Vef MA, Moti N, Süß T *et al.* GekkoFS — A temporary burst buffer file system for HPC applications. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 72–91 Jan. 2020. DOI 10.1007/s11390-020-9797-6

GekkoFS — A Temporary Burst Buffer File System for HPC Applications

Marc-André Vef[1], Nafiseh Moti[1], Tim Süß[1], Markus Tacke[1], Tommaso Tocci[2], Ramon Nou[2], Alberto Miranda[2] Toni Cortes[2,3], and André Brinkmann[1], *Member, ACM*

[1] *Zentrum für Datenverarbeitung, Johannes Gutenberg University Mainz, Mainz 55128, Germany*

[2] *Barcelona Supercomputing Center, Barcelona 08034, Spain*

[3] *Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona 08034, Spain*

E-mail: {vef, moti, suesst, tacke}@uni-mainz.de
{tommaso.tocci, ramon.nou, alberto.miranda, toni.cortes}@bsc.es; brinkman@uni-mainz.de

Received June 30, 2019; revised October 3, 2019.

Abstract Many scientific fields increasingly use high-performance computing (HPC) to process and analyze massive amounts of experimental data while storage systems in today's HPC environments have to cope with new access patterns. These patterns include many metadata operations, small I/O requests, or randomized file I/O, while general-purpose parallel file systems have been optimized for sequential shared access to large files. Burst buffer file systems create a separate file system that applications can use to store temporary data. They aggregate node-local storage available within the compute nodes or use dedicated SSD clusters and offer a peak bandwidth higher than that of the backend parallel file system without interfering with it. However, burst buffer file systems typically offer many features that a scientific application, running in isolation for a limited amount of time, does not require. We present GekkoFS, a temporary, highly-scalable file system which has been specifically optimized for the aforementioned use cases. GekkoFS provides relaxed POSIX semantics which only offers features which are actually required by most (not all) applications. GekkoFS is, therefore, able to 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.

Keywords distributed file system, high-performance computing (HPC), burst buffer, POSIX (portable operating system interface)

1 Introduction

High-performance computing (HPC) applications are currently significantly changing. Traditional HPC applications have been compute-bound, large-scale simulations, while today's HPC community is additionally moving towards the generation, processing, and analysis of massive amounts of experimental data. This trend, known as data-driven science, is affecting many different scientific fields, some of which have made significant progress tackling previously unaddressable challenges thanks to newly developed techniques[1, 2].

Most data-driven workloads are based on new algorithms and data structures like graph databases, which impose new requirements on HPC file systems[3, 4]. Among others, they include large numbers of metadata operations, data synchronization, non-contiguous and random access patterns, and small I/O requests[3, 5]. These access patterns differ significantly from past workloads which mostly performed sequential I/O operations on large files. These new access patterns do not only slow down the data-driven applications themselves but can also heavily disrupt other applications that are

Regular Paper

Special Section on Selected I/O Technologies for High-Performance Computing and Data Analytics

This work has been funded by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" and the ADA-FS project, and also partially supported by the Spanish Ministry of Science and Innovation under Grant No. TIN2015–65316, the Generalitat de Catalunya under Contract 2014–SGR–1051, as well as the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 671951 (NEXTGenIO).

concurrently accessing the shared storage system [6, 7]. Consequently, conventional parallel file systems (PFS) cannot handle these workloads efficiently and data-driven applications suffer from prolonged I/O latencies, reduced throughput, and long waiting times.

Software-based approaches to supporting data-driven applications try to align the new access patterns to the capabilities of the underlying PFS. These approaches include application modifications or use middleware and high-level libraries (e.g., ADIOS [8], or HDF5 [9]). Adapting the software is typically time-consuming, difficult to couple with big data and machine learning libraries, or sometimes (based on the underlying algorithms) just impossible.

Hardware-based approaches move from magnetic disks, which are still the main backend technology for PFSs, to NAND-based solid-state drives (SSDs). Nowadays, many supercomputers deploy SSDs, which provide a high sequential and random access performance[1]–[6]. SSDs can be used as dedicated burst buffers [10] or as *node-local* burst buffers. To achieve high metadata performance, burst buffers can be deployed in combination with a dynamic *burst buffer file system* [11, 12].

Generally, burst buffer file systems increase performance compared to a PFS without modifying an application. Therefore, they typically support POSIX (portable operating system interface) which provides the standard semantics accepted by most application developers. Nevertheless, enforcing POSIX can severely reduce a PFS's peak performance [13]. Further, many POSIX features are not required for most scientific applications [14], especially if they can exclusively access the file system. Similar argumentations hold for other advanced features like fault tolerance or security.

We present *GekkoFS*, a temporarily deployed, highly-scalable distributed file system for HPC applications. GekkoFS pools together fast node-local storage resources and provides a global namespace accessible by all participating nodes. It relaxes POSIX by removing some of the semantics that most impair I/O performance in a distributed context, and it takes previous studies on the behavior of HPC applications into account [14] to heavily optimize the most used file system operations.

For load-balancing, all data and metadata are distributed across all nodes using the HPC RPC framework Mercury [15] in combination with the HPC threading framework *Argobots* [16, 17]. The file system runs in user-space and can be easily deployed in under 20 seconds on a 512 node cluster by any user. It, therefore, can be used in a number of temporary scenarios, e.g., during the lifetime of a compute job or in longer-term use cases, such as campaigns in which data is simultaneously accessed by many nodes in short bursts.

Besides being able to offer the combined storage capabilities of node-local storage devices, GekkoFS' goal is to accelerate I/O operations in common HPC workloads that are challenging for modern PFSs. We demonstrate how our lightweight, yet highly distributed file system can achieve scalable data and metadata performance, achieving tens of millions of metadata operations per second on a 512 node cluster. At the same time, GekkoFS is able to run complex applications, such as OpenFOAM solvers [18]. These features are achieved while GekkoFS operates synchronously and with a strong consistency model for file system operations that target a specific file or directory.

In this work, we built on the conference paper by Vef *et al.* [19] and extended it by a more detailed description of its architecture and core techniques in Section 3. Section 4 offers a deeper analysis of additional experiments with an in-depth evaluation of the file system's metadata and data performance on the MOGON II supercomputer. In Subsection 4.4, we evaluate GekkoFS' I/O variability on the MareNostrum IV supercomputer and compare it with the capabilities of its GPFS installation. Further, Subsection 4.5 investigates GekkoFS' and Lustre's effects on the network communication when the user application OpenFOAM [18] is run. Finally, we conclude in Section 5 and discuss further research directions.

2 Related Work

In this section, we give an overview over existing HPC file systems and discuss the differences to

[1] Aurora. http://aurora.alcf.anl.gov, Nov. 2019.

[2] Cori. http://www.nersc.gov/users/computational-systems/cori, Nov. 2019.

[3] Marenostrum. https://www.bsc.es/marenostrum/marenostrum, Nov. 2019.

[4] Mogon II. https://hpc.uni-mainz.de, Nov. 2019.

[5] Sierra. https://computation.llnl.gov/computers/sierra, Nov. 2019.

[6] Summit. https://www.olcf.ornl.gov, Nov. 2019.

GekkoFS.

General-Purpose Parallel File Systems. Most HPC systems are equipped with a backend storage system which is globally accessible using a parallel file system (e.g., GPFS [20], Lustre [21, 22], BeeGFS [23], or PVFS [24]). These file systems offer a POSIX-like interface, which allows applications to run as they were accessing a local file system, focusing on data consistency and long-term storage. However, due to the nature of the file system being globally accessible, single applications can disrupt the I/O performance of other applications as well. In addition, these file systems are not well suited for small file accesses, in particular on shared files, which can be found in scientific applications [3].

GekkoFS' design does not focus on long-term storage. Contrary to previously mentioned and permanently available file systems, it offers a separate namespace only accessible to nodes participating within the context of an HPC job or other temporary defined groups. When such a context is finished, all data is deleted. Further, GekkoFS offers a relaxed POSIX environment. As such, our file system is able to provide a significant increase in metadata performance and reduces the impact on other applications running on the same HPC system.

Burst Buffers. Burst buffers are fast, intermediate storage systems that aim to reduce the load on the global file system and on reducing an applications' I/O overhead [10]. Essentially, burst buffers can be categorized into two groups [11]: remote-shared and node-local. Remote-shared burst buffers, e.g., DDN's IME[7] and Cray's DataWarp[8], are centralized, dedicated I/O nodes structured as a forwarding layer.

Node-local burst buffers are generally collocated with the compute nodes and can be dependent on the PFS (e.g., PLFS [12]). In some cases, these burst buffers can also be managed directly by the PFS itself [25]. The Hermes [26] I/O middleware library provides a distributed I/O buffering system which transparently combines multi-tiered storage (e.g., node-local SSDs) and memory hierarchies of supercomputer environments. It uses multiple data placement techniques to place data on all storage layers efficiently and therefore considers both local and shared resources as burst buffers.

BurstFS [11], perhaps the most related work to ours, is a standalone burst buffer file system which does not require a centralized instance as well. However, GekkoFS is not limited to write data locally like BurstFS. Instead, all data is distributed across all participating file system nodes to balance data workloads for write and read operations without sacrificing scalability. BeeOND [23] can create a job-temporal file system on a number of nodes similar to GekkoFS. BeeOND is, in contrast to our file system, POSIX-compliant and our GekkoFS' measurements show a much higher metadata throughput than what is offered by BeeOND (see Subsection 4.2).

POSIX. The term portable operating system interface[9] (POSIX) is rather broad and at times ambiguous. When used within a file system context, POSIX is typically referred to as *POSIX I/O*, targeting the behavior of write and read operations. Henceforth, POSIX refers in this paper to POSIX I/O. POSIX was developed over 25 years ago for file accesses by a single process. Yet, until today there is no support for parallel I/O which involves multiple I/O operations on a single shared file.

To provide POSIX semantics in a distributed environment with a strong consistency model and shared accesses, parallel file systems typically rely on expensive distributed locking mechanisms to avoid conflicts, such as byte-range locking (e.g., Lustre [21] and GPFS [20]). However, strong consistency semantics and poor parallel I/O support in file systems [27] combined with scientific applications' access patterns [3] are the main reasons for HPC I/O challenges. Despite ever-increasing computing performance, data-intensive applications running on HPC systems face scalability challenges which prevents them to fully utilize the offered computing power [28].

One solution to reduce locking induced overhead is to give the responsibility that no shared conflicts occur to the application or external libraries. PVFS, e.g., does not implement a file system locking mechanism [24, 29]. GekkoFS supports this argumentation and does not provide a locking mechanism to avoid any global locking overhead.

