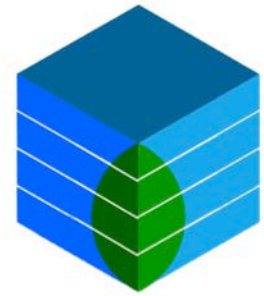


BigStorage

STORAGE-BASED CONVERGENCE BETWEEN HPC AND CLOUD TO HANDLE BIG DATA

Deliverable number D3.2
Deliverable title I/O Middleware *for HPC clouds*
Final Report
WP3 HPC-CLOUD Convergence

Editor
Adrien Lebre (Inria)

Authors
Pierre Matri (UPM, ESR03), Nafiseh Moti (JGU, ESR 4) Linh Thuy Nguyen (Inria, ESR07), Fotis Nikolaidis (CEA, ESR09)

Grant Agreement number	642963
Project ref. no	MSCA-ITN-2014-ETN-642963
Project acronym	BigStorage
Project full name	BigStorage: Storage-based convergence between HPC and Cloud to handle Big Data
Starting date (dur.)	1/1/2015 (48 months)
Ending date	31/12/2018
Project website	http://www.bigstorage-project.eu

Coordinator	María S. Pérez
Address	Campus de Montegancedo sn. 28660 Boadilla del Monte, Madrid, Spain
Reply to	mperez@fi.upm.es
Phone	+34- 910672857

Executive Summary

This document provides an overview of the work that has been achieved during the Project BigStorage (from 01-01-2015 until 31-12-2018) in WP3 HPC-Cloud Convergence.

Although available storage mechanisms in HPC and Cloud environments significantly differ from their design as well as the services they deliver (files vs. key/value systems, hierarchical vs. flat namespaces, semantics, resilience) understanding the specific techniques used in both areas has been a key element to identify pros/cons of available systems (D3.1), and propose new storage building blocks offering the best of both worlds.

To illustrate this convergence, we choose to present three systems that have been proposed through the BigStorage project :

- **YOLOFS, a caching technique to faster VM provisioning -** several works have shown that the time to boot one virtual machine (VM) can last up to a few minutes in high consolidated cloud scenarios. This time is critical as VM boot duration defines how an application can react w.r.t. demands' fluctuations (horizontal elasticity). To limit as much as possible the time to boot a VM, we design the YOLO mechanism (You Only Load Once), a new storage service that uses HPC caching approaches to speed VM Boot Time duration.
- **GekkoFS, a temporary, highly-scalable burst buffer file system** - GekkoFS has been specifically designed for new access patterns of data-intensive HPC applications. The file system provides relaxed POSIX semantics, only offering features which are actually required by most (not all) applications. It is able to provide scalable I/O performance and reaches millions of metadata operations already for a small number of nodes, significantly outperforming the capabilities of general-purpose parallel file systems.
- **Týr, a Transactional Object Storage for HPC and Big Data Convergence -** Týr takes the benefit of the growing trend of equipping HPC compute nodes with local storage in order to deploy object storage systems alongside the application on the compute nodes. By leveraging Blobs (Binary Large Objects) as as an alternative to files, Tyr improves storage throughput for a variety of existing applications.

For each system, we first discuss the problem it addresses, and second how the solution tackles it.

Through this final report, we aim to demonstrate also, the expertise ESRs gained during the BigStorage project. The contributions they proposed have been published through research reports or scientific publications in conferences and journals. As an example, the publication « Týr : Blob Storage Meets Built-In Transactions » has been selected as Best Student Paper Finalist at the ACM Super Computing Conference in Nov 2016.

Document Information

IST Project Number	MSCA-ITN-2014-ETN-642963
Acronym	BigStorage
Title	Storage-based convergence between HPC and Cloud to handle Big Data
Project URL	http://www.bigstorage-project.eu
Deliverable	D3.2 Final Report on WP3
Workpackage	WP3 Storage Solutions
Date of Delivery	Planned: 31.12.2018 Actual: 26.12.2018
Status	Version 1.0 final draft □
Nature	prototype □ report X dissemination □
Dissemination level	public □ consortium X
Distribution List	Consortium Partners
Responsible Editor	Adrien Lebre (Inria), adrien.lebre@inriafr, Tel: +33 251 858 243
Authors (Partner)	Linh NGuyen (Inria), Nafiseh Moti (Mainz Univ), Pierre Matri (UPM)
Reviewers	BigStorage Advisors
Abstract (for dissemination)	Executive Summary
Keywords	HPC, Clouds, Virtual Machine Image, distributed file systems, key-value store, BLOBS

Version	Modification(s)	Date	Author(s)
0.1	Initial template and structure	20.07.2018	Adrien Lebre, Inria
0.2	Sections from all authors	20.12.2018	All authors
0.3	Internal version for review	26.12.2018	Adrien Lebre, Inria
0.4	Internal review (Add Project information and Project consortium pages)	27.12.2018	María S. Pérez
0.5	Final version to commission	31.12.2018	Adrien Lebre, Inria

Project Consortium Information

Participants		Contact
Universidad Politécnica de Madrid (UPM), Spain		María S. Pérez Email: mperez@fi.upm.es
Barcelona Supercomputing Center (BSC), Spain		Toni Cortes Email: toni.cortes@bsc.es
Johannes Gutenberg University (JGU) Mainz, Germany		André Brinkmann Email: brinkman@uni-mainz.de
Inria, France		Gabriel Antoniu Email: gabriel.antoniu@inria.fr Adrian Lebre Email: adrien.lebre@inria.fr
Foundation for Research and Technology - Hellas (FORTH), Greece		Angelos Bilas Email: bilas@ics.forth.gr
Seagate, UK		Sai Narasimhamurthy Email:sai.narasimhamurthy@seagate.com
DKRZ, Germany		Thomas Ludwig Email: ludwig@dkrz.de
CA Technologies Development Spain (CA), Spain		Victor Muntes Email: Victor.Muntes@ca.com
CEA, France		Jacque Charles Lafoucriere Email: Charles.LAFOUCRIERE@CEA.FR
Fujitsu Technology Solutions GMBH, Germany		Sepp Stieger Email: sepp.stieger@ts.fujitsu.com

Table of contents

1. YOLO: Speeding up VM Boot Time

1.1 Context and Problematic

The promise of elasticity of cloud computing brings the benefits for clients of adding and removing new VMs in a manner of seconds. However, in reality, users may have to wait several minutes to get a new VM in public IaaS clouds such as Amazon EC2, Microsoft Azure or RackSpace [14]. Such long startup duration has a strong negative impact on services deployed in a cloud system. For instance, when an application (e.g., a web service) faces peak demands, it is important to provision additional VMs as fast as possible to prevent loss of revenue for this service. Therefore, the startup time of VMs plays an essential role in provisioning resources in a cloud infrastructure. In this first result, we present how HPC caching techniques can significantly reduce this time.

The startup time of VMs can be divided into two major parts: (i) the time to transfer the VMI from the repository to the selected compute node and (ii) the time to perform the VM boot process. While a lot of efforts focused on mitigating the penalty of the VMI transferring time either by using deduplication, caching and chunking techniques or by avoiding it thanks to remote attached volume approaches [a-9,a-17,a-19,a-20], only a few works addressed the boot duration challenge. To the best of our knowledge, the solutions that investigated the boot time issue proposed to use either cloning techniques [a-2,a-12] or suspend/resume capabilities of VMs [a-10,a-21,a-32]. The former relies on lives available on each compute node so that it is possible to spawn new identical VMs without performing the VM boot process. The latter consists in saving the entire state of each possible VM and resuming it when necessary (each time a VM is requested, the new VM is created from the master snapshot). Once the new VM is available, both approaches may use hot-plug mechanisms to reconfigure the VM physical characteristics according to the users' expectations (in terms of number CPU, RAM size, network ...). Although these two solutions enable speeding up the boot duration, they have major drawbacks. The cloning technique requires to allocate dedicated resources for each live VM, which limits the number of master copy that can be executed on each node. The suspend/resume approach eliminates this issue but requires a large amount of storage space on each compute node to save a copy of the snapshot of each VM that might be instantiated according to the existing VMIs. Besides, these two approaches have not been designed with high-consolidated scenarios in mind. In other words, the process to launch a VM on the compute note (boot, cloning or resuming) performs I/O and CPU operations that impact the performance. Such an issue has been investigated for traditional boot approaches in recent studies [a-15,a-28] where the authors show that the duration of VM boot process is highly variable depending on the effective system load and the number of simultaneous provisioning requests the compute node should satisfy.

To deal with each of the aforementioned limitations (mitigate resource wasting as well as resource competition on each compute node), we designed the YOLO mechanism (You Only Load Once). YOLO speeds up the boot process by manipulating mandatory data to boot a VM as less as possible.

In the following, we give an overview of our proposal and its implementation. First, we introduce the boot image abstraction and explain how boot images are created. Second, we introduce how yolofs, our custom file system, intercepts I/O requests to speed up the VM boot process. Finally, we discuss two experiments we made to validate our proposal.

1.2. YOLO Design and Implementation

At coarse-grained, YOLO has been built on the observation that only a small portion of a VMI is required to boot a VM [a-16,a-19,a-32]. Hence, for each VMI, we construct a boot image,i.e., a subset of the VMI that contains the mandatory data needed for booting a VM, and