Brinkmann A, Mohror K, Yu W *et al.* Ad hoc file systems for high-performance computing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 4–26 Jan. 2020. DOI 10.1007/s11390-020-9801-1

Ad Hoc File Systems for High-Performance Computing

André Brinkmann[1,*], *Member, ACM*, Kathryn Mohror[2,*], *Member, ACM, IEEE*
Weikuan Yu[3,*], *Senior Member, IEEE, Member, ACM*, Philip Carns[4], Toni Cortes[5], Scott A. Klasky[6]
Alberto Miranda[7], Franz-Josef Pfreundt[8], *Member, ACM*, Robert B. Ross[4], and Marc-André Vef[1]

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

[2] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, U.S.A.*

[3] *Department of Computer Science, Florida State University, Tallahassee, FL 32306, U.S.A.*

[4] *Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, U.S.A.*

[5] *Department of Computer Architecture, Universitat Politecnica de Catalunya, Barcelona 08034, Spain*

[6] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, U.S.A.*

[7] *Computer Science Department, Barcelona Supercomputing Center, Barcelona 08034, Spain*

[8] *Fraunhofer Institute for Industrial Mathematics ITWM, Fraunhofer-Platz 1, Kaiserslautern 67663, Germany*

E-mail: brinkman@uni-mainz.de; mohror1@llnl.gov; yuw@cs.fsu.edu; carns@mcs.anl.gov; toni.cortes@bsc.es
klasky@ornl.gov; alberto.miranda@bsc.es; pfreundt@itwm.fhg.de; rross@mcs.anl.gov
vef@uni-mainz.de

Received June 30, 2019; revised August 30, 2019.

Abstract Storage backends of parallel compute clusters are still based mostly on magnetic disks, while newer and faster storage technologies such as flash-based SSDs or non-volatile random access memory (NVRAM) are deployed within compute nodes. Including these new storage technologies into scientific workflows is unfortunately today a mostly manual task, and most scientists therefore do not take advantage of the faster storage media. One approach to systematically include node-local SSDs or NVRAMs into scientific workflows is to deploy ad hoc file systems over a set of compute nodes, which serve as temporary storage systems for single applications or longer-running campaigns. This paper presents results from the Dagstuhl Seminar 17202 "Challenges and Opportunities of User-Level File Systems for HPC" and discusses application scenarios as well as design strategies for ad hoc file systems using node-local storage media. The discussion includes open research questions, such as how to couple ad hoc file systems with the batch scheduling environment and how to schedule stage-in and stage-out processes of data between the storage backend and the ad hoc file systems. Also presented are strategies to build ad hoc file systems by using reusable components for networking and how to improve storage device compatibility. Various interfaces and semantics are presented, for example those used by the three ad hoc file systems BeeOND, GekkoFS, and BurstFS. Their presentation covers a range from file systems running in production to cutting-edge research focusing on reaching the performance limits of the underlying devices.

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

Survey

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

This work has also been partially funded by the German Research Foundation (DFG) through the German Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) and the ADA-FS project, and by the European Union's Horizon 2020 Research and Innovation Program under the NEXTGenIO Project under Grant No. 671591, the Spanish Ministry of Science and Innovation under Contract No. TIN2015-65316, and the Generalitat de Catalunya under Contract No. 2014-SGR-1051. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 (LLNL-JRNL-779789) and also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract No. DE-AC02-06CH11357. This work is also supported in part by the National Science Foundation of USA under Grant Nos. 1561041, 1564647, 1744336, 1763547, and 1822737.

*Corresponding Author

1 Introduction

For decades, magnetic disks have served as the storage backbone of high-performance computing (HPC) clusters, and their physical properties have significantly influenced the design of parallel file systems. Magnetic disks are composed of rotating platters and a mechanical arm positioning the read and write heads on the platter tracks. The bandwidth of magnetic disks is limited by the disk rotation speed and access latency (the seek time of the disk arm) [1]. Sequential accesses to magnetic disks can easily achieve sustained transfer rates of more than 200 MByte/s, and thus thousands of disks accessed in parallel can read or write at speeds of more than 1 TByte/s.

In contrast to the high throughput achievable with sequential accesses to magnetic disks, random accesses to a single disk reduce disk transfer rates to less than 1% of peak performance. Random access patterns executed by a single application on a storage system can easily lead to a denial of service to the complete HPC system [2]. Applications that perform huge amounts of random accesses can be highly problematic for HPC clusters and parallel file systems that are optimized for sequential read and write access patterns, which occur, for example, during the stage-in of data at application start-up or during checkpointing [3].

The mechanical composition of magnetic disks also leads to a high failure rate compared with that of pure semiconductor components. As a result, magnetic disks are used mostly for maintenance-friendly, dedicated storage clusters that are physically separated from the compute nodes. In this configuration, failing disks can easily be replaced when compared with configurations with disks attached to compute nodes, where there is generally no easy manual disk access. Unfortunately, the dedicated storage cluster configuration hinders the scalability of the storage backend with respect to an increased compute node count.

The challenges and limitations associated with magnetic disks have led to the introduction of new storage technologies into HPC systems, including non-volatile random-access memory (NVRAM) devices such as flash-based solid state drives (SSDs), as new storage tiers in addition to the parallel file system (PFS). SSDs are fully semiconductor-based and can benefit from decreasing process technologies. Today, a single non-volatile memory express (NVMe) SSD can deliver an order of magnitude higher bandwidth than that of a magnetic disk with little difference in the performance between random and sequential access patterns. Additionally, the reliability of SSDs is higher than that of magnetic disks. The replacement rates of SSDs in the field are much smaller than those of magnetic disks [4] and depend mostly on the amount of data written to the SSD [5]. Different bit error rate characteristics during the lifetime of an SSD also enable administrators to identify the likelihood that a device will fail and therefore to proactively exchange it [6].

The use of SSDs is already widespread in most new HPC systems and they can be used as metadata storage for parallel file systems [7], in-system burst buffers [8], and node-local storage. The bandwidth of node-local SSDs typically exceeds the peak bandwidth of the attached parallel file system, while the maximum number of I/O operations (IOPS) can even be more than 10 000x higher than that of the parallel file system (see Table 1).

Several efforts have been made to explore best methods for using node-local SSDs, including as an additional caching layer for parallel file systems [9] or as file systems run over node-local SSDs, with a larger number of these file systems being implemented at the user level [10, 11].

This paper focuses on ad hoc file systems that aim to efficiently use temporarily available storage on either node-local SSDs or global burst buffers. While existing parallel file systems can be in principle built on top of node-local SSDs or global burst buffers, it is important to adhere to the following definitions to be useful in the context of ad hoc file systems for HPC.

• Ad hoc file systems can be deployed on HPC clusters for lifetimes as small as the runtime of a single job to use node-local SSDs or external burst buffers. It is

Table 1. SSD Usage in Different Sized Cluster Environments

	Node Capacity (GB)	Node Bandwidth (MB/s)	Node Count	Cluster Capacity (TB)	Cluster Bandwidth (GB/s)
MOGON II	400	500	1 868	≈ 747	934
Summit	1 600	6 000	4 608	7 300	≈ 27 648
MareNostrum 4	240	320	3 456	≈ 829	1 106
Theta	128	⩾ 2 150	4 392	562	⩾ 9 307

therefore important that the deployment overhead of the ad hoc file systems is low compared with the application runtime.

• Ad hoc file systems provide a global namespace for all nodes being linked to the ad hoc file system, while the semantics of the file system can be optimized for the application scenario, enabling optimized parallel access schemes.

• Ad hoc file systems interact with the backend storage system using data staging, and most ad hoc file systems have been and will be implemented completely in user space, while they also may contain a kernel component.

User-level file systems are an attractive implementation option for ad hoc file systems because it is relatively easy to swap in new, specialized implementations for use by applications on a case-by-case basis, as opposed to the current mainstream approach of using general-purpose, system-level file systems, which may not be optimized for specific HPC workloads and must be installed by administrators. In contrast, user-level file systems can be tailored for specific HPC workloads for high performance and can be used by applications without administrator intervention.

Although the benefits of hierarchical storage have been adequately demonstrated, critical questions remain for supporting hierarchical storage systems including ad hoc file systems.

• How should we manage data movement through a storage hierarchy for best performance and resilience of data?

• Are user-level file systems fast enough to be used in HPC systems? And how should we present hierarchical storage systems to user applications, such that they are easy to use and that application code is portable across systems?

• How do the particular I/O use cases mandate the way we manage data?

• Is it possible to reuse components like building blocks when designing user-level file systems?

The Dagstuhl Seminar 17202 "Challenges and Opportunities of User-Level File Systems for HPC" brought together experts in I/O performance, file systems, and storage and collectively explored the space of current and future problems and solutions for I/O on hierarchical storage systems [12].

This paper provides a summary of the Dagstuhl seminar and puts the results into the context of the related work. The paper therefore starts in Section 2 with use cases and continues in Section 3 with a presentation of reusable components as basic building blocks of ad hoc file systems that can simplify the implememtation of ad hoc file systems. Section 4 analyzes different implementation choices to present the ad hoc file system to the client application. Section 5 discusses existing ad hoc file systems and their approaches. Ad hoc file systems also require changes to the HPC infrastructure. It is for example in many cases necessary to synchronize data between the backend file system and the ad hoc file system. This staging process is discussed in Section 6. Section 7 provides a summary of our conclusions.

2 Use Cases for Ad Hoc File Systems

While parallel file systems such as Lustre, GPFS, or BeeGFS have already been serving as reliable backbones for HPC clusters for more than two decades, a need for changes in the HPC storage architecture arose with the arrival of data-intensive applications in HPC. These applications shift the bottleneck from being compute intensive, and thus being restricted by the performance of the CPUs, to being bound by the quantity of data, its complexity, and the speed at which it changes [13].

Node-local SSDs and burst buffers have been introduced into the HPC storage hierarchy to support the new application requirements. This hierarchy level can be used by ad hoc file systems to share data and to provide better performance than that provided by general-purpose storage backends. Ad hoc file systems can be tailored for specific application semantics and can be applied, for example, in the following use cases.

Big Data Workloads. Data processing and analysis have always been important applications for smaller and mid-sized HPC clusters. Researchers, for example from high-energy physics, astronomy, or bioinformatics, have developed community-specific workflow and processing environments that often have been adapted to the specific properties of HPC backend storage systems [14–17]. Using HPC backend storage as primary file systems, however, unfortunately also restricted these big data applications to the drawbacks of centralized storage, for example that bandwidths and IOPS are shared between all concurrently running applications. Big data processing is nevertheless not restricted to HPC. Also, cloud-specific big data environments such as Hadoop or Spark [18, 19] as well as scalable NoSQL databases such as MongoDB or Cassandra [20, 21] attracted researchers to adapt their applications to new and more easily programmmed

environments [22, 23]. Thus, unified environments are needed in order to process both HPC and big data workloads, where the converged environment should keep the promises and benefits of both approaches [24–26]. Ad hoc file systems can help couple locality with a global namespace, while additionally providing the random access rates of node-local SSDs. Nevertheless, in this case the ad hoc file systems must also support long-running campaigns, so that data staging between the backend storage and the backend parallel file system can be reduced to a minimum (see also Section 6).

