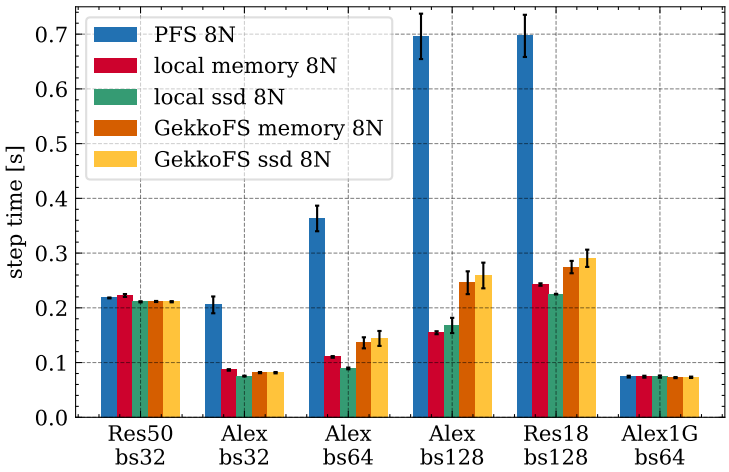


Figure 4.21.: Comparison of the average training step times for different setups, models and file systems using the ImageNet data set [218].

training which, as expected, showed the best performance in both memory and SSD cases. Because GekkoFS is aggressively distributing the data across all nodes, there is a certain performance penalty due to remotely accessing the training data visible when compared with the local cases. The cost of remote access has also been shown earlier in Section 4.5.4.

However, when comparing the local and GekkoFS cases, it is important to consider the data volumes that need to be copied. In the above local case, the complete data set was available on each of the eight nodes. In other words, the training data was copied to node-local storage eight times, wasting the bandwidth of the parallel file system. In the GekkoFS case, the data only needed to be copied once to its shared global namespace. What is more, in cases where the training set exceeds the node's storage capabilities, the data must be partitioned in a way to minimize biases, called data shards, whereas the shards are then distributed across the nodes. While shards do not necessarily impact training accuracy, minimizing biases in shards is not trivial and can result in biases that are not immediately visible [218]. Therefore, GekkoFS can offer a transparent way for shard-free distributed training.

In summary, Schimmelpfennig and Vef et al. [218] proposed a transparent workflow using GekkoFS which does not rely on shards or other means of data packing. As a result of its shared namespace, researchers do not need to minimize biases when creating shards because each node has access to the full training set, requiring only a single copy from the parallel file system. Finally, further evaluation in [218] showed that GekkoFS scales linearly with common DL workloads (not shown in this thesis) while the file system does not interfere with the all-reduce model synchronization process.

4.6 Future work

At the time of writing, GekkoFS remains in continuous development due to the file system being funded through ADMIRE (*Adaptive multi-tier intelligent data manager for Exascale*) and the FIDIUM (*Federated Digital Infrastructures for Research on Universe and Matter*) projects. This section will outline some of the work that is planned for GekkoFS in the context of these projects and beyond.

Malleability and interference In the context of the ADMIRE project, GekkoFS will be significantly extended to support malleability, allowing various internal configurations changeable during runtime. Such configurations include increasing or decreasing the number of file system nodes, caching policies relaxing consistency guarantees, data placement strategies, and QoS to control the used network or computational resources. The specific malleability policies will be implemented into GekkoFS after a detailed analysis of the HPC applications used in ADMIRE. Finally, we will investigate GekkoFS's interference with other applications (that are running on the same node) and how to minimize it. For example, this can be achieved by reducing the general RPC throughput, decreasing CPU utilization, or prioritizing application network traffic over the file system, as supported by Intel's Omni-Path.

Reliability We also plan to support reliability with GekkoFS when a file system instance runs for more extended periods, e.g., weeks or months. This allows the file system to be used in long-term use cases, such as when compute jobs access the same data repeatedly. For this, GekkoFS needs to support resilience mechanisms, e.g., mirroring or two-error correcting erasure codes. This could be complemented by a reliability protocol with recovery mechanisms in the future. However, GekkoFS is not planned to run for years as a general parallel file system does.

Data and metadata backends Today, GekkoFS can use any node-local storage device accessible via a path to the user. Therefore, GekkoFS can use storage hardware ranging from spinning HDDs and SSDs to memory. We plan to extend GekkoFS to efficiently use future storage technologies, e.g., *non-volatile main memory* (NVMM) devices. While kernel-based file systems exist that are specifically optimized for NVMM devices and provide a path that GekkoFS could use, Moti et al. [160] showed that the OS and its software stack complexity hold back NVMM devices from unfolding their full potential. Their Simurgh file system runs fully in user

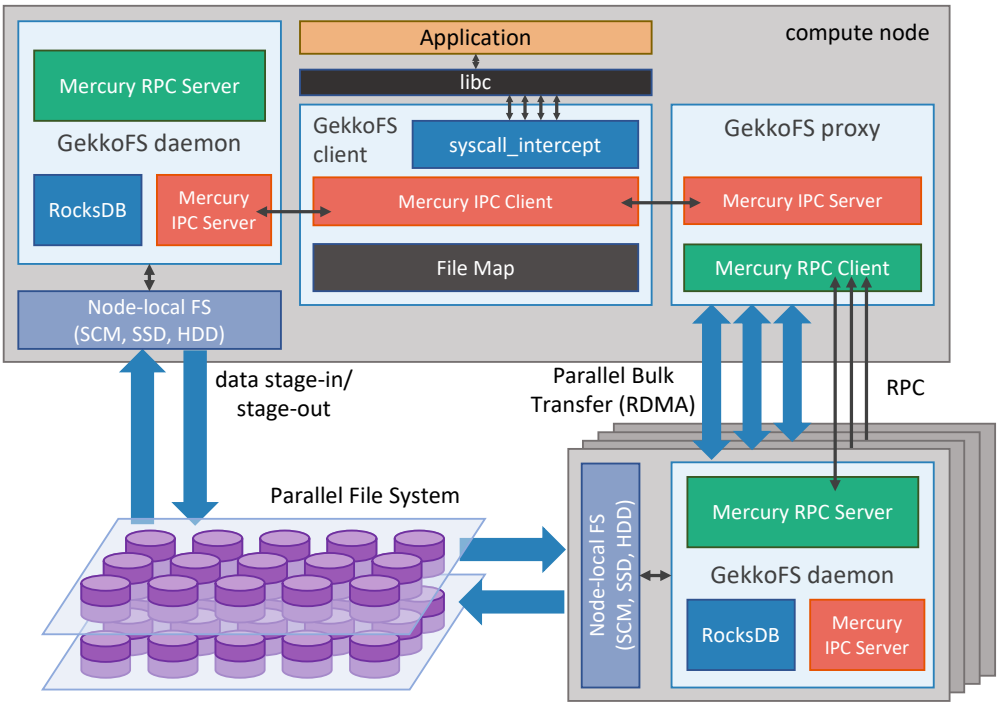


Figure 4.22.: GekkoFS architecture and its components with the GekkoFS proxy.

space, providing significant performance improvements (up to an 89% increase in application runtime), and could be used in the future for NVMM by GekkoFS.

Proxy forwarding Because a GekkoFS client is started implicitly when an application (process) is launched via library preloading, any potential optimizations, e.g., client caching, is limited to the corresponding preloading process. Especially, batching of I/O requests and support for asynchronous operations showed significant performance benefits in other work [197]. Therefore, we plan to extend the GekkoFS architecture by an additional forwarding component, called *GekkoFS proxy*, that runs on each client node. All GekkoFS client requests must thus go through this component as depicted in Figure 4.22.

The first preliminary performance results were promising, showing up to 85% efficiency in metadata workloads and similar performances for data workloads in comparison to GekkoFS's original design. The corresponding performance results are visualized in Figures A.3 and A.4 in the appendix for zero-byte files and 4 KiB sized files metadata experiments, respectively. Further, sequential data experiments are also presented in Figure A.5 in the appendix.

4.7 Summary

Larger supercomputers and higher I/O expectations and requirements have increased the pressure on common parallel file systems, starring new workloads, e.g., from the data-driven science domain. To combat I/O interference, slow metadata operations, and more predictable performance, both software-based and hardware-based approaches have been proposed. One of the software-based approaches is to use a burst buffer file system that is placed between the application and the parallel file system and utilizes already available node-local storage, which applications often underutilize.

This chapter has presented GekkoFS[5], a high-performing distributed burst buffer file system in user space that can be deployed ad hoc by any non-privileged users in a matter of seconds. By choosing the `LD_PRELOAD` method and intercepting I/O-related system calls within the GekkoFS namespace, the file system can run entirely in user space. As a result, GekkoFS does not need to implement common central data structures that are required by the virtual file system (VFS), e.g., inodes or directory blocks. Avoiding such data structures and running in user space is essential for GekkoFS's design, which is fundamentally based on a decoupled architecture and can, as a result, avoid inter-node locking communication and other synchronization protocols. By additionally relaxing certain file system operations and aggressively distributing metadata and data, GekkoFS has shown to linearly scale both metadata and data operations with the number of compute nodes. As such, it can scale to millions of metadata operations already for a small number of nodes and efficiently use node-local storage for data operations. GekkoFS achieves these features while still offering strong consistency semantics when accessing a single file system resource, e.g., a file.

Since its initial publication, GekkoFS has been the basis for other works and student theses [8, 13, 22]. For instance, Bez et al. [22] demonstrated in their paper the benefits of dynamic *I/O forwarding* [110, 175, 258] by adding a forwarding module as part of GekkoFS, called *GekkoFWD*. To reduce interferences at the parallel file system, GekkoFWD uses the AGIOS [25] library, which includes several *I/O scheduling algorithms* [124], aiming to alleviate these interferences by using machine learning to pick the best algorithm for a particular application. In their experiments, GekkoFWD could improve the aggregated bandwidth of a queue of nine heterogeneous applications by up to 85% compared with the typically-used static allocation approach. Moreover, GekkoFS has placed 4th in its first submission to the IO500's

[5] `https://storage.bsc.es/gitlab/hpc/gekkofs`

10-node challenge [@105] at the *"International Conference for HPC, Networking, Storage, and Analysis"* (SC) conference in 2019. The IO500 list represents an effort of the storage community to provide a set of experiments (based on mdtest and IOR [@106]) that should capture a file system's performance that is representative, comparable, and trustworthy. GekkoFS has further inspired other researchers to build on its architecture with *MadFS* [@146], showing that GekkoFS's design philosophy can scale to significantly larger supercomputers than those presented in this thesis. MadFS uses a similar architecture compared to GekkoFS [@146], including the hash-based distribution algorithm, syscall intercept via `LD_PRELOAD` for their clients, and RocksDB for metadata and chunks for data on the server-side. Nevertheless, MadFS differs in other areas, e.g., in their used network layer (not using Mercury), and offering other optimizations, e.g., placing small files within RocksDB. At the time of writing, MadFS remains in first place on IO500's main list.

In the future, we plan to significantly extend GekkoFS in several directions. These additions mainly focus on I/O malleability, longer-running workflows, and reliability. In addition, GekkoFS will feature further data and metadata backends, data distribution algorithms, such as random slicing as proposed by Miranda et al. [156], and a proxy forwarding component allowing GekkoFS to utilize client caching mechanisms.

5

DelveFS: An event-driven semantic file system for object stores

Data-driven applications are becoming increasingly important in numerous industrial and scientific fields, growing the need for scalable data storage, such as object storage (see Section 2.4). Yet, many data-driven applications cannot use the custom object interfaces directly and typically have to rely on third-party file system connectors that offer a standard POSIX file system interface. What is more, these connectors only provide a basic representation of objects as files in a flat namespace, that is, a single directory. Given that object store buckets can grow to millions of objects, such a simple organization can be insufficient for users and applications who are usually interested in only a small subset of objects in a bucket. Therefore, these huge buckets lack basic semantic properties and structure and are further challenging to manage from a technical perspective because object store file systems cannot cope with the corresponding large directory sizes.

This chapter introduces DelveFS, the first object store file system that aims to solve this challenge by offering the ability to compose a custom user-defined semantic file system, allowing multiple unique *views* onto the object store. Through flexible filters, users can specify each view's content, tailored to their unique interests or an application's requirements. With user-defined views, the file system can improve the usability of huge directories and allow new ways to implement complex workflows. DelveFS delivers similar file system throughput compared to the native object store interfaces or other file system connectors for both metadata and data operations. Moreover, due to how DelveFS handles its views, the file system can considerably increase the performance for common directory operations.

First, Section 5.1 will motivate this work and give an introduction to the topic. Section 5.2 will provide a background to describe the important core functionalities of modern object stores for DelveFS. In addition, related work will be discussed in detail. Next, Section 5.3 will describe DelveFS's goals, its central components, and main design considerations. In Section 5.4, DelveFS will be evaluated with regard to its event processing capabilities, event latencies, and metadata and data

performance. Finally, Section 5.5 will conclude this chapter and outlines our future work.

5.1 Motivation

Data-intensive applications drive a significant fraction of today's innovations in research and industry. Big data processing or machine learning, among many other underlying technologies, depend on the availability of huge data sets. Consequently, with more accurate techniques than ever before, data is being generated and collected at an increasingly rapid pace. For instance, in the bioinformatics community, the requirements to store sequenced genomes are growing faster than the storage density of hard drives [10, 11]. Already in 2014, the European Centre for Medium-Range Weather Forecasts (ECMWF) reached a storage capacity of over 100 petabytes with a 45% annual growth rate [88]. The particle physics community stores hundreds of petabytes of generated data by the Large Hadron Collider (LHC) at CERN [189], and the Square Kilometre Array (SKA) telescope will require more than 100 petabytes of short-term buffer storage for a 12-hour observation duration and exabytes of long-term storage [200].

Recently, companies also began to manage their data in *data lakes* in which all information is stored in a raw format [89]. Before data is accessed in data lakes, applications first need to locate the desired files among all files in the lake which adds to their complexity. Object storage, with its high scalability, low maintenance overhead, flat namespace hierarchy, and ease of use, has made this storage solution a popular option for such scale-out use cases. Nowadays, there exist many commercial and academic objects stores, e.g., Amazon S3 [161], BlobSeer [166], Ceph [268], Swift [7], OpenIO [176], or Ursa Minor [2]. Their use cases are manifold and include archiving, media and entertainment, scientific data, and the *internet of things* (IoT).

In addition, new internet-scale applications like Netflix or Dropbox have been designed around using object storage systems through their native object API [3, 61]. However, because these custom APIs are not adhering to established standards, such as a standard file system interface that is semantically similar to POSIX, data-driven applications may not be able to use these new interfaces offered by object stores. Adapting the I/O layer of these applications is typically time-consuming and may not be a feasible option. This is especially the case for many proprietary applications

or applications that have a decade-long development history, e.g., Nek5000 [69, @164].

As a result, several object stores added a corresponding file system interface or depend on third-party *file system connectors* [21, 130, @214, 267, 279]. Object store containers (or buckets) are then represented as file system directories and the containing objects as files. Therefore, applications can then use object stores through a file system interface without the need to modify the application's I/O layer. Nevertheless, since object stores operate in a flat namespace, these file system connectors only offer a basic representation of objects as files in a similar flat namespace, i.e., one directory per object store container. This is especially challenging when millions of objects are stored in single containers [@12] because of the lack of basic semantic properties and the typical file system structure.

Yet, even if a hierarchical directory structure of file systems was used, such hierarchies would often be insufficient when handling large amounts of data. For this reason, the usefulness of the hierarchical file system model has been questioned several times over the past three decades [82, 101, 220]. An alternative approach was proposed that involves a semantic model where data is dynamically searchable so that a user can create a data organization that fits how applications access their files. In this context, some object stores support search capabilities, e.g., basic filters for object name wildcards [@86] or time ranges [@56], but also more sophisticated metadata search capabilities [127, @216]. However, these are not available in file system connectors.

Another challenge that arises from the missing structure and data semantics is that the object store's flat namespace is mapped directly into the file system. This can result in severe challenges concerning metadata when a large amount of data is presented [269]. The reason is that POSIX-like file systems are built around a hierarchical structure, utilizing many mechanisms to optimize for this model, such as the *dentry cache* which caches directory entries in memory (see Section 2.1). This is the opposite in object stores which may gain scalability benefits by employing a flat namespace design [128].

DelveFS is an object store file system in user space that offers the ability to compose a custom semantic file system. It keeps consistency by processing *object store events* which describe changes in the object store, such as the creation of an object. Tailored to the user's requirements, DelveFS creates unique *views* onto the object store that can be flexibly filtered by several criteria, e.g., object names, sizes, access times, or object properties (similar to extended attributes in file systems). In addition, DelveFS allows a hierarchical structure on top of the user-defined view, emulated

within the object store's flat namespace to support applications that internally expect a hierarchical structure. Further, the file system operates in a distributed setting, supporting clients from many machines, each having potentially different but possibly overlapping views. DelveFS achieves these features without restricting access to the native object store API or the usage of other object store interfaces which can run alongside DelveFS.

This chapter will demonstrate how events can be used to map objects in a flat namespace to file system files and how events can be utilized to provide meaningful semantic subsets of data. Via object store events, DelveFS can update its views through its *event handler* at low latency and can operate without interfering with the object store by otherwise constantly contacting it for changes. Because DelveFS is using the object store as a black box, it offers an eventual consistency model with close-to-open semantics [94] between its different interfaces. Further, it achieves similar throughputs compared to the native object store interfaces and provides up to 110× increased performance for common directory operations (`ls -l`) compared with other object store file systems.

Overall, DelveFS aims for a number of use cases: 1. (virtually) reducing the number of objects in large containers, 2. filtering for specific types of objects to ease access to target data of, e.g., data lake applications, and 3. automatically classifying new objects to the context of user-defined views. For instance, the latter can be beneficial in short read alignment tools in computational biology where a large number of (short) sequences, classified by object metadata, are aligned to a reference genome. This allows users and applications to filter for desired metadata via DelveFS's views instead of scanning potentially huge containers for input objects with specific metadata and putting file system connectors and object stores under high loads.

5.2 Related work

Earlier in this thesis, Section 2.4 introduced the concept of object storage systems which has several advantages over traditional storage techniques, e.g., file systems. Most prominently, compared to file systems, object stores use a flat namespace and an object storage API that does not follow the standard POSIX semantics or established interfaces whatsoever. As a result, the object stores tend to scale well since their container complexity does not increase when objects are added [153]. It is therefore a popular choice for storing large amounts of unstructured data, e.g., photos or media files. In the context of DelveFS, this section will discuss

more advanced object storage topics, such as how object properties work, how file systems can use the object store via the standard file system interface, and how event notifications work.

5.2.1 Containers and objects

In general, object stores use containers (also called buckets) to accommodate objects [176, @214]. But, due to the flat namespace, there is no hierarchy of containers. Therefore, there is little overall structure (unlike the hierarchical structures in file systems), often resulting in large numbers of objects within a single container that can reach millions of objects [@12].

An object is a distinct unit of data with a globally unique identifier and some attached metadata. However, compared to a file in file systems, an object is generally immutable. In other words, to change parts of an object, the entire object needs to be downloaded, locally modified, and then uploaded, thereby replacing the old object in the process. In particular, this can become an issue when uploading large objects (hundreds of megabytes and more) as an upload failure requires repeating the entire upload process. S3, for example, offers the *multipart upload* feature which splits an object into multiple parts that can be individually uploaded. Hence, a failed upload of a single part does not fail the entire object upload. Only after all parts are uploaded, an object is formed that can be accessed.

5.2.2 Object properties

Object properties are custom user-defined metadata similar to extended file attributes (*xattr*) supported by many file systems [125, 129, 148]. They consist of key-value pairs that allow users to further classify an object beyond its name. Most object storage solutions support object properties but they differ in their usage.