Metadata Scalability. At its core, metadata in Unix-like operating systems can be categorized into three components: an object's metadata, a file's data, and a directory's contents [30]. Metadata is typically stored in an *inode*, containing information about the object's

[7] ddn.com/products/ime-flash-native-data-cache, Nov. 2019.

[8] https://www.cray.com/datawarp, Nov. 2019.

[9] http://pubs.opengroup.org/onlinepubs/9699919799/, Nov. 2019.

type (e.g., file, directory, symbolic links), its size, access permissions, and last modification time (`mtime`), for example. For data organization, file systems offer directories that store information about their contents in directory blocks.

However, inodes and directory blocks were not designed for parallel accesses because a single block can only be accessed by one process at a time. This is particularly relevant in distributed systems when a huge number of files are created in a single directory from multiple processes, a common workload in HPC environments [12, 25, 31, 32]. In general, such systems distribute data across all available storage targets. While this technique works well for data, it does not achieve the same throughput when handling metadata [33, 34]. This is caused by complex distributed locking of central data structures (generally managed by a metadata server instance) that are required to be accessed in parallel [31]. The file system community presented various techniques for handling metadata [12, 32, 35–38], but this challenge is still prevailing and is becoming an even bigger challenge for upcoming data-science applications.

IndexFS [33], for example, is one such attempt to drastically improve metadata performance by using it as a middleware software on top of existing file systems, such as PVFS or Lustre. Similar to GekkoFS, IndexFS uses a key-value database to store metadata information and it distributes metadata across multiple IndexFS servers. In addition, IndexFS uses various client caches to increase RPC efficiency further and assigns directories to IndexFS servers. GekkoFS, on the other hand, uses loosely coupled components and does not offer any caching functionality on clients. Instead, GekkoFS aggressively stripes all metadata across all servers. Moreover, in contrast to GekkoFS, IndexFS is still bound to implement POSIX file I/O semantics, requiring IndexFS to work with the file system protocols of the used underlying file system, e.g., pathname traversal or permission checking. GekkoFS relaxes POSIX and weakens the concept of directories which internally are no longer treated as mentioned directory blocks, directory entries, and inodes (see Subsection 3.5). In fact, GekkoFS entirely removes directory blocks and replaces directory entries by objects, stored within a strongly consistent key-value database which relinquishes the need for locking mechanisms within the file system. As a result, GekkoFS achieves tens of millions of metadata operations for 512 nodes and billions of files.

3 Design and Implementation

In this section we introduce the goals and the design of GekkoFS. First, we present the goals of our file system. Second, we give a brief overview of the system components and the file system's architecture. Finally, we present details to each component of our system.

3.1 Design Goals

We define the following goals for GekkoFS' design.

Functionality. Any user should be able to deploy the file system on an arbitrary number of nodes without administrative assistance. The mount point of GekkoFS and the data directory, which the file system uses to store user data, is given when the file system is started. The mount point should then present the user with a single global namespace, consisting of the aggregated node-local storage of each node.

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

Consistency Model. GekkoFS should provide the same consistency as POSIX for any file system operation that accesses a specific data file. This includes read and write operations as well as any metadata operations that target a single file, e.g., file creation. Nevertheless, the consistency of directory operations, for instance, 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' startup should be finished within one minute to be used immediately by applications once the startup succeeds.

Hardware Independence. GekkoFS should be able to utilize all networking hardware that is commonly used in HPC clusters, such as Infiniband or Omni-Path, and fully support the native protocols of these fabrics to efficiently move the data between file system nodes. In addition, 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 path permitted to the user.

3.2 Overview

GekkoFS aims to offer a user-space file system for the lifetime of a particular use case, e.g., within the context of an HPC job. The file system uses the available local storage of compute nodes and combines their node-local storage into a single global namespace. For the scalability and balanced usage of all available storage, data and metadata are distributed across all file system nodes.

Before an application is started, a file system server process is launched on each node with information about the file system mount point, the location where file system data is stored, and a list of participating nodes. An application uses the file system by *preloading* the GekkoFS client interposition library that intercepts file system operations on the GekkoFS mount point. The client library can also be used to stage-in or stage-out data into GekkoFS from the PFS or vice versa, if necessary. For example, to copy data from a PFS path to a GekkoFS path, a user may use the `cp` command on the *command line interface* (CLI) while the interposition library is preloaded. When a GekkoFS operation is intercepted, the client forwards the operation to the responsible server, determined by hashing the file's path, where it is directly executed. To achieve a balanced data distribution for large files, each file is additionally split into equally sized chunks by the client and then distributed among the servers. Due to this communication scheme, there is no communication between server instances.

3.3 POSIX Relaxation

From definitions on how I/O interfaces work, come certain requirements and expectations on how the file system retrieves results. The POSIX model inherently leads to a consistency model, which requires atomicity and locking mechanisms in a distributed environment. While local file systems are able to efficiently provide such a consistency model, it can result in scalability issues in a PFS (see Section 2).

The POSIX consistency is especially challenging for the scalability of a PFS in the following cases.

1) *Atomic operations within a distributed environment that require exclusive write or read access to a central data structure.* Such atomicity is typically achieved by acquiring a global lock on the desired data structure, which can impair concurrent work. Examples of such operations are file `create()`, since it involves modifying the parent directory, and `readdir()` as it needs to create a snapshot of a directory's current state and may target an arbitrary number of file system objects.

2) *Cache coherency protocols.* While there are several advantages to provide various forms of caching in a PFS, 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 body of network communication to keep synchronization, which is commonly impracticable at larger scales.

Therefore, similar to PVFS[39] and OrangeFS[29], GekkoFS does not provide a global lock mechanism. In this sense, applications should be responsible to ensure that no conflicts occur, in particular, w.r.t. overlapping file regions, to avoid complex locking within the file system. However, the lack of distributed locking has consequences for operations where the number of affected file system objects is unknown a priori, e.g., when requesting the contents of a directory. In these *indirect file system operations*, GekkoFS does not guarantee to return the current state of the directory. In other words, `readdir()` operations, which are called by the `ls -l` or `rm -rf /*` commands, for instance, follow an eventual-consistency model. Also for the above-stated reasons, each operation in GekkoFS is synchronous without any form of caching within the context of file system operations. This not only reduces file system complexity but allows for an evaluation of GekkoFS' raw performance capabilities.

Moreover, studies on the behavior of HPC applications have shown that many features that file systems offer are rarely used or not used at all, such as move/rename operations[14]. Instead, these operations are typically only called from a console after the actual simulation ends and when parallel access is not required anymore. Taking these observations into account, GekkoFS' does not optimize towards move or rename operations and linking functionality although supporting it rudimentarily.

Finally, security management in the form of access permissions is not maintained by GekkoFS as it already implicitly follows the security protocols of the node-local file system that is used to store the file system's data and metadata. Data can only be staged into or out of GekkoFS if the user has the corresponding access rights on the backend file system.

3.4 Architecture

GekkoFS' architecture (see Fig.1) consists of two main components: a client library and a server process. An application that uses GekkoFS must first preload the client interposition library through the `LD_PRELOAD` environment variable which intercepts file system operations. However, because any function can be intercepted by the preloading library, all used I/O functions

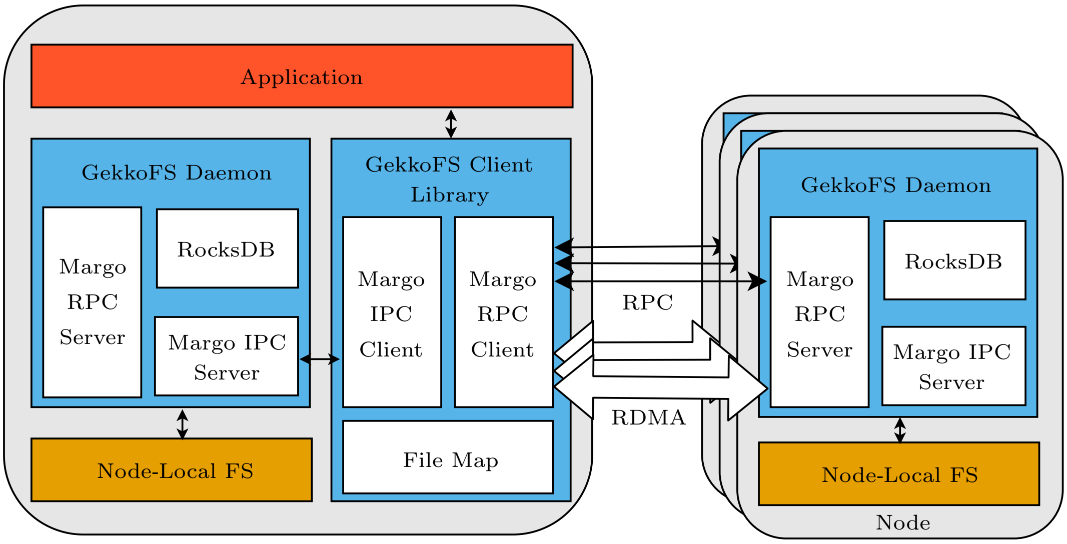


Fig.1. GekkoFS architecture with its components.

by an application must be reinterpreted by GekkoFS for the application to work. Therefore, the more complex an application, the more the functions need to be intercepted, potentially reimplementing a large percentage of the *GNU C Library* (glibc).

To limit the number of functions that need to be intercepted, GekkoFS uses the *system call intercepting library*[10] (syscall_intercept) which aims to solve this challenge by providing a low-level interface for hooking Linux system calls while still using the `LD_PRELOAD` method. As a result, the GekkoFS client only needs to intercept system calls, such as `sys_mknod` or `sys_write`, while leaving the functionality of glibc, for instance, untouched.

Once the client has intercepted an I/O call, it forwards the I/O call to a server (*GekkoFS daemon*), if required. The GekkoFS daemon, which runs on each file system node, receives forwarded file system operations from clients and processes them, sending a response when finished. The daemons operate independently and do not communicate with other server processes on remote nodes, therefore being effectively unaware from each other. In the following paragraphs, we describe the client and the daemon in detail.

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

When the client library is first invoked by an application, it requests basic information about the mount point and the participating nodes from the local GekkoFS daemon. The interception library then checks for each file system operation if the GekkoFS mount point is used. If this is not the case, the call is passed through to the underlying file system. Whenever a file is opened, a new file map entry is created which associates the file handle with its path, among other information. Upon closing the file this entry is removed.