Bulk-Synchronous Applications. Bulk-synchronous applications are the dominant workload seen on today's HPC systems. Here, applications run in a loosely synchronized fashion, generally synchronizing on major timestep boundaries. At these boundaries, the applications perform collective communication and I/O operations, for example output or visualization dumps and checkpoint/restart. In the general case, the I/O operations per process are independent and written either to per-process files or to process-isolated offsets in a shared file. Additionally, read and write operations occur in bulk phases, without concurrent interleaving of reads and writes.

These behaviors can easily be supported by ad hoc file systems that can provide higher performance than general-purpose file systems. In the shared file case, the ad hoc file system can create a shared namespace across disjoint storage devices, enabling applications with this behavior to use fast storage tiers. Also, because we know that the processes will not read and write concurrently and that each process will write to its own isolated offsets, the ad hoc file system does not need to implement locking around write operations and hence can greatly improve performance.

Checkpoint/Restart. HPC applications can survive failures by regularly saving their global state in checkpoints, which are often stored in the backend parallel file systems. The application can, in case of a failure, restart from the last checkpoint. Especially long-running applications benefit from the ability to restart failed simulation runs. However, the time to take a single application checkpoint increases linearly with the size of the application, and the overall checkpointing overhead increases with the checkpoint frequency. Previous studies have shown that up to 65% of applications' runtimes were spent in performing checkpoints [27, 28], and studies indicate that up to 80% of HPC I/O traffic is induced by checkpoints [29].

Many HPC backend storage systems have been designed according to the peak demands of their checkpointing load. The node-internal SSDs often have a higher peak performance than the backend file system, and the number of node-internal SSDs available for a checkpoint linearly scales with the job's size. Several approaches for using node-internal storage as the checkpoint target have been developed, for example by integrating them into the MPI-IO protocol [30] or by offering dedicated checkpointing libraries [31]. These can be combined with strategies to reduce the checkpoint size, for example by using compression or deduplication [32, 33].

These approaches are partly bound to the usage of MPI or the availability of libraries. An interesting alternative is the use of dedicated ad hoc checkpointing file systems, which can store the checkpoint either in main memory or on node-local SSDs and which then can asynchronously flush (some of the) checkpoints to persistent storage [3]. The asynchronous nature of flushing the checkpoints in a background process even allows first storing the checkpoint locally and later moving the data to storage that cannot be affected by a local failure [30]. One can even overcome network latencies and store a checkpoint at memory speed [3].

Machine and Deep Learning Workloads. The desired input data sizes of machine and deep learning workloads are increasing rapidly. The reason is that the use of small dataset sizes can fail to produce adequately generalized models that can recognize real-world variations in input, such as poses, positions, and scales in images. The typical I/O workload for learning applications is that random samples from the full dataset are read repeatedly from the backend store during training. These random reads from a parallel file system can be a bottleneck, especially for learning frameworks being run on GPU clusters with very high computational throughput. For smaller datasets, the parallel file system cache, or perhaps node-local storage such as an SSD, can hold the entire dataset, and performance is not an issue. Larger datasets, however, may not fit in the file system cache or node-local storage, and the performance of the learning workload can suffer because of the I/O bottleneck [34].

Learning workloads can benefit from ad hoc file systems. One such ad hoc file system could distribute the very large datasets across the memory or storage on other compute nodes of a job and serve the randomly requested input to each process as needed. Another file system implementation could prefetch the randomly se-

lected portions of the dataset from the parallel file system to the compute nodes if the dataset cannot fit in node-local storage. In both cases, the learning workload would see vast improvements in I/O performance resulting in better training throughput.

Producer-Consumer Workloads. Several variations of producer-consumer workloads on HPC systems exist. Perhaps the most canonical of these is found in climate model codes, for example in the Energy Exascale Earth System Modeling E3SM [35], where different physical components are modeled in individual executables (e.g., land, ocean, atmosphere, or ice). The individual component executables consume data files as input and produce output files that can be in turn consumed by another component executable. A common workflow for these models is to run them concurrently and have the components produce and consume files to compute the overall simulation.

Another emerging example of producer-consumer workloads for HPC is data analytics [36]. Here, traditional HPC simulations produce simulation output that is read in and analyzed by processes in the same allocation, for example coupled simulation and machine learning tasks for climate analytics [37]. The analysis could be done in situ, as a library linked into the application or an executable running on the same compute node, or simply coscheduled in the allocation on separate compute nodes. The analysis processes can perform tasks such as feature extraction or machine learning.

In both these producer-consumer workloads, the workflow can benefit from ad hoc file systems that are able to facilitate the sharing of data between the components in the job without resorting to using the parallel file system. The ad hoc file system can keep the data on fast, local storage and manage moving the bytes as appropriate for best performance with respect to the producer and consumer components. For example, writing bytes to the local storage of the producer will result in best performance from the producer's perspective. When it is time for the consumer to read the bytes, however, the file system could move the file contents to the compute node where the consumer is running, for best read performance.

3 Reusable Components

Ad hoc file system implementations can be specialized for a variety of use cases (Section 2) and interfaces (Section 4). Despite this specialization, however, each ad hoc file system will encounter a common set of underlying technical challenges posed by HPC platforms and HPC application workloads. These challenges include the following:

- HPC network fabric compatibility;
- storage device compatibility;
- maximizing concurrency;
- minimizing resource consumption.

Reusable building block components can help address these challenges while improving file system developer productivity, reducing software maintenance cost, and enhancing portability. In this section, we highlight each of these challenges and survey the state of the art in reusable components to address them.

3.1 HPC Network Fabric Compatibility

Ad hoc file systems that operate "in-system" must, by definition, interact with the system's HPC network fabric. HPC networks are characterized by low latency (to accommodate tightly coupled computations), RDMA access (to minimize CPU impact on data transfers), and low message loss (because the environment tends to be static and homogeneous). No single standard for HPC network hardware exists that meets these requirements, however. Many of the world's most powerful computers use different network technologies [38–40], each with its own distinct API and optimization strategy. Data services must therefore employ network abstraction layers to ensure portability.

TCP/IP sockets are the most widely-used and most portable network abstraction model, but they lack the specialized features needed to accommodate the latency, RDMA, and message loss characteristics of a dedicated HPC network. The Message Passing Interface (MPI) provides a network abstraction and programming model that is directly tailored to HPC environments [41]. However, MPI was designed for application-level use and is not readily applicable to system services in practice [42]. More generalized HPC network fabric abstractions such as OFI/libfabric [43], UCX [44], and Portals [45] are not tied to MPI semantics or programming models and are thus more appropriate foundational building blocks for ad hoc file system implementations.

Remote procedure call frameworks can be used in conjunction with network abstractions to further ease the task of constructing ad hoc file system services. Examples in general-purpose distributed computing in-

clude gRPC① and Apache Thrift②, while examples in HPC include Mercury [46] and Nessie [47]. RPC frameworks implement common client/server functionality such as request/response matching, protocol encoding, service handler invocation, and programmatic API bindings.

3.2 Storage Device Compatibility

Storage devices are the second crucial resource that ad hoc file systems must manage. Parallel and distributed file systems have long utilized local file systems such as EXT4 and XFS as the abstraction point between distributed service daemons and local storage resources. Local file system abstractions are still a valid and highly portable design choice, but the emergence of new storage device technologies has prompted renewed exploration of alternative interfaces (see also Section 4).

Persistent memory devices in particular have more in common with dynamic memory than they do with rotating magnetic media and can thus be accessed more efficiently through direct user-space load/store operations than through indirect kernel-space block device and page cache operations. This property has led to the creation of storage abstractions such as the Persistent Memory Developers Kit (PMDK)③ that provide simple transactional storage primitives atop memory-mapped devices rather than block devices.

Faster block device interfaces can also benefit from lower-latency access paths. For example, the Storage Performance Development Kit (SPDK)④ provides an alternative interface to NVMe devices that relies on user-space poll-driven device drivers to minimize latency. PMDK and SPDK are intended for use with two different storage technologies, but they share the common goal of minimizing access cost for devices that do not conform to the design assumptions and performance characteristics of legacy hard drives.

3.3 Maximizing Concurrency

The following are three major drivers of concurrency in ad hoc file system services:

- request arrival rate from parallel applications;
- availability of multicore processors on service nodes;
- storage resources that require parallel request issue to maximize bandwidth.

Balancing these factors while limiting implementation complexity is a daunting task. Several techniques and strategies are available to help, however. Services can leverage an asynchronous event-driven model using frameworks such as Seastar⑤ or libraries such as libevent⑥ or libev⑦, and storage device concurrency can be achieved by offloading operations to asynchronous APIs such as the POSIX asynchronous I/O interface or the SPDK framework described in subsection 3.2.

Multithreading can also be used to manage concurrency without adopting an event-driven programming model. Conventional POSIX threads are functionally effective but introduce excessive overhead when concurrency exceeds the number of compute cores available on a system. User-level threads seek to find a middle ground between the efficiency of event-driven frameworks and the programmability of POSIX threads. Examples of modern user-level threading packages include Qthreads [48], MassiveThreads [49], and Argobots [50]. Argobots is particularly amenable to distributed service development because it supports customized schedulers and flexible mappings of work units to compute resources.

3.4 Minimizing Resource Consumption

The term “ad hoc file system” implies that the file system does not necessarily have dedicated resources but is instead provisioned on-demand within an existing system. An ad hoc file system must therefore be able to coexist with other services or even application processes without dominating available resources or causing excessive jitter. Common resource consumption pitfalls include the following:

- memory consumption;
- busy polling of network resources;
- CPU or NUMA contention.

No single component solves these challenges; they

① https://grpc.io/, Nov. 2019.
② https://thrift.apache.org/, Nov. 2019.
③ https://pmem.io/pmdk/, Nov. 2019.
④ https://spdk.io/, Nov. 2019.
⑤ http://seastar.io/, Nov. 2019.
⑥ https://libevent.org/, Nov. 2019.
⑦ http://software.schmorp.de/pkg/libev.html, Nov. 2019.

are more readily addressed by design patterns that account for the capabilities of HPC hardware resources. For example, non-volatile memory can be combined with RDMA-capable networks (either in user space or via kernel drivers) to limit memory consumption by transferring data directly between network and storage without intermediate buffering. RPC frameworks and network abstraction layers can use adaptive polling strategies to limit network and CPU use when services are idle. CPU affinity and NUMA control can prevent colocated services and applications from contending for CPU and memory resources. These design patterns can be used in any ad hoc file system implementation.

3.5 Mochi Project

Mochi [51] seeks to combine many of the components and best practices described in this subsection into a coherent environment to accelerate data service development. This type of environment also enables utility libraries to *span* components in order to implement complementary best practices. This is exemplified by the Margo [52] and abt-io [50] libraries that integrate reusable RPC and file I/O functionality into the Argobots threading framework.

Regardless of development environment and runtime system, however, the core principles of reusable components and design patterns can be applied to accelerate the development of ad hoc file systems and allow their creators to spend more time on data service innovation.

4 Interfacing Ad Hoc File Systems

Since the arrival of parallel file systems over two decades ago, file systems have continued to become increasingly more complex, spanning intricate logic over million of lines of code [53] with the goal of offering a general-purpose solution for all applications (see Section 2). In order to allow for a familiar and portable interface that most applications can agree and rely on, such general-purpose file systems follow a set of rules in the POSIX family of standards that define the syntax and semantics of the I/O interface, also known as *POSIX I/O*. As a result, the POSIX I/O interface is deeply embedded within the operating system and common to users and applications, usually accessed via the GNU C library (glibc). Note that these application-oriented interfaces should not be confused with other interfaces such as block I/O, object-oriented I/O, and file-based I/O, which are defined for lower-level storage media access.