For example, Amazon S3 allows two types of object properties: metadata and tags. Object metadata is tied to the immutability of the object and cannot be changed while tags are managed separately. OpenStack Swift only offers to upload a set of all properties instead of modifying each property individually. Thus, each property is also essentially immutable.

For DelveFS, flexible object property implementations, such as those offered by OpenIO or Google Cloud Storage, work best because they work similarly to extended attributes in file systems where individual key-value pairs can be updated. This is

because DelveFS allows the modification of object properties. It not only maps them to extended attributes but also uses them to build a hierarchical namespace on top of the object store's flat namespace.

5.2.3 Object storage file systems

Since object stores offer custom object-based APIs to manipulate objects, applications that are built around a file system POSIX interface are unable to use them. Hence, numerous file system *connectors* have been developed which allow an object store to be used as a UNIX file system [21, @84, 130, @177, @206, @214, @215, @236, 261, 267].

At their core, file system connectors present object store containers as directories and objects as files. However, they differ significantly in their features and how they operate. For example, S3QL supports multiple object stores and offers on-the-fly encryption [@215]. BlueSky employs a log-structured design to hide the latency of accessing the backend object store [261]. CephFS [267] uses its own *Reliable Autonomic Distributed Object Store* (RADOS) to offer a file system interface.

The above-listed file systems do not feature *dual-access*, that is, objects can only be accessed through the file system interface if consistency is required. In this context, the authors in [130] discuss the importance and efficiency of dual-accessing the object store through multiple interfaces. Object store file systems allowing dual-access include S3FS [@214], Goofys [@84], SVFS [@236], RioFS [@206], and OIO-FS [@177].

Nevertheless, these file systems do not consider containers with millions or even billions of objects which have become increasingly common in today's object stores [@12]. They only offer a basic representation of objects as files in a flat namespace, lacking any structure or semantics in addition to technical challenges due to metadata management [269].

However, even if a hierarchical structure would be used to retain some structure, it is still not enough when managing large data volumes. Consequentially, *semantic file systems* have been proposed several times in the past decades [82, 101, 220]. In a semantic file system, data is dynamically searchable to achieve a representation that fits the way it is accessed. For instance, Gifford et al. [82] allow suffixes to file system paths to filter files by ownership, among others. Some object stores also offer basic filtering mechanisms, such as for object name wildcards [@86] or time

ranges [@56], or more sophisticated search capabilities for object properties [@216]. Similar features are not available in current object store connectors.

DelveFS offers both dual-access and provides considerably higher usability than above-mentioned file systems by allowing semantic file system views based on a metadata filtering mechanism. Further, DelveFS overcomes existing consistency restrictions by coupling the object store's view and the user-defined file system view via the event interface of modern object stores. Because DelveFS relies on the event interface for fast client namespace updates at low latency, the file system achieves the features without actively disrupting the object store to retrieve the most recent namespace changes.

5.2.4 Event notifications

To ensure that both the object interface and the DelveFS views can become eventually consistent, DelveFS requires notifications whenever the namespace changes. For this, the file system processes an object store's *events* (or *notifications*) and uses object properties to form the file system views. More precisely, an event refers to a piece of data describing an operation in the object store, e.g., object write/upload operations. The event can then be (actively) retrieved by an application whenever a container or an object is updated. Such events are available in object stores, such as OpenIO, Amazon S3, or in Google Cloud, but they are still missing in Swift or RADOS. However, there is no standardized interface for events and, as a result, they differ significantly by how events are accessed, by the events' content, and by the general features the event interface offers, i.e., which operations produce events.

For DelveFS, events that describe changes of object properties or object sizes are most important as they are crucial to update a client's view on its defined namespace. This is because object metadata, e.g., properties and sizes, can be used when a client defines its view (see Section 5.3.4 later). Nevertheless, such object properties events are only supported by OpenIO and by Google Cloud Storage. In this chapter, DelveFS uses the OpenIO event interface as an example to leverage these kinds of events.

Agni [130] and YAS3FS [@275] use events to notify file system clients of changes in the object store. Such FUSE-based file systems, which are usually used in object storage connectors [21, @84, 130, @177, @206, @214, @215, @236], are, however, known to struggle with workloads that involve a large number of metadata operations [250] – also shown in Section 5.4.3 later. DelveFS addresses this shortcoming by processing events significantly further to allow user-defined semantic file system

views, allowing fine-grained searches and reducing the overhead of listing otherwise potentially huge directories.

5.3 Design

This section will present DelveFS's goals, architecture, and system components and will also discuss how object store events are processed in the file system and how it influences its consistency model.

5.3.1 Goals

DelveFS was designed to achieve the following objectives:

Functionality DelveFS should provide a file system interface to an existing object store through the Linux *virtual file system* (VFS). The file system should maintain its own persistent view of the object store via metadata information, but it should not store data outside the object store except for caching.

Dual-access The object store's interfaces should be able to be used at the same time as DelveFS. In addition, multiple DelveFS clients should be able to operate concurrently and access the object store in parallel. However, we assume that a single object is only modified by one client at a time, either DelveFS or through an object store interface.

Object property filtering Users should be able to define rules within DelveFS which allow them to view objects in the file system that are of particular interest. Such rules filter the object namespace for containers, objects, object sizes, object access times, or object properties. Filtering for container and object names should allow exact name searches and more complex expressions, such as filtering for all objects with a characteristic prefix or suffix.

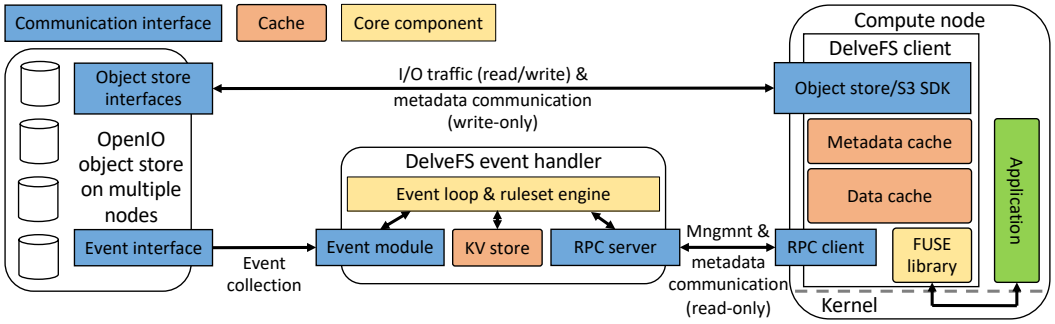


Figure 5.1.: DelveFS's architecture with its components and interaction with the object store.

Consistency model DelveFS is not intended to strictly follow the POSIX I/O semantics. Instead, DelveFS should provide an eventual consistency model with *close-to-open* semantics [94]. Close-to-open semantics guarantee that other processes can see the latest changes once a file is closed but not while it is still open and being modified. This allows the client to locally cache all changes until closing the file. These semantics are sufficient for our assumption that only a single client is modifying an object at a time.

A stronger consistency model, e.g., provided by several parallel file system implementations [219, 270], would also require a distributed locking mechanism (DLM) and hence full control of the object store. However, we use the object store as a black box so that DelveFS does not interfere with other simultaneously used object store interfaces. Further, aligning DelveFS more to NFS semantics [94] allows us to significantly reduce the file system's complexity and avoid otherwise intricate and communication-intensive distributed locking mechanisms [271]. Using such an eventual consistency model implies that changes applied to files within DelveFS may not be immediately visible in the object store and vice versa. This is due to caching mechanisms and event processing affecting both metadata and data operations.

5.3.2 Architecture

Figure 5.1 presents DelveFS's architecture, its main components, and its interaction with an object store. In this work, DelveFS uses the OpenIO object store [176] as its backend that is running on multiple nodes. OpenIO offers suitable interfaces to communicate metadata and bulk data between the object store and its clients. DelveFS includes the event handler and allows multiple FUSE file system clients.

At the core of the DelveFS ecosystem, the event handler is running as a single instance. Its three primary responsibilities are: 1. maintaining the state of the object store's namespace by processing its events, 2. filtering the namespace for user-defined rules, and 3. providing a unique client namespace's state to the requesting client. DelveFS allows multiple concurrently running clients which can be added to and removed from the DelveFS ecosystem at any time. To considerably reduce the load on the single event handler instance and increase overall file system performance, clients heavily utilize metadata and data caches.

Overall, the DelveFS architecture focuses on managing the object store's metadata (updated by events) to form a semantic file system *view* of objects of interest. *Rulesets* define these views, each containing a set of filters for relevant objects. All matching objects are then placed as files into a directory defined by the ruleset.

The life cycle of a file starts with its creation either via the object store interface or the file system interface. Once the object is created in the object store, a corresponding event is produced that the event handler captures, updating all affected file system views of DelveFS clients. Clients do not get notified by these updates directly, but have to poll the event handler explicitly when required, e.g., when a user lists a directory. Notice that views may also be affected by objects created or removed outside DelveFS since it supports dual-access. For instance, this can occur through the native object interface or other connected object store file systems, producing the appropriate events that are then processed by the event handler.

For maintaining consistency, DelveFS requires at least three event types: an object creation event, an object removal event, and an object update event when its size or properties are modified. Properties, i.e., key-value pairs similar to file system extended attributes, allow a fine-grained way of filtering the namespace.

The communication between DelveFS's components uses the *Mercury* RPC communication library [228]. DelveFS interfaces Mercury through the *Margo* library [42] which provides *Argobots*-aware wrappers [221] to Mercury's API (see Section 4.3.4 for details on Mercury and Margo).

5.3.3 Event handler

The event handler keeps the object store's state, i.e., its containers and objects, for the DelveFS client-registered rulesets. Therefore, the event handler maintains and provides the metadata for objects that are part of a view, including their directory structure. In principle, any user can launch the event handler, e.g., within the context

of an HPC compute job. However, modifying the configurations for the object store's event interface may require administrative support, depending on the used object store. Overall, the event handler consists of four main components (see Figure 5.1): 1. a key-value (KV) store to keep track of the object store's state, 2. an event loop which handles incoming *events*, describing a change within the object store, 3. a parallel rule-matching engine to process the content of each event, and 4. an RPC communication layer to exchange messages with DelveFS clients.

For storing object metadata, the event handler utilizes a local RocksDB KV store, offering a high-performance embedded database for key-value data [59]. When a client first connects to the event handler, the latter queries the object store to list all objects of containers that are part of the client's rulesets if not already available in RocksDB. Otherwise, a previous client already triggered the initial scan for a particular container. Hence, the KV store does not necessarily store the complete object store state but lazily populates it when required, which avoids scanning the entire object store when the event handler is started. This is important because, depending on the object store size, scanning the object store with possibly millions of objects is time-consuming and heavily disrupts the object store, impeding access times for other users.

Moreover, keeping a part of the object store's namespace allows DelveFS to discard irrelevant events to a ruleset. In other words, when the container was not scanned yet, it is not part of any ruleset, and therefore the event can be dropped. Note that the DelveFS event handler postpones applying the event to objects when it notices that the container is currently being scanned for the first time. For low-latency access, some of the object's metadata that matches a connected client's ruleset is kept in memory. Adapting to Google's object store would further allow listening only for specified containers, which makes discarding events unnecessary. In the case of OpenIO, events are exposed by one *beanstalkd* [@14] work queue per OpenIO node. Each event in the beanstalkd work queue describes a single change in the object store and is removed from the work queue after it has been processed.

The *event loop* pulls events from the beanstalkd work queues and prepares further processing. Essentially, it consists of multiple *progress threads*, accessing the beanstalkd queues, and *worker threads*, processing new events that the progress threads have accessed. First, the progress thread identifies the event type and passes its content to a worker thread created from a dedicated thread pool. In the *event matching* process, each worker checks if the event refers to a tracked container and if the described changes apply to any registered ruleset. In that case, the worker thread updates the KV store and ruleset (if applicable) accordingly. Notice that the

event handler does not notify a DelveFS client of an incoming event that affects its rulesets. Instead, the client lazily requests this information when required.

In the case of an event handler failure, no client data is lost because the event handler is a read-only medium. It does not immediately affect clients, with the exception of them being unable to update their ruleset metadata information. Hence, an interruption does not break the eventual consistency guarantees. Moreover, since the event handler does not store information beyond what is already available in the object store, there is no immediate need for advanced crash consistency procedures to conserve events while the event handler is shut down. Such a procedure is part of future work and could decrease the time when clients lazily ask for their data after the event handler is restarted.

5.3.4 File system client

An application uses the DelveFS client directly via the standard file system interface. It connects both to the DelveFS event handler and the object store. It is based on the FUSE library, providing an interface to export a user space file system implementation to the kernel. More precisely, DelveFS is using FUSE's low-level API which allows a flexible file system design that avoids various overheads of the high-level API [250]. The high-level API operates entirely on paths and does not allow manipulating internal file system data structures, such as inodes or dentries. This can reduce overall file system complexity and the required development time but induces additional costs for general metadata and data performance.

Like any other kernel file system, it can be mounted at a specified path. The client consists of four components: 1. A file system implementation based on FUSE's low-level API, 2. an in-memory caching module to store metadata as well as data for a short time frame, 3. an object store module to directly communicate with the object store, and 4. an RPC communication layer based on Mercury to exchange messages with the DelveFS event handler.

Rulesets When the client is started, the user passes a mounting path and a file that contains a *ruleset*. A ruleset is a collection of rules in the *YAML* syntax [@18] that the object store should be filtered for. Each ruleset can use rules of five categories with each rule narrowing down the results: container name (mandatory), object name, object size, object access time, and object properties. Then, all rulesets are sent to the event handler which processes them and responds with the resulting metadata of all matched objects. Figure 5.2 visualizes the overall startup process.

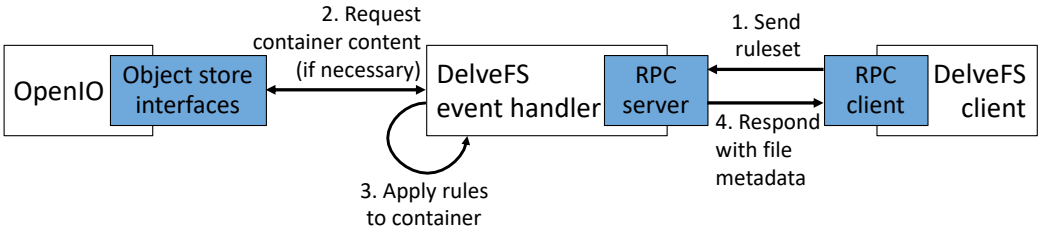


Figure 5.2.: DelveFS client startup procedure.

```
1  ---
2  Gecko:
3    account: "delvefs"
4    container: "gecko_genome"
5    object: "*.fasta"
6    properties:
7      color: "black"
8      color: "green"
9  Leopard:
10   [...]
11 ---
```

Figure 5.3.: An example of the `Gecko` ruleset in YAML syntax and its defined rules to filter for objects in the `gecko_genome` container.

An example YAML ruleset is shown in Figure 5.3 with the name `Gecko` (defined in line 2), showcasing some of the rule categories. All objects that match all rules of the set are placed into a single directory named after the ruleset. In this example, the ruleset matches all objects in the `gecko_genome` container with the properties `black` color **or** `green` color. In addition to exact matches, DelveFS supports partial matches via the `*` character. Here, the `object` keyword represents a suffix-wildcard and matches all objects whose name ends with `"*.fasta"`. The fasta extension is used for files storing nucleotide sequences or protein sequences in the *FASTA* format [185].

Metadata management The client queries the event handler via read-only RPC messages to get the latest update of a directory's state, that is, the ruleset's metadata. The directory state is represented as inodes and directory entries which are kept in in-memory caches. The directory is then considered *recent* for a configurable amount of time, during which the client does not ask the event handler for updates on that directory. Hence, changes in the object store may not be immediately visible to the client. Note that a file can disappear from a client's ruleset directory under specific conditions. For instance, the corresponding object could have been removed from

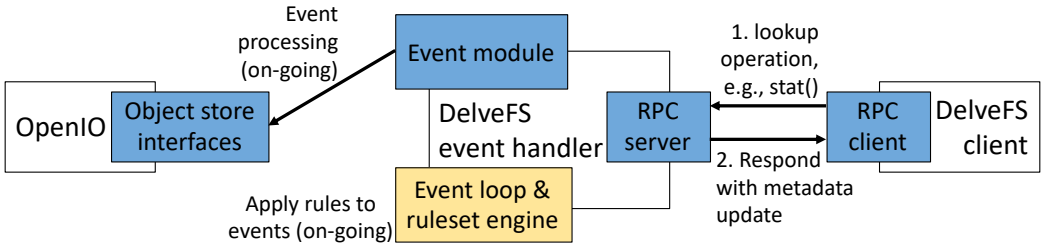


Figure 5.4.: DelveFS client lookup procedure.

the object store, or an object's properties have changed and no longer match all ruleset criteria.

This process is similar to file lookup operations issued, e.g., via a `stat` system call. If the directory was recently listed, a file's metadata, i.e., the inode, is cached and therefore served locally. Otherwise, the event handler is queried for a potential update of the corresponding object's metadata that was modified by an event. Figure 5.4 provides an example of a stat operation that queries the event handler, including the ongoing event processing that updates the ruleset information stored at the event handler.

File system operations that change the namespace, i.e., creating and removing files or modifying their extended attributes, communicate directly with the object store's native interface. This avoids potential consistency issues between DelveFS and the object store. By and large, the object store is always treated as the primary source of truth because it is updated before DelveFS. For example, when a client creates a file, it also creates the equivalent object in the object store. During this process, the event handler is unaware of this operation until it receives a respective create event. That is to say, the event handler relies on the object store's event interface to update its database, even if a DelveFS client changes the object store's namespace. Notice that the event handler cannot determine whether a given event was modified by a DelveFS client or another source.

Data management Like metadata operations, I/O operations do not notify other DelveFS nodes. Instead, objects are directly accessed via the object store's native API. However, for I/O operations, the file system interface and object-based interface are not immediately compatible due to how data is written and read. A (local) file system assumes that data is available in mutable blocks, changeable at any offset. On the contrary, object storage uses the concept of immutable objects which cannot be modified in place after their creation [7, 161].

As a result, modifications of an object require downloading the object, modifying it locally, and then uploading a new object with the same name that replaces the old object. It is worth mentioning that object replacement is not available in all object stores and can depend on container configurations, in which case the object has to be removed first. FUSE-based file systems, on the other hand, split file system operations for reads and writes with a maximum buffer size of 1 MiB and 128 KiB, respectively. Therefore, a read operation on a 10 MiB file involves 10 FUSE-internal read function invocations. In the case of a write operation, downloading the full object for each operation and replacing at most 128 KiB of data before uploading it back to the object store is, thus, inefficient and slow. Moreover, it causes unnecessary additional loads over the network and disrupts the object store for other accessing users.

For this issue, DelveFS implements a write-back data cache. It supports the OpenIO API and the S3 API to translate (remote) objects to files (henceforward called *object files*). The cached object files are then stored at a user-defined local file system path outside of the FUSE environment. This allows DelveFS's write operations to operate locally until a `close` or `fsync` command is called, indicating that no further write operations are issued or the object file should be synchronized with the object store, respectively. The file system then uploads the object file to the object store. DelveFS's semantics assume that data is not changed outside of this client until the data is uploaded back to the object store and, therefore, only allow a single writer to each file. This is not a special limitation of using DelveFS but applies to any object store client, whether the native API is used or another object store file system because of the object's immutability that does not allow parallel access.

For read operations, DelveFS downloads the remote object during the first encountered read operation, placing the resulting object file into its cache. All subsequent read operations can then work locally. A user-defined configuration decides how long an object is considered *recent* in the DelveFS caches before it is downloaded again. To avoid downloading an object repeatedly after the caching time expires, DelveFS first compares the checksum value of the object file and the remote object. This is a small metadata operation between the client and the object store.