While a file is open, the client uses the file path p for each related file system operation 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:

$$nodeID = h(p) \pmod{\text{number of GekkoFS nodes}}.$$

The corresponding operation is then forwarded via an RPC message to the daemon with the unique *nodeID* where it is directly executed. In other words, GekkoFS uses a pseudo-random distribution to spread data and metadata across all nodes, also known as *wide-striping*. Because each client is able to independently resolve the responsible node for a file system operation, GekkoFS does not require central data structures that keep track of where metadata or data is located.

[10] https://github.com/pmem/syscall_intercept, Nov. 2019.

In addition, to achieve a balanced data distribution for large files, data requests are split into equally sized chunks before they are distributed across file system nodes. During data transfers between client and GekkoFS daemons, the client exposes the relevant chunk memory region to the daemon which accesses it via *remote-direct-memory-access* (RDMA), if supported by the underlying network fabric protocol.

GekkoFS Daemon. A GekkoFS daemon's purpose is to process forwarded file system operations of clients to store and retrieve data and metadata that hashes to a daemon. 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.

Each daemon operates a single local RocksDB key-value store which provides a high-performance embedded database for key-value data, based on a log-structured merge-tree (LSM)[40]. RocksDB is optimized for NAND storage technologies with low latencies and thus fits GekkoFS' needs as SSDs are primarily used as node-local storage in today's HPC clusters.

For the communication layer, we leverage on the *Mercury* RPC framework, which allows GekkoFS to be network-independent, achieving one of our design goals (see Subsection 3.1). Mercury is an RPC communication library developed by the Argonne National Laboratories (ANL), which focuses on HPC environments[15]. In contrast to other RPC frameworks, Mercury is able to use the native network transport layer and can, therefore, handle large data transfers efficiently. Mercury's *network abstraction layer*, which provides a high-level interface on top of the lower-level network fabrics, offers a wide variety of plugins to natively support common fabric protocols, e.g., InfiniBand or Omni-Path. When available, large data transfers are processed via RDMA or *cross-memory attach* (CMA) in remote and local communications, respectively. This allows GekkoFS to efficiently transfer data within the file system.

Within a GekkoFS, Mercury is interfaced indirectly through the *Margo* library[11] which provides *Argobots-aware* wrappers to Mercury's API with the goal to provide a simple multi-threaded execution model[17]. Argobots is a lightweight low-level threading and tasking framework, developed to support massive on-node concurrency in modern HPC environments[16]. Using Margo allows GekkoFS daemons to minimize the resource consumption of Margo's progress threads and handlers that accept and handle RPC requests[17].

3.5 Rethinking Metadata Management

Modern distributed storage systems typically employ several strategies to distribute metadata and data across all available storage targets[20–23]. As described in Section 2 this technique usually works well for data but does not achieve the same efficiency and performance (throughput or operations per second) when handling metadata due to expensive distributed locking mechanisms. Based on these observations, we aim at forming a file system that performs well for any type of direct metadata operations and allows them to scale to an arbitrary number of nodes. We present our methodology in two steps targeting metadata management and its contents.

Decoupled Wide-Striping. We aim to achieve metadata scalability by decoupling directory contents from directory blocks, allowing GekkoFS to operate without a global lock manager, and by wide-striping metadata across all file system nodes. Instead of using directory blocks which are challenging to use in distributed environments, each directory entry is stored in a daemon's 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 as the children of a directory. Finally, by using the previously described distribution algorithm (see Subsection 3.4), metadata entries are striped across all nodes.

However, using the path of a file system object as an index within a flat namespace comes at a price. If a directory is, e.g., 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, as metadata might need to be migrated if the hashes for the new paths lead to different file system nodes. Nonetheless, as described in Subsection 3.3, GekkoFS explicitly disallows such operations, since they are rarely used in HPC applications.

Metadata Contents. A file system that is only temporarily accessed during a specific use case, e.g., within the context of an HPC job, may not require exposing

[11] https://xgitlab.cels.anl.gov/sds/margo, Dec. 2019.

all metadata fields to an application. First, we categorize metadata fields into three categories: redundant, rarely used, and mandatory. The first category includes permission bits, user ID, and group ID information, which are used for security purposes, but that we consider as redundant as GekkoFS has to follow the node-local file system's security protocols (see Subsection 3.3). The second category contains metadata fields which are *rarely used* by HPC applications, including timestamps, the inode number, and block size information. The metadata fields of the first two categories are disabled by default to save space within the KV store and to reduce the amount of metadata that is sent over the network. However, they can still be enabled if needed.

The third category contains *mandatory* metadata fields which cannot be turned off: the file type and the file size. Although we do not use traditional directories which store all entries in a directory block, we still support 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 keep track of a file's data boundaries. This is particularly of interest when working with a *sparse file* in which file systems generally do not write regions to disk that contain no information. In other words, the file size is used to differentiate between an application accessing outside a file's dimensions or within a sparse region of the file.

3.6 Data Management

GekkoFS manages data by utilizing a compute node's local storage system. In the following paragraphs, we describe GekkoFS' I/O protocol and explain how shared file access is managed in an environment where global locks are not used.

I/O Protocol. Similar to metadata, data (split into equally-sized chunks) is evenly distributed across all file system nodes. Within a node-local file system, each chunk is represented as a file and is named after the chunk's numeric identifier, which also describes the chunk's data offset. For example, in a 1 MiB chunk-sized file system, the chunk 2 references a file's offset starting at address 2 MiB. Chunks of the same file in GekkoFS are stored within the scope of the same directory in the node-local file system.

Fig.2 shows a write operation from the clients point of view. In the given case, the write buffer is split into six chunks (dependent on the chunk size). For each chunk, GekkoFS computes the target node with the help of the file's path and chunk identifier, grouping chunks that target the same node. The client then sends an RPC message to each target daemon node, each 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 CMA, which allows accessing a set memory region of another process on the same machine without copying it. In the event of serving multiple chunks per daemon, data transfer and disk I/O may be done in parallel. The read operation works similarly but in reverse. As explained in Subsection 3.3, all I/O operations are synchronous without any caching mechanisms on clients or daemons, while the caching mechanisms of the local file systems are used.

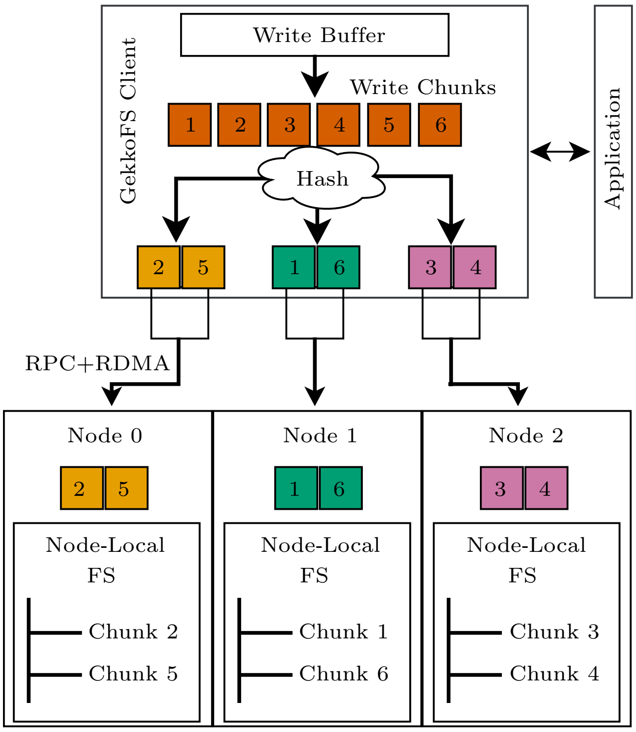


Fig.2. GekkoFS' write operation where a write buffer is split into six chunks and then distributed among three daemons. Each daemon stores its chunk in a node-local file system.

Shared Write Conflicts. As shown in Subsection 3.3, GekkoFS does not implement a global locking manager. This can impose challenges when working with shared files. For instance, when two or more processes write to the same file region at the same time they could cause a shared write conflict, resulting in an undefined behavior with regard to the data which is written to the underlying storage device. However, because of GekkoFS' decoupled design, the above-described locking conflicts can, in fact, be handled by any file system daemon locally. This is owing to the use of a POSIX-compliant node-local file system which stores the corresponding data chunks and, in the case of a shared write conflict, serializes access to the same chunk file. Moreover, because a file may be distributed across many chunk files and nodes, multiple conflicts in the same file only affect one chunk at a time. Hence, other chunks of the file are not disrupted during such a potential shared write conflict.

4 Evaluation

In this section, we evaluate the performance of GekkoFS based on various microbenchmarks which catch access patterns that are common in HPC applications. First, we describe the experimental setup and introduce the workloads that we simulate with microbenchmark applications. Then, we investigate GekkoFS' startup time and compare GekkoFS' metadata performance against a Lustre parallel file system. Although GekkoFS and Lustre have different goals, we point out the performances that can be gained by using GekkoFS as a burst buffer file system. In addition, we compare GekkoFS' metadata performance with BeeGFS' BeeOND burst buffer file system which is used similarly as GekkoFS. Then, we evaluate GekkoFS' data performance and compare them with BeeGFS' BeeOND and discuss the measured results. Further, we investigate GekkoFS' I/O variability and worst-case performance. Finally, we explore GekkoFS' with Lustre's effects on the network when the OpenFOAM application is used.

4.1 Experimental Setup

We use three supercomputers in our experiments: MOGON II at the Johannes Gutenberg University Mainz in Germany[12], MareNostrum IV at the Barcelona Supercomputing Center in Spain[13], and the NEXTGenIO prototype[14].

4.1.1 MOGON II

MOGON II consists of 1 876 nodes in total, with 822 nodes using Intel 2630v4 Intel® Broadwell processors (two sockets each) and 1 046 nodes using Xeon® Gold 6130 Intel Skylake processors (four sockets each). If not otherwise noted, Intel® Broadwell processors are used in all presented experiments. The node main memory capacity ranges from 64 GiB up to 512 GiB of memory.

The cluster uses 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 multiple Lustre parallel file systems. In addition, each node provides a data center Intel® SATA SSD DC S3700 Series with 200 GiB or 400 GiB of available storage as a scratch-space (*XFS* formatted) usable within a compute job. In our experiments, we use these SSDs for storing data and metadata of GekkoFS or BeeGFS. Both GekkoFS and BeeGFS use internal chunk sizes of 512 KiB.

All Lustre experiments were performed on a Lustre scratch file system with 12 object storage targets (OSTs), two object storage servers (OSSs), and one metadata service (MDS) with a total of 1.2 PiB of storage.