The I/O interface can be implemented at the user level or the kernel level. As shown in Fig.1, kernel-based file systems such as EXT4 may work entirely within the kernel, or they can utilize an additional user-space implementation where application I/O calls are redirected to, for example, FUSE. The kernel-based ap-

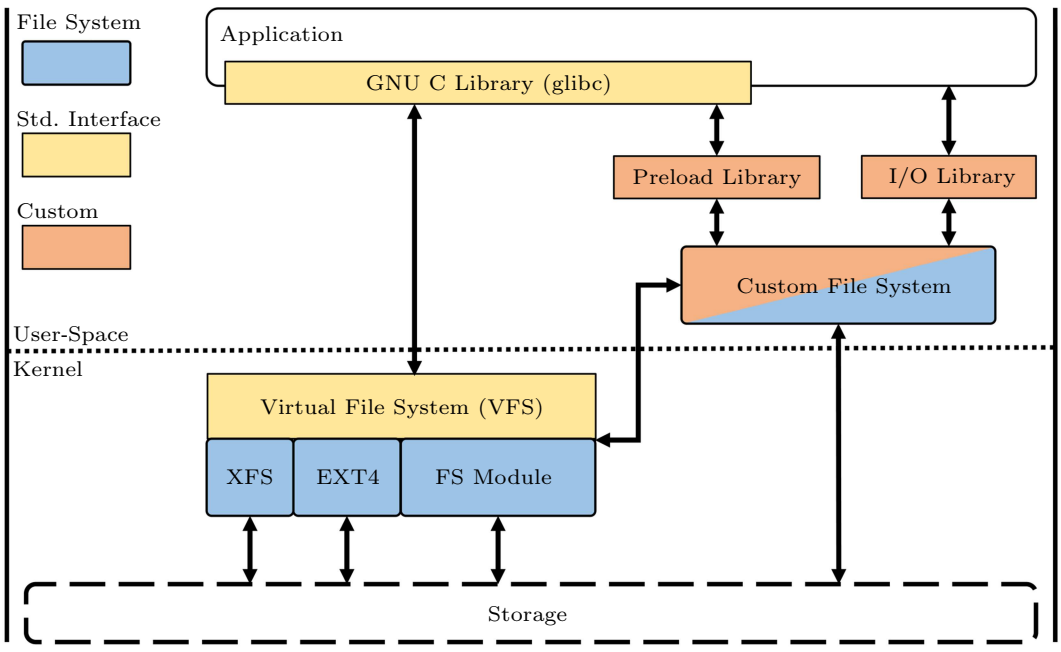

Fig.1. Techniques to interface file systems with their components.

proach works by registering the file system to the virtual file system (VFS), which can be accessed by standard libraries. Exclusive user-space file systems, on the other hand, can also operate without a kernel component, providing their own implementation by *preloading* a library through the `LD_PRELOAD` environment variable, which intercepts defined I/O calls and redirects them to the user-space file system. These types of file systems are becoming increasingly popular (including in ad hoc file systems) because of their advantages for code development, porting, debugging, maintenance, and often performance.

An alternative to the standard I/O interface is proposed by so-called I/O libraries or I/O wrappers, providing a thinner set of API functions that may ease deployment and maintenance. Such libraries are not necessarily backed by a user-space file system and align access patterns with the capabilities of the underlying PFS (e.g., ADIOS [54]). They can also be used as an API to a separate user-space file system such as OrangeFS [55].

In this context, the key-value interface has recently gained traction. Volos *et al.* [56] developed a flexible file system architecture called Aerie that can support user-level I/O accesses to storage-class memory. Nonetheless, a completely non-standardized custom interface usually requires the modification of applications and is not suitable in all cases.

In the following, we discuss some of the existing techniques to implement user-space file systems, and we highlight the most popular approach when developing an ad hoc file system in user space.

4.1 User-Space File Systems

User-space file systems that are interfaced via the traditional standard interface cannot be easily registered to an operating system compared with kernel-based file systems. Instead, exclusive user-space file systems are directly linked to an application at compile time or loaded at runtime, which on UNIX systems can be achieved by a preloading library. The latter technique is most popular and has been adopted by several ad hoc file systems, such as CRUISE [3], BurstFS [11], GekkoFS [10], and DeltaFS [57]. They all intercept the application I/O via a set of wrapper functions that are implemented in the form of a user-level preloading library, albeit not always being fully POSIX-compliant [10].

In addition to easing development or maintenance by using such a library, avoiding the kernel when calling I/O functions can yield significant performance benefits in terms of I/O throughput and latency. Volos *et al.* [56] argued that the existing kernel-based stack of components, although well suited for disks, unnecessarily limits the design and implementation of file systems for faster storage (e.g., storage-class memory). Their Aerie framework, a POSIX-like file system in user space, has been implemented with performance similar to or better than a kernel implementation.

PMDK bypasses the kernel and builds on Linux's DAX features, which allow applications to use persistent memory as memory-mapped files. Therefore, techniques such as `LD_PRELOAD` are attractive choices for ad hoc file systems, which often utilize flash-based node-local storage. With more advanced future storage technologies (e.g., persistent memory), reducing the time spent within the operating system to increase I/O performance is going to become even more critical.

Because of the ability to intercept any function in preloading libraries, however, all I/O functions used by an application must be implemented and reinterpreted in the user-space file system. Hence, the more complex an application, the more the functions required in the file system for the application to work, potentially intercepting a large percentage of functions that are part of glibc, for instance. The *system call intercepting library*[8] (*syscall_intercept*), as part of the pmem project, aims to solve this challenge by providing a low-level interface for hooking Linux system calls in user space while still using the established `LD_PRELOAD` method. This can dramatically reduce the number of functions that need to be implemented in the user-space file system because the set of functions is limited to only system calls, such as `sys_mknod` or `sys_write`.

Other efforts have also explored ways to standardize the interception of POSIX-related I/O calls, including *libsysio*[9] by Sandia National Laboratories and *Gotcha*[10] by Lawrence Livermore National Laboratory. Libsysio provides a POSIX-like interface that redirects I/O function calls to file systems and supports the conventional VFS/vnode architecture for file-based accesses [58]. Gotcha is a wrapper library that works

[8] https://github.com/pmem/syscall_intercept, Nov. 2019.
[9] https://github.com/hpc/lustre/tree/master/libsysio, Dec. 2019.
[10] https://github.com/LLNL/GOTCHA, Nov. 2019.

similar to `LD_PRELOAD` but operates via a programmable API.

4.2 FUSE File Systems

The FUSE (Filesystem in Userspace) [59, 60] framework is another popular approach when developing user-space file systems. It consists of two components, the FUSE kernel module and the *libfuse* user-space library, that support 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 and communicates through the FUSE kernel module with the kernel. 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-level file systems [59]. FUSE therefore provides a traditional file system interface, but it allows users to quickly implement a file system based on FUSE's API by avoiding much of the complexity within the kernel. As a result, a large number of FUSE-based file systems have been developed, including ChunkFS [61], SSHFS [62], FusionFS [63], and GlusterFS [64].

While convenient, FUSE-based user-level file systems also face several drawbacks inherent to its architecture. Since the *libfuse* library is typically executed as a separate user process, the communication round trip between an application and the FUSE process leads to nontrivial performance overhead [60]. To make the situation worse, an application leveraging the kernel module for I/O interception will experience even more overhead due to context switches across the user-kernel boundary. Furthermore, root permission is required to mount FUSE file systems, or system administrators have to give nonprivileged users the ability to mount FUSE file systems. In HPC environments, users do not typically own such superuser privileges and cannot easily mount the desired FUSE file systems without help from system administrators.

In ad hoc file systems, FUSE's disadvantages outweigh the advantage of convenient development, and it is therefore rarely used in these file systems.

5 Ad Hoc File System Implementations

The previous two sections have discussed basic building blocks of ad hoc file systems and how to interface them. This section now puts these components together and presents three different ad hoc file system implementations for different use cases.

• BeeGFS-On-Demand, or BeeOND for short, is a production-quality BeeGFS wrapper that simplifies the deployment of multiple independent BeeGFS instances on one cluster to aggregate the performance and capacity of internal SSDs or hard disks. Important aspects of BeeOND clients are developed as kernel modules.

• GekkoFS has been designed to overcome scalability limitations of the POSIX protocol [10]. It both slightly relaxes the POSIX semantics and reduces security guarantees, while still being able to ensure confidentiality in the setting of ad hoc file systems. Performance is achieved by spreading data and metadata as widely as possible.

• The Burst Buffer File System (BurstFS) uses techniques such as scalable metadata indexing, colocated I/O delegation, and server-side read clustering and pipelining to support scalable and efficient aggregation of I/O bandwidth from node-local storage [11]. Clients furthermore write to node-local logs to increase write performance, providing a different approach from that of GekkosFS and being suited for different applications.

All file systems have been developed to create temporary file systems that run for the duration of a compute job. Hence, the ad hoc file systems must be integrated with a workload manager such as Slurm or Torque, and the deployment time must be as short as possible so that the runtime of the compute job is not negatively affected by the start-up phase of the file system.

5.1 BeeGFS and BeeOND

BeeGFS, initially named *Fraunhofer Parallel File System* (FhGFS), was started in 2004 as a result of a project to integrate the Lustre PFS in a video streaming environment and as storage in an existing 64-node Linux cluster. The initial requirements for the development were to distribute metadata, to require no kernel patches, and to support zero-config clients with the goal of providing a scalable multithreaded architecture and dynamic failover for native InfiniBand and Ethernet.

The file system is built to run on any underlying POSIX-compliant local file system, providing users the choice and flexibility if only a specific local file system is available. BeeGFS is designed to be easily deployed and maintained while focusing on performance instead of on features. The client is built as a Linux kernel module, while other software components were moved to user space, allowing increased usage flexibility.

Staying POSIX-compliant in a distributed world without sacrificing performance throughout the years is a challenging process, requiring detailed analyses and careful implementation. Today, BeeGFS has become a true high-performance distributed file system that includes a reliable and fast distributed locking algorithm. After five years of heavy development, BeeGFS was ready for installation in larger environments, delivering the highest single-thread performance on the market up to this day and showing its strengths in $N : 1$ shared file I/O cases (see Fig.2).

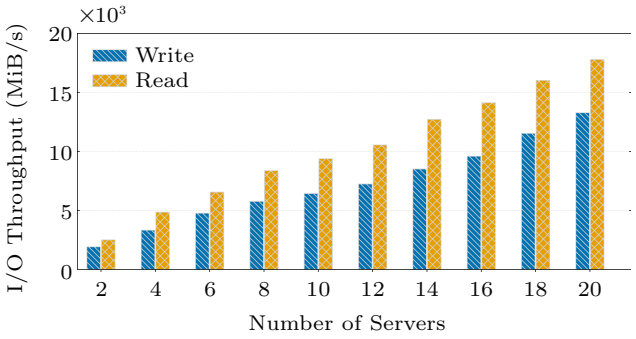

Fig.2. BeeGFS's I/O throughput for increasing server numbers for 192 client processes on one shared file.

Fig.3 describes BeeGFS' overall architecture. The client works as a kernel module with its own caching strategy, which is kept updated and follows the latest kernel developments. The metadata server, the storage server, and the management server are independent processes that can be installed on a single server or in a distributed setting, depending on the needs of the user.