Virtual directories Rulesets allow users to create a semantic file system view, but due to the object store's flat namespace, ruleset directories would not allow a further directory structure that applications and users might require. Therefore, DelveFS emulates a directory hierarchy within a ruleset, e.g., to represent the source code of a complex application with many subdirectories. These directories are entirely virtual

since object stores do not support recursive container structures. In DelveFS, the object's position in the (virtual) directory hierarchy is stored as an object property that is resolved only on the client side. As a result, the directory hierarchy of an object is fixed for all clients.

Alternatively, the file system could use dedicated directory objects, similar to directory blocks in file systems, only containing information on object names belonging to the directory. We decided against this approach because it would require a central entity (e.g., a global lock manager) to control access to such an object so that file system clients do not issue conflicting write requests, effectively overwriting the directory information of each other. Moreover, directory objects are of no value to other object store users and would be considered junk objects.

Nevertheless, in both techniques, virtual directories are tied to the object and fixed for all clients, although their content can differ based on which rules are applied to the ruleset directory. If necessary, showing virtual directories can be disabled for a ruleset.

5.4 Evaluation

This section will present the evaluation of DelveFS concerning the event processing performance, the file system's metadata handling and (ruleset) directory capabilities, as well as its data accesses. Finally, we will present an example of how rulesets can benefit an application's workflow.

5.4.1 Experimental setup

Our experiments used the Intel Skylake partition of the MOGON II cluster at Johannes Gutenberg University Mainz. MOGON II consists of 1,876 nodes in total, from which 1,046 nodes contain two Xeon Gold 6130 Intel Skylake processors with main memory capacities ranging from 64 GiB up to 1,536 GiB. The cluster nodes are connected via a 100 Gbit/s Intel Omni-Path network to establish a fat tree network, and it offers a Lustre parallel file system as a storage backend.

The DelveFS event handler uses local-node SATA SSDs as backend storage for RocksDB. The DelveFS clients and the event handler use TCP/IP over Omni-Path as the network protocol. In all DelveFS experiments, the DelveFS clients and the applications under test are pinned to separate processor sockets to minimize possible

interference. The FUSE client library runs in multi-threaded mode and uses up to ten threads.

An OpenIO object store is used (running on ten compute nodes) to demonstrate DelveFS's functionality and performance. Nine nodes serve as storage nodes for data and metadata, and one node serves as a management node and gateway for metadata requests. For data and metadata, OpenIO's storage nodes use data center SATA SSDs, offering approximately 1.8 TB of storage in total. The object store was used in isolation without any other user accessing it. Internally, OpenIO communicates using TCP/IP over Omni-Path. External nodes that use the object store use TCP/IP over Omni-Path as well to download and upload objects. In all S3FS experiments, the S3 interface of OpenIO was used. Finally, data replication was disabled in OpenIO and versioning was limited to two versions per object to reduce general overheads.

5.4.2 Event processing

DelveFS clients rely on the performance capabilities of the object store's event interface and the event handler's ability to process events quickly. For OpenIO, events are exposed by the beanstalkd work queue (see Section 5.3.3) with each of the nine OpenIO nodes running a separate beastalkd instance. Henceforward, we use the terms beanstalkd and work queue interchangeably.

Because the event handler stores metadata outside the object store, it avoids that read-heavy metadata requests, e.g., `lookup()` or `readdir()`, are issued to the object store directly. Moreover, the event handler manages rulesets that filter the namespace in a user-defined way. Therefore, depending on a central entity in this use case requires that the event handler can process events at a reasonable throughput. In addition, each event must be available for the clients at a low latency if it modifies one of their defined rulesets.

We designed three scenarios to evaluate the various components of the event handler and to measure its event processing throughput and event latencies. The first scenario (`RAW`) covers the event handler's raw throughput capabilities when events are not matching any client ruleset. This scenario is the best case with the least amount of functionality triggered since events are merely removed from the work queues and then discarded. The second scenario (`MATCH`) measures performance degradation by continuously matching events to rulesets. `MATCH` only accesses a single object that is matching to 100 different rulesets that are maintained in memory.

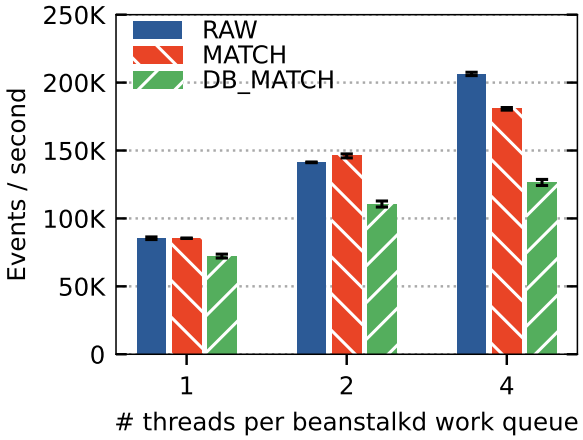


Figure 5.5.: DelveFS's event throughput in three scenarios.

Operating on a single object allows RocksDB to (almost) completely work in memory, minimizing potential performance degradation due to the KV store accessing the underlying SSD. The latter case is then shown in the third scenario (`DB_MATCH`), stressing the KV store. `DB_MATCH` events do not only match each event to 100 rulesets but also refer to enough objects in total (10,000 in this case) so that RocksDB must continuously read and write *static sorted table files* (SST files) from and to the underlying storage. SST files contain parts of the KV store's entries to persist the KV store's state to disk. Hence, the `DB_MATCH` workload targets all event handler components. On a separate node, a DelveFS client sets up the required rulesets for the event handler, but it remains idle during the experiments to not interfere with the event handling process.

Event throughput The event throughput for all three scenarios was measured by artificially inserting two million events into each of the nine OpenIO nodes' work queues. The insertion rate was set to ∼390,000 events per second in total (resulting in over a 1-minute runtime) because it exceeds the event handler's processing capabilities in the `RAW` scenario. This allows for examining the event handler's sustained event throughput.

We did not trigger events using OpenIO directly because it emitted fewer than 1,000 events per second due to its limited installation size. Instead, by using similar artificial events, we were able to explore the event handler's processing capabilities in an environment that generates a considerably higher number of events per second.

Figure 5.5 shows the event handler's throughput for the three discussed scenarios for up to four threads per work queue (x-axis). The y-axis depicts the total event

throughput for each case. For the RAW scenario, the event handler processed up to 200K events per second by using four threads per work queue.

For the MATCH scenario, the throughput decreased to 180K events per second. However, even for a single object, the performance degradation is mainly attributed to the repeated KV store accesses because each event causes a KV store put operation to update the object's metadata. Increasing the number of threads for each work queue did not yield further improvements in the RAW and MATCH scenarios.

For the DB_MATCH scenario, the event throughput reached up to 126K events per second for four work queue threads. Like in the MATCH scenario, the performance degradation is caused by the continuous access and modification of all objects' metadata in RocksDB. As the KV store must use more storage to accommodate the metadata of all objects, it created several SST files on the backend SSD storage that is constantly written and read. Overall, this slightly delays access to each KV store entry, resulting in a ∼30% reduced throughput compared to MATCH for four work queue threads.

Event latency The time from when an event enters the work queue until its corresponding changes are available to the client is defined as the *event latency*. As a result, a client can access the update (if they actively ask for it) as soon as the event handler has processed it.

The event latency was evaluated by running the MATCH and DB_MATCH workloads. RAW was omitted since it only processes events that are discarded and thus irrelevant for clients. For each of the two workloads, eight million events were inserted into all work queues in total. The event insertion rate was set to ∼50,000 events per second so that the maximum latency of MATCH remained below 30 milliseconds. We also ensured that this insertion rate does not lead to a saturation in the DB_MATCH scenario since the event latencies of the more complex DB_MATCH scenario significantly differ from MATCH. Finally, the event handler and workload-inducing benchmarks were run on the same node to compute accurate latencies. To minimize interference, the event handler and benchmark were run on separate processor sockets.

Figure 5.6 visualizes the latency distribution for the MATCH scenario for eight million events. The x-axis presents a non-linear range of latencies to capture all processed events' latencies, ranging from 100 microseconds to 30 milliseconds. The y-axis depicts the percentage of events corresponding to a latency bucket (on a logarithmic scale). For example, the latencies of all events in the 200 μs bucket range from 101 to 200 μs. For this experiment, over 99.8% of the events were processed in less than

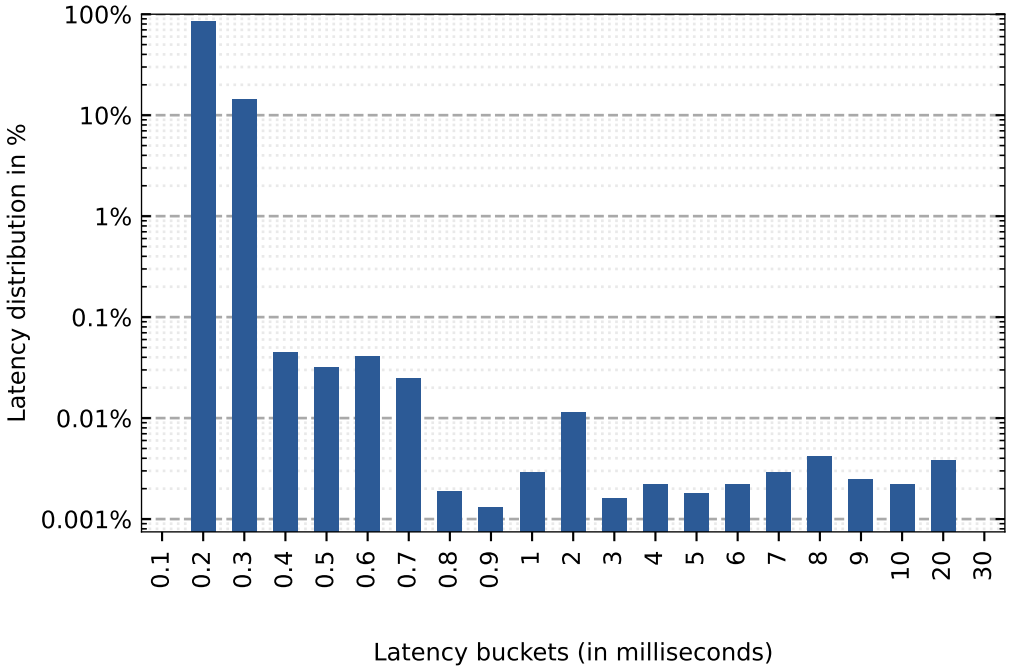


Figure 5.6.: DelveFS's latency distribution for the event handler's event processing in the `MATCH` scenario.

400 μs. The event with the highest latency required 30 ms. These low latencies were achieved because the rulesets and nearly all updates to RocksDB were kept in memory.

Figure 5.7 presents the latency distribution for the `DB_MATCH` scenario for 8 million events in which all events refer to a total of 10,000 objects in addition to each event being matched to 100 rulesets. Compared to the `MATCH` scenario's latencies, this workload showed a considerably higher range of latencies attributed to congestion at the KV store, taking up to several seconds. Nevertheless, more than 92% of events were processed in less than 300 μs with an overall event throughput well above the load our experimental setup can emit.

5.4.3 Metadata performance

This section will cover DelveFS's metadata performance for file metadata and directory operations.

Microbenchmarks The metadata performance was evaluated by running a parallel microbenchmark that *creates*, *removes*, and *stats* 1,000 zero-byte files per process

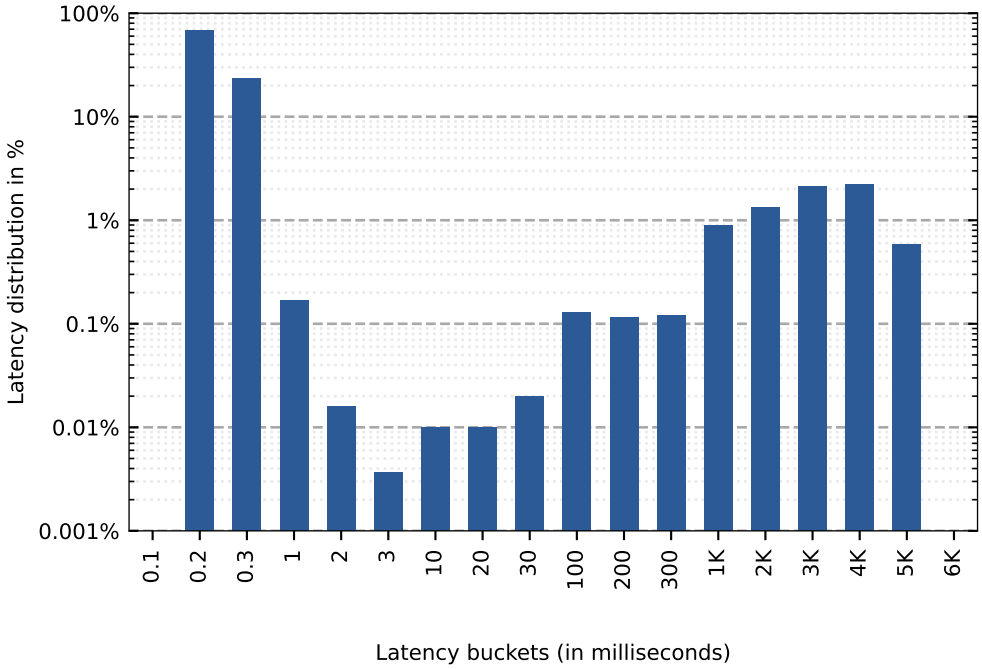
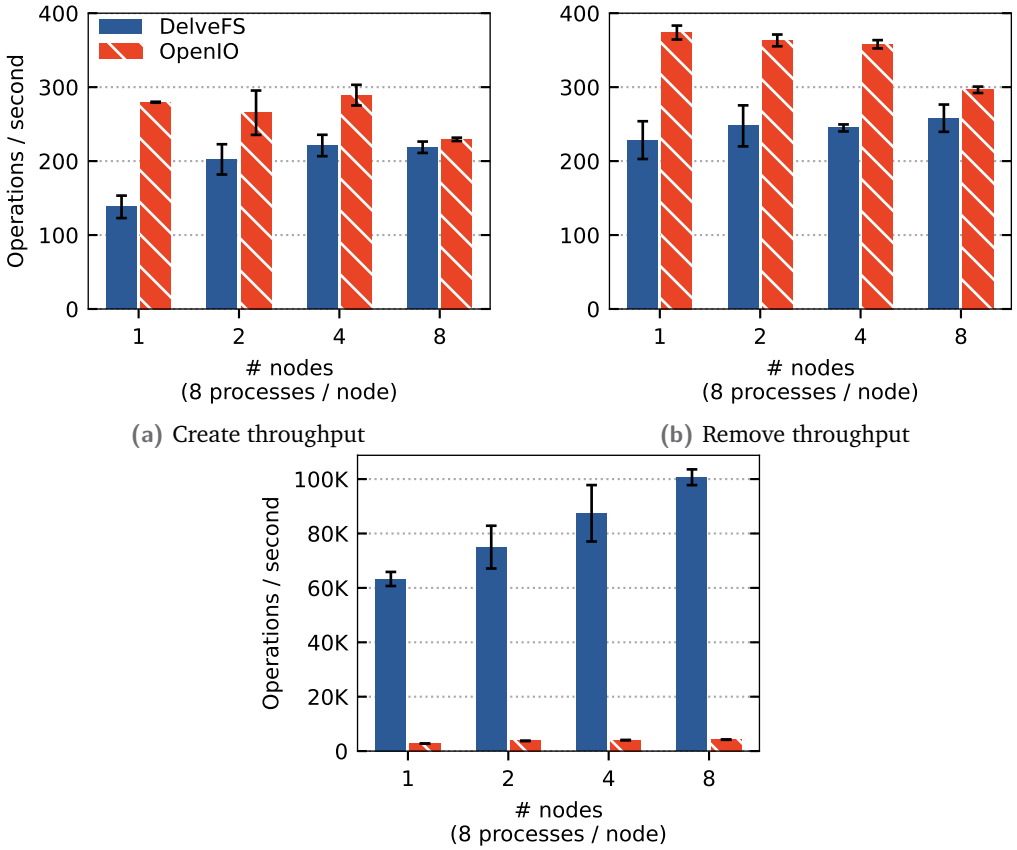


Figure 5.7.: DelveFS's latency distribution for the event handler's event processing in the `DB_MATCH` scenario.

in a single container for 8 processes per node. The workload was also run with the equivalent operations when using OpenIO's SDK directly (i.e., the native object store API) to measure the impact of DelveFS's file system layer.

Figure 5.8 presents the results, each data point showing the mean of five iterations and the corresponding relative standard deviation. OpenIO's native object API via its SDK achieved a throughput of up to 290 creates per second, 370 removes per second, and 4,200 stats per second. OpenIO reaches its maximum throughput for create and removal operations already at a single client node, indicating metadata congestion at the object store. DelveFS did not achieve the same throughput for a single client node since each create and remove operation causes multiple consecutive internal FUSE function invocations that cause several overheads in FUSE's internal components [250]. Overall, the observed overhead was significantly more prevalent in the metadata experiments compared to the data experiments that will be discussed in a later section. Nevertheless, these effects become less severe when multiple nodes are used, showing similar throughputs as the native object store API.

On the other hand, the `stat` throughput of DelveFS was over 23× higher compared to OpenIO's SDK. This is because the DelveFS client does not use the object store API for requesting an object's metadata but instead retrieves it from the event handler. The event handler can then serve the results from memory without additional



(a) Create throughput

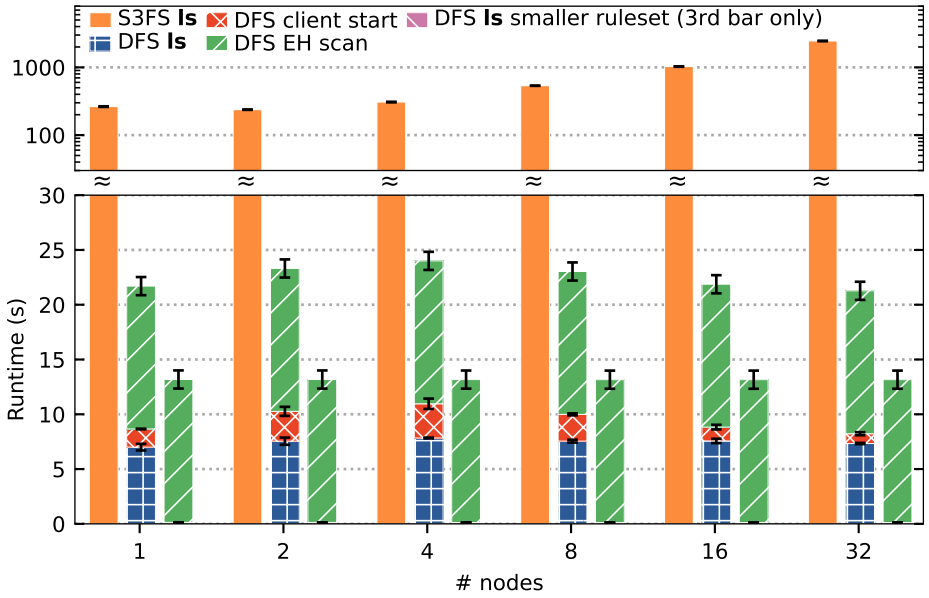
(b) Remove throughput

(c) Stat throughput

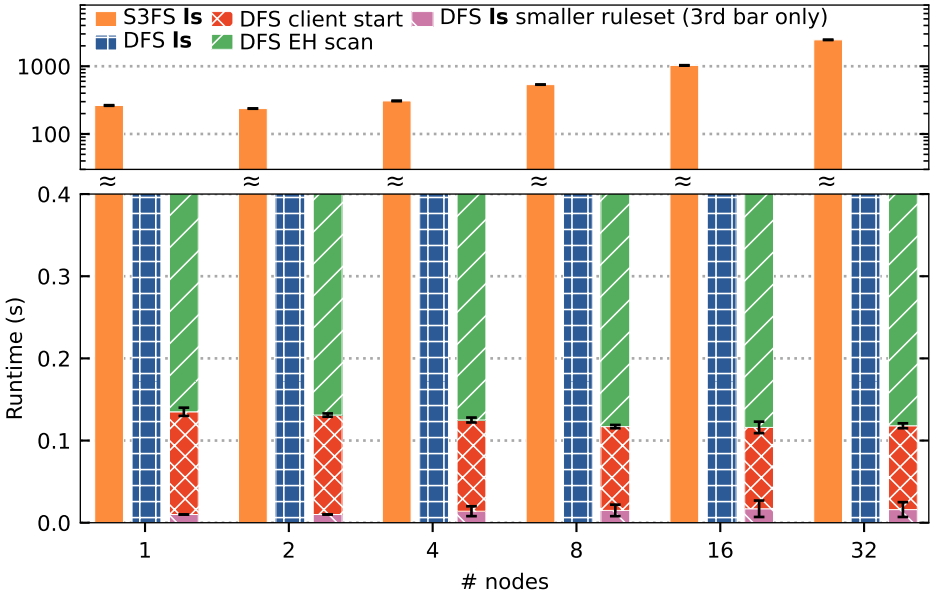
Figure 5.8.: DelveFS's metadata throughput when compared to the direct usage of OpenIO's SDK.

lookups to RocksDB. Bigger systems may show a lower stat throughput when not all metadata can be served from memory.