For GekkoFS and BeeGFS experiments, all SSD contents are removed, and all kernel caches, i.e., buffer, inode, and dentry caches, are flushed before each experiment iteration. In addition, the GekkoFS daemons are restarted (requiring less than 20 seconds for 512 nodes) before each experiment iteration as well. The GekkoFS daemon and the BeeGFS storage and metadata services, and the application under test are pinned to separate processor sockets to ensure that file system and application do not interfere with each other.

4.1.2 MareNostrum IV

MareNostrum IV uses 3 456 Lenovo ThinkSystem SD530 compute nodes on 48 racks. Each node uses two Intel® Xeon® Platinum 8160 24C chips with 24 processors each at 2.1 GHz which totals to 165 888 processes and 390 TB of main memory. In addition, each node provides an Intel® SSD DC S3520 Series with 240 GiB of available storage, usable within a compute job. All GekkoFS experiments use these SSDs as their underlying storage.

A 100 Gb Intel® Omni-Path Full-Fat Tree is used for the interconnection network connected to a total 14 PB of storage capacity offered by IBM's GPFS. All GPFS experiments were run in the projects file system which offers 4.4 PB of storage with an 8 MB set file system block size.

[12] Mogon II. https://hpc.uni-mainz.de, Nov. 2019.

[13] Marenostrum. https://www.bsc.es/marenostrum/marenostrum, Nov. 2019.

[14] NEXTGenIO prototype. http://www.nextgenio.eu/, Nov. 2019.

4.1.3 *NEXTGenIO Prototype*

NEXTGenIO prototype is composed of 34 compute nodes[15]. Each node has a dual Intel® Xeon® Platinum 8260M CPU @ 2.40 GHz (i.e., 48 cores per node), 192 GiB of main memory and 3 TBytes of node-local Intel DCPMM memory. All GekkoFS experiments use these DCPMM memories as their underlying storage.

Compute nodes are interconnected with an Omni-Path fabric with two fabrics per node, henceforward called `ib0` and `ib1`, and they have a 56 Gbps InfiniBand to communicate with a Lustre server with 6 OSTs. GekkoFS uses TCP/IP over Omni-Path (emulated TCP) while Lustre uses the Omni-Path fabric with InfiniBand emulation.

4.2 Metadata Performance

On MOGON II, we simulated common metadata intensive HPC workloads using the *mdtest*[16] microbenchmark to evaluate GekkoFS' metadata performance. In our experiments, mdtest performs *create*, *stat*, and *remove* operations in parallel in a single directory. In particular, concurrent metadata operations in a single directory are an important workload in many HPC applications and are among the most difficult workloads for a general-purpose PFS to handle efficiently [31].

Each operation was performed on GekkoFS using 100 000 zero-byte files per process with 16 processes used per node. We chose the high number of files to force RocksDB to flush its in-memory tables to the underlying SSD to show RocksDB's consistent performance. As a result, a number of static sorted table files (*sst* files) were created on the backend SSD storage during our experiments.

All files created in the GekkoFS experiments are stored from the user application's perspective within a single directory. However, due to the flat namespace that is kept internally in the KV stores, there is conceptually no difference in which directory files are created. This is in contrast to a traditional PFS that may perform better if the workload is distributed among many directories instead of in a single directory. This is because inserting files into a directory is mostly sequentialized as multiple process cannot access the same file system block in parallel (see Section 2). 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.

Fig.3 compares GekkoFS with Lustre in three scenarios with up to 512 nodes: file creation, file stat, and file removal. The y-axis depicts the corresponding operations per second that were achieved for a particular workload on a logarithmic scale. During the experiments GekkoFS was running exclusively on the cluster without other compute jobs interfering. Each experiment was run at least five times with each data point representing the mean of all iterations. GekkoFS' workload scaled with 100 000 files per process, while Lustre's workload was fixed to four million files for all experiments. We fix the number of files for Lustre's metadata experiments because Lustre was detecting hanging nodes when scaling to too many files.

Lustre experiments were run in two configurations: all processes operated in a single directory (`single dir`) or each process worked in its own directory (`unique dir`). Moreover, Lustre's metadata performance was evaluated while the system was accessible by other applications as well.

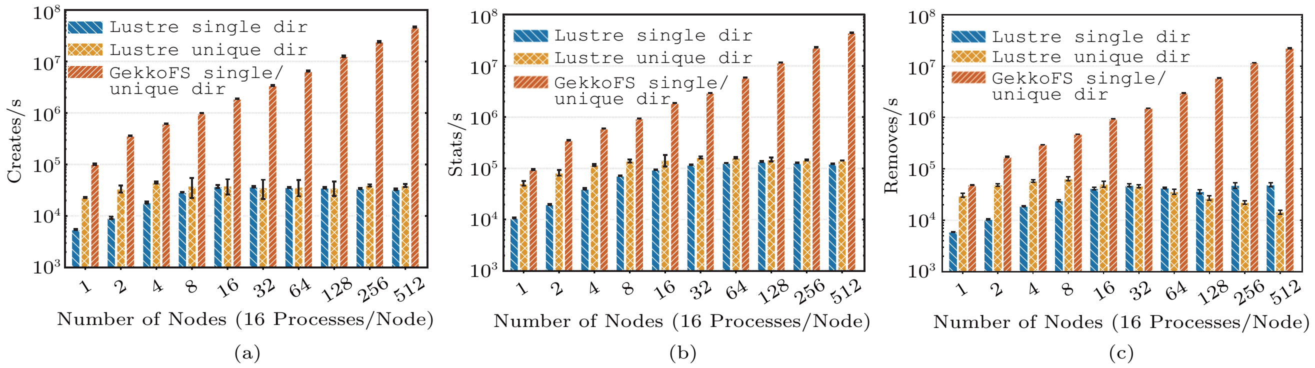


Fig.3. GekkoFS' file create, stat, and remove throughput for an increasing number of nodes compared to a Lustre parallel file system. (a) Create throughput. (b) Stat throughput. (c) Remove throughput.

[15] http://www.nextgenio.eu/, Nov. 2019.

[16] https://github.com/hpc/ior, Nov. 2019.

As seen in Fig.3, 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 with 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 standard deviation was computed as the percentage of the mean which, in all GekkoFS cases, was less than 3.5%. For GekkoFS, each operation is performed synchronously without any caching mechanisms in place and showed close to linear scaling. Therefore, we achieve our scalability goal, demonstrating the performance benefits of distributing metadata and decoupling directory entries from non-scalable directory blocks (see Subsection 3.5).

The GekkoFS experiments were also run while Mogon II was used by other users during production, revealing network interference within the cluster. Up to 128 nodes we were unable to measure a difference in metadata operation throughput outside of the margin for errors compared with the experiments in an undisturbed environment. For 256 and 512, we measured a reduced metadata operation throughput between 10% and 20% for create and stat operations. Remove operation throughput remained unaffected.

Lustre's metadata performance did not scale beyond approximately 32 nodes, demonstrating the aforementioned metadata scalability challenges in such a general-purpose PFS. Moreover, processes in Lustre experiments that operated within their own directory achieved a higher performance in most cases, except for 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. This is because the time required to remove the directory of each process (in which it creates its workload) is included in the remove throughput and the number of created `unique` directories increases with the number of used processes in an experiment. Similarly, the time to create the process directories is also included in the create throughput but does not show the same behavior as the remove throughput, indicating optimizations towards create operations.

Fig.4 compares GekkoFS with a POSIX-compliant burst buffer file system: BeeGFS' BeeOND. Similar to GekkoFS, we configured BeeGFS to distribute metadata across all file system nodes. All experiments are run in two configurations: all processes operated in a single directory (`single dir`) or each process worked in its own directory (`unique dir`). GekkoFS' workload and all BeeGFS `unique dir` experiments were weakly scaled with 100 000 files per process. The workload for BeeGFS `single dir` experiments, on the other hand, was fixed to four million files. This is because BeeGFS' distribution to multiple metadata servers is coupled with the number of directories. Hence, operations in a single directory are only utilizing a single metadata server which inherently causes congestion on the corresponding node, affecting scalability. Further, BeeGFS is equivalently deployed to GekkoFS, that is, BeeGFS is deployed in an ad hoc fashion for the lifetime of a compute job and destroyed afterwards while all metadata is stored on the node-local SSD. It is therefore only accessible by the benchmark application.

As seen in Fig.4, both GekkoFS' and BeeGFS' `unique dir` experiments show close to linear scalability with the number of nodes for create and stat operations. BeeGFS' remove operations, however, do not scale beyond four nodes, although throughput doubled from 256 to 512 nodes. GekkoFS' `unique dir` and `single dir` performance is equivalent as both scenarios are internally treated indifferently (see Subsection 3.5). In

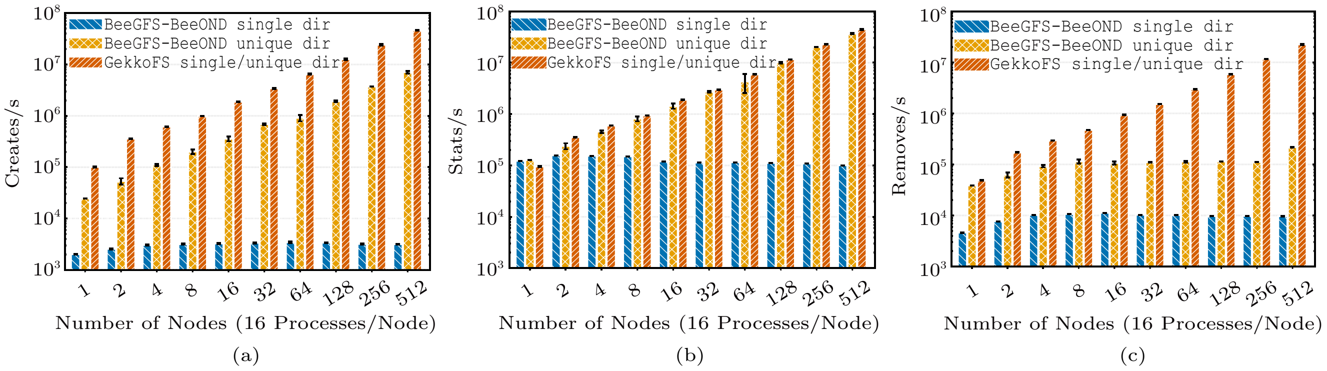


Fig.4. GekkoFS' file create, stat, and remove throughput for an increasing number of nodes compared with a BeeGFS' BeeOND. (a) Create throughput. (b) Stat throughput. (c) Remove throughput.

summary, at 512 nodes, GekkoFS achieved a ~6.5x higher create throughput, a ~1.2x higher stat throughput, and a ~102x higher remove throughput, compared with BeeGFS `unique dir` scenario. All BeeGFS `single dir` experiments did not scale beyond four nodes due to above described limitations.