When a QDR InfiniBand system (with SSDs in each node) was installed at Fraunhofer in 2012, BeeOND was born out of the idea to use BeeGFS as an ad hoc file system on nodes that a user got assigned to by the cluster's batch system. BeeOND creates an empty, private distributed file system across all job nodes to separate challenging I/O patterns from the I/O load of the PFS. The file system is then started during the prolog process of the batch system. Using BeeGFS for this task was ideal because all server processes are already running in user-space and do not require further patches to be installed in advance. One of its first use cases was a data-sorting routine for seismic data that reads data from the PFS and uses the SSD storage as an intermediate data buffer before the sorted data is written back to the PFS. BeeGFS supports advanced features such as data mirroring, because of the high availability requirements in hyperconverged solutions to store large datasets economically, for instance.

Today, BeeOND is used around the world as an alternative to expensive burst buffers and allows users to manage the demanding requirements of deep learning applications. It inherits all the functionality of BeeGFS and offers a POSIX-compliant and scalable distributed file system. The installation at Tsubame 3.0 and the ABCI system in Japan are the most prominent

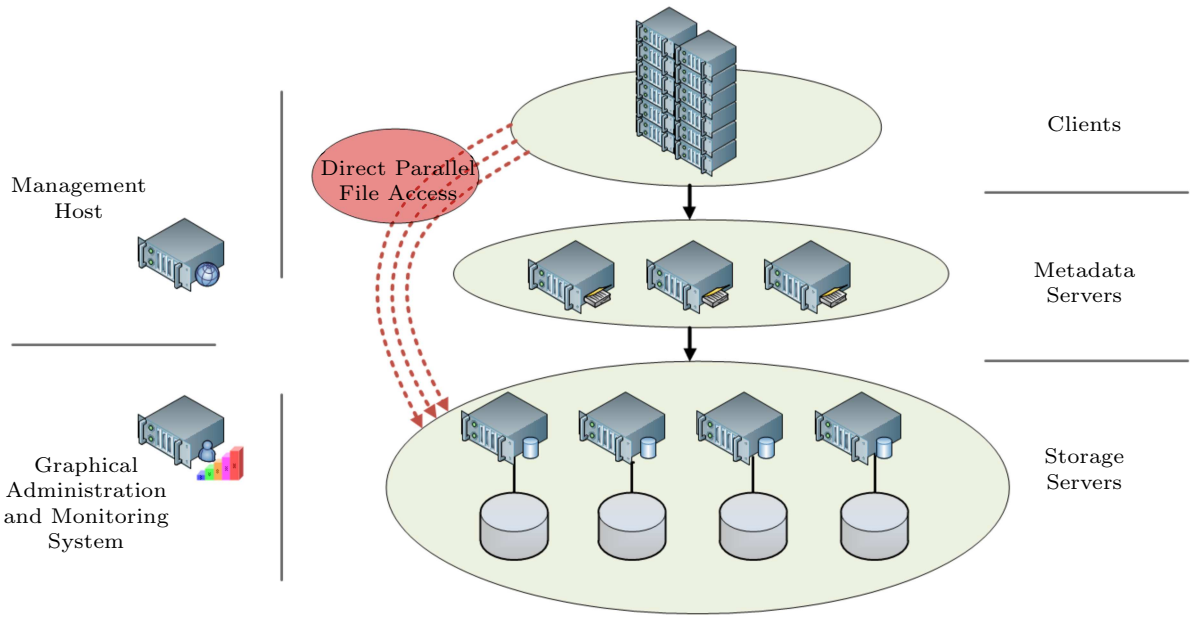

Fig.3. Architecture of the BeeGFS file system.

BeeOND installations and can reach 1 TB/s streaming throughput. In the future, BeeGFS and BeeOND will be continuously developed to improve performance and to satisfy the needs of arising use cases.

5.2 GekkoFS

The basic design idea behind GekkoFS is to distribute data and metadata among cluster nodes as evenly as possible [10]. GekkoFS therefore aggressively uses hashing to distribute data and metadata. Each file inode is managed by one cluster node, which can be determined by simply calculating a hash function of the file path and name and then by taking the result modulo the number of nodes participating in the file system.

A client node, for example, creates a new file by first computing the managing cluster node and then by running the file create protocol between the client and the managing node. The serialization inside the managing node guarantees that an existing file cannot be created a second time. The protocol ensures that most metadata operations linearly scale in the number of participating cluster nodes.

Distributing inode metadata over all cluster nodes nevertheless also requires a different directory handling. GekkoFS therefore significantly relaxes the POSIX directory semantics. Directories are created similar to files, while the content of a directory is only implicitly available by collecting distributed inode entries using broadcast operations. GekkoFS is therefore not suited for applications that regularly require listing the content of directories using `ls`-operations or which rename directories, since this requires expensive one-to-all and all-to-one communication patterns and in the second case also requires updating all distributed inode entries. Fortunately, studies have shown that these operations are extremely rare while running a parallel job [65, 66].

GekkoFS provides the same consistency as POSIX for any file system operation that accesses a specific file. These include read and write operations as well as any metadata operation that targets a single file, for example, file creation. Nevertheless, similar to PVFS [67], GekkoFS does not provide a global byte-range lock mechanism. In this sense, applications are responsible for ensuring that no conflicts occur, in particular w.r.t. overlapping file regions, in order to avoid complex locking within the file system.

The GekkoFS architecture shown in Fig.4 consists of two main components: a client library and a server process. An application that uses GekkoFS must first preload the client interposition library that intercepts all file system operations and forwards them to a server (GekkoFS daemon), if necessary. 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 of each other.

The client consists of an interception interface that catches relevant calls to GekkoFS and forwards unrelated calls to the node-local file system, a file map that manages the file descriptors of opened files and directories, independently of the kernel, and an RPC-based

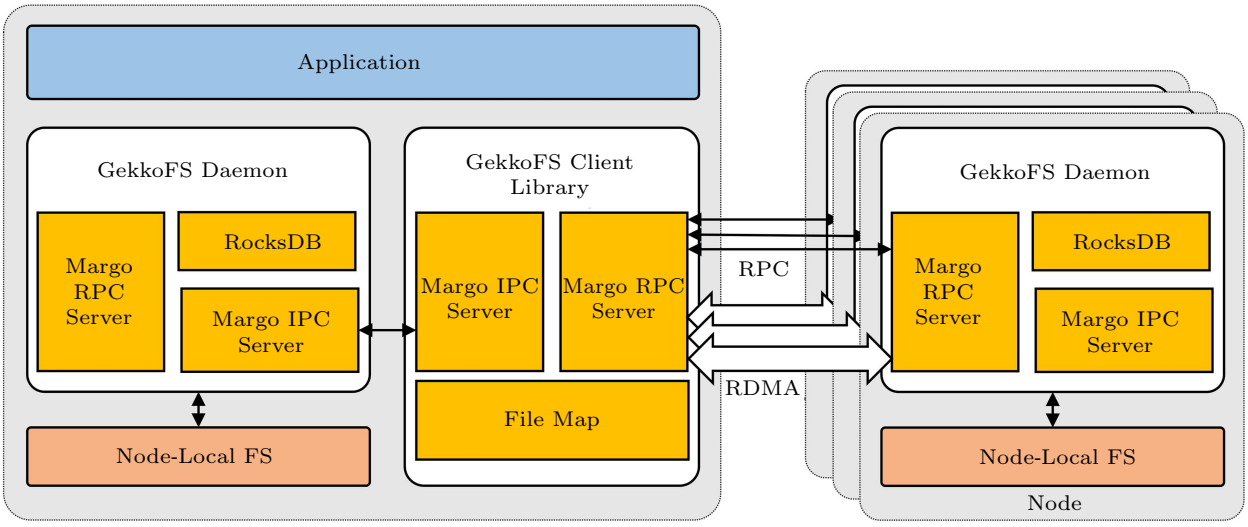

Fig.4. GekkoFS architecture.

communication layer that forwards file system requests to local/remote GekkoFS daemons.

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 use a RocksDB [68] key-value store (KV store) for handling metadata operations, an I/O persistence layer that reads/writes data from/to the underlying node-local storage system, and an RPC-based communication layer that accepts local and remote connections to handle file system operations.

The communication layer uses the Mercury RPC framework, which allows GekkoFS to be independent from the network implementation [46]. Mercury is interfaced indirectly through the Margo library, which provides wrappers to Mercurys API with the goal of providing a simple multithreaded execution model [52]. It uses the lightweight, low-level threading and tasking framework Argobots, which has been developed to support massive on-node concurrency.

The experiments for the results presented in Fig.5 and Fig.6 have been performed on the MOGON II cluster at the Johannes Gutenberg University Mainz, Germany. The cluster 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 (two sockets each). The Intel® Broadwell processors have been used in all presented experiments. The main memory capacity inside the nodes ranges from 64 GiB to 512 GiB, and the nodes are connected by a 100 Gbit/s Intel Omni-Path interconnect. The cluster is attached to a 7.5 PiB Lustre storage backend.

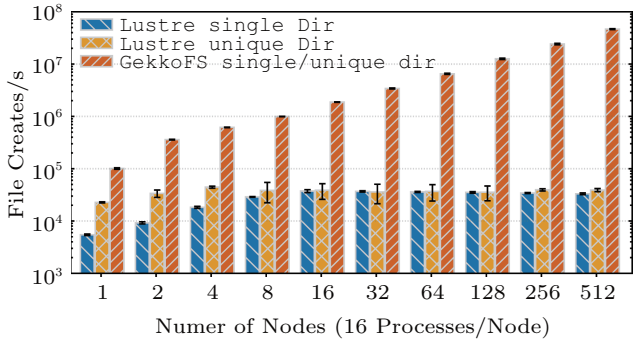

Fig.5. GekkoFS's file create performance compared with a Lustre file system for increasing the number of nodes.

In addition, each node includes an Intel® SATA SSD DC S3700 Series with 200 GiB or 400 GiB, which has been used for storing data and metadata of GekkoFS. All Lustre experiments were performed on a Lustre scratch file system with 12 object storage targets (OSTs), 2 object storage servers (OSSs), and 1 metadata service (MDS) with a total of 1.2 PiB of storage. The experiments were run at least five times with each data point representing the mean of all iterations.

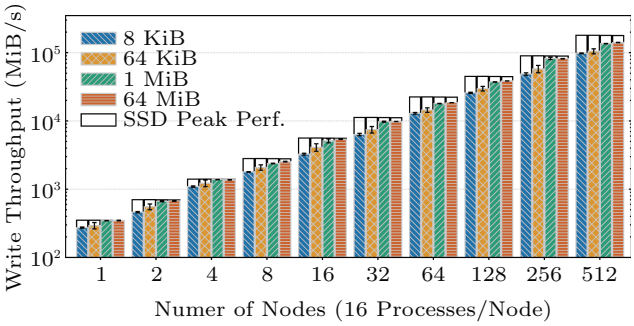

Fig.6. GekkoFS's write throughput for increasing the number of nodes.

Fig.5 compares GekkoFS with Lustre for file creates for up to 512 nodes on a logarithmic scale. GekkoFS's workload was scaled with 100 000 files per process. Lustre's workload was fixed to 4 million files for all experiments. We fixed 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, Lustres metadata performance was evaluated while the system was accessible by other applications as well.

GekkoFS outperforms Lustre by a large margin, regardless of whether Lustre processes operated in a single directory or in isolated directories. GekkoFS achieved around 46 million creates per second, while each operation was performed synchronously without any caching mechanisms in place, showing close to linear scaling. Lustre's create performance did not scale beyond approximately 32 nodes and has been ∼1 405x lower than the create-performance of GekkoFS, demonstrating the well-known metadata scalability challenges of general-purpose PFS.