Directory operations Several experiments measured the time required to list large containers with the typically used `ls -l` command for both DelveFS and S3FS. The container size was restricted to 100K objects since the `ls -l` command did not complete when using larger container sizes (e.g., 150K objects) for S3FS. As is typical for object store file systems, S3FS was mounted to a single container whereas DelveFS clients used two rulesets within the same mount point: 1. A ruleset containing all 100K objects of the container, and 2. a ruleset that filters the container for a rule-defined object name prefix, reducing the number of files within the directory to 100. This configuration compares DelveFS's directory listing capabilities with the established S3FS object store connector, highlighting the performance benefits of rulesets if they operate on a large container. Figures 5.9a and 5.10 present the results with regards to the time required in seconds (y-axis) to run the



(a) Cold cache



(b) Cold cache (zoomed-in)

Figure 5.9.: Cold cache: Listing a directory containing 100K objects in DelveFS and S3FS compared to using a ruleset, reducing its size to 100 objects.

`ls -l` command for an increasing number of nodes (x-axis) for a cold and hot cache, respectively. Each node ran a single process, and each data point presents the mean of three iterations.

Figure 5.9a shows the cold cache case whereas we flushed the operating system caches and restarted the file system and the DelveFS event handler before each iteration. It is important to understand that DelveFS's cold cache case for `ls -l` must also include the time for two additional operations since the client relies on the initial ruleset processing when DelveFS is started. Therefore, it must include the time for the event handler's initial container scanning (`DFS EH scan`) and the time required to process the ruleset of the client (`DFS client start`). For a single node, S3FS needed ∼360 seconds to list the container contents compared to DelveFS's ∼22 seconds in total. When the number of nodes is increased, the time for S3FS increased to ∼2450 seconds (over 40 minutes) at 32 nodes, while DelveFS's performance remained constant. Further node numbers are omitted as DelveFS's performance remained stable for up to 128 tested client nodes.

The reason for S3FS's 110× longer runtime at 32 nodes is that clients must share a connection to the object store. This bottleneck already becomes visible at eight nodes and worsens when more nodes are used. DelveFS's event handler, on the other hand, must read the content of the container only once for all DelveFS clients using `DFS EH scan`. Afterward, it can use its event processing without directly querying the object store. The startup for the DelveFS clients (`DFS client start`) for a 100K file ruleset required less than 4 seconds to complete in all cases. This time includes communication with the event handler and processing the given ruleset, which creates the directory metadata for the client for all matching objects. In fact, the average client startup decreased when more nodes were used since the event handler can reuse similar existing rule set definitions of other clients while they were starting up simultaneously (all clients used the same ruleset definition). Accounting for `DFS EH scan` and `DFS client start` leaves the runtime of `ls -l`, which was less than eight seconds in all cases. The time is mainly attributed to the repeated invocation of FUSE's internal `readdir()` function that can only serve ∼100 entries at a time (depending on the file name length).

When a smaller ruleset was used that contained only 100 objects (`DFS ls smaller ruleset`), the client startup took less than 150 milliseconds, with `ls -l` requiring less than 20 milliseconds. Because of the y-axis's range of several seconds, the startup time and `ls -l` are not visible in Figure 5.9a. Figure 5.9b provides a zoomed-in view where the y-axis range is capped at 0.4 seconds, revealing the runtime for the client startup and for the `ls -l` command. Therefore, a ruleset that decreases the

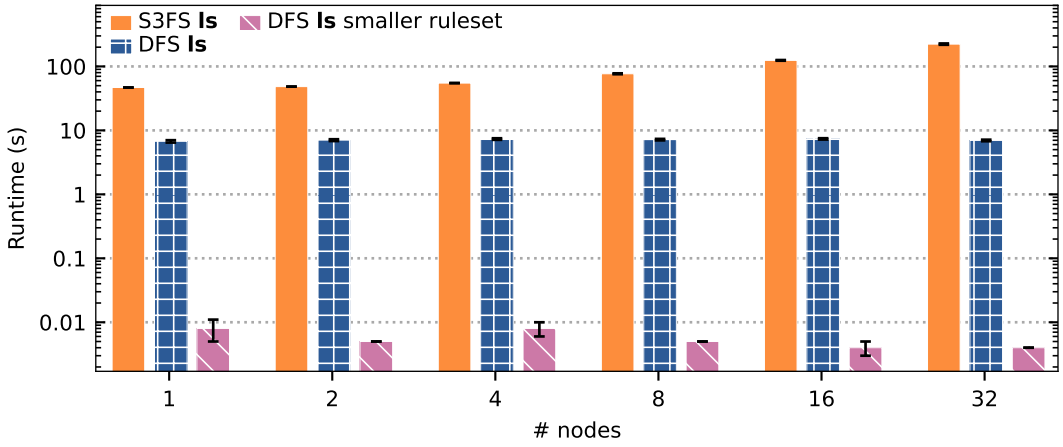


Figure 5.10.: Hot cache: Listing a 100K object directory in DelveFS and S3FS compared to using a ruleset, reducing its size to 100 objects.

number of objects in a large container can both reduce the working set and also significantly decrease the necessary time for directory operations.

Figure 5.10 presents the same cases on a logarithmic scale for a hot cache, that is, the directory has recently been listed. This allows each DelveFS client to operate in its local caches. Thus, the time required to read the container content and to start the client is not included in the figure, leaving less than 8 seconds to list the 100K file ruleset directory (mainly due to FUSE's overhead) and less than 10 milliseconds to list the 100 file ruleset directory. For a single node, S3FS required ∼46 seconds to list the directory but again showed a bottleneck at around 8 nodes, indicating that the file system is not operating entirely locally.

5.4.4 Data performance

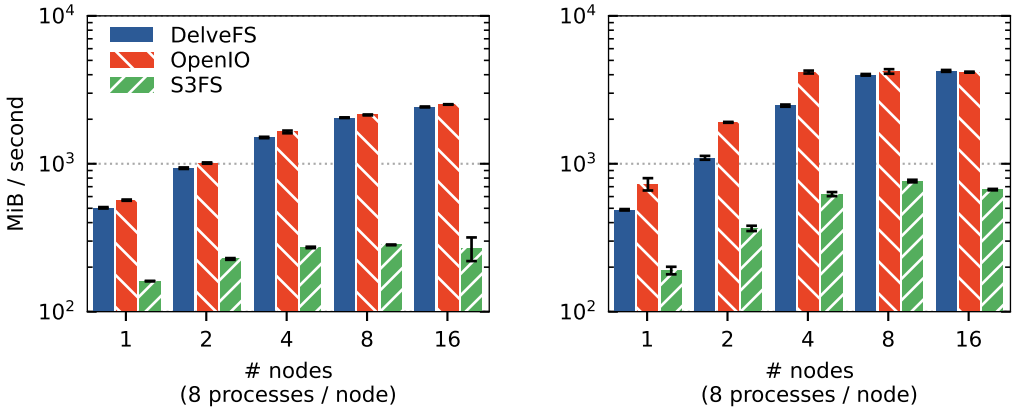
The following paragraphs will investigate DelveFS's I/O performance and present an example of how rulesets can be used in a complex workflow by using the Bowtie [122] application to align short sequences to a human reference sequence.

Microbenchmarks To evaluate DelveFS's I/O performance, the file system was compared directly to the OpenIO SDK, i.e., the native object-based interface presenting the upper bound, and S3FS. Each experiment iteration accessed its own file/object and performed 4 GiB (worth) of writes and reads per process in succession for up to 16 nodes. For DelveFS, each iteration is broken down into four *core I/O operations*: 1. Writing test data to local storage, 2. uploading the data from local storage to the

object store, 3. downloading the data from the object store to local storage, and 4. reading the test data from local storage. Therefore, in the I/O benchmarks, the former two I/O operations form the overall *write* operation, and the latter two I/O operations represent the overall *read* operation.

First of all, the experiments were designed to investigate potential FUSE-induced overheads and to compare DelveFS directly to the equivalent object store interactions. Thus, when measuring the OpenIO SDK I/O throughput, the above core I/O operations were run in succession when using the OpenIO SDK. As a result, when comparing the OpenIO SDK to DelveFS, the equivalent local and remote I/O operations are performed, leaving only the performance degradation of the file system and FUSE. Further, for the S3FS comparisons, we disabled its *multipart* feature. Multipart allows users to split the object upload process into multiple parts. If one part upload were to fail, the user can attempt to reupload the failed part without the need to re-upload all other parts. When all parts are uploaded, the object store forms the final object out of all parts on the server side. This is particularly useful for large objects because a failure to upload the entire object without multipart would require the user to upload the entire object again. Nevertheless, in the S3FS experiments, using multipart did not come without cost and reduced the overall throughput by up to 10%. Since the OpenIO SDK does not offer similar functionality, multipart was disabled on S3FS. Moreover, because the write and read operations include writing and reading to and from local storage and because the network bandwidth is considerably higher than the node-local SSDs bandwidth, the DelveFS client's and S3FS's data caches were placed on a RAM disk. However, to prevent read operations from accessing the RAM disk, the file systems are restarted and their caches are flushed after each write experiment. Finally, to avoid measuring the buffer cache instead of the sustained node-local SSD's performance in OpenIO, the amount of available memory was limited on the object store nodes by hogging about 80% of their main memory.

Figure 5.11 presents the combined I/O throughput on a logarithmic scale on the y-axis for up to 16 client nodes. Each data point represents the mean of five iterations with the corresponding relative standard deviation. For 16 nodes in the OpenIO SDK case, the upload (write) and download (read) throughput reached 2,500 and 4,100 MiB per second, respectively. For smaller node numbers, the throughput is bound by the network, while OpenIO achieved to saturate the expected SSD I/O throughput starting at 4 clients for reads and 16 clients for writes. DelveFS was slightly slower for writes, but it could saturate the object backend for reads starting at 16 client nodes. We omit further node numbers as the data throughputs did not increase for up to 128 tested client nodes.



(a) Write throughput

(b) Read throughput

Figure 5.11.: DelveFS's sequential write and read throughput when compared to the direct usage of OpenIO's SDK and S3FS.

S3FS achieved at most ∼282 MiB per second write throughput and ∼764 MiB per second read throughput for 8 nodes. However, note that S3FS was unable to complete the workload for 16 nodes, causing I/O errors upon closing the files. This also resulted in an incomplete upload to the object store. We decreased the number of processes to 4 per node for S3FS's 16 node experiment to allow for a complete run.

Note that the above-described experiments only directly compare the raw performance capabilities of each interface, but it does not consider using the data caches of DelveFS. This is because we are interested in DelveFS's and S3FS's I/O interface efficiency compared to using the native object store API directly, which does not implement a client cache. Nevertheless, if the data cache was placed on the node-local SSD and the experiments were run repeatably with caching enabled, the corresponding object was accessed locally and therefore showed the expected SSD I/O throughput of multiple clients, as shown in Section 4.5.4 earlier in this thesis. This specific configuration of *stacking* file systems, i.e., running a FUSE file system on top of another local file system, was already evaluated by [250].

In summary, DelveFS can achieve close to similar bandwidths as when the native object interface was used, albeit requiring a larger number of nodes for the read case to catch up to the object store's I/O capabilities. This shows that FUSE adds some amount of performance degradation, but not to the extent seen for the metadata experiments. The reason for this behavior is that FUSE's I/O path requires significantly fewer internal function invocations that communicate between the user space component and FUSE's kernel component compared to FUSE's metadata operation code paths. These observations are consistent with the results shown by

Vangoor et al. [250], who investigated the performance overhead of various FUSE operations.

Short read alignment As an example of how DelveFS rulesets could be used in a scientific workflow and to demonstrate its usage, we ran DelveFS in a typical workload from the biology field with the popular *Bowtie* application [122]. In this workload, many *short reads*, i.e., short DNA sequences of fewer than 200 nucleobases produced by *Next Generation Sequencing* (NGS) devices, were aligned to a reference DNA sequence. Bowtie was run to align a total of ∼100 GiB short reads (consisting of 16 *fastq* input files[1]) to a human reference sequence. In the alignment process, around 11 GiB of output data was generated.

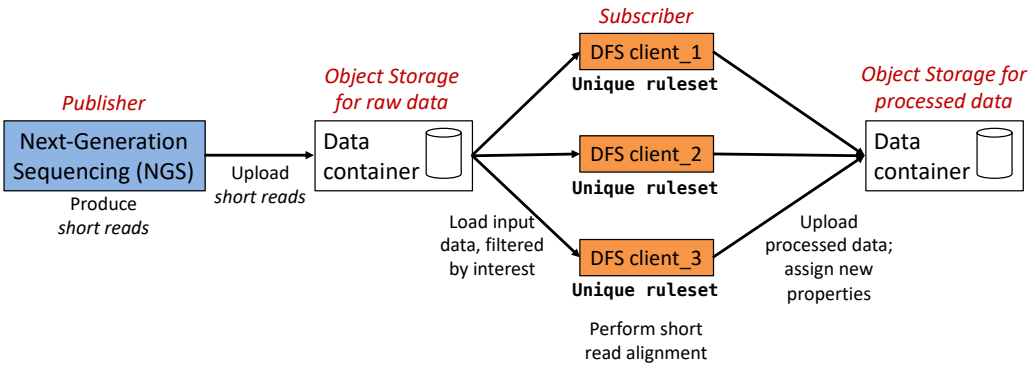


Figure 5.12.: DelveFS in a publish-subscribe model workflow.

In this case, the rules describe a *publish-subscribe* model in which the publisher (the sequencer) addressed its subscribers (the alignment processes) based on their interests that are defined as object properties in rulesets (see Figure 5.12 for a three-node example). DelveFS clients were run on up to eight nodes, each client with its own unique ruleset. The publisher then put the 16 fastq input objects into a container via the native object interface. For each experiment, the objects' properties were defined to address the rulesets of a defined group of DelveFS clients to distribute the overall workload.

On each node, a subscriber process checked the ruleset directory regularly for new input data. If input data was available, the subscriber launched the Bowtie application for each file in sequence. Figure 5.13 presents the Bowtie runtimes for these four experiments, presenting the impact of the implicit load balancing. Finally, the alignment results were written to a separate output ruleset directory.

[1]The short read sequences were downloaded from the National Center for Biotechnology Information's (NCBI) Sequence Read Archive (SRA) [@163]

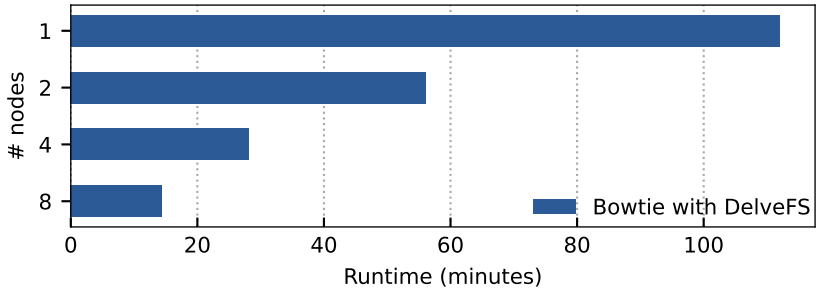


Figure 5.13.: Bowtie runtimes with DelveFS.

Each output ruleset directory contained a number of rules, e.g., a specific container name and some defined properties. Because of this, each Bowtie output file that was placed in the said directory was automatically assigned its constraints. In other words, the corresponding objects that were uploaded to OpenIO were automatically assigned the matching object properties.

This example shows that rulesets can improve overall performance, e.g., shown for the `DFS ls smaller ruleset` above, and also simplify the development of complex workflows and load balancing schemes.

5.5 Summary

As the amount of experimental data increases over time, scalable data storage solutions will become even more critical assets in the future. Object storage systems have successfully established themselves in numerous fields, but the lack of interface standards makes them challenging to use for application developers. Although file system connectors exist for this use case, they only provide a basic representation of the object store and often prohibit the simultaneous access of multiple object store interfaces.

This chapter has presented DelveFS[2], a novel approach to how object store events can form unique semantic file system views for each user. DelveFS has shown to process up to 200,000 events per second, distributing modifications of the object store's namespace to client views at low latency. The file system provides dual-access, and we have shown that it can provide similar metadata and data throughputs compared to the native object store interface. With an ever-increasing amount of experimental data stored in object storage containers, users can leverage views to considerably reduce the size of containers via rulesets. DelveFS's evaluation has

[2] `https://gitlab.rlp.net/DelveFS/delvefs`

demonstrated that these rulesets can significantly improve directory operations from a usability and performance perspective, and rulesets have shown how they could simplify complex workflows.

For future work, we plan to extend DelveFS with several features. First, we intend to develop more sophisticated techniques to persistently store rulesets on the event handler's backend storage to allow failure recovery. Further, the rulesets should be grouped based on their similarities to pack metadata between rulesets more efficiently. Second, we plan to explore using a distributed event handler to increase DelveFS's overall event processing capabilities, removing its single point of failure when updating file system clients. Finally, we aim to support additional object store interfaces (e.g., Google Cloud Storage) and investigate other consistency concepts, e.g., allowing DelveFS to periodically upload parts of an object while the writing process is running (if supported by the underlying object store).

6 Conclusion

The first exascale HPC systems have become operational, and their storage systems employ tens of thousands of HDDs, providing hundreds of petabytes of capacity. In the future, the size of HPC systems is set to continuously increase with higher I/O demands on the corresponding backend parallel file systems. These higher I/O demands are not solely because of higher data volumes due to a higher resolution of modern instruments that leads to more raw experimental data, for instance. Coupled with challenging I/O patterns, e.g., random I/O in the case of machine learning, and with more compute nodes, more applications can therefore run in parallel and increase the overall load on these file systems. Already today, some users of HPC applications, e.g., the NEK5000 [69] application for computational fluid dynamics, need to discard some of the simulation output due to I/O bottlenecks. Supercomputers have expanded the I/O hierarchy with additional storage layers to reduce such bottlenecks, offering fast flash-based storage devices at each compute node. However, distributed HPC applications usually require a single namespace and therefore node-local storage often remains unused.

On another spectrum, object storage has established itself in numerous fields and, because of its scalability and low maintenance, can also serve as data lakes for HPC systems running applications that process this data. Nevertheless, object stores offer a non-standardized interface, and applications that expect a POSIX-like file system interface are unable to use them. While such file system connectors are available, they suffer various drawbacks ranging from performance degradations to usability limitations due to rising data volumes in flat namespaces.