4.3 Data Performance

On MOGON II, we evaluated GekkoFS' I/O performance with experiments that were designed to reflect some of the most difficult I/O patterns that scientific applications request from the PFS, such as small I/O requests or random access patterns. We used the IOR[17] microbenchmark which offers a rich configuration interface for evaluating a file system's I/O performance in various scenarios. We performed experiments with sequential and random access patterns in two configurations: each process is accessing its own file (file-per-process) and all processes access a single file (shared-file).

We used four different write and read sizes for each configuration, in the following called *transfer sizes*: 8 KiB, 64 KiB, 1 MiB, and 64 MiB to assess the performances for many small I/O accesses as well as for few large I/O requests. In all experiments 16 processes ran on each client with each process writing and reading 4 GiB in total.

GekkoFS data performance is not compared with the Lustre scratch file system as its peak performance, around 12 GiB per second, is already reached for no more than 10 nodes for sequential I/O patterns. Moreover, Lustre has shown to scale linearly for sequential access patterns in larger deployments with more OSSs and OSTs being available[41]. Instead, we first focus on GekkoFS' behavior with various transfer sizes and I/O patterns with the goal to achieve close to linear scalability, and then compare GekkoFS' I/O performance with BeeGFS' BeeOND where both file systems write and read data on the node-local SSDs.

We first evaluate file-per-process throughput performances, write and read operations per second (IOPS), and latencies. Then we discuss shared-file performance and its challenges.

File-per-Process and Sequential Access Patterns. Fig.5 shows GekkoFS' sequential read and write throughput in MiB/s for an increasing number of nodes for different transfer sizes. Each data point represents the mean of at least five iterations. The standard deviation, calculated as the percentage of the mean, was smaller than 5% in all cases, except for 64 KiB writes which varied up to 13%. Further, each data point is compared to the peak performance that all aggregated SSDs could deliver for a given node configuration, visualized as a white rectangle, indicating GekkoFS' SSD usage efficiency.

In general, every result demonstrates GekkoFS' close to linear scalability, achieving about 141 GiB/s (~80% of the aggregated SSD peak bandwidth) and 204 GiB/s (~70% of the aggregated SSD peak bandwidth) in respective write and read operations for 64 MiB transfer sizes for 512 nodes.

Any I/O operation which is larger than the used chunk size of 512 KiB will internally be split into equally sized chunk files on the node-local file system on one or more nodes (see Subsection 3.6). Chunks that hash to the same node are packaged together into the same RPC request, resulting in parallel RDMA accesses to the client's memory from multiple nodes. Therefore, chunks of 64 MiB I/O requests can be served in parallel, achieving a higher throughput than 1 MiB I/O requests. I/O requests smaller than the chunk size always target a single chunk, and hence, a single node.

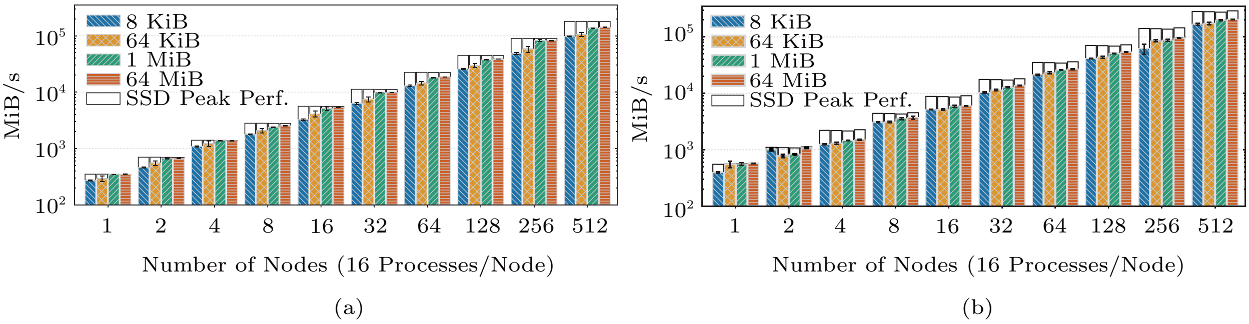
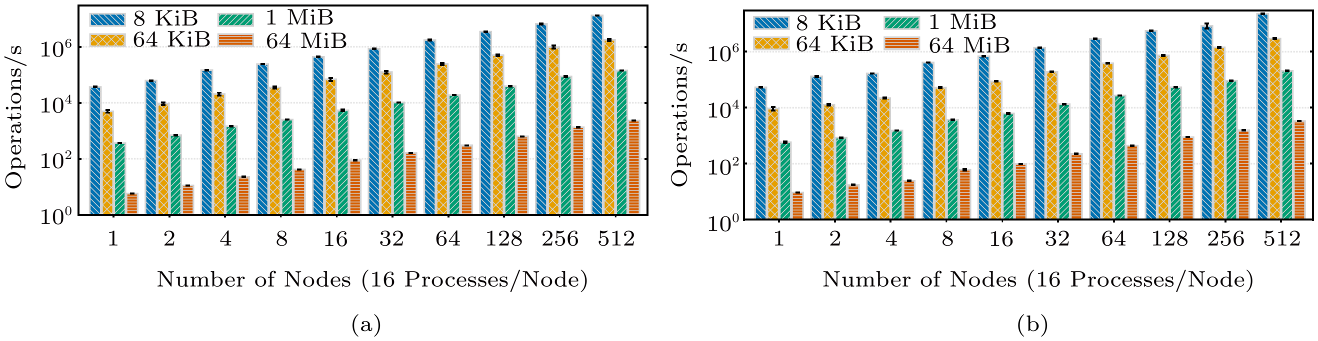


Fig.5. GekkoFS' sequential throughput for each process operating on its own file compared to the plain SSD peak throughput. (a) Write throughput. (b) Read throughput.

[17] https://github.com/hpc/ior, Nov. 2019.

Fig.6 shows the I/O operations per second (IOPS) for an increasing number of nodes (x-axis). GekkoFS achieved more than 13 million write IOPS and more than 22 million read IOPS at 512 nodes with an 8 KiB transfer size. In general, the smaller the transfer size, the higher the IOPS in all cases. This is because small I/O requests reduce the amount of time the target node spends during an I/O operation of a chunk file. Therefore, the latency of each I/O request becomes the predominant factor in such an operation causing a decrease in throughput and an increase in IOPS.

Fig.7 depicts the I/O latencies for such small write and read requests, exemplarily shown for 8 KiB transfer sizes. Each data point represents the mean latency for all write or read requests over five iterations. Because the experiments were weakly scaled, an increasing node number translates into more I/Os. For instance, at 512 nodes the I/O latency of over 167 million operations were taken into account. In all cases, the I/O latencies were well below the capabilities of modern hard disk drives with access times of several milliseconds, showing GekkoFS' efficient utilization of the SSD storage backend. Note that the latencies of read operations are lower than those of write operations as the latter involves an additional communication step to update the size of the file.



(a)

(b)

Fig.6. GekkoFS' sequential write and read operations per second for each process operating on its own file. (a) Write IOPS. (b) Read IOPS.

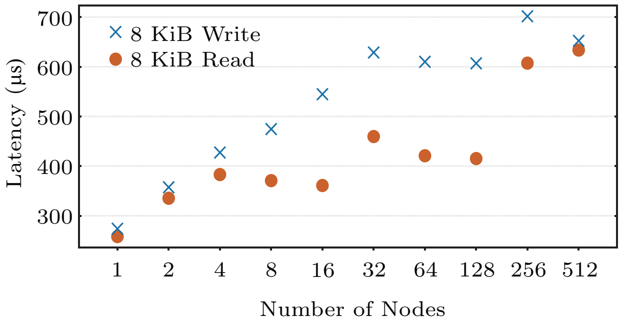
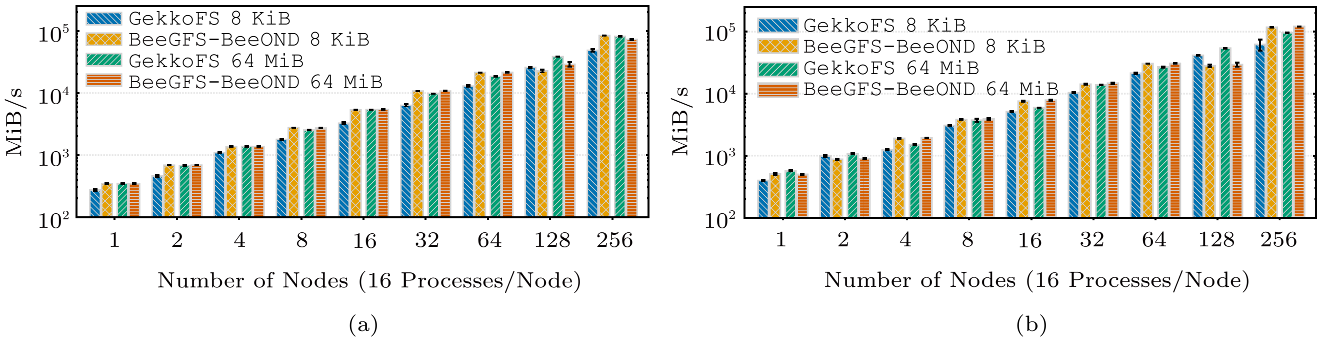


Fig.7. GekkoFS' I/O latencies for 8 KiB transfer sizes.

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

Fig.8 compares GekkoFS with BeeGFS' BeeOND up to 256 nodes in two configurations: 64 MiB and 8 KiB transfer size to evaluate large and small transfer sizes alike. In both configurations, BeeGFS scales almost



(a)

(b)

Fig.8. GekkoFS' sequential throughput for each process operating on its own file compared with BeeGFS' BeeOND. (a) Write throughput. (b) Read throughput.

linearly with a standard deviation smaller than 9% in all cases. Similar to GekkoFS, BeeGFS used node-local SSDs for storage with the same workload, and it distributed all data across all available nodes with the test directory's stripe size set to −1. At 256 nodes with 64 MiB transfer sizes, GekkoFS' write throughput is ∼1.12x higher than BeeGFS' while BeeGFS' read throughput is ∼1.26x higher than GekkoFS'. For operations with an 8 KiB transfer size, on the other hand, BeeGFS' write and read throughput are ∼1.73x and ∼1.86x higher than GekkoFS, respectively. Further experiments with BeeGFS and transfer sizes ranging from 1 MiB to 8 KiB showed that write and read throughput remained similar, suggesting caching mechanisms for such transfer sizes. Since GekkoFS is not utilizing any caching mechanisms, a reduction in I/O throughput for smaller transfer sizes is expected as each I/O request is sent individually and therefore becoming increasingly more latency-dependent in the process.