GekkoFS's data performance is not compared with the Lustre scratch file system because Lustre's peak performance of 12 GByte/s is already reached for 10 nodes for sequential accesses. Moreover, Lustre has shown to scale linearly for sequential access patterns in larger deployments with more OSSs and OSTs being available [69]. Fig.6 shows GekkoFS's sequential I/O

throughput in MiB/s for an increasing number of nodes for different transfer sizes. In addition, each data point is compared with the peak performance that all aggregated SSDs could deliver for a given node configuration, visualized as a white rectangle. The results demonstrate GekkoFS's 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) for write and read operations with a transfer size of 64 MiB for 512 nodes. At 512 nodes, this translates to more than 13 million write IOPS and more than 22 million read IOPS, while the average latency can be bounded by at most 700 µs for file system operations with a transfer size of 8 KiB.

5.3 Burst Buffer File System (BurstFS)

BurstFS shares a number of basic design considerations with GekkoFS. It has been designed to have the same temporary life cycle as a batch-submitted job, while it uses node-local burst buffers to improve applications' read and write performance [11]. The main distinction between the two ad hoc file systems is that BurstFS clients always write to local storage in a log-structured way. This can significantly improve write performance since no network latency is involved; but it also requires building a metadata directory to reconstruct writes coming from multiple clients to one file in case of $N-1$ access file patterns.

When a batch job is allocated over a set of compute nodes, an instance of BurstFS will be constructed on the fly across these nodes, using the locally attached burst buffers, which may consist of memory, SSDs, or other fast storage devices. These burst buffers enable very fast log-structured local writes; in other words, all processes can store their writes to the local logs. Next, one or more parallel programs launched on a portion of these nodes can leverage BurstFS to write data to or read data from the burst buffers.

BurstFS is mounted with a configurable prefix and transparently intercepts all POSIX functions under that prefix. Data sharing between different programs can be accomplished by mounting BurstFS using the same prefix. Upon the unmount operation from the last program, all BurstFS instances flush their data for data persistence (if requested), clean up their resources, and exit.

BurstFS uses MDHIM as distributed KV store (KVS) for metadata, along with log-structured writes for data segments [70]. Fig.7 shows the organization of data and metadata for BurstFS. Each process stores its data to the local burst buffer as data logs, which are organized as data segments. New data are always appended to the data logs. With such log-structured writes, all segments from one process are stored together regardless of their global logical position with respect to data from other processes.

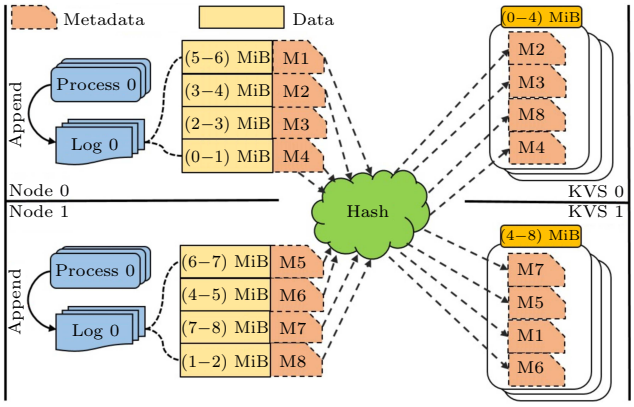

Fig.7. Diagram of the distributed key-value store for BurstFS.

When the processes in a parallel program create a global shared file, a key-value pair (e.g., M1 or M2) is generated for each segment. A key consists of the file ID (8-byte hash value) and the logical offset of the segment in the shared file. The value describes the actual location of the segment, including the hosting burst buffer, the log containing the segment (there can be more than one log from multiple processes on the same node), the physical offset in the log, and the length. The key-value pairs for all the segments then provide the global layout for the shared file. All KV pairs are consistently hashed and distributed among the key-value servers (e.g., KVS0 and KVS1). With such an organization, the metadata storage and services are spread across multiple key-value servers.

Lazy synchronization provides efficient support for bursty writes. Each process holds a small memory pool for metadata KV pairs from write operations, and, at the end of a configurable interval, KV pairs are periodically stored to the key-value stores. An `fsync` operation can force an explicit synchronization. BurstFS leverages the batch put operation from MDHIM to transfer these KV pairs together in a few round trips, minimizing the latency incurred by single put operations. During the synchronization interval, BurstFS searches for contiguous KV pairs in the memory pool to combine, which can span a bigger range and reduce the number of data segments. As shown in Fig.7, segments [2-3) MiB and [3-4) MiB are contiguous and map to the

same server; therefore their KV pairs are combined into one.

Read operations involve a metadata look-up for the distributed data segments. Thus, they search for all KV pairs whose offsets fall in the requested range. With batched read requests, BurstFS needs to search for all KV pairs that are targeted by the read requests in the batch. However, range queries are not directly supported by MDHIM and indirectly performing them by iterating over consecutive KV pairs induces additive round-trip latencies.

BurstFS therefore includes parallel extensions for both MDHIM clients and servers [11]. On the client side, incoming range requests are broken into multiple small range queries to be sent to each server based on consistent hashing [71]. Compared with sequential cursor operations, this extension allows a range query to be broken into many small range queries, one for each range server. On the server side, for the small range query within its scope, all KV pairs inside that range are retrieved through a single sequential scan in the key-value store.

Scalable read and write services are furthermore achieved through a mechanism called co-located I/O delegation. BurstFS launches an I/O proxy process on each node, a delegator. Delegators are decoupled from the applications in a batch job and are launched across all compute nodes. The delegators collectively provide data services for all applications in the job by offering a request manager and an I/O service manager. In this way, a conventional client-server model for I/O services is transformed into a peer-to-peer model among all delegators.

The delegators allow BurstFS to leverage the existing techniques of batched reads from the client side, where POSIX commands such as `lio_listio` allow read requests to be transferred in batches. BurstFS exploits the visibility of read requests at the server side for further performance improvements by introducing a mechanism called server-side read clustering and pipelining (SSCP) in the I/O service manager. In the two-level request queue, SSCP first creates several categories of request sizes, ranging from 32 KiB to 1 MiB (see Fig.8). Incoming requests are inserted into the appropriate size category either individually or, if contiguous with other requests, combined with the existing contiguous requests and then inserted into the suitable size category. As shown in the figure, two contiguous requests of 120 KiB and 200 KiB are combined by the service manager. Within each size category, all requests are queued based on their arrival time. A combined request will use the arrival time from its oldest member. For best scheduling efficiency, the category with the largest request size is prioritized for service. Within the same category, the oldest request will be served first. BurstFS enforces a threshold on the wait time of each category (default 5 ms). If any category has not been serviced longer than this threshold, BurstFS selects the oldest read request from this category for service.

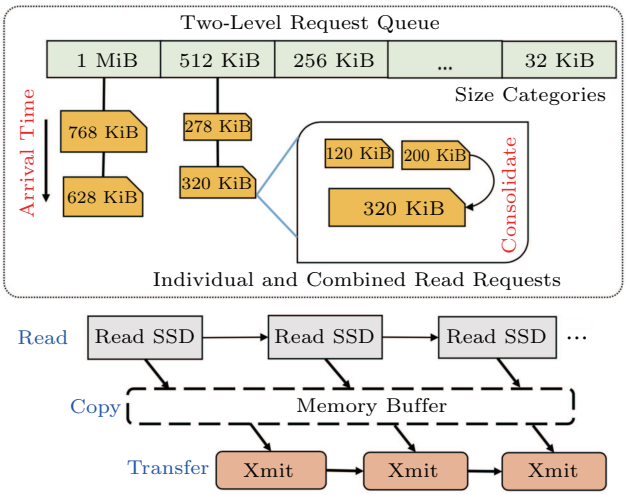

Fig.8. Server-side read clustering and pipelining.

Experiments comparing BurstFS with OrangeFS 2.8.8 [55] and the Parallel Log-Structured File System 2.5 (PLFS) [72] have been conducted on the Catalyst cluster at Lawrence Livermore National Laboratory, consisting of 384 nodes. Each node is equipped with two 12-core Intel® Xeon® Ivy Bridge E5-2695v2 processors, 128 GB of DRAM, and an 800 GB burst buffer comprising PCIe SSDs.

In the experiments, OrangeFS instantiated server instances across all the compute nodes allocated to a job to manage all node-local SSDs. PLFS is designed to accelerate N–1 writes by transforming random, dispersed N–1 writes into sequential N–N writes in a log-structured manner. Data written by each process is stored on the backend PFS as a log file. In the experiments, OrangeFS over node-local SSDs has been used as the PLFS backend. In PLFS with burst buffer support, processes store their metalinks on the backend PFS, which point to the real location of their log files in the burst buffers. Within the experiments, each process wrote to its node-local SSD, and the location was recorded on the center-wide Lustre parallel file system.

Fig.9 compares the write bandwidth with the PLFS burst buffer (PLFS-BB), PLFS, and OrangeFS. Sixteen

processes are placed on each node, each writing 64 MB data following an N–1 strided pattern. Both BurstFS and PLFS-BB scale linearly with process count. The reason is that processes in both systems write locally and the write bandwidth of each node-local SSD is saturated. OrangeFS and PLFS also scale linearly, while their bandwidths increase at a much slower rate. The reason is that both PLFS and OrangeFS stripe their files across multiple nodes, which can cause degraded bandwidth due to contention when different processes write to the same node. On average, BurstFS delivers 3.5x the performance of OrangeFS and 1.6x the performance of PLFS.

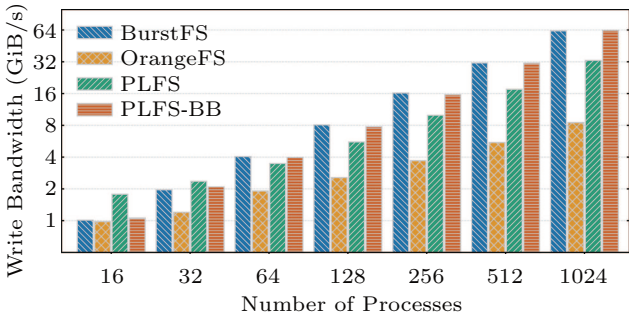

Fig. 9. Comparison of BurstFS with PLFS and OrangeFS for $N-1$ segmented writes.

PLFS initially delivers a higher bandwidth than BurstFS at small process counts. PLFS internally transforms the N–1 writes into N–N writes. However, when `fsync` is called to force these N–N files to be written to the backend file system, OrangeFS does not completely flush the files to the SSDs before `fsync` returns.

In order to evaluate the support for data sharing among different programs in a batch job, read tests with IOR were conducted. A varying number of processes read a shared file written by another set of processes from a Tile-IO program. Processes in both MPI programs were launched in the same job. Each node hosted 16 Tile-IO processes and 16 IOR processes. Once Tile-IO processes completed writing on the shared file, this file was read back by IOR processes using the N–1 segmented read pattern. The same transfer sizes were used for IOR and Tile-IO. Since the read pattern did not match the initial write pattern of Tile-IO, each process needed to read from multiple logs on remote nodes. The size of each tile was fixed to 128 MiB, and the number of tiles along the y axis to 4, while the number of tiles along the x axis was increased with the number of reading processes.