This dissertation has discussed three main subjects, covering challenges in developing and maintaining the GPFS parallel file system and presenting the design and evaluation of two distributed file systems that target the usage of node-local SSDs and object stores. First, we have shown rare insights into how decades of development have shaped the complex tracing facilities in the closed-source GPFS parallel file system and discussed the corresponding performance and storage overheads that make examining system behavior difficult. To alleviate these overheads, we have proposed a new tracing interface, called *FlexTrace*, which offers significantly more control over which traces are collected and can even make always-on tracing

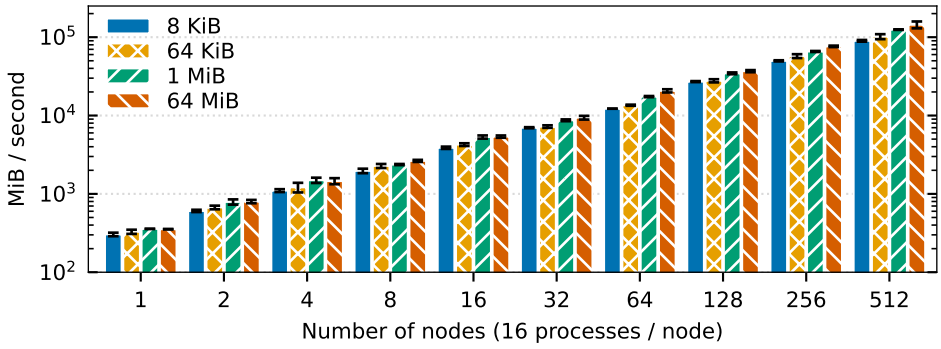
a possibility. Second, we have introduced the temporary *GekkoFS* burst buffer file system, which uses a compute node's local storage capabilities to pool together the storage and performance of many nodes to provide a single global namespace. It can be deployed *ad hoc* within seconds and can be used by any non-privileged user. GekkoFS has shown to significantly outperform the metadata and data capabilities of parallel file systems and remains in continuous international development. Finally, we have presented the *DelveFS* event-driven semantic file system for object stores which provides users the ability to define custom file systems allowing multiple unique *views* onto the object store, severely improving the usability of huge containers. Like GekkoFS, DelveFS clients can be deployed ad hoc for a given *view* configuration. DelveFS keeps these views consistent by exploiting object store events, further provides dual-access, and offers similar metadata and data performances as the native object store interface.

Appendix

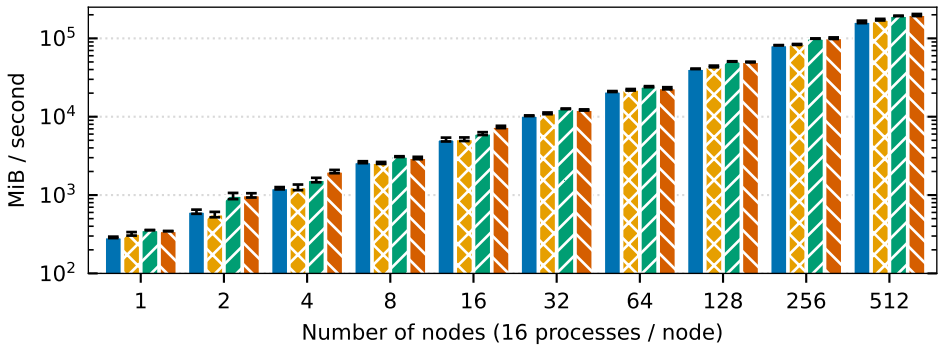
A

The appendix contains additional experiments in the context of this thesis.

GekkoFS metadata cache

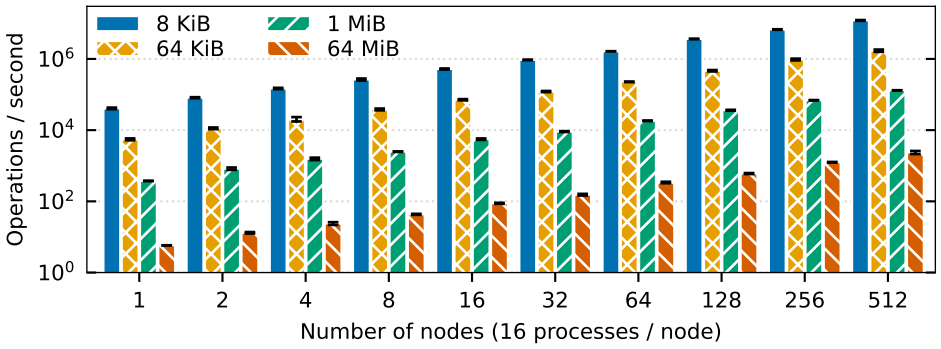


(a) Write throughput

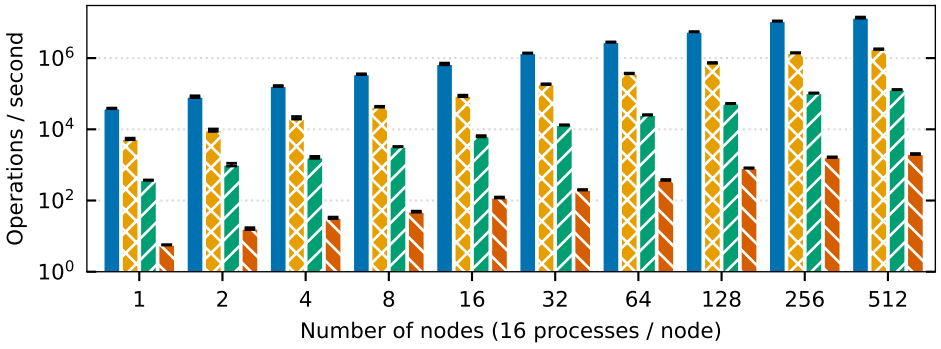


(b) Read throughput

Figure A.1.: GekkoFS's sequential write and read throughput with a metadata client cache for all processes operating on a single shared file (see Section 4.5.4).



(a) Write IOPS



(b) Read IOPS

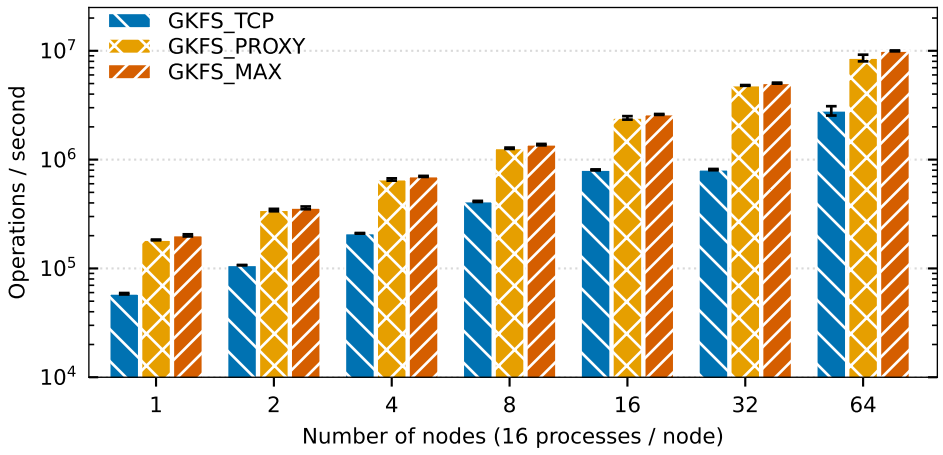
Figure A.2.: GekkoFS's IOPS for a sequential workload with a metadata client cache for all processes operating on a single shared file (see Section 4.5.4).

GekkoFS proxy

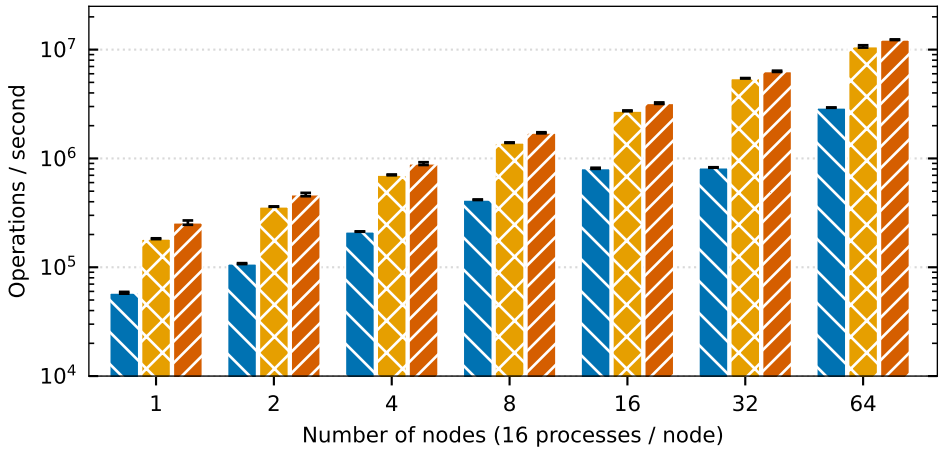
The following figures will compare the GekkoFS proxy (see Section 4.6) with the original GekkoFS implementation. Each data point represents the average of ten iterations unless otherwise specified. Further, unless otherwise specified, the GekkoFS daemons are using shared memory as its data backend. The used abbreviations of the compared network configurations are defined as follows:

1. GekkoFS without proxy using the native Omni-Path protocol (psm2) presenting GekkoFS's maximum performance (`GKFS_MAX`).
2. GekkoFS without the proxy using TCP/IP over Omni-Path (`GKFS_TCP`).
3. GekkoFS with the proxy using the native Omni-Path protocol (psm2) for communication between the proxy and the daemons (`GKFS_PROXY`).
4. GekkoFS using the node-local SATA SSDs of MOGON II (`GKFS_SSD`). Since the SSDs represent the performance bottleneck, the throughput of the three above network configurations are similar for all `GKFS_SSD` experiments.

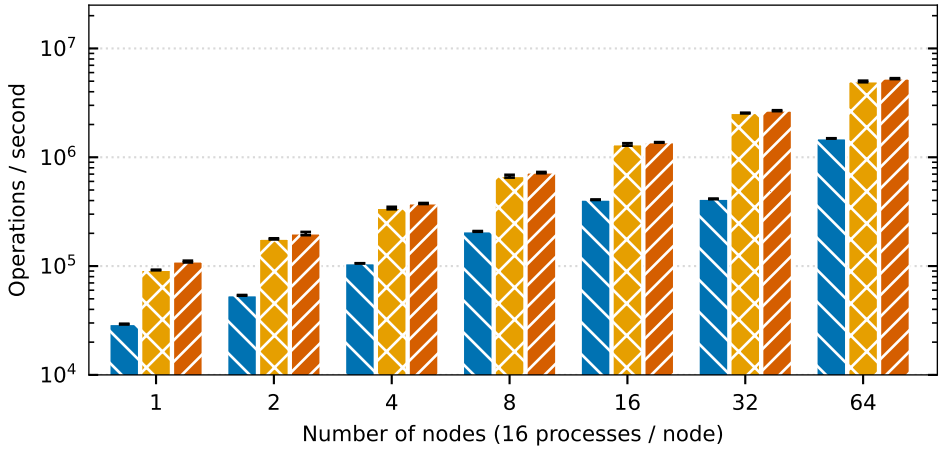
For these experiments, we used the Skylake compute nodes of MOGON II.



(a) Create throughput

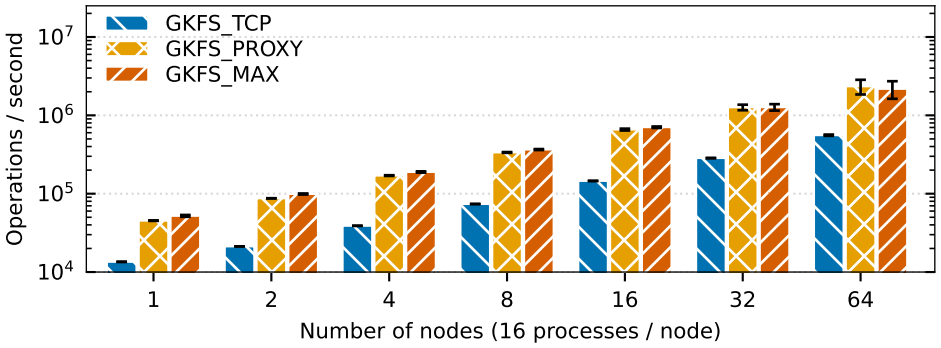


(b) Stat throughput

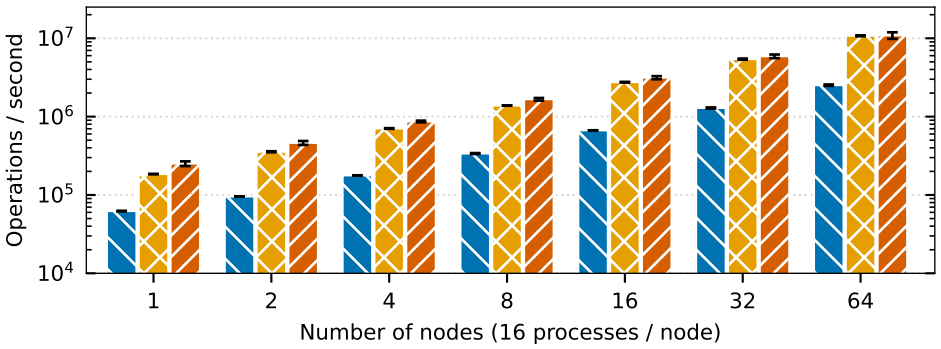


(c) Remove throughput

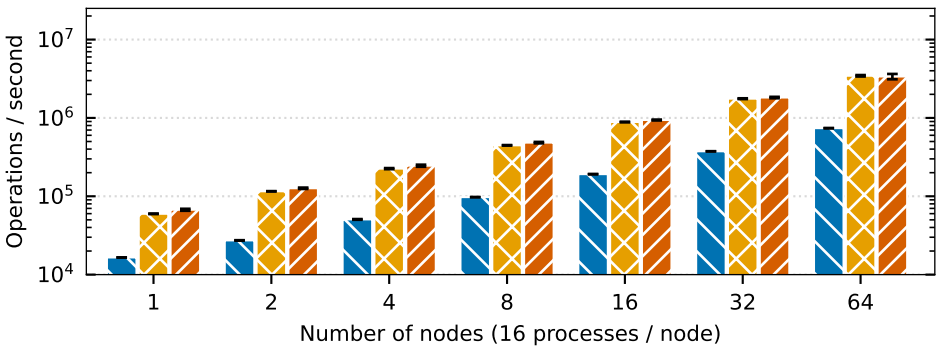
Figure A.3.: GekkoFS's average file create, stat, and remove throughput for an increasing number of nodes using the GekkoFS proxy. For each operation, we used a workload of 100,000 files per process with 16 processes per node comparing several GekkoFS configurations (see Section 4.6).



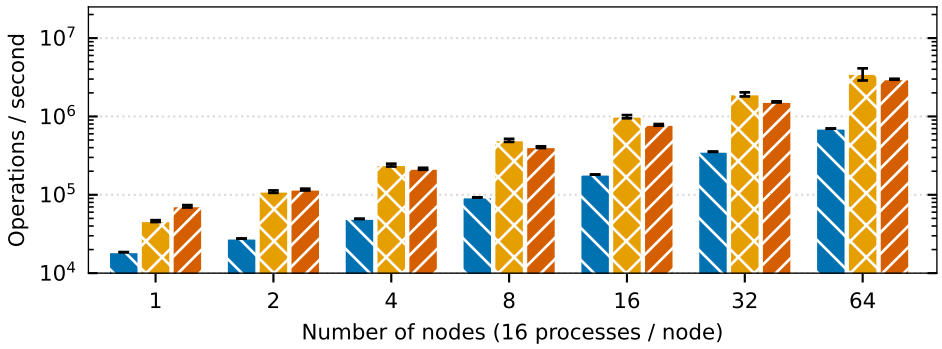
(a) Create throughput



(b) Stat throughput

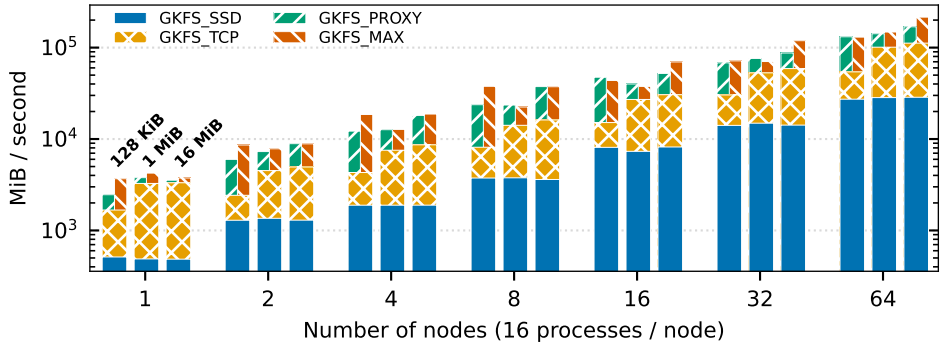


(c) Read throughput

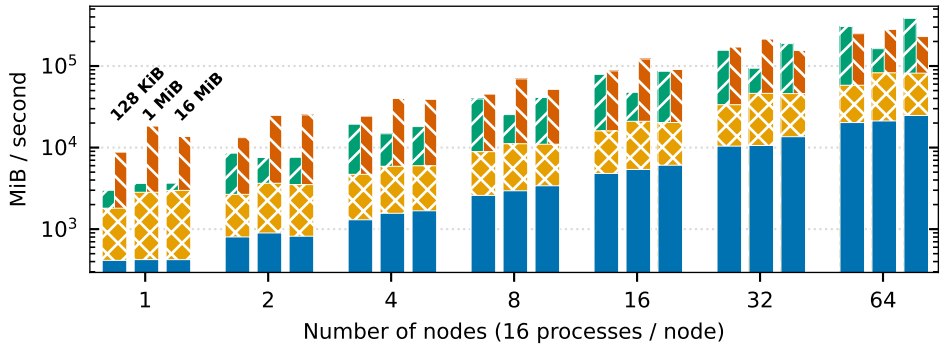


(d) Remove throughput

Figure A.4.: GekkoFS's file create, stat, read, and remove throughput for 4 KiB-sized files for an increasing number of nodes using the GekkoFS proxy. For each operation, we used a workload of 100,000 files per process with 16 processes per node comparing several GekkoFS configurations (see Section 4.6).



(a) Write throughput



(b) Read throughput

Figure A.5.: GekkoFS's maximum sequential write and read throughput for up to 64 nodes using the GekkoFS proxy compared to other network protocols for various I/O sizes. In each experiment, 16 processes were run per node, with each process sequentially writing and reading a 1 GiB file to and from memory (see Section 4.6).

List of Figures

List of Tables

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 265–283 (cit. on pp. 56, 100).

[2] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. "Ursa Minor: Versatile Cluster-based Storage". In: *Proceedings of the Conference on File and Storage Technologies (FAST), December 13-16, San Francisco, California, USA*. 2005 (cit. on pp. 21, 108).

[3] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z. Zhang. "Unreeling netflix: Understanding and improving multi-CDN movie delivery". In: *Proceedings of the IEEE INFOCOM, Orlando, FL, USA, March 25-30*. 2012, pp. 1620–1628 (cit. on p. 108).

[4] A. Aggarwal, W. Scott, E. Rustici, D. Bucciero, A. Haskins, and W. Matthews. *Method and apparatus for determining a communications path between two nodes in an Internet Protocol (IP) network*. US Patent 5,675,741. Oct. 1997 (cit. on p. 27).

[5] M. Ahrens, V. H. Jeff Bonwick, M. Maybee, and M. Shellenbaum. "The Zettabyte File System". In: *FAST '03 Conference on File and Storage Technologies, Work-in-Progress Reports, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. 2003 (cit. on p. 26).