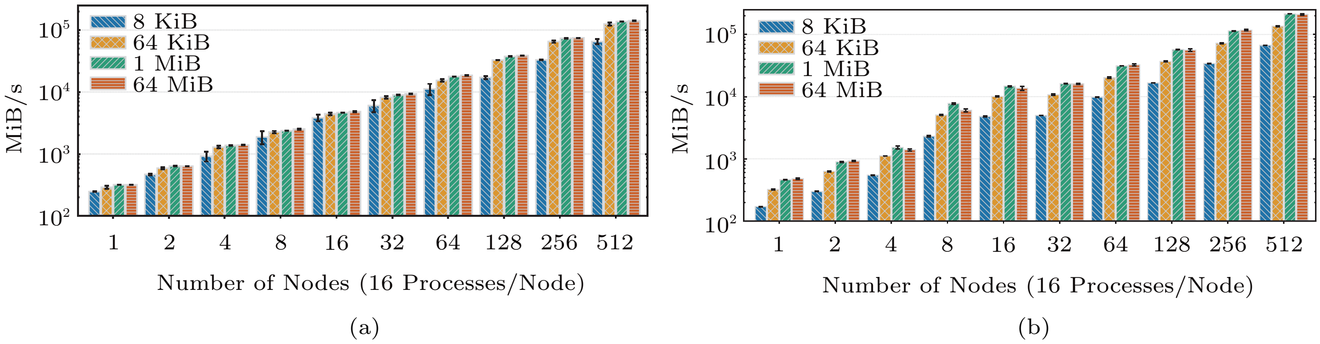
File per Process and Random Access Patterns. Fig.9 shows GekkoFS' throughput for random accesses for an increasing number of nodes, showing close to linear scalability in all cases. The file system achieved up to 141 GiB/s write throughput and up to 204 GiB/s read throughput for 64 MiB transfer sizes at 512 nodes.

The throughput for 64 MiB and 1 MiB transfer sizes is comparable to the sequential results. Nevertheless, write and read throughput decreased by approximately 33% and 60% for 512 nodes and a transfer size of 8 KiB. The reason is that transfer sizes larger than the chunk size internally access whole chunks while smaller transfer sizes access one chunk at a random offset.

Consequently, random accesses for large transfer sizes are conceptually the same as sequential accesses. In both cases whole chunks are written or read, resulting in a similar performance. Small transfer sizes, on the other hand, are slower than sequential accesses due to the resulting random access to chunks. Hence, applications may benefit from choosing smaller chunk sizes if their transfer sizes are small.

Fig.10 compares GekkoFS' throughput for random accesses with BeeGFS' with up to 256 nodes. At 256 nodes with 64 MiB transfer sizes, GekkoFS' write and read throughput is ∼1.86x and ∼2.7x higher than BeeGFS', respectively, potentially showing the above-described benefits of transfer sizes larger than the chunk size. For 8 KiB transfer sizes, GekkoFS' write throughput is ∼2x higher than BeeGFS' while BeeGFS' read throughput is ∼1.95x higher than GekkoFS'.

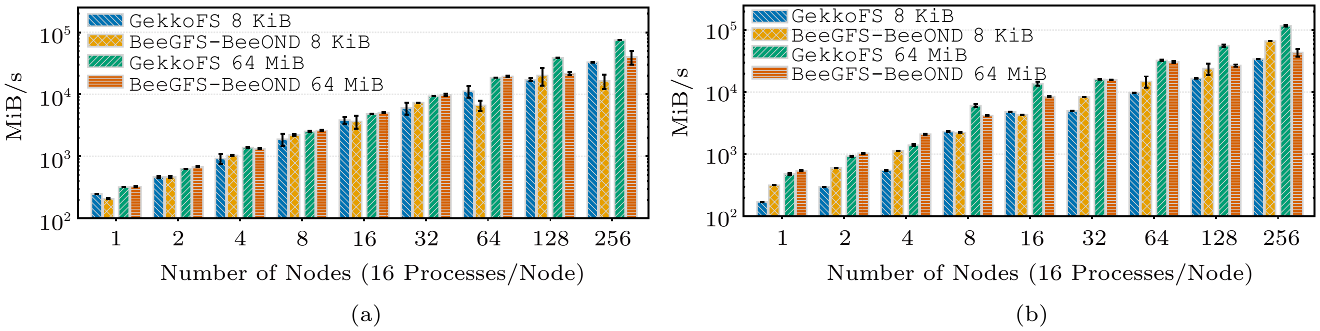
Single Shared File. Shared file operations have many similarities to the previously presented experiments in which each process operated on its own file.



(a)

(b)

Fig.9. GekkoFS' random (a) write and (b) read throughput for each process operating on its own file.



(a)

(b)

Fig. 10. GekkoFS' random (a) write and (b) read throughput for each process operating on its own file compared with BeeGFS' BeeOND.

For instance, in cases where the transfer size is bigger 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. If two processes try to access the same offset in the same chunk file, locking mechanisms of the node-local file system serialize the access to this file.

Fig.11 presents the write throughput and the write IOPS for 64 KiB and 8 KiB transfer sizes for a sequentially written single shared file for up to 32 nodes. In these examples a drawback of GekkoFS' synchronous and cache-less design becomes visible and no more than approximately 150 000 write operations per second were achieved. We omit experiments for more than 32 nodes as throughput and IOPS stagnate. This behavior is caused by the size of a single metadata entry that is maintained by only a single node and needs to be updated constantly by all nodes in a mutually exclusive way.

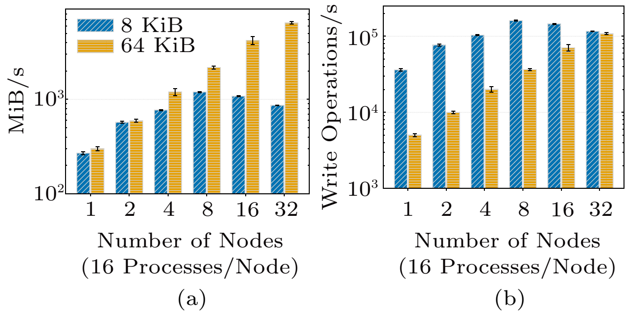


Fig.11. GekkoFS' sequential write throughput and write IOPS for each process operating on a single shared file. (a) Write throughput. (b) Write IOPS.

This bottleneck becomes worse for more processes participating in an experiment — eventually reaching a bottleneck, visible at the number of write IOPS. Note that each node in the previous file-per-process experiments only handled at most 25 000 write operations per second on average. Therefore, such behavior did not appear during the previous experiments. The root-cause lies within the number of small RPC messages updating the file size for each write operation, causing network contention on the metadata maintaining node. This observation is supported by the fact that read operations did not show a similar behavior as no metadata updates are necessary and that increasing the number of RPC handler threads did not result in a higher number of write IOPS.

To reduce the number of RPC messages being sent to a single node, we add a rudimentary client cache to locally buffer size updates of a number of write operations before they are sent to the node that manages the size of the file. Fig.12 shows the resulting throughput for the cache case for sequential accesses on a single shared file for up to 512 nodes, now showing close to linear scalability. Due to the internal similarities in data distribution, shared file performances are now comparable to the previously discussed file-per-process performances. While the usage of such a cache may not be ideal for all applications, it significantly prevents node contention.

4.4 I/O Variability and Worst-Case

On MareNostrum IV, we used the IOR benchmark to measure the I/O variability and worst-case I/O performance, which applications experience when using GekkoFS compared to the cluster's GPFS. All experiments were run during production, and they were co-located with the ordinary HPC workload. For each file system, we ran 25 independent repetitions of the same benchmark set with each using different node allocations at different time throughout one week.

Each benchmark set used varying I/O request sizes ranging between 512 bytes and 64 MiB to evaluate a more realistic representation of real applications whose

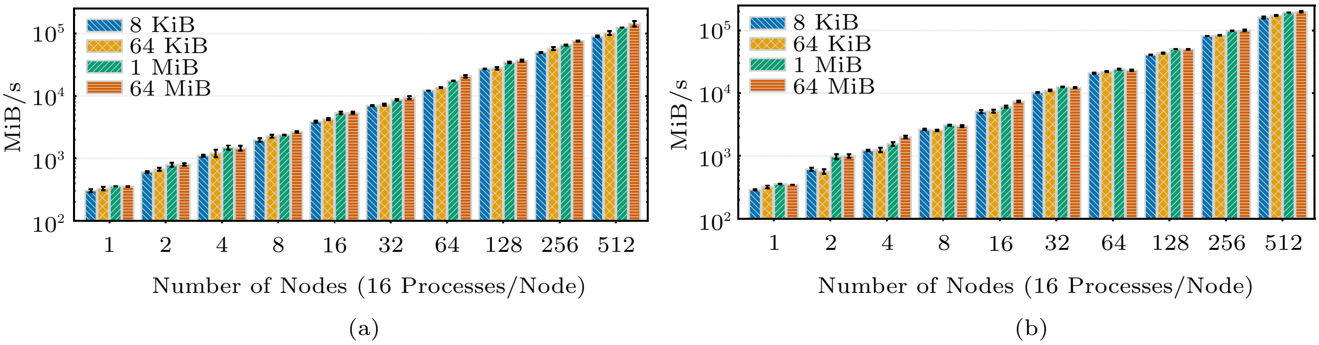


Fig.12. GekkoFS' sequential throughput with a metadata client cache for all processes operating on a single shared file. (a) Write throughput. (b) Read throughput.

I/O request sizes usually differ during execution. In addition, GekkoFS chunk size was varied as well, ranging between 128 KiB and 64 MiB. GPFS' block size is not dynamically changeable and therefore remained 8 MiB. The benchmarks used 24 out of the 48 available cores on each node, and the written file sizes were set large enough to fill the node's main memory to avoid cache effects. The files IOR writes data to and reads from are initially created with each process working on a dedicated file.

Fig.13 presents the I/O variability of GPFS and GekkoFS as squares and triangles, respectively, up to 32 nodes for each IOR run. GPFS' measured throughput shows significant variability, often scattered by orders of magnitude, with writes showing a higher variability than reads. In fact, such I/O behavior is not uncommon at many HPC sites[42–44], and it is generally known as the so-called *cross-application interference*. Cross-application interference is caused by an *I/O bottleneck* where a shared resource, such as a PFS, is accessed by multiple, uncoordinated applications. With this I/O bottleneck already being a great challenge at many HPC sites today, some studies suggest this issue could become one of the core challenges for Exascale machines in the future[45–47].