Fig.10 compares the read bandwidth of BurstFS with those of PLFS and OrangeFS. Both PLFS and OrangeFS are vulnerable to small transfer size (32 KiB). BurstFS maintains high bandwidth because of locally combining small requests and server-side read clustering and pipelining. On average, when reading data produced by Tile-IO, BurstFS delivers 2.3x and 2.5x the performance of OrangeFS and PLFS, respectively.

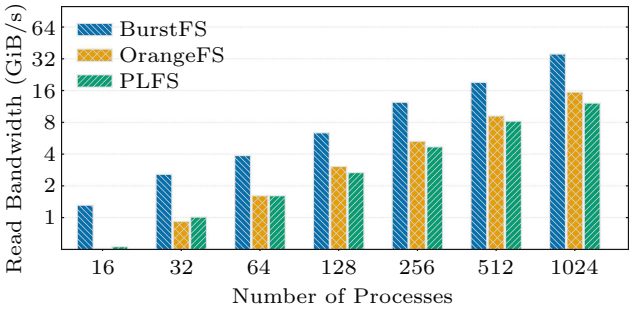

Fig.10. IOR read bandwidth on a shared file written by Tile-IO.

6 Feeding Ad Hoc File Systems: Data Staging

Ad hoc file systems can significantly speed up individual data accesses for an application when compared with a backend PFS, given that they allow exploiting faster node-local storage hardware such as NVRAM, and also since they provide application-specific data distributions and access semantics that can improve application I/O latency and/or bandwidth. Nonetheless, for ad hoc file systems to be useful, input data must be transferred (or "staged") into the file systems before running the targeted application and output data must be staged out after the application terminates. Since ad hoc file systems are ephemeral in nature, such *data staging* is typically done from/to the backend PFS. This section discusses requirements for coupling a batch scheduler, the backend PFS, and the ad hoc file system during stage-in and stage-out activities. Additionally, it discusses the potentially positive and negative interactions between concurrent jobs on a system while these stage-in and stage-out activities occur.

Data-intensive workloads are challenging for I/O subsystems because they present substantially larger I/O requirements than those of traditional compute-bound workloads [11, 73, 74]. Thus, even if modern HPC clusters can run multiple concurrent applications on top of millions of cores, severe I/O performance degradation is often observed because of cross-application interference [75, 76], a phenomenon that originates due to competing accesses to the supercomputer's shared

resources. Nevertheless, the inclusion of fast node-local storage in compute nodes (see Table 1) enables the creation of a high-performance distributed staging layer where applications can efficiently store and retrieve data in isolation from each other which, if done correctly, can help reduce cross-application interference. Ad hoc file systems can become very useful in this scenario, both as enforcers of this isolation and as mediators for applications to transparently access this staging layer. Compared with traditional approaches, where applications directly access the PFS at their convenience, such a *staging architecture* has the advantage that arbitrary application I/O workloads would be substituted by system-controlled stage-in/stage-out I/O workloads, which could be scheduled to minimize interference between concurrent staging phases. Moreover, from the point of view of the PFS, global application I/O would be transformed from a stream of unrelated, random data accesses to a well-defined series of sequential read/write phases, which are better suited to be optimized.

Nonetheless, despite these benefits to HPC I/O performance, the presence of this node-local staging layer further increases the complexity of managing the storage hierarchy. When the hierarchy of the HPC storage system consists mainly of the parallel file system and archival storage, users of HPC clusters can easily manage the data movements required by their applications explicitly, either in the application code itself or in the scripts controlling their batch jobs. Explicitly managing data transfers between storage tiers, however, is not optimal since end users lack the required real-time information about the state of the cluster to decide the best moment to execute a transfer of data. Moreover, given the increasing complexity of current HPC storage architectures—which may include as many layers as node-local storage, storage on I/O nodes, parallel file systems, campaign storage, and archival storage (see Fig.11)—explicit transfer management forces users (i.e., scientists and researchers) to spend time learning the best way to use these technologies in their applications, an effort that would be better spent on their scientific problems. This means that opportunities for global I/O optimization are being missed by not communicating application data flows to the HPC services in charge of resource allocation and, as such, any architecture that does not expose information about the storage tiers to applications, and relies solely on the hardware and OS to transparently manage the I/O stack will lead to sub-optimal performance. Although with the advent of *shared burst buffers* [8], vendors are providing APIs for transferring data to/from the parallel file system into/out of burst buffers (e.g., Cray DataWarp API [77] and IBM BBAPI [78]), such APIs 1) do not yet schedule PFS I/Os by taking into account cross-application interference [79], and 2) do not yet concern themselves with node-local storage.

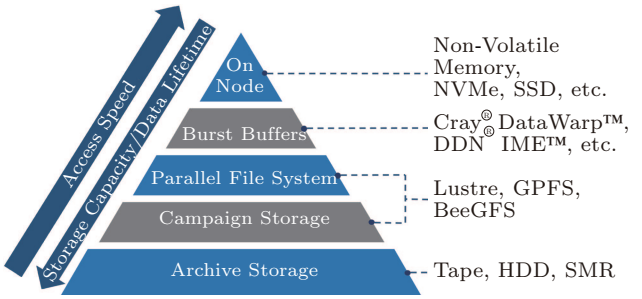

Fig.11. Growing complexity of the storage hierarchy of modern HPC systems, raising the need for research in order to understand how data and programming models expose and interact with this hierarchy.

Addressing these challenges and utilizing this new data-driven staging architecture effectively require developing new interfaces and APIs that allow end users to convey application *data flow requirements* (e.g., expected data lifetime, type of access, or visibility to related applications) to the services responsible for managing the HPC infrastructure. For instance, conveying data flow dependencies between jobs can help utilize the storage stack more effectively: if `Job A` generates output data that is going to be fed as input to `Job B`, a data-aware job scheduler could reuse `Job A`'s compute nodes for `Job B` and keep data in NVRAM. Unfortunately, today's users have no way to either express these dependencies or to influence the job-scheduling process so that `Job A`'s output data is kept in local storage until `Job B` starts. Worse yet, given that the I/O stack remains essentially a black box for today's job schedulers, `Job A`'s output data could be staged out to the cluster's parallel file system and, at some point in the near future, staged back into a new set of compute nodes for `Job B`, which might end up including some of the original nodes allocated to `Job A`.

Thus, we argue that integrating application data flows with scheduling and resource managers is critical for effectively using and managing an HPC hierarchical storage stack powered by ad hoc file systems. The development and deployment of data-aware scheduling services that ingest application data flow requirements and facilitate data movement across storage tiers, can

improve the coordination between HPC resource managers and the storage stack, resulting in reduced PFS I/O contention and, in turn, improved job run times and system efficiency (see Fig.12). While several services have been proposed with similar goals [80–83], to the best of our knowledge no resource scheduling algorithms have yet been proposed that take into account a job's dynamic requirements in terms of I/O (e.g., capacity, latency, and bandwidth) in addition to the more static computing requirements (e.g., compute nodes). There are thus plenty of research opportunities to investigate and develop APIs, infrastructure services and scheduling algorithms that can include application data flow needs into resource allocation decisions for I/O optimization. Provided with this information, and since many HPC applications exhibit relatively regular I/O patterns that would run in isolation in an architecture based on ad hoc file systems, these scheduling algorithms should be able to coordinate/interleave the staging phases from/to node-local storage in order to minimize the I/O contention of the parallel file system. This kind of algorithms have proved successful in avoiding contention in shared burst buffers and we believe that they could be extended to node-local storage [84].

Note, however, that major challenges still remain that are associated with scheduling storage resources on HPC systems. For example, is it possible to predict the best moment to start staging data into a compute node [85]? Would the data scheduler background transfers affect the cluster's networking subsystem so much that they impaired normal application execution? How should elastic workflows be addressed? What should the scheduling algorithms do when a job crashes while it is staging in data? Could these algorithms increase the energy efficiency of the supercomputer?

Besides exploiting user-provided information conveyed through APIs and services, a step further consists of taking advantage of self-describing I/O [54, 86–88], which has become a key aspect in managing large-scale datasets. By enriching the self-describing nature of large datasets, and integrating it with data-aware infrastructure services, we are not simply moving and storing large numbers of bytes but rather creating a vehicle to extract the most possible information as efficiently as possible. The idea is to promote intelligent I/O, a mechanism to allow information to be published and later subscribed to at all scales, for all types of data. The key to this is the ability to have self-describing data in streams and to think of data in motion in the same context as data at rest. We envision an extension to the publish/subscribe metaphor to include a clerk that will sit between the publisher and the subscriber and mediate or orchestrate data streams in a dynamic fashion.

In order to support the emerging analytics, processing, and storage use cases, data cannot be considered passive, hence directly falling through a chute connecting publishers and subscribers. Instead, a service-oriented architecture must connect them; actors must be involved to touch, manage, maintain, and abstract the data and to support inspection tasks such as in code coupling, in situ analysis, or visualization. These sets of actions must be managed and orchestrated across the wide array of resources in a way that enables not just imperative connections ("Output A must go to Input B") but also new models of learning and intelligence in the system ("Make this data persistent, but watch what I have been doing to other datasets and preprocess this data based on that").

While one can build systems that can be tuned dynamically by a human in the loop, intelligent systems with the capability to automatically tune workflows and drive them according to data events observed at runtime will lead the way in the design of modern computing infrastructure. In this case, storage for applications, which can be in terms of memory, individual burst buffers, burst buffers put together in an ad hoc file system, and parallel file systems need data placed and retrieved with enough contextual meaning that different codes for these workflows can publish and subscribe to this data. Thus, self-describing data streams can be at the core of all these storage systems both for on-line processing and for eventual postprocessing during the scientific campaign.

7 Conclusions

Ad hoc file systems enable the usage of node-local SSDs in many application scenarios. The three presented file systems BeeOND, GekkoFS, and BurstFS can show only a small fraction of the possibilities of such ad hoc file systems. Nevertheless, they start from a production file system that transfers BeeGFS's very high client performance to BeeOND's dynamic setting, including distributed SSDs, and range to research file systems showing possible performance capabilities either when using local writes or when spreading metadata and data.

The research file systems also show that applications and usage scenarios have to be partially adapted

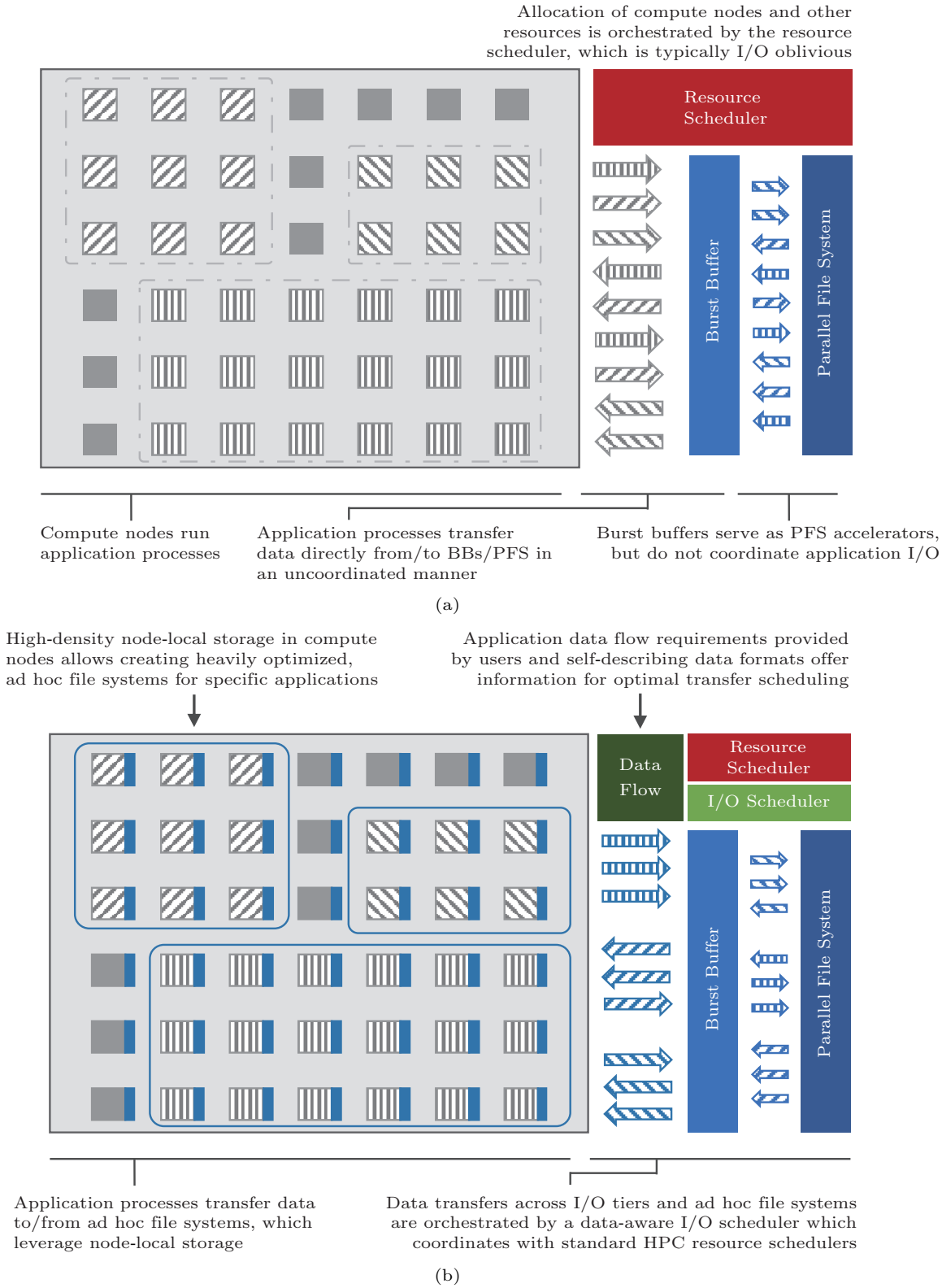

Fig. 12. Designing a data-driven HPC cluster is possible by coupling application-aware data staging with the I/O isolation provided by ad hoc file systems. On traditional HPC clusters applications access backend storage directly causing contention due to uncoordinated I/Os. On a data-driven HPC cluster, application I/O is absorbed by node-local storage and transfers of application input and output artifacts are coordinated to maximize the performance of backend storage. (a) Cluster with uncoordinated backend access. (b) Data-driven cluster with coordinated staging.

to ad hoc file systems. Many commands such as `ls -a` or `mv` are not widespread within parallel applications, but completely abstaining from them would simplify the development of ad hoc file systems while at the same time significantly increasing performance. Codesigning applications with storage systems can therefore benefit both sides. The benefit of this codesign can even be improved if reusable components can be applied as building blocks to shorten the time to develop production-ready file systems.

This paper has shown that ad hoc file systems help use the additional storage layer of node-local SSDs and NVRAMs. Nevertheless, simply using these file systems without considering data stage-in and stage-out within the batch environment makes using them a manual and error-prone task. It is therefore necessary to extend the overall HPC framework to make the usage of ad hoc file systems automatic, still leaving plenty of room for new research directions in the scheduling domain, on codesign, component isolation and performance tuning.

Acknowledgments We thank the team managing the Dagstuhl seminar series and all participants of the Dagstuhl Seminar 17202. We also gratefully acknowledge the computing time granted on the supercomputer Mogon II at Johannes Gutenberg University Mainz.

References

[1] Ruemmler C, Wilkes J. An introduction to disk drive modeling. *IEEE Computer*, 1994, 27(3): 17-28.

[2] 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*, November 2017, Article No. 6.

[3] Rajachandrasekar R, Moody A, Mohror K, Panda D K. A 1 PB/s file system to checkpoint three million MPI tasks. In *Proc. the 22nd Int. Symp. High-Performance Parallel and Distributed Computing*, June 2013, pp.143-154.

[4] Schroeder B, Lagisetty R, Merchant A. Flash reliability in production: The expected and the unexpected. In *Proc. the 14th USENIX Conference on File and Storage Technologies*, February 2016, pp.67-80.

[5] Meza J, Wu Q, Kumar S, Mutlu O. A large-scale study of flash memory failures in the field. In *Proc. the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2015, pp.177-190.

[6] Narayanan I, Wang D, Jeon M, Sharma B, Caulfield L, Sivasubramaniam A, Cutler B, Liu J, Khessib B M, Vaid K. SSD failures in datacenters: What? When? and Why? In *Proc. the 9th ACM International on Systems and Storage Conference*, June 2016, Article No. 7.

[7] Welch B, Noer G. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *Proc. the 29th IEEE Symposium on Mass Storage Systems and Technologies*, May 2013, Article No. 29.

[8] 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.

[9] Qian Y, Li X, Ihara S, Dilger A, Thomaz C, Wang S, Cheng W, Li C, Zeng L, Wang F, Feng D, Süß T, Brinkmann A. LPCC: Hierarchical persistent client caching for Lustre. In *Proc. the Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2019.

[10] Vef M A, 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 Int. Conf. Cluster Computing*, September 2018, pp.319-324.

[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] Brinkmann A, Mohror K, Yu W. Challenges and opportunities of user-level file systems for HPC (Dagstuhl Seminar 17202). *Dagstuhl Reports*, 2017, 7(5): 97-139.

[13] Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (1st edition). O'Reilly Media, 2017.

[14] Antcheva I, Ballintijn M, Bellenot B *et al*. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 2011, 182(6): 1384-1385.

[15] Edgar R C. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 2010, 26(19): 2460-2461.

[16] Köster J, Rahmann S. Snakemake—A scalable bioinformatics workflow engine. *Bioinformatics*, 2018, 34(20): 3600.

[17] Zhang Z, Barbary K, Nothaft F A, Sparks E R, Zahn O, Franklin M J, Patterson D A, Perlmutter S. Scientific computing meets big data technology: An astronomy use case. In *Proc. the 2015 IEEE International Conference on Big Data*, October 2015, pp.918-927.

[18] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In *Proc. the 26th IEEE Symp. Mass Storage Systems and Technologies*, May 2010, Article No. 9.

[19] Zaharia M, Xin R S, Wendell P, Das T *et al*. Apache spark: A unified engine for big data processing. *Commun. ACM*, 2016, 59(11): 56-65.

[20] Banker K. MongoDB in Action (2nd edition). Manning Publications, 2016.

[21] Carpenter J, Hewitt E. Cassandra: The Definitive Guide (2nd edition). O'Reilly Media, 2016.

[22] Jacob J C, Katz D S, Berriman G B, Good J, Laity A C, Deelman E, Kesselman C, Singh G, Su M, Prince T A, Williams R. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *Int. J. Computational Science and Engineering*, 2009, 4(2): 73-87.

[23] O'Driscoll A, Daugelaite J, Sleator R D. 'Big data', Hadoop and cloud computing in genomics. *Journal of Biomedical Informatics*, 2013, 46(5): 774-781.

[24] Conejero J, Corella S, Badia R M, Labarta J. Task-based programming in COMPSs to converge from HPC to big data. *International Journal of High Performance Computing Applications*, 2018, 32(1): 45-60.

[25] Fox G C, Qiu J, Jha S *et al*. Big data, simulations and HPC convergence. In *Lecture Notes in Computer Science 10044*, Rabl T, Nambiar R, Baru C, Bhandarkar M, Poess M, Pyne S (eds.), Springer-Verlag, 2015, pp.3-17.

[26] Wasi-ur-Rahman M, Lu X, Islam N S, Rajachandrasekar R, Panda D K. High-performance design of YARN MapReduce on modern HPC clusters with Lustre and RDMA. In *Proc. the 2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp.291-300.

[27] Ferreira K, Riesen R, Oldfield R, Stearley J, Laros J, Pedretti K, Brightwell R, Kordenbrock T. Increasing fault resiliency in a message passing environment. Technical Report, Sandia National Laboratories, 2009. https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/rMPI_tech.pdf, August 2019.

[28] Philp I R. Software failures and the road to a petaflop machine. In *Proc. the 1st Workshop on High Performance Computing Reliability Issues*, February 2005.

[29] Petrini F. Scaling to thousands of processors with Buffered Coscheduling. In *Proc. the 2002 Scaling to New Height Workshop*, May 2002.

[30] Congiu G, Narasimhamurthy S, Süß T, Brinkmann A. Improving collective I/O performance using non-volatile memory devices. In *Proc. the 2016 IEEE International Conference on Cluster Computing*, September 2016, pp.120-129.

[31] Moody A, Bronevetsky G, Mohror K, de Supinski B R. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proc. the 2010 Conference on High Performance Computing Networking, Storage and Analysis*, November 2010, Article No. 22.

[32] Islam T Z, Mohror K, Bagchi S, Moody A, de Supinski B R, Eigenmann R. McrEngine: A scalable checkpointing system using data-aware aggregation and compression. In *Proc. the 2012 Conf. High Performance Computing Networking, Storage and Analysis*, Nov. 2012, Article No. 17.

[33] Kaiser J, Gad R, Süß T, Padua F, Nagel L, Brinkmann A. Deduplication potential of HPC applications' checkpoints. In *Proc. the 2016 IEEE International Conference on Cluster Computing*, September 2016, pp.413-422.

[34] Zhu Y, Chowdhury F, Fu H, Moody A, Mohror K, Sato K, Yu W. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *Proc. the 26th IEEE Int. Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2018, pp.145-156.

[35] Rasch P, Xie S, Ma P L *et al.* An overview of the atmospheric component of the Energy Exascale Earth System Model. *Journal of Advances in Modeling Earth Systems*, 2019, 11(8): 2377-2411.

[36] Ross R, Ward L, Carns P, Grider G, Klasky S, Koziol Q, Lockwood G K, Mohror K, Settlemyer B, Wolf M. Storage systems and I/O: Organizing, storing, and accessing data for scientific discovery. Technical Report, US Department of Energy, 2018. https://www.osti.gov/biblio/1491994, May 2019.

[37] Kurth T, Treichler S, Romero J *et al.* Exascale deep learning for climate analytics. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 51.

[38] Liu J, Wu J, Panda D K. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004, 32(3): 167-198.

[39] Chen D, Eisley N, Heidelberger P, Senger R M, Sugawara Y, Kumar S, Salapura V, Satterfield D L, Steinmacher-Burow B D, Parker J J. The IBM Blue Gene/Q interconnection network and message unit. In *Proc. the 2011 Conference on High Performance Computing Networking, Storage and Analysis*, November 2011, Article No. 26.

[40] Faanes G, Bataineh A, Roweth D, Court T, Froese E, Alverson R, Johnson T, Kopnick J, Higgins M, Reinhard J. Cray cascade: A scalable HPC system based on a Dragonfly network. In *Proc. the 2012 Conference on High Performance Computing Networking, Storage and Analysis*, November 2012, Article No. 103.

[41] Gropp W D, Lusk E L, Skjellum A. Using MPI — Portable Parallel Programming with the Message-Passing Interface (3rd edition). MIT Press, 2014.

[42] Latham R, Ross R B, Thakur R. Can MPI be used for persistent parallel services? In *Proc. the 13th European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, September 2006, pp.275-284.

[43] Grun P, Hefty S, Sur S, Goodell D, Russell R D, Pritchard H, Squyres J M. A brief introduction to the OpenFabrics interfaces — A new network API for maximizing high performance application efficiency. In *Proc. the 23rd IEEE Annual Symposium on High-Performance Interconnects*, August 2015, pp.34-39.

[44] Shamis P, Venkata M G, Lopez M G *et al.* UCX: An open source framework for HPC network APIs and beyond. In *Proc. the 23rd IEEE Annual Symposium on High-Performance Interconnects*, August 2015, pp.40-43.

[45] Barrett B W, Brightwell R, Hemmert S *et al.* The portals 4.0 network programming interface. Technical Report, Sandia National Laboratories, 2012. http://www.cs.sandia.gov/Portals/portals40.pdf, May 2019.

[46] 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.

[47] Oldfield R, Widener P M, Maccabe A B, Ward L, Kordenbrock T. Efficient data-movement for lightweight I/O. In *Proc. the 2006 IEEE International Conference on Cluster Computing*, September 2006, Article No. 60.

[48] Wheeler K B, Murphy R C, Thain D. Qthreads: An API for programming with millions of lightweight threads. In *Proc. the 22nd IEEE International Symposium on Parallel and Distributed Processing*, April 2008.

[49] Nakashima J, Taura K. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond — Papers Dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, Agha G, Igarashi A, Kobayashi N, Masuhara H, Matsuoka S, Shibayama E, Taura K (eds.), Springer, 2014, pp.222-238.

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

[51] Dorier M, Carns P H, Harms K *et al.* Methodology for the rapid development of scalable HPC data services. In *Proc. the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, November 2018, pp.76-87.

[52] 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.

[53] 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.

[54] 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.

[55] Moore M, Bonnie D, Ligon B, Marshall M, Ligon W, Mills N, Quarles E, Sampson S, Yang S, Wilson B. OrangeFS: Advancing PVFS. In *Proc. the 9th USENIX Conference on File and Storage Technologies*, February 2011.

[56] Volos H, Nalli S, Panneerselvam S, Varadarajan V, Saxena P, Swift M M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. the 9th Eurosys Conference*, April 2014, Article No. 14.

[57] Zheng Q, Cranor C D, Guo D, Ganger G R, Amvrosiadis G, Gibson G A, Settlemyer B W, Grider G, Guo F. Scaling embedded in-situ indexing with deltaFS. In *Proc. the 2018 Int. Conf. High Performance Computing, Networking, Storage, and Analysis*, Nov. 2018, Article No. 3.

[58] Kelly S M, Brightwell R. Software architecture of the light weight kernel, Catamount. In *Proc. the 2005 Cray User Group Annual Technical Conference*, May 2005, pp.16-19.

[59] Rajgarhia A, Gehani A. Performance and extension of user space file systems. In *Proc. the 2010 ACM Symposium on Applied Computing*, March 2010, pp.206-213.

[60] Vangoor B K R, Tarasov V, Zadok E. To FUSE or not to FUSE: Performance of user-space file systems. In *Proc. the 15th USENIX Conference on File and Storage Technologies*, February 2017, pp.59-72.

[61] Henson V, van de Ven A, Gud A, Brown Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proc. the 2nd Workshop on Hot Topics in System Dependability*, November 2006, Article No. 8.

[62] Hoskins M E. SSHFS: Super easy file access over SSH. *Linux Journal*, 2006, 2006(146): Article No. 4.

[63] Zhao D, Zhang Z, Zhou X, Li T, Wang K, Kimpe D, Carns P H, Ross R B, Raicu I. FusionFS: Toward supporting data intensive scientific applications on extreme-scale high-performance computing systems. In *Proc. the 2014 IEEE Int. Conf. Big Data*, October 2014, pp.61-70.

[64] Davies A, Orsaria A. Scale out with GlusterFS. *Linux Journal*, 2013, 2013(235): Article No. 1.

[65] Lensing P H, Cortes T, Brinkmann A. Direct lookup and hash-based metadata placement for local file systems. In *Proc. the 6th Annual International Systems and Storage Conference*, June 2013, Article No. 5.

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

[67] 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.

[68] 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.

[69] Oral S, Dillow D A, Fuller D, Hill J, Leverman D, Vazhkudai S S, Wang F, Kim Y, Rogers J, Simmons J, Miller R. OLCF's 1 TB/s, next-generation Lustre file system. In *Proc. the 2013 Cray User Group Conference*, April 2013.

[70] Greenberg H, Bent J, Grider G. MDHIM: A parallel key/value framework for HPC. In *Proc. the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, July 2015, Article No. 10.

[71] Karger D R, Lehman E, Leighton F T, Panigrahy R, Levine M S, Lewin D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. the 29th Annual ACM Symposium on the Theory of Computing*, May 1997, pp.654-663.

[72] 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.

[73] Carns P H, Harms K, Allcock W E, Bacon C, Lang S, Latham R, Ross R B. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. Storage*, 2011, 7(3): Article No. 8.

[74] Yildiz O, Dorier M, Ibrahim S, Ross R B, Antoniu G. On the root causes of cross application I/O interference in HPC storage systems. In *Proc. the 2016 IEEE Int. Parallel and Distributed Processing Symposium*, May 2016, pp.750-759.

[75] 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.

[76] 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 Conference on High Performance Computing Networking, Storage and Analysis*, November 2012, Article No. 8.

[77] Paul D, Landsteiner B. Datawarp administration tutorial. https://cug.org/proceedings/cug2018proceedings/includes/_les/tut105s2-_le1.pdf, May 2019.

[78] Gonsiorowski E. Using sierra burst buffers. https://computing.llnl.gov/tutorials/SierraGettingStarted, May 2019.

[79] 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 Int. Conf. Cluster Computing*, September 2018, pp.290-301.

[80] Dong B, Byna S, Wu K, Prabhat, Johansen H, Johnson J N, Keen N. Data elevator: Low-contention data movement in hierarchical storage system. In *Proc. the 23rd IEEE Int. Conf. High Performance Computing*, December 2016, pp.152-161.

[81] Miranda A, Jackson A, Tocci T, Panourgias I, Nou R. NORNS: Extending Slurm to support data-driven workflows through asynchronous data staging. In *Proc. the 2019 IEEE International Conference on Cluster Computing*, September 2019.

[82] Subedi P, Davis P E, Duan S, Klasky S, Kolla H, Parashar M. Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *Proc. the 2018 Int. Conf. for High Performance Computing, Networking, Storage, and Analysis*, Nov. 2018, Article No. 73.

[83] Wang T, Oral S, Pritchard M, Wang B, Yu W. TRIO: Burst buffer based I/O orchestration. In *Proc. the 2015 IEEE Int. Conf. Cluster Computing*, Sept. 2015, pp.194-203.

[84] 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.

[85] Soysal M, Berghoff M, Klusácek D, Streit A. On the quality of wall time estimates for resource allocation prediction. In *Proc. the 48th International Conference on Parallel Processing*, August 2019, Article No. 23.

[86] Folk M, Heber G, Koziol Q, Pourmal E, Robinson D. An overview of the HDF5 technology suite and its applications. In *Proc. the 2011 EDBT/ICDT Workshop on Array Databases*, March 2011, pp.36-47.

[87] Li J, Liao W, Choudhary A N, Ross R B, Thakur R, Gropp W, Latham R, Siegel A R, Gallagher B, Zingale M. Parallel netCDF: A high-performance scientific I/O interface. In *Proc. the 2003 ACM/IEEE Conf. High Performance Networking and Computing*, Nov. 2003, Article No. 39.

[88] Rew R, Davis G. NetCDF: An interface for scientific data access. *IEEE Computer Graphics and Applications*, 1990, 10(4): 76-82.

André Brinkmann is a full professor at the Computer Science Department of Johannes Gutenberg University Mainz (JGU) 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, Paderborn, 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.

Kathryn Mohror is the group leader for the Data Analysis Group at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory (LLNL), Livermore. Kathryn's research on high-end computing systems is currently focused on scalable fault tolerant computing and I/O for extreme scale systems. Her other research interests include scalable performance analysis and tuning, and parallel programming paradigms. Kathryn has been working at LLNL since 2010.

Weikuan Yu is a professor in the Department of Computer Science at Florida State University, Tallahassee. He received his Ph.D. degree in computer science and Master's degree in neurobiology from the Ohio State University, Columbus. He also holds his Bachelor's degree in genetics from Wuhan University, Wuhan. Yu's main research interests include big data management and analytics frameworks, parallel I/O and storage, GPU memory architecture, and high performance networking. Yu's research has won the 2012 Alabama Innovation Award and the First Prize of 2012 ACM Student Research Competition Grand Finals. He is a senior member of IEEE and life member of ACM.

Philip Carns is a principal software development specialist in the Mathematics and Computer Science Division of Argonne National Laboratory, Lemont. He is also an adjunct associate professor of electrical and computer engineering at Clemson University and a fellow of the Northwestern-Argonne Institute for Science and Engineering. His research interests include characterization, modeling, and development of storage systems for data-intensive scientific computing.

Toni Cortes is an associate professor at Universitat Politècnica de Catalunya, Barcelona, (since 1998), and researcher at the Barcelona Supercomputing Center. 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.

Scott A. Klasky is a distinguished scientist and the group leader for scientific data in the Computer Science and Mathematics Division at the Oak Ridge National Laboratory, Oak Ridge. He holds an appointment at the University of Tennessee, and Georgia Tech University. He obtained his Ph.D. degree in physics from the University of Texas at Austin (1994). Dr. Klasky is a world expert in scientific computing and scientific data management, co-authoring over 200 papers.

Alberto Miranda is a senior researcher in advanced storage systems in the Computer Science Department of the Barcelona Supercomputing Center (BSC) 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) in 2014, and has been working at BSC since 2007.

Franz-Josef Pfreundt is the director of the Competence Center for HPC & Visualization at Fraunhofer ITWM since 1999. He studied mathematics, physics and computer science resulting in a Diploma in mathematics and a Ph.D. degree in mathematical physics (1986). In 2001 the prestigious Fraunhofer Research Prize was awarded to Franz-Josef Pfreundt, Konrad Steiner and their research group for their work on microstructure simulation. The developments in the area of visualization and implementations on IBM Cell Processor gained the Fraunhofer Research Price in 2005 and the IBM Faculty Award in 2006. His main research focuses are parallel file systems and new parallel programming approaches.

Robert B. Ross is a senior computer scientist at Argonne National Laboratory, Lemont, and the director of the DOE SciDAC RAPIDS Institute for Computer Science and Data. Rob's research interests are in system software for high performance computing systems, in particular distributed storage systems and libraries for I/O and message passing. Rob received his Ph.D. degree in computer engineering from Clemson University, Clemson, in 2000. Rob was a recipient of the 2004 Presidential Early Career Award for Scientists and Engineers.

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. His master thesis was in cooperation with IBM Research about analyzing file create performance in the IBM Spectrum Scale parallel file system (formerly GPFS). Marc's research interests focus on parallel and ad-hoc file systems and system analytics.