[6] A. Aranya, C. P. Wright, and E. Zadok. "Tracefs: A File System to Trace Them All". In: *Proceedings of the FAST '04 Conference on File and Storage Technologies, March 31 - April 2, 2004, Grand Hyatt Hotel, San Francisco, California, USA*. USENIX, 2004, pp. 129–145 (cit. on pp. 2, 27).

[7] J. Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. 1st ed. O'Reilly and Associates, Oct. 2014 (cit. on pp. 20, 21, 108, 120).

[8] D. Auer. "Dynamic data placement in distributed ad hoc file systems". Bachelor's Thesis. Johannes Gutenberg University Mainz, 2021 (cit. on p. 104).

[10] E. Ayday, E. D. Cristofaro, J. Hubaux, and G. Tsudik. "Whole Genome Sequencing: Revolutionary Medicine or Privacy Nightmare?" In: *IEEE Computer* 48.2 (2015), pp. 58–66 (cit. on p. 108).

[11] M. Baker. “Next-generation sequencing: adjusting to data overload”. In: *Nature Methods* 7 (2010), pp. 495–499 (cit. on p. 108).

[13] O. Bause. “Slurm Burst Buffer Implementation and Coupling with Lustre Bandwidth Guarantees”. Bachelor’s Thesis. Johannes Gutenberg University Mainz, 2020 (cit. on p. 104).

[15] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, and B. Mohr. “Automatic Trace-Based Performance Analysis of Metacomputing Applications”. In: *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*. IEEE, 2007, pp. 1–10 (cit. on p. 28).

[19] J. Bent, G. A. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. “PLFS: a checkpoint filesystem for parallel applications”. In: *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA*. 2009 (cit. on pp. 2, 16, 18, 19, 54, 55, 58, 78, 79).

[20] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. *XRay: A Function Call Tracing System*. White Paper. 2016 (cit. on pp. 25, 29).

[21] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo. “SCFS: A Shared Cloud-backed File System”. In: *USENIX Annual Technical Conference (ATC), Philadelphia, PA, USA, June 19-20*. 2014, pp. 169–180 (cit. on pp. 2, 21, 109, 112, 113).

[22] J. L. Bez, A. Miranda, R. Nou, F. Z. Boito, T. Cortes, and P. O. A. Navaux. “Arbitration Policies for On-Demand User-Level I/O Forwarding on HPC Platforms”. In: *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 2021, pp. 577–586 (cit. on p. 104).

[23] A. Birrell and B. J. Nelson. “Implementing Remote Procedure Calls”. In: *ACM Trans. Comput. Syst.* 2.1 (1984), pp. 39–59 (cit. on p. 67).

[24] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller. “Twizzler: a Data-Centric OS for Non-Volatile Memory”. In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 2020, pp. 65–80 (cit. on p. 38).

[25] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin. “Automatic I/O scheduling algorithm selection for parallel file systems”. In: *Concurr. Comput. Pract. Exp.* 28.8 (2016), pp. 2457–2472 (cit. on p. 104).

[27] L. Bottou. “Large-Scale Machine Learning with Stochastic Gradient Descent”. In: *19th International Conference on Computational Statistics, COMPSTAT 2010, Paris, France, August 22-27, 2010 - Keynote, Invited and Contributed Papers*. Physica-Verlag, 2010, pp. 177–186 (cit. on p. 79).

[28] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel - from I/O ports to process management: covers version 2.6 (3. ed.)* O’Reilly, 2005 (cit. on p. 6).

[29] P. Braam. “The Lustre Storage Architecture”. In: *CoRR* abs/1903.01955 (2019) (cit. on pp. 12, 15, 20, 26, 72).

[30] P. J. Braam and P. Schwan. “Lustre: The intergalactic file system”. In: *Ottawa Linux Symposium*. 2002, p. 50 (cit. on pp. 2, 16, 17, 26, 28, 30, 56, 58, 71).

[32] A. Brinkmann, K. Mohror, W. Yu, P. H. Carns, T. Cortes, S. Klasky, A. Miranda, F. Pfreundt, R. B. Ross, and M. Vef. “Ad Hoc File Systems for High-Performance Computing”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 4–26 (cit. on pp. xi, xiv, 3, 19, 54, 76, 78, 79).

[33] B. M. Bruffey, G. S. Sidhu, P. W. Dirks, and C. R. McFall. *Hierarchical file system to provide cataloging and retrieval of data*. US Patent 4,945,475. 1990 (cit. on p. 1).

[34] H. Brunst, H. Hoppe, W. E. Nagel, and M. Winkler. “Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach”. In: *Computational Science - ICCS 2001, International Conference, San Francisco, CA, USA, May 28-30, 2001. Proceedings, Part II*. Vol. 2074. Lecture Notes in Computer Science. Springer, 2001, pp. 751–760 (cit. on p. 28).

[35] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, et al. “Windows Azure Storage: a highly available cloud storage service with strong consistency”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal, October 23-26, 2011*. 2011, pp. 143–157 (cit. on p. 21).

[36] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. “Dynamic Instrumentation of Production Systems”. In: *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004, Boston Marriott Copley Place, Boston, MA, USA*. 2004, pp. 15–28 (cit. on p. 29).

[37] M. Cao, S. Bhattacharya, and T. Ts’o. “Ext4: The Next Generation of Ext2/3 Filesystem”. In: *Linux Storage and Filesystem Workshop, February 12–13, 2007, San Jose, CA*. 2007 (cit. on pp. 11, 59).

[38] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. “State of the art: Where we are with the ext3 filesystem”. In: *Proceedings of the Ottawa Linux Symposium (OLS)*. 2005, pp. 69–96 (cit. on p. 11).

[39] R. Card, T. Ts’o, and S. Tweedie. “Design and implementation of the second extended filesystem”. In: *Proceedings of the First Dutch Interational Symposium on Linux*. 1995 (cit. on p. 10).

[40] P. Carns, Y. Yao, K. Harms, R. Latham, R. Ross, and K. Antypas. “Production I/O characterization on the Cray XE6”. In: *Proceedings of the Cray User Group meeting*. Vol. 2013. 2013 (cit. on p. 18).

[41] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. “PVFS: A Parallel File System for Linux Clusters”. In: *4th Annual Linux Showcase & Conference 2000, Atlanta, Georgia, USA, October 10-14, 2000*. USENIX Association, 2000 (cit. on pp. 2, 66).

[42] P. H. Carns, J. Jenkins, C. D. Cranor, S. Atchley, S. Seo, S. Snyder, and R. B. Ross. “Enabling NVM for Data-Intensive Scientific Services”. In: *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW@OSDI 2016, Savannah, GA, USA, November 1, 2016*. 2016 (cit. on pp. 55, 68, 69, 116).

[43] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. “24/7 Characterization of petascale I/O workloads”. In: *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*. IEEE Computer Society, 2009, pp. 1–10 (cit. on pp. 27, 28).

[44] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure. “Characterizing Deep-Learning I/O Workloads in TensorFlow”. In: *3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS@SC, Dallas, TX, USA, November 12*. 2018, pp. 54–63 (cit. on p. 54).

[45] S. W. D. Chien, A. Podobas, I. B. Peng, and S. Markidis. “tf-Darshan: Understanding Fine-grained I/O Performance in Machine Learning Workloads”. In: *IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, September 14-17*. 2020, pp. 359–370 (cit. on p. 54).

[46] A. Choudhary, W.-k. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham. “Scalable I/O and analytics”. In: *Journal of Physics: Conference Series*. Vol. 180. 1. 2009, p. 012048 (cit. on pp. 17, 58).

[47] G. Congiu, S. Narasimhamurthy, T. Süß, and A. Brinkmann. “Improving Collective I/O Performance Using Non-volatile Memory Devices”. In: *IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, September 12-16*. 2016, pp. 120–129 (cit. on p. 78).

[48] G. Cooper. “DTrace: dynamic tracing in oracle Solaris, Mac OS X, and free BSD by Brendan Gregg and Jim Mauro”. In: *ACM SIGSOFT Softw. Eng. Notes* 37.1 (2012), p. 34 (cit. on pp. 27–29).

[49] F. J. Corbató, F. H. Saltzer, and C. T. Clingen. “Multics: the first seven years”. In: *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972*. Vol. 40. AFIPS Conference Proceedings. AFIPS, 1972, pp. 571–583 (cit. on p. 1).

[52] P. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. “Input/Output Characteristics of Scalable Parallel Applications”. In: *Proceedings Supercomputing ’95, San Diego, CA, USA, December 4-8, 1995*. 1995, p. 59 (cit. on p. 54).

[54] A. Davies and A. Orsaria. “Scale out with GlusterFS”. In: *Linux Journal* 2013.235 (2013) (cit. on pp. 2, 61).

[57] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 2009, pp. 248–255 (cit. on pp. 79, 100).

[58] M. Desnoyers and M. R. Dagenais. “The LTTng tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux”. In: *Proceedings of the Ottawa Linux Symposium, July 19th – 22nd, 2006 Ottawa, Ontario, Canada*. 2006, pp. 209–223 (cit. on pp. 27, 28).

[59] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. “Optimizing Space Amplification in RocksDB”. In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 2017 (cit. on pp. 70, 117).

[60] M. Dorier, G. Antoniu, R. B. Ross, D. Kimpe, and S. Ibrahim. “CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. 2014, pp. 155–164 (cit. on pp. 3, 54).

[61] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras. “Inside dropbox: understanding personal cloud storage services”. In: *Proceedings of the 12th ACM SIGCOMM Internet Measurement Conference (IMC), Boston, MA, USA, November 14-16*. 2012, pp. 481–494 (cit. on p. 108).

[62] R. Duncan. *The MS-Dos encyclopedia*. Vol. 124320. Microsoft Press, 1988 (cit. on p. 1).

[63] F. C. Eigler. “Problem Solving With Systemtap”. In: *Proceedings of the Ottawa Linux Symposium, July 19th – 22nd, 2006 Ottawa, Ontario, Canada*. 2006, pp. 261–268 (cit. on pp. 27–29).

[64] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. “Passive NFS Tracing of Email and Research Workloads”. In: *Proceedings of the FAST ’03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003 (cit. on p. 27).

[65] D. Ellard and M. I. Seltzer. “New NFS Tracing Tools and Techniques for System Analysis”. In: *Proceedings of the 17th Conference on Systems Administration (LISA 2003), San Diego, California, USA, October 26-31, 2003*. USENIX, 2003, pp. 73–86 (cit. on p. 27).

[66] Ú. Erlingsson, M. Peinado, S. Peter, and M. Budiu. “Fay: extensible distributed tracing from kernels to clusters”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. ACM, 2011, pp. 311–326 (cit. on p. 29).

[67] D. G. Feitelson and A. M. Weil. “Utilization and Predictability in Scheduling the IBM SP2 with Backfilling”. In: *12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP ’98), March 30 - April 3, 1998, Orlando, Florida, USA, Proceedings*. IEEE Computer Society, 1998, pp. 542–546 (cit. on p. 96).

[68] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, R. Brightwell, and T. Kordenbrock. *Increasing Fault Resiliency in a Message-Passing Environment*. Tech. rep. SAND2009-6753. Sandia National Laboratories, 2009 (cit. on pp. 54, 78).

[69] P. Fischer, J. Lottes, and H. Tufo. *Nek5000*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States), 2007 (cit. on pp. 77, 109, 137).

[70] M. Folk, A. Cheng, and K. Yates. "HDF5: A file format and I/O library for high performance computing applications". In: *Proceedings of supercomputing*. Vol. 99. 1999, pp. 5–33 (cit. on p. 54).

[71] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. "X-Trace: A Pervasive Network Tracing Framework". In: *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX, 2007 (cit. on p. 28).

[72] A. Frank, M. Baumgartner, R. Salkhordeh, and A. Brinkmann. "Improving checkpointing intervals by considering individual job failure probabilities". In: *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 2021, pp. 299–309 (cit. on pp. 18, 78).

[73] W. Frings, F. Wolf, and V. Petkov. "Scalable massively parallel I/O to task-local files". In: *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA*. 2009 (cit. on pp. 19, 58).

[77] R. M. A. Gad. "Enhancing application checkpointing and migration in HPC". PhD thesis. University of Mainz, Germany, 2018 (cit. on pp. 18, 78).

[78] J. Garcia-Blas, D. E. Singh, and J. Carretero. "IMSS: In-Memory Storage System for Data Intensive Applications". In: *ISC High Performance 2022 International Workshops*. Vol. 13387. Lecture Notes in Computer Science. Springer, 2022 (cit. on p. 57).

[79] S. L. Garfinkel. "Commodity Grid Computing with Amazon's S3 and EC2". In: *;login:* 32.1 (2007) (cit. on pp. 20, 21).

[80] D. Geels, G. Altekar, S. Shenker, and I. Stoica. "Replay Debugging for Distributed Applications (Awarded Best Paper!)" In: *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*. USENIX, 2006, pp. 289–300 (cit. on p. 27).

[81] S. Ghemawat, H. Gobioff, and S. Leung. "The Google file system". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. ACM, 2003, pp. 29–43 (cit. on pp. 12, 26).

[82] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. O'Toole. "Semantic File Systems". In: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. 1991, pp. 16–25 (cit. on pp. 109, 112).

[83] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers. "Recovering system metrics from kernel trace". In: *Linux Symposium*. Vol. 109. 2011 (cit. on p. 26).

[87] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *CoRR* abs/1706.02677 (2017) (cit. on p. 79).

[88] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth. “Analysis of the ECMWF Storage Landscape”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. 2015, pp. 15–27 (cit. on p. 108).

[89] R. Hai, S. Geisler, and C. Quix. “Constance: An Intelligent Data Lake System”. In: *Proceedings of the International Conference on Management of Data (SIGMOD), San Francisco, CA, USA, June 26 - July 01*. 2016, pp. 2097–2100 (cit. on pp. 21, 108).

[90] S. Haria, M. D. Hill, and M. M. Swift. “MOD: Minimally Ordered Durable Datastructures for Persistent Memory”. In: *ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 775–788 (cit. on p. 38).

[91] Y. Hashimoto and K. Aida. “Evaluation of Performance Degradation in HPC Applications with VM Consolidation”. In: *Third International Conference on Networking and Computing, ICNC 2012, Okinawa, Japan, December 5-7, 2012*. 2012, pp. 273–277 (cit. on p. 98).

[92] R. L. Haskin and F. B. Schmuck. “The Tiger Shark File System”. In: *Forty-First IEEE Computer Society International Conference: Technologies for the Information Superhighway, COMPCON 1996, Santa Clara, California, USA, February 25-28, 1996, Digest of Papers*. IEEE Computer Society, 1996, pp. 226–231 (cit. on p. 12).

[94] T. Haynes. “Network File System (NFS) Version 4 Minor Version 2 Protocol”. In: *RFC* 7862 (2016) (cit. on pp. 2, 24, 110, 115).

[95] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright. *Architecture and design of cray datawarp*. `https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf`. White Paper. Accessed on Feb, 23, 2021. 2016 (cit. on p. 19).

[96] V. Henson, A. van de Ven, A. Gud, and Z. Brown. “Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair”. In: *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep), Seattle, WA, USA, November 8*. 2006 (cit. on p. 61).

[97] F. Herold, S. Breuner, and J. Heichler. *An Introduction to BeeGFS*. `https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf`. White Paper. Accessed on Feb, 23, 2021. 2014 (cit. on pp. 2, 28, 56, 57, 71, 72).

[98] D. Hildebrand and F. Schmuck. “Chapter 9—GPFS”. In: *High Performance Parallel I/O*. CRC Press, 2014 (cit. on p. 13).

[99] D. Hildebrand and F. B. Schmuck. “On Making GPFS Truly General”. In: *login Usenix Mag.* 40.3 (2015) (cit. on pp. 13, 25).

[100] M. E. Hoskins. “SSHFS: Super Easy File Access over SSH”. In: *Linux Journal* 2006.146 (June 2006) (cit. on pp. 2, 61).

[101] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. “Semantic-Aware Metadata Organization Paradigm in Next-Generation File Systems”. In: *IEEE Trans. Parallel Distrib. Syst.* 23.2 (2012), pp. 337–344 (cit. on pp. 109, 112).

[104] A. C. Inc. *Inside Macintosh Volume II*. Addison-Wesley Publishing Company, Inc., 1985 (cit. on p. 1).

[108] S. A. Jacobs, J. Gaffney, T. Benson, P. B. Robinson, J. L. Peterson, B. K. Spears, B. V. Essen, D. Hysom, J. Yeom, T. Moon, R. Anirudh, J. J. Thiagarajan, S. Liu, and P. Bremer. “Parallelizing Training of Deep Generative Models on Massive Scientific Datasets”. In: *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*. IEEE, 2019, pp. 1–10 (cit. on p. 79).

[109] H. Jasak, A. Jemcov, Z. Tukovic, et al. “OpenFOAM: A C++ library for complex physics simulations”. In: *International workshop on coupled methods in numerical dynamics*. Vol. 1000. 2007, pp. 1–20 (cit. on pp. 56, 70, 77, 80, 98).

[110] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue. “Automatic, Application-Aware I/O Forwarding Resource Allocation”. In: *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 2019, pp. 265–279 (cit. on p. 104).

[112] S. J. Kim, S. W. Son, W. Liao, M. T. Kandemir, R. Thakur, and A. N. Choudhary. “IOPin: Runtime Profiling of Parallel I/O in HPC Systems”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*. IEEE Computer Society, 2012, pp. 18–23 (cit. on p. 29).

[113] S. R. Kleiman. “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”. In: *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*. USENIX Association, 1986, pp. 238–247 (cit. on p. 2).

[114] A. Kougkas, H. Devarajan, and X. Sun. “Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, AZ, USA, June 11-15, 2018*. 2018, pp. 219–230 (cit. on pp. 57, 97).

[115] R. Krishnakumar. “Kernel korner: kprobes – a kernel debugger”. In: *Linux Journal* 2005.133 (2005), p. 11 (cit. on p. 29).

[116] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. H. Phillips, A. Mahesh, M. A. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston. “Exascale deep learning for climate analytics”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE, 2018, 51:1–51:12 (cit. on p. 79).

[121] S. Lang. *Parallel File Systems*. Guest Lecture. Argonne National Laboratory, 2010 (cit. on p. 16).

[122] B. Langmead. “Aligning short sequencing reads with Bowtie”. In: *Current protocols in bioinformatics* 32.1 (2010), pp. 11–7 (cit. on pp. 131, 134).

[123] R. Latham, R. B. Ross, and R. Thakur. “The Impact of File Systems on MPI-IO Scalability”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*. 2004, pp. 87–96 (cit. on pp. 17, 58).

[124] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa. “I/O Scheduling Service for Multi-Application Clusters”. In: *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. IEEE Computer Society, 2006 (cit. on p. 104).

[125] C. Lee, D. Sim, J. Y. Hwang, and S. Cho. “F2FS: A New File System for Flash Storage”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 16-19*. 2015, pp. 273–286 (cit. on p. 111).

[126] S. Lee and C. Shields. “Tracing the source of network attack: A technical, legal and societal problem”. In: *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*. Vol. 6. Citeseer. 2001 (cit. on p. 27).

[127] P. H. Lensing, T. Cortes, and A. Brinkmann. “Direct lookup and hash-based metadata placement for local file systems”. In: *6th Annual International Systems and Storage Conference, SYSTOR ’13, Haifa, Israel - June 30 - July 02, 2013*. ACM, 2013, 5:1–5:11 (cit. on p. 109).

[128] P. H. Lensing, T. Cortes, J. Hughes, and A. Brinkmann. “File System Scalability with Highly Decentralized Metadata on Independent Storage Devices”. In: *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, May 16-19*. 2016, pp. 366–375 (cit. on pp. 55, 67, 72, 109).