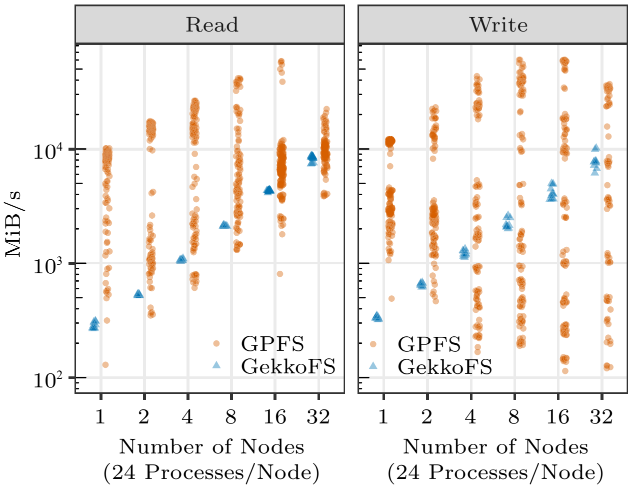


Fig.13. I/O variability of GPFS and GekkoFS of multiple IOR runs on different time and node allocations throughout one week.

GekkoFS, on the other hand, shows steady and predictable performance with low cross-application interference, showing the benefits of a private, job-exclusive file system that is accessed by a single application. In both read and write cases, GekkoFS shows close to linear scalability as the available bandwidths and storage capabilities increase with the number of nodes, eventually closing the gap to GPFS' best-case performance. Nonetheless, in terms of the worst-case performance both file systems experience, GekkoFS considerably excels GPFS' throughput in most cases due to the low impact of cross-application interference on GekkoFS, visualized in Fig.14.

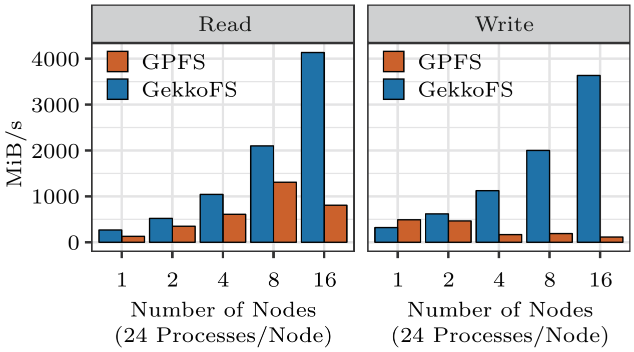


Fig.14. I/O worst-case of GPFS and GekkoFS of multiple IOR runs on different time and node allocations throughout one week.

4.5 Effects on the Network

On the NEXTGenIO prototype, we use the user application *OpenFOAM* to investigate GekkoFS' effects on the network. OpenFOAM[18] is a C++ library for developing user-customized numerical solvers for the solution of Continuum Mechanics problems, including Computational Fluid Dynamics. OpenFOAM solvers often require multiple stages to complete, involve large amounts of I/O, and benefit from using multiple nodes while parallelization is achieved with MPI. The application is used in industry as well as academia in large-scale computations.

In our experiments, we used the *simpleFOAM* solver which is a steady-state solver for incompressible, turbulent flow, using the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm. We used 100 iterations with a time step 1 setting which generates around 25 GiB of data in over 170 000 files. The experiments were run on four nodes (24 processes per node) with simpleFOAM pinned to socket 0. GekkoFS and Lustre both use the `ib0` Omni-Path adapter. SimpleFOAM's MPI communication was run in three configurations: MPI using the `ib0`, `ib1`, or `ib0` and `ib1` Omni-Path adapters. Hence, `ib1` separates MPI from Lustre's and GekkoFS' internal file system network traffic.

Fig.15 shows the runtime (in seconds) of the simpleFOAM experiments when used with Lustre or GekkoFS. The three bar groups depict the above-described simpleFOAM MPI configurations for each file system. Although simpleFOAM in this configuration does not generate significant I/O, GekkoFS is still ∼7%–∼9% faster in all cases compared with Lustre. Further profiling revealed that the performance improvements are caused by `MPI_Waitall` and `MPI_Allreduce`, instead of from direct file system operations. As a result, both MPI functions potentially benefit from less generated network pollution of GekkoFS. This is because GekkoFS is writing parts of the data locally (see Subsection 3.6) with less file system traffic being put onto the network in general compared with Lustre, where all file system communication is remote. Therefore, in addition to GekkoFS' linear scaling for metadata and data operations (see Subsection 4.2 and Subsection 4.3), GekkoFS can also be beneficial for an application's inter-node communication by reducing file system network pollution.

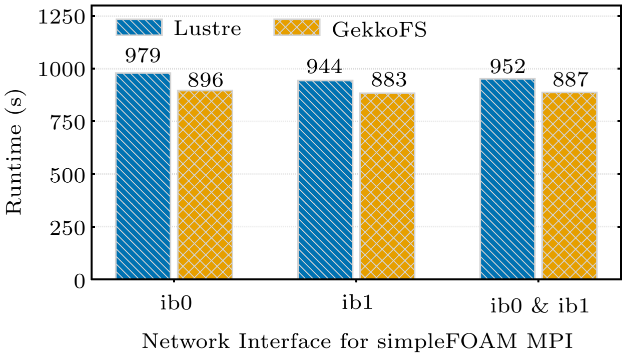


Fig.15. Runtime of the simpleFOAM application when its MPI communication is run on different network interfaces.

5 Conclusions

Increasingly more powerful HPC clusters will lead to even more I/O pressure on a general-purpose PFS, as more applications will concurrently access the shared storage. At the same time, applications with access to node-local SSDs at the compute nodes can use burst buffer file systems to benefit from low latencies and modern network fabrics, reducing the load on the global PFS. Yet, burst buffer file systems often offer features that scientific applications do not require when running in isolation in an exclusive file system environment.

We introduced and evaluated GekkoFS, a new burst buffer file system with relaxed POSIX-semantics, which is optimized for access patterns of HPC applications that are known to not work well on a traditional PFS. The POSIX relaxation allows GekkoFS to especially scale metadata operations for accesses to a single directory or even to a single file. We evaluated GekkoFS' advantages for typical metadata workloads and showed that it is able to achieve millions of metadata operations already for a small number of nodes. Moreover, we presented its linear scalability and low impact of cross-application interference on the file system. We discussed challenges that occur for shared file I/O operations and how client caches can overcome resulting bottlenecks. Finally, we evaluated GekkoFS' I/O variability and its effects on the network with the OpenFOAM application.

We plan to extend GekkoFS into three directions. First, we plan an extensive survey on use cases which might benefit from even smaller chunk sizes than the investigated 512 KiB, including metadata effects on node-local file systems. Second, we plan to investigate how applications can benefit from caching without compromising GekkoFS' lightweight design. Third, we aim to carefully explore in detail how the file system itself and other applications interfere with GekkoFS to further improve its predictable performance.

Acknowledgments We greatly appreciate Franz-Joseph Pfreundt with Fraunhofer ITWM for his assistance on configuring and running BeeGFS' BeeOND. This research was conducted using the supercomputer Mogon II and services offered by Johannes Gutenberg University Mainz. The authors gratefully acknowledge the computing time granted on Mogon II.

References

[1] Hey T, Tansley S, Tolle K M. The Fourth Paradigm: Data-Intensive Scientific Discovery (1st edition). Microsoft Research, 2009.

[2] Ross R, Thakur R, Choudhary A. Achievements and challenges for I/O in computational science. *Journal of Physics: Conference Series*, 2005, 16(1): 501-509.

[3] Nieuwejaar N, Kotz D, Purakayastha A, Ellis C S, Best M L. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 1996, 7(10): 1075-1089.

[4] Wang F, Xin Q, Hong B, Brandt S A, Miller E, Long D, McLarty T. File system workload analysis for large scientific computing applications. In *Proc. the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004, pp.139-152.

[5] Crandall P, Aydt R A, Chien A A, Reed D A. Input/output characteristics of scalable parallel applications. In *Proc. the 1995 Supercomputing*, December 1995, Article No. 59.

[6] Dorier M, Antoniu G, Ross R B, Kimpe D, Ibrahim S. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *Proc. the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp.155-164.

[7] Thapaliya S, Bangalore P, Lofstead J F, Mohror K, Moody A. Managing I/O interference in a shared burst buffer system. In *Proc. the 45th International Conference on Parallel Processing*, August 2016, pp.416-425.

[8] Lofstead J F, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proc. the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, June 2008, pp.15-24.

[9] Folk M, Cheng A, Yates K. HDF5: A file format and I/O library for high performance computing applications. In *Proc. the 1999 Supercomputing* (*CD-ROM*), November 1999, pp.5-33.

[10] Liu N, Cope J, Carns P H, Carothers C D, Ross R B, Grider G, Crume A, Maltzahn C. On the role of burst buffers in leadership-class storage systems. In *Proc. the 28th IEEE Symposium on Mass Storage Systems and Technologies*, April 2012, Article No. 5.

[11] Wang T, Mohror K, Moody A, Sato K, Yu W. An ephemeral burst-buffer file system for scientific applications. In *Proc. the 2016 International Conference for High Performance Computing*, November 2016, pp.807-818.

[12] Bent J, Gibson G A, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate M. PLFS: A checkpoint filesystem for parallel applications. In *Proc. the 2009 ACM/IEEE Conference on High Performance Computing*, November 2009, Article No. 26.

[13] Vilayannur M, Nath P, Sivasubramaniam A. Providing tunable consistency for a parallel file store. In *Proc. the 2005 Conference on File and Storage Technologies*, December 2005, Article No. 3.

[14] Lensing P H, Cortes T, Hughes J, Brinkmann A. File system scalability with highly decentralized metadata on independent storage devices. In *Proc. the 16th the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2016, pp.366-375.

[15] Soumagne J, Kimpe D, Zounmevo J A, Chaarawi M, Koziol Q, Afsahi A, Ross R B. Mercury: Enabling remote procedure call for high-performance computing. In *Proc. the 2013 IEEE International Conference on Cluster Computing*, September 2013, Article No. 50.

[16] Seo S, Amer A, Balaji P, Bordage C *et al.* Argobots: A lightweight low-level threading and tasking framework. *IEEE Trans. Parallel Distrib. Syst.*, 2018, 29(3): 512-526.

[17] Carns P H, Jenkins J, Cranor C D, Atchley S, Seo S, Snyder S, Ross R B. Enabling NVM for data-intensive scientific services. In *Proc. the 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, November 2016, Article No. 4.

[18] Jasak H, Jemcov A, Tukovic Z *et al.* OpenFOAM: A C++ library for complex physics simulations. In *Proc. the International Workshop on Coupled Methods in Numerical Dynamics*, September 2007, Article No. 3.

[19] Vef M, Moti N, Süß T, Tocci T, Nou R, Miranda A, Cortes T, Brinkmann A. GekkoFS — A temporary distributed file system for HPC applications. In *Proc. the 2018 IEEE International Conference on Cluster Computing*, September 2018, pp.319-324.

[20] Schmuck F B, Haskin R L. GPFS: A shared-disk file system for large computing clusters. In *Proc. the 2002 Conference on File and Storage Technologies*, January 2002, pp.231-244.

[21] Braam P J, Schwan P. Lustre: The intergalactic file system. In *Proc. the 2002 Ottawa Linux Symposium*, June 2002, pp.50-54.

[22] Qian Y, Li X, Ihara S, Zeng L, Kaiser J, Süß T, Brinkmann A. A configurable rule based classful token bucket filter network request scheduler for the Lustre file system. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 6.

[23] Herold F, Breuner S. An introduction to BeeGFS. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, August 2019.

[24] Ross R B, Latham R. PVFS — PVFS: A parallel file system. In *Proc. the 2006 ACM/IEEE Conference on High Performance Networking and Computing*, November 2006, Article No. 34.

[25] Oral S, Shah G. Spectrum scale enhancements for CORAL. http://files.gpfsug.org/presentations/2016/SC16/11_Sarp_Oral_Gautam_Shah_Spectrum_Scale_Enhancements_for_CORAL_v2.pdf, August 2019.

[26] Kougkas A, Devarajan H, Sun X. Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proc. the 27th International Symposium on High-Performance Parallel and Distributed Computing*, June 2018, pp.219-230.

[27] Latham R, Ross R B, Thakur R. The impact of file systems on MPI-IO scalability. In *Proc. the 11th European PVM/MPI Users' Group Meeting*, September 2004, pp.87-96.

[28] Choudhary A, Liao W K, Gao K, Nisar A, Ross R, Thakur R, Latham R. Scalable I/O and analytics. *Journal of Physics: Conference Series*, 2009, 180(1): Article No. 012048.

[29] Moore M, Bonnie D, Ligon B, Marshall M, Ligon W, Mills N, Quarles E, Sampson S, Yang S, Wilson B. OrangeFS: Advancing PVFS. https://www.usenix.org/legacy/event/fast11/posters_files/Moore.pdf, August 2019.

[30] Ritchie D, Thompson K. The UNIX time-sharing system (reprint). *Commun. ACM*, 1983, 26(1): 84-89.

[31] Vef M A, Tarasov V, Hildebrand D, Brinkmann A. Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Trans. Storage*, 2018, 14(2): Article No. 18.

[32] Patil S, Gibson G A. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proc. the 9th USENIX Conference on File and Storage Technologies*, February 2011, pp.177-190.

[33] Ren K, Zheng Q, Patil S, Gibson G A. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proc. the 2014 International Conference for High Performance Computing*, November 2014, pp.237-248.

[34] Carns P, Yao Y, Harms K, Latham R, Ross R, Antypas K. Production I/O characterization on the Cray XE6. In *Proc. the Cray User Group Meeting*, May 2013, Article No. 121.

[35] Xing J, Xiong J, Sun N, Ma J. Adaptive and scalable metadata management to support a trillion files. In *Proc. the 2009 ACM/IEEE Conference on High Performance Computing*, November 2009, Article No. 31.

[36] Frings W, Wolf F, Petkov V. Scalable massively parallel I/O to task-local files. In *Proc. the 2009 ACM/IEEE Conference on High Performance Computing*, November 2009, Article No. 22.

[37] Yang S, Ligon III W B, Quarles E C. Scalable distributed directory implementation on orange file system. In *Proc. the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, May 2011.

[38] Patil S, Ren K, Gibson G. A case for scaling HPC metadata performance through de-specialization. In *Proc. the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, November 2012, pp.30-35.

[39] Carns P H, Ligon III W B, Ross R B, Thakur R. PVFS: A parallel file system for Linux clusters. In *Proc. the 4th Annual Linux Showcase & Conference*, October 2000, Article No. 4.

[40] Dong S, Callaghan M, Galanis L, Borthakur D, Savor T, Strum M. Optimizing space amplification in RocksDB. In *Proc. the 8th Biennial Conference on Innovative Data Systems Research*, January 2017, Article No. 30.

[41] Oral S, Dillow D A, Fuller D *et al.* OLCF's 1 Tb/s, next-generation Lustre file system. In *Proc. the 2013 Cray User Group Conference*, May 2013, Article No. 151.

[42] Lofstead J F, Zheng F, Liu Q, Klasky S, Oldfield R, Kordenbrock T, Schwan K, Wolf M. Managing variability in the IO performance of petascale storage systems. In *Proc. the 2010 Conference on High Performance Computing Networking, Storage and Analysis*, November 2010, Article No. 35.

[43] Xie B, Chase J S, Dillow D, Drokin O, Klasky S, Oral S, Podhorszki N. Characterizing output bottlenecks in a supercomputer. In *Proc. the 2012 International Conference on High Performance Computing Networking, Storage and Analysis*, November 2012, Article No. 8.

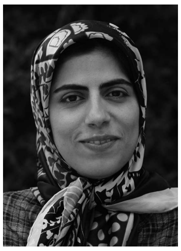
[44] Kougkas A, Devarajan H, Sun X, Lofstead J F. Harmonia: An interference-aware dynamic I/O scheduler for shared non-volatile burst buffers. In *Proc. the 2018 IEEE International Conference on Cluster Computing*, September 2018, pp.290-301.

[45] Hashimoto Y, Aida K. Evaluation of performance degradation in HPC applications with VM consolidation. In *Proc. the 3rd International Conference on Networking and Computing*, December 2012, pp.273-277.

[46] Lofstead J F, Ross R. Insights for exascale IO APIs from building a petascale IO API. In *Proc. the 2013 International Conference for High Performance Computing*, November 2013, Article No. 87.

[47] Reed D A, Dongarra J J. Exascale computing and big data. *Commun. ACM*, 2015, 58(7): 56-68.

Marc-André Vef is a third-year Ph.D. candidate in André Brinkmann's research team at the Johannes Gutenberg University Mainz, Mainz. He started his Ph.D. in 2016 after receiving his B.Sc. and M.Sc. degrees in computer science from the Johannes Gutenberg University Mainz, Mainz. His master thesis was in cooperation with IBM Research about analyzing file create performance in the IBM Spectrum Scale parallel file system (formerly GPFS). His research interests focus on parallel and ad-hoc file systems and system analytics.



Nafiseh Moti is a Ph.D. student in computer science at the Johannes Gutenberg University Mainz. Currently, she is a research assistant at Efficient Computing and Storage Group at Center for Data Processing at the University of Mainz, Mainz. Her main research interests include user-level file systems and file systems and data structures for storage class memories.

Tim Süß is a full professor at the Computer Science Department of the Fulda University of Applied Science (since 2019). He received his Ph.D. degree in computer science in 2011 from the Paderborn University, Paderborn, and has been post-doc at the Johannes Gutenberg University Mainz, from 2012 to 2019 with an intermission as temporal professor for Applied Computer Science from 2017 to 2018. His research interests focus on techniques for accelerating parallel computing, effective parallel scheduling, and storage systems.

Markus Tacke is the technical head of HPC (since 2015) at the Center for Data Processing (ZDV) at the Johannes Gutenberg University Mainz, Mainz, Germany. He studied mathematic (and astronomy) at the WWU Münster, Germany, where he received his M.Sc. degree in 1987 in applied mathematics. He joined the ZDV in 1991 and has been in charge of several responsibilities, including the design of the university's two supercomputers (since 1994). However, his true passion lies in parallel file systems.

Tommaso Tocci graduated in computer science at Sapienza University of Rome, Rome, and he concluded his Master's degree in distributed systems at Universitat Politècnica de Catalunya (UPC), Barcelona. At the moment he is employed at Barcelona Supercomputing Center (BSC), investigating new distributed storage solutions for the upcoming exascale computing era.

Ramon Nou has been working at Barcelona Supercomputing Center (BSC) on the Autonomic System and e-Business Platforms group of the Barcelona Supercomputing Center (BSC), Barcelona, until 2008 when he switched to the Storage-System group at BSC since 2009 as a researcher. In 2008, he obtained his Ph.D. degree with the topic "Using online simulation to improve QoS on middleware". Ramon has a wide view on all computer levels, with expertise in optimization, performance measurements and simulation/modelling of complex systems. Now he has been co-leader of the Storage Systems for Extreme Computing research group since January 2019.

Alberto Miranda is a senior researcher in advanced storage systems in the Computer Science Department of the Barcelona Supercomputing Center (BSC), Barcelona, and co-leader of the Storage Systems for Extreme Computing research group since January 2019. His research interests include scalable storage technologies, architectures for distributed systems, operating system internals, and high performance networking. He received a Ph.D. Cum Laude in computer science from the Technical University of Catalonia (UPC), Barcelona, in 2014, and has been working at BSC since 2007.

Toni Cortes is an associate professor at Universitat Politècnica de Catalunya (since 1998), Barcelona, and researcher at the Barcelona Supercomputing Center, Barcelona. He received his M.S. degree in computer science in 1992 and his Ph.D. degree also in computer science in 1997 (both at Universitat Politècnica de Catalunya). Currently he develops his research at the Barcelona Supercomputing Center, where he acted as the leader of the Storage Systems Research Group from 2006 until 2019. His research concentrates in storage systems, programming models for scalable distributed systems and operating systems. He is also editor of the Cluster Computing Journal and served as the coordinator of the SSI task in the IEEE TCSS. He has also served in many international conference program committees and/or organizing committees and was general chair for the Cluster 2006 and 2021 conference, LaSCo 2008, XtreemOS summit 2009, and SNAPI 2010. He is also served as the chair of the steering committee for the Cluster conference series (2011−2014). His involvement in IEEE CS has been awarded by the "Certificate of Appreciation" in 2007.

André Brinkmann is a full professor at the Computer Science Department of Universitat Politècnica de Catalunya and head of the university's Data Center ZDV (Zentrum für Datenverarbeitung) (since 2011). He received his Ph.D. degree in electrical engineering in 2004 from the Paderborn University and has been an assistant professor in the Computer Science Department of the Paderborn University from 2008 to 2011. Furthermore, he has been the managing director of the Paderborn Centre for Parallel Computing PC^2 during this time frame. His research interests focus on the application of algorithm engineering techniques in the area of data centre management, cloud computing, and storage systems.