[129] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. “Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems”. In: *7th USENIX Conference on File and Storage Technologies (FAST), February 24-27, San Francisco, CA, USA. Proceedings*. 2009, pp. 153–166 (cit. on p. 111).

[130] K. Lillaney, V. Tarasov, D. Pease, and R. C. Burns. “Agni: An Efficient Dual-access File System over Object Storage”. In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC), Santa Cruz, CA, USA, November 20-23*. 2019, pp. 390–402 (cit. on pp. 21, 109, 112, 113).

[134] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. “On the role of burst buffers in leadership-class storage systems”. In: *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*. 2012, pp. 1–11 (cit. on pp. 19, 55).

[136] J. F. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton. “DAOS and friends: a proposal for an exascale storage system”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. IEEE Computer Society, 2016, pp. 585–596 (cit. on pp. 59, 61).

[137] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. 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, CLADE@HPDC 2008, Boston, MA, USA, June 23, 2008*. 2008, pp. 15–24 (cit. on pp. 54, 60).

[138] J. F. Lofstead and R. Ross. “Insights for exascale IO APIs from building a petascale IO API”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC’13, Denver, CO, USA - November 17 - 21, 2013*. 2013, 87:1–87:12 (cit. on p. 98).

[139] J. F. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. “Managing Variability in the IO Performance of Petascale Storage Systems”. In: *Conference on High Performance Computing Networking, Storage and Analysis,SC 2010, New Orleans, LA, USA, November 13-19, 2010*. 2010, pp. 1–12 (cit. on pp. 3, 97).

[141] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. ACM, 2005, pp. 190–200 (cit. on p. 29).

[142] X. Luo, F. Mueller, P. H. Carns, J. Jenkins, R. Latham, R. B. Ross, and S. Snyder. “ScalaIOExtrap: Elastic I/O Tracing and Extrapolation”. In: *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 585–594 (cit. on p. 28).

[144] H. Luu, M. Winslett, W. Gropp, R. B. Ross, P. H. Carns, K. Harms, Prabhat, S. Byna, and Y. Yao. “A Multiplatform Study of I/O Behavior on Petascale Supercomputers”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*. ACM, 2015, pp. 33–44 (cit. on p. 18).

[145] J. Mace, R. Roelke, and R. Fonseca. “Pivot tracing: dynamic causal monitoring for distributed systems”. In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 2015, pp. 378–393 (cit. on pp. 25, 27, 29).

[148] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivie. “The new ext4 filesystem: current status and future plans”. In: *Proceedings of the 2007 Linux Symposium, Volume Two, Ottawa, Ontario, Canada*. 2007, pp. 21–32 (cit. on pp. 11, 111).

[149] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy. “Capsule: an energy-optimized object storage system for memory-constrained sensor devices”. In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys), Boulder, Colorado, USA*. 2006, pp. 195–208 (cit. on p. 21).

[150] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. S. Pérez. “Týr: blob storage meets built-in transactions”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Salt Lake City, UT, USA, November 13-18*. 2016, pp. 573–584 (cit. on p. 21).

[151] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. “Data Ingestion for the Connected World”. In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017 (cit. on p. 18).

[152] Q. Meng, W. Chen, Y. Wang, Z. Ma, and T. Liu. “Convergence analysis of distributed stochastic gradient descent with shuffling”. In: *Neurocomputing* 337 (2019), pp. 46–57 (cit. on p. 79).

[153] M. P. Mesnier, G. R. Ganger, and E. Riedel. “Object-based storage”. In: *IEEE Communications Magazine* 41.8 (2003), pp. 84–90 (cit. on pp. 20, 21, 110).

[154] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. C. López-Hernández, J. Hendricks, G. R. Ganger, and D. R. O’Hallaron. “//TRACE: Parallel Trace Replay with Approximate Causal Events”. In: *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*. USENIX, 2007, pp. 153–167 (cit. on pp. 27, 28).

[155] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, and Y. Kageyama. “ImageNet/ResNet-50 Training in 224 Seconds”. In: *CoRR* abs/1811.05233 (2018) (cit. on p. 79).

[156] A. Miranda, S. Effert, Y. Kang, E. L. Miller, I. Popov, A. Brinkmann, T. Friedetzky, and T. Cortes. “Random Slicing: Efficient and Scalable Data Placement for Large-Scale Storage Systems”. In: *ACM Trans. Storage* 10.3 (2014), 9:1–9:35 (cit. on p. 105).

[158] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System”. In: *Conference on High Performance Computing Networking, Storage and Analysis (SC), New Orleans, LA, USA, November 13-19, 2010*. 2010 (cit. on p. 78).

[159] M. Moore, D. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Quarles, S. Sampson, S. Yang, and B. Wilson. “OrangeFS: Advancing PVFS”. In: *FAST poster session* (2011) (cit. on pp. 2, 58, 60, 66).

[160] N. Moti, F. Schimmelpfennig, R. Salkhordeh, D. Klopp, T. Cortes, U. Rückert, and A. Brinkmann. “Simurgh: a fully decentralized and secure NVMM user space file system”. In: *SC ’21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*. ACM, 2021, 46:1–46:14 (cit. on pp. 38, 59, 62, 102).

[161] J. Murty. *Programming Amazon web services - S3, EC2, SQS, FPS, and SimpleDB: outsource your infrastructure*. O'Reilly, 2008 (cit. on pp. 18, 20, 108, 120).

[162] D. Nagle, M. Factor, S. Iren, D. Naor, E. R. and Ohad Rodeh, and J. Satran. "The ANSI T10 object-based storage standard and current implementations". In: *IBM Journal of Research and Development* 52.4-5 (2008), pp. 401–412 (cit. on p. 20).

[166] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. "BlobSeer: Next-generation data management for large scale infrastructures". In: *Journal of Parallel and Distributed Computing* 71.2 (2011), pp. 169–184 (cit. on p. 108).

[167] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello. "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 911–920 (cit. on pp. 78, 79).

[168] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. "File-Access Characteristics of Parallel Scientific Workloads". In: *IEEE Trans. Parallel Distrib. Syst.* 7.10 (1996), pp. 1075–1089 (cit. on pp. 17, 54, 56, 58).

[169] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. "ScalaTrace: Scalable compression and replay of communication traces for high-performance computing". In: *J. Parallel Distributed Comput.* 69.8 (2009), pp. 696–710 (cit. on p. 28).

[170] R. Nou, A. Miranda, M. Siquier, and T. Cortes. "Improving OpenStack Swift interaction with the I/O Stack to Enable Software Defined Storage". In: *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC^2), Kanazawa, Japan, November 22-25*. 2017, pp. 63–70 (cit. on p. 21).

[174] S. Oeste, M. Vef, M. Soysal, W. E. Nagel, A. Brinkmann, and A. Streit. "ADA-FS - Advanced Data Placement via Ad hoc File Systems at Extreme Scales". In: *Software for Exascale Computing - SPPEXA 2016-2019*. Vol. 136. Lecture Notes in Computational Science and Engineering. Springer, 2020, pp. 29–59 (cit. on p. xiii).

[175] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. B. Ross, and Y. Ishikawa. "Optimization Techniques at the I/O Forwarding Layer". In: *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*. IEEE Computer Society, 2010, pp. 312–321 (cit. on p. 104).

[176] OpenIO. *OpenIO Core Solution Description*. `https://www.openio.io/resources/`. White Paper. Accessed on Jan, 16, 2021. 2016 (cit. on pp. 108, 111, 115).

[178] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons, et al. "Olcfs 1 tb/s, next-generation lustre file system". In: *Proceedings of Cray User Group Conference (CUG 2013)*. 2013, pp. 1–12 (cit. on pp. 17, 89).

[180] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. V. Essen. “The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs With Hybrid Parallelism”. In: *IEEE Trans. Parallel Distributed Syst.* 32.7 (2021), pp. 1641–1652 (cit. on p. 79).

[181] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari. “Revisiting I/O behavior in large-scale storage systems: the expected and the unexpected”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19*. 2019, 65:1–65:13 (cit. on p. 54).

[182] S. Patil and G. A. Gibson. “Scale and Concurrency of GIGA+: File System Directories with Millions of Files”. In: *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*. 2011, pp. 177–190 (cit. on pp. 18, 19, 58, 78).

[183] S. Patil, K. Ren, and G. Gibson. “A Case for Scaling HPC Metadata Performance through De-specialization”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*. 2012, pp. 30–35 (cit. on pp. 19, 58).

[184] D. A. Patterson, G. A. Gibson, and R. H. Katz. “A Case for Redundant Arrays of Inexpensive Disks (RAID)”. In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*. ACM Press, 1988, pp. 109–116 (cit. on p. 12).

[185] W. R. Pearson. “Rapid and sensitive sequence comparison with FASTP and FASTA”. In: *Methods in enzymology* 183 (1990), pp. 63–98 (cit. on p. 119).

[186] D. Pease, A. Amir, L. V. Real, B. Biskeborn, M. Richmond, and A. Abe. “The Linear Tape File System”. In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. IEEE Computer Society, 2010, pp. 1–8 (cit. on p. 2).

[188] D. Petcu, C. Craciun, and M. Rak. “Towards a Cross Platform Cloud API - Components for Cloud Federation”. In: *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 7-9 May, 2011*. 2011, pp. 166–169 (cit. on p. 20).

[189] A. J. Peters, E. Sindrilaru, and G. Adde. “EOS as the present and future solution for data storage at CERN”. In: *21st International Conference on Computing in High Energy and Nuclear Physics (CHEP)*. 2015 (cit. on p. 108).

[191] F. Petrini. “Scaling to Thousands of Processors with Buffer Coscheduling”. In: *Scaling to New Height Workshop*. 2002 (cit. on pp. 54, 78).

[192] I. R. Philp. “Software Failures and the Road to a Petaflop Machine”. In: *Proceedings of the 1st Workshop on High Performance Computing Reliability Issues (HPCRI)*. 2005 (cit. on pp. 54, 78).

[194] D. Poliakoff and M. LeGendre. “Gotcha: An Function-Wrapping Interface for HPC Tools”. In: *Programming and Performance Visualization Tools - International Workshops, ESPT 2017 and VPA 2017, Denver, CO, USA, November 12 and 17, 2017, and ESPT 2018 and VPA 2018, Dallas, TX, USA, November 16 and 11, 2018, Revised Selected Papers*. Vol. 11027. Lecture Notes in Computer Science. Springer, 2018, pp. 185–197 (cit. on p. 63).

[196] *Private communication with GPFS developers*. 2016 (cit. on pp. xii, 23, 25, 28, 30, 33, 37–39, 41, 44).

[197] Y. Qian, W. Cheng, L. Zeng, M.-A. Vef, O. Drokin, A. Dilger, S. Ihara, W. Zhang, Y. Wang, and A. Brinkmann. “MetaWBC: POSIX-compliant metadata write-back caching for distributed file systems”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2022, Dallas, Texas, USA, November 13-18, 2022*. IEEE, 2022 (cit. on pp. xiv, 103).

[198] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann. “A configurable rule based classful token bucket filter network request scheduler for the lustre file system”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, November 12 - 17*. 2017, 6:1–6:12 (cit. on pp. 26, 56).

[199] X. Qiu, H. Li, C. Wu, Z. Li, and F. C. M. Lau. “Cost-Minimizing Dynamic Migration of Content Distribution Services into Hybrid Clouds”. In: *IEEE Trans. Parallel Distributed Syst.* 26.12 (2015), pp. 3330–3345 (cit. on p. 18).

[200] P. Quinn, T. Axelrod, I. Bird, R. Dodson, A. Szalay, and A. Wicenec. “Delivering SKA Science”. In: *CoRR* abs/1501.05367 (2015) (cit. on p. 108).

[201] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda. “A 1 PB/s file system to checkpoint three million MPI tasks”. In: *The 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC), New York, NY, USA - June 17 - 21*. 2013, pp. 143–154 (cit. on pp. 59, 62, 79).

[202] A. Rajgarhia and A. Gehani. “Performance and extension of user space file systems”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*. 2010, pp. 206–213 (cit. on p. 60).

[203] D. A. Reed and J. J. Dongarra. “Exascale computing and big data”. In: *Commun. ACM* 58.7 (2015), pp. 56–68 (cit. on p. 98).

[205] K. Ren, Q. Zheng, S. Patil, and G. A. Gibson. “IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE Computer Society, 2014, pp. 237–248 (cit. on pp. 18, 58, 72).

[207] D. Ritchie and K. Thompson. “The UNIX Time-Sharing System (Reprint)”. In: *Commun. ACM* 26.1 (1983), pp. 84–89 (cit. on p. 17).

[208] O. Rodeh, J. Bacik, and C. Mason. “BTRFS: The Linux B-Tree Filesystem”. In: *ACM Trans. Storage* 9.3 (2013), 9:1–9:32 (cit. on p. 26).

[209] R. Ross, R. Thakur, and A. Choudhary. “Achievements and challenges for I/O in computational science”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. 2005, p. 501 (cit. on p. 54).

[210] R. B. Ross, G. Amvrosiadis, P. H. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, R. W. Robey, D. Robinson, B. W. Settlemyer, G. M. Shipman, S. Snyder, J. Soumagne, and Q. Zheng. “Mochi: Composing Data Services for High-Performance Computing Environments”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 121–144 (cit. on pp. 55, 68).

[211] R. B. Ross and R. Latham. “PVFS - PVFS: a parallel file system”. In: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. 2006, p. 34 (cit. on pp. 2, 56, 58).

[212] S. Rostedt. “Finding origins of latencies using ftrace”. In: *11th Real-Time Linux Workshop*. 2009, pp. 117–130 (cit. on pp. 27–29).

[213] R. Russon and Y. Fledel. “NTFS documentation”. In: *Recuperado el* 1 (2004) (cit. on p. 2).

[217] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. “Diagnosing Performance Changes by Comparing Request Flows”. In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011 (cit. on p. 28).

[218] F. Schimmelpfennig, M. Vef, R. Salkhordeh, A. Miranda, R. Nou, and A. Brinkmann. “Streamlining distributed Deep Learning I/O with ad hoc file systems”. In: *IEEE International Conference on Cluster Computing, CLUSTER 2021, Portland, OR, USA, September 7-10, 2021*. IEEE, 2021, pp. 169–180 (cit. on pp. xi, xii, xiv, 79, 100, 101).

[219] F. B. Schmuck and R. L. Haskin. “GPFS: A Shared-Disk File System for Large Computing Clusters”. In: *Proceedings of the FAST ’02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*. USENIX, 2002, pp. 231–244 (cit. on pp. 2, 3, 11–13, 16, 17, 24, 26, 56, 58, 71, 72, 115).

[220] M. I. Seltzer and N. Murphy. “Hierarchical File Systems Are Dead”. In: *Proceedings of HotOS’09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland*. 2009 (cit. on pp. 109, 112).

[221] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. H. Carns, A. Castelló, D. Genet, T. Hérault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. H. Beckman. “Argobots: A Lightweight Low-Level Threading and Tasking Framework”. In: *IEEE Trans. Parallel Distrib. Syst.* 29.3 (2018), pp. 512–526 (cit. on pp. 55, 69, 70, 116).

[222] A. Sergeev and M. D. Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *CoRR* abs/1802.05799 (2018) (cit. on p. 100).

[223] S. D. Sharma and M. Dagenais. “Enhanced Userspace and In-Kernel Trace Filtering for Production Systems”. In: *J. Comput. Sci. Technol.* 31.6 (2016), pp. 1161–1178 (cit. on pp. 25, 27, 28).

[225] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. IEEE Computer Society, 2010, pp. 1–10 (cit. on pp. 12, 26).

[227] S. Snyder, P. H. Carns, K. Harms, R. B. Ross, G. K. Lockwood, and N. J. Wright. “Modular HPC I/O Characterization with Darshan”. In: *5th Workshop on Extreme-Scale Programming Tools, ESPT SC 2016, Salt Lake City, UT, USA, November 13, 2016*. IEEE, 2016, pp. 9–17 (cit. on pp. 28, 29, 77).

[228] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross. “Mercury: Enabling remote procedure call for high-performance computing”. In: *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*. 2013, pp. 1–8 (cit. on pp. 55, 67, 116).

[229] M. Soysal, M. Berghoff, T. Zirwes, M. Vef, S. Oeste, A. Brinkmann, W. E. Nagel, and A. Streit. “Using On-Demand File Systems in HPC Environments”. In: *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*. IEEE, 2019, pp. 390–398 (cit. on pp. xiv, 83).

[231] M. Subramanian, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, S. Viswanathan, L. Tang, and S. Kumar. “f4: Facebook’s Warm BLOB Storage System”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, October 6-8*. 2014, pp. 383–398 (cit. on p. 21).

[233] P. Sun, W. Feng, R. Han, S. Yan, and Y. Wen. “Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes”. In: *CoRR* abs/1902.06855 (2019) (cit. on p. 79).

[234] T. Süß, L. Nagel, M. Vef, A. Brinkmann, D. Feld, and T. Soddemann. “Pure Functions in C: A Small Keyword for Automatic Parallelization”. In: *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 552–556 (cit. on p. xiv).

[235] T. Süß, L. Nagel, M. Vef, A. Brinkmann, D. Feld, and T. Soddemann. “Pure Functions in C: A Small Keyword for Automatic Parallelization”. In: *Int. J. Parallel Program.* 49.1 (2021), pp. 1–24 (cit. on p. xiv).

[239] S. Tannenbaum Andrew and S. Woodhull Albert. *Operating Systems: Design and Implementation*. 2003 (cit. on p. 1).

[240] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. “Extracting flexible, replayable models from large block traces”. In: *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*. 2012, p. 22 (cit. on p. 28).

[241] O. Tatebe, S. Moriwake, and Y. Oyama. “Gfarm/BB - Gfarm File System for Node-Local Burst Buffer”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 61–71 (cit. on p. 16).

[242] J. Terrace and M. J. Freedman. “Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads”. In: *USENIX Annual Technical Conference (ATC), San Diego, CA, USA, June 14-19*. 2009 (cit. on p. 21).

[243] S. Thapaliya, P. Bangalore, J. F. Lofstead, K. Mohror, and A. Moody. “Managing I/O Interference in a Shared Burst Buffer System”. In: *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*. 2016, pp. 416–425 (cit. on pp. 3, 54).

[244] *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009 (cit. on p. 54).

[245] E. Thereska, B. Salmon, J. D. Strunk, M. Wachs, M. Abd-El-Malek, J. C. López-Hernández, and G. R. Ganger. “Stardust: tracking activity in a distributed storage system”. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006*. ACM, 2006, pp. 3–14 (cit. on p. 28).

[246] R. G. Tinedo, P. G. López, M. S. Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortes, W. Oppermann, and P. Michiardi. “IOStack: Software-Defined Object Storage”. In: *IEEE Internet Comput.* 20.3 (2016), pp. 10–18 (cit. on p. 21).

[248] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker. “Parallel I/O performance: From events to ensembles”. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010, pp. 1–11 (cit. on p. 28).

[250] B. K. R. Vangoor, V. Tarasov, and E. Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems”. In: *15th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 27 - March 2, 2017*. 2017, pp. 59–72 (cit. on pp. 59–61, 113, 118, 127, 133, 134).

[251] M.-A. Vef. “Analyzing File Create Performance in IBM Spectrum Scale”. `https://seafile.rlp.net/f/683e1ca86c7d497ca6fb/`. MA thesis. Johannes Gutenberg University Mainz, 2016 (cit. on pp. xi, 18, 38, 48, 73).

[252] M. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. “GekkoFS - A Temporary Burst Buffer File System for HPC Applications”. In: *J. Comput. Sci. Technol.* 35.1 (2020), pp. 72–91 (cit. on pp. xi, xiii).

[253] M. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. “GekkoFS - A Temporary Distributed File System for HPC Applications”. In: *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, 2018, pp. 319–324 (cit. on pp. xi, xiii).

[254] M. Vef, R. Steiner, R. Salkhordeh, J. Steinkamp, F. Vennetier, J. Smigielski, and A. Brinkmann. “DelveFS - An Event-Driven Semantic File System for Object Stores”. In: *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*. IEEE, 2020, pp. 35–46 (cit. on pp. xi–xiii).

[255] M.-A. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann. “Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale”. In: *ACM Trans. Storage* 14.2 (2018), 18:1–18:24 (cit. on pp. xi–xiii, 2, 12, 18, 84).

[256] M.-A. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann. “Tracing of Complex Production Systems: Obstacles and Solutions”. In: *System Analytics and Characterization (SAC)* 1 (2016) (cit. on pp. xi–xiii, 25).

[257] M. Vilayannur, P. Nath, and A. Sivasubramaniam. “Providing Tunable Consistency for a Parallel File Store”. In: *Proceedings of the FAST ’05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. 2005 (cit. on p. 55).

[258] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. A. Morozov, M. E. Papka, R. B. Ross, and K. Yoshii. “Accelerating I/O Forwarding in IBM Blue Gene/P Systems”. In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 2010, pp. 1–10 (cit. on p. 104).

[259] W. Vogels. “Eventually consistent”. In: *Commun. ACM* 52.1 (2009), pp. 40–44 (cit. on p. 66).

[260] H. Volos, S. N. and Sankaralingam Panneerselvam and Venkatanathan Varadarajan and Prashant Saxena, and M. M. Swift. “Aerie: flexible file-system interfaces to storage-class memory”. In: *Ninth Eurosys Conference 2014 (EuroSys), Amsterdam, The Netherlands, April 13-16, 2014*. 2014, 14:1–14:14 (cit. on pp. 60, 62).

[261] M. Vrable, S. Savage, and G. M. Voelker. “BlueSky: a cloud-backed file system for the enterprise”. In: *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), San Jose, CA, USA, February 14-17*. 2012, p. 19 (cit. on pp. 21, 112).

[262] S. R. Walli. “The POSIX family of standards”. In: *ACM Stand.* 3.1 (1995), pp. 11–17 (cit. on pp. 2, 16, 24).

[263] C. Wang, K. Mohror, and M. Snir. “File System Semantics Requirements of HPC Applications”. In: *HPDC ’21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021*. ACM, 2021, pp. 19–30 (cit. on pp. 16, 55, 67, 72, 78).

[264] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty. *File system workload analysis for large scale scientific computing applications*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2004 (cit. on p. 54).

[265] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. “An ephemeral burst-buffer file system for scientific applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. 2016, pp. 807–818 (cit. on pp. 3, 19, 55, 57, 59, 62).

[266] A. M. Weil and D. G. Feitelson. “Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling”. In: *IEEE Trans. Parallel Distributed Syst.* 12.6 (2001), pp. 529–543 (cit. on p. 96).

[267] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. “Ceph: A Scalable, High-Performance Distributed File System”. In: *7th Symposium on Operating Systems Design and Implementation (OSDI ’06), November 6-8, Seattle, WA, USA*. USENIX Association, 2006, pp. 307–320 (cit. on pp. 21, 26, 28, 30, 109, 112).

[268] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. “RADOS: a scalable, reliable storage service for petabyte-scale storage clusters”. In: *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW), November 11, Reno, Nevada, USA*. 2007, pp. 35–44 (cit. on pp. 20, 21, 108).

[269] B. Welch and G. A. Gibson. “Managing Scalability in Object Storage Systems for HPC Linux Clusters”. In: *21st IEEE Conference on Mass Storage Systems and Technologies / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, Maryland, USA, April 13-16*. 2004, pp. 433–445 (cit. on pp. 21, 109, 112).

[270] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. “Scalable Performance of the Panasas Parallel File System”. In: *6th USENIX Conference on File and Storage Technologies (FAST), February 26-29, San Jose, CA, USA*. 2008, pp. 17–33 (cit. on pp. 20, 115).

[271] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis. “Parallel File System Analysis Through Application I/O Tracing”. In: *Comput. J.* 56.2 (2013), pp. 141–155 (cit. on pp. 27, 28, 115).

[272] B. Xie, J. S. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. “Characterizing output bottlenecks in a supercomputer”. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, p. 8 (cit. on pp. 3, 97).

[273] J. Xing, J. Xiong, N. Sun, and J. Ma. “Adaptive and scalable metadata management to support a trillion files”. In: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. 2009 (cit. on pp. 19, 58).

[274] S. Yang, W. B. Ligon III, and E. C. Quarles. “Scalable distributed directory implementation on orange file system”. In: *Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI)* (2011) (cit. on pp. 19, 58).

[276] O. Yildiz, M. Dorier, S. Ibrahim, R. B. Ross, and G. Antoniu. “On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems”. In: *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 750–759 (cit. on p. 3).

[277] A. B. Yoo, M. A. Jette, and M. Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*. Vol. 2862. Lecture Notes in Computer Science. Springer, 2003, pp. 44–60 (cit. on p. 83).

[278] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer. “ImageNet Training in Minutes”. In: *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 1:1–1:10 (cit. on p. 79).

[279] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu. “High-Performance Storage Support for Scientific Applications on the Cloud”. In: *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud), Portland, Oregon, USA, June 16*. 2015, pp. 33–36 (cit. on p. 109).

[280] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. H. Carns, R. B. Ross, and I. Raicu. “FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems”. In: *2014 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, October 27-30, 2014*. 2014, pp. 61–70 (cit. on p. 61).

[281] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo. “Scaling embedded in-situ indexing with deltaFS”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC) , Dallas, TX, USA, November 11-16*. 2018 (cit. on pp. 59, 62).

[282] S. Zhou, H. D. Costa, and A. J. Smith. “A File System Tracing Package for Berkeley UNIX”. In: *Proceedings of the USENIX Summer Conference*. Portland, OR, 1984, pp. 407–419 (cit. on p. 27).

[283] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. “Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems”. In: *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, September 25-28, 2018*. 2018, pp. 145–156 (cit. on pp. 54, 79).

Webpages

[@9] *Aurora supercomputer*. 2021. URL: `http://aurora.alcf.anl.gov` (visited on Feb. 15, 2021) (cit. on p. 55).

[@12] J. Barr. *Amazon S3 Batch Operations*. 2019. URL: https://aws.amazon.com/blogs/aws/new-amazon-s3-batch-operations/ (visited on Feb. 15, 2021) (cit. on pp. 18, 20, 109, 111, 112).

[@14] *Beanstalk is a simple, fast work queue*. 2020. URL: https://github.com/beanstalkd/beanstalkd (visited on Mar. 25, 2021) (cit. on p. 117).

[@16] BeeGFS. *BeeOND: BeeGFS On Demand*. 2018. URL: https://www.beegfs.io/wiki/BeeOND (visited on Feb. 23, 2021) (cit. on p. 57).

[@17] *BeeGFS logging configurations*. 2017. URL: https://git.beegfs.com/pub/v6/blob/6.1/fhgfs_client_module/build/dist/etc/beegfs-client.conf/#L224 (visited on Mar. 10, 2021) (cit. on p. 28).

[@18] O. Ben-Kiki, C. Evans, and B. Ingerson. *YAML Ain't Markup Language Version 1.2, 3rd Edition*. 2009. URL: https://yaml.org/spec/1.2/spec.html (visited on Apr. 15, 2021) (cit. on p. 118).

[@26] R. Borris. *Customized developments for AUDI's OpenFOAM processes*. 2016. URL: https://www.esi-group.com/sites/default/files/resource/other/1898/abstract_keynote_audi_borris_customizeddevelopments.pdf (visited on July 9, 2022) (cit. on p. 99).

[@31] T. Brant. *SSD vs. HDD: What's the Difference?* 2022. URL: https://www.pcmag.com/news/ssd-vs-hdd-whats-the-difference (visited on Dec. 18, 2022) (cit. on p. 1).

[@50] J. Corbet. *BF tracing filters*. 2013. URL: https://lwn.net/Articles/575531/ (visited on Mar. 10, 2021) (cit. on p. 27).

[@51] *Cori supercomputer*. 2016. URL: https://docs.nersc.gov/systems/cori/ (visited on Feb. 15, 2021) (cit. on p. 55).

[@53] *Dardel supercomputer*. 2021. URL: https://www.pdc.kth.se/hpc-services/computing-systems/dardel-1.1043529 (visited on Dec. 16, 2022) (cit. on p. 77).

[@55] DDN. *INFINITE MEMORY ENGINE (IME)*. 2021. URL: https://www.ddn.com/products/ime-flash-native-data-cache/ (visited on Feb. 23, 2021) (cit. on p. 19).

[@56] M. Deck. *Building and Maintaining an Amazon S3 Metadata Index without Servers*. 2015. URL: https://aws.amazon.com/blogs/big-data/building-and-maintaining-an-amazon-s3-metadata-index-without-servers/ (visited on Mar. 23, 2021) (cit. on pp. 109, 113).

[@74] *Frontier supercomputer*. 2022. URL: https://www.olcf.ornl.gov/2021/05/20/olcf-announces-storage-specifications-for-frontier-exascale-system/ (visited on May 15, 2022) (cit. on p. 2).

[@75] *Fugaku supercomputer*. 2020. URL: https://www.fujitsu.com/global/about/innovation/fugaku/ (visited on Feb. 15, 2021) (cit. on p. 55).

[@76] *FUSE (Filesystem in Userspace)*. 2002. URL: https://github.com/libfuse/libfuse (visited on Feb. 4, 2021) (cit. on p. 59).

[@84] *Goofys*. 2019. URL: https://github.com/kahing/goofys (visited on Mar. 10, 2021) (cit. on pp. 112, 113).

[@85] Google. *Glog – C++ implementation of the Google logging module*. 2015. URL: https://github.com/google/glog (visited on Mar. 10, 2021) (cit. on p. 28).

[@86] Google. *Wildcard names*. 2020. URL: https://cloud.google.com/storage/docs/gsutil/addlhelp/WildcardNames (visited on Mar. 10, 2021) (cit. on pp. 109, 112).

[@93] R. Hat. *Ceph Logging and Debugging*. 2016. URL: https://docs.ceph.com/en/nautilus/rados/troubleshooting/log-and-debug/ (visited on Mar. 9, 2021) (cit. on pp. 26, 28, 30).

[@102] T. Huuva and S. Törnros. *CFD and OpenFOAM at Caterpillar with a Main Focus on Marine Applications*. 2016. URL: https://www.esi-group.com/sites/default/files/resource/other/1900/abstract_keynote_caterpillar_huuva_cfd_and_openfoam_at_caterpillar_with_a_main_focus_on_marine_applications.pdf (visited on July 9, 2022) (cit. on p. 99).

[@103] IBM. *Burst Buffer Shared Checkpoint File System*. 2018. URL: https://cast.readthedocs.io/en/cast_1.7.x/burst-buffer/bbapi.html (visited on Apr. 4, 2022) (cit. on p. 79).

[@105] *IO500*. 2022. URL: https://io500.org/ (visited on May 5, 2022) (cit. on pp. 12, 105).

[@106] *IOR and MDTest*. 2020. URL: https://github.com/hpc/ior (visited on Mar. 1, 2021) (cit. on pp. 37, 45, 84, 88, 105).

[@107] J. Jacobi. *NVMe SSDs: Everything you need to know about this insanely fast storage*. 2019. URL: https://www.pcworld.com/article/432532/everything-you-need-to-know-about-nvme.html (visited on Dec. 18, 2022) (cit. on p. 1).

[@111] *Kernel Summit 2011 Summary*. 2016. URL: http://lwn.net/Articles/464268/ (visited on Mar. 10, 2021) (cit. on pp. 27, 28).

[@117] A. N. Laboratory. *The Mochi Project*. 2021. URL: https://www.mcs.anl.gov/research/projects/mochi/ (visited on Feb. 15, 2021) (cit. on p. 69).

[@118] L. L. N. Laboratory. *GOTCHA*. 2018. URL: https://github.com/LLNL/GOTCHA (visited on Mar. 10, 2021) (cit. on p. 63).

[@119] L. L. N. Laboratory. *UnifyFS*. 2019. URL: https://github.com/LLNL/UnifyFS (visited on Mar. 15, 2021) (cit. on pp. 16, 57, 61).

[@120] L. Lancaster. *Intel Optane is high-octane memory*. 2017. URL: https://www.cnet.com/tech/computing/intel-optane-is-high-octane-memory/ (visited on Dec. 18, 2022) (cit. on p. 1).

[@131] Linux. *iostat(1) - Linux man page*. URL: https://linux.die.net/man/1/iostat (visited on Mar. 17, 2021) (cit. on p. 46).

[@132] *Linux manual page: write(2)*. 2019. URL: https://www.man7.org/linux/man-pages/man2/write.2.html (visited on Feb. 15, 2021) (cit. on p. 16).

[@133] *Linux perf tools*. 2020. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on Mar. 9, 2021) (cit. on pp. 27, 46).

[@135] *lmbench*. 2013. URL: http://lmbench.sourceforge.net/ (visited on Mar. 15, 2021) (cit. on p. 45).

[@140] X. Lucas. *Ext4 file system*. 2020. URL: https://www.kernel.org/doc/Documentation/filesystems/ext4.txt (visited on Feb. 4, 2021) (cit. on pp. 2, 12, 59, 72).

[@143] *Lustre Diagnostic and Debugging Tools*. 2021. URL: http://wiki.lustre.org/index.php/Diagnostic_and_Debugging_Tools (visited on Feb. 17, 2021) (cit. on pp. 26, 28, 30).

[@146] *MadFS*. 2020. URL: https://github.com/rustcc/RustChinaConf2020/blob/master/rustchinaconf2020/RustChinaConf2020-24.%E7%8E%8B%E6%B6%A6%E5%9F%BA-%E3%80%8ArCore%EF%BC%9ARust%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8%E7%9A%84%E6%8E%A2%E7%B4%A2+MadFS%EF%BC%9A%E5%B0%8F%E5%B7%A7%E7%B2%BE%E6%82%8D%E7%9A%84%E5%88%86%E5%B8%83%E5%BC%8F%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F%E3%80%8B.pdf (visited on Jan. 9, 2022) (cit. on p. 105).

[@147] *MareNostrum 4 supercomputer*. 2017. URL: https://www.bsc.es/marenostrum/marenostrum (visited on Feb. 15, 2021) (cit. on pp. 55, 80).

[@157] *Mogon II supercomputer*. 2017. URL: https://hpc-en.uni-mainz.de/high-performance-computing/systeme/ (visited on Feb. 15, 2021) (cit. on pp. 55, 80).

[@163] *National Center For Biotechnology Information: Sequence Read Archive*. 2020. URL: https://www.ncbi.nlm.nih.gov/sra (visited on Feb. 15, 2021) (cit. on p. 134).

[@164] *NEK5000 – fast high-order scalable computational fluid dynamics*. 2022. URL: https://nek5000.mcs.anl.gov/ (visited on May 15, 2021) (cit. on p. 109).

[@165] *NEXTGenIO prototype*. 2017. URL: http://www.nextgenio.eu/ (visited on Feb. 15, 2021) (cit. on p. 80).

[@171] *NVIDIA Collective Communication Library*. 2022. URL: https://developer.nvidia.com/nccl (visited on Apr. 1, 2022) (cit. on pp. 82, 100).

[@172] *NVIDIA CUDA*. 2022. URL: https://developer.nvidia.com/cuda-toolkit (visited on Apr. 1, 2022) (cit. on p. 82).

[@173] *NVIDIA DGX SYSTEMS*. 2022. URL: https://www.nvidia.com/en-us/data-center/dgx-systems/ (visited on Apr. 1, 2022) (cit. on p. 79).

[@177] *OpenIO FS Architecture*. 2020. URL: https://docs.openio.io/latest/source/arch-design/fs_overview.html#label-oiofs-architecture (visited on Mar. 9, 2021) (cit. on pp. 21, 112, 113).

[@179] S. Oral and G. Shah. *Spectrum Scale Enhancements for CORAL. Presentation slides at Supercomputing'16.* 2016. URL: http://files.gpfsug.org/presentations/2016/SC16/11_Sarp_Oral_Gautam_Shah_Spectrum_Scale_Enhancements_for_CORAL_v2.pdf (visited on Feb. 23, 2021) (cit. on pp. 18, 19, 38).

[@187] *Persistent Memory Development Kit*. 2018. URL: https://pmem.io/pmdk/ (visited on Mar. 9, 2021) (cit. on p. 62).

[@190] T. K. Petersen. *Inside the Lustre File System*. Seagate. 2014. URL: http://www.seagate.com/files/www-content/solutions-content/cloud-systems-and-solutions/high-performance-computing/_shared/docs/clusterstor-inside-the-lustre-file-system-ti.pdf (visited on Feb. 15, 2021) (cit. on p. 15).

[@193] T. N. Platform. *What's So Bad About POSIX I/O?* 2017. URL: https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/ (visited on July 9, 2022) (cit. on p. 16).

[@195] POSIX. *POSIX*. 2001. URL: https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/stat.h.html (visited on Feb. 15, 2022) (cit. on p. 5).

[@204] P. S. C. news release. *PSC Wins a Record Five HPCwire Readers', Editors' Choice Awards*. 2017. URL: https://www.cmu.edu/mcs/news-events/2017/1114-psc-hpcwire-awards.html (visited on July 9, 2022) (cit. on p. 99).

[@206] *RioFS*. 2018. URL: https://github.com/skoobe/riofs (visited on Mar. 9, 2021) (cit. on pp. 112, 113).

[@214] *S3FS-FUSE*. 2019. URL: https://github.com/s3fs-fuse/s3fs-fuse (visited on Mar. 9, 2021) (cit. on pp. 21, 109, 111–113).

[@215] *S3QL – a full-featured file system for online data storage*. 2019. URL: https://github.com/s3ql/s3ql (visited on Mar. 9, 2021) (cit. on pp. 112, 113).

[@216] Y. Sadeh. *New in Luminous: RGW Metadata Search*. 2020. URL: https://ceph.io/rgw/new-luminous-rgw-metadata-search/ (visited on Apr. 9, 2021) (cit. on pp. 109, 113).

[@224] A. Shilov. *SSDs Outsell HDDs in Unit Sales 3:2: 99 Million Vs. 64 Million in Q1*. 2021. URL: https://www.tomshardware.com/news/ssd-market-shares-q1-2021-trendfocus (visited on May 3, 2022) (cit. on p. 1).

[@226] *Sierra supercomputer*. 2018. URL: https://computation.llnl.gov/computers/sierra (visited on Feb. 15, 2021) (cit. on p. 55).

[@230] *strace: Linux syscall tracer*. 2021. URL: https://strace.io/ (visited on Mar. 9, 2021) (cit. on pp. 27, 28).

[@232] *Summit supercomputer*. 2018. URL: https://www.olcf.ornl.gov/summit/ (visited on Feb. 15, 2021) (cit. on p. 55).

[@236] *SVFS*. 2019. URL: https://github.com/ovh/svfs (visited on Mar. 9, 2021) (cit. on pp. 112, 113).

[@237] *Syscall intercept*. 2018. URL: https://github.com/pmem/syscall_intercept (visited on Mar. 9, 2021) (cit. on p. 62).

[@238] *Sysdig*. 2021. URL: https://sysdig.com/ (visited on Mar. 9, 2021) (cit. on p. 27).

[@247] *Top500*. 2022. URL: https://www.top500.org/ (visited on May 5, 2022) (cit. on pp. 2, 12).

[@249] *Using the Linux Kernel Tracepoints*. 2016. URL: https://www.kernel.org/doc/Documentation/trace/tracepoints.txt (visited on Mar. 8, 2021) (cit. on pp. 26–28).

[@275] *YAS3FS: Yet Another S3-backed File System*. 2019. URL: https://github.com/danilop/yas3fs (visited on Feb. 15, 2021) (cit. on p. 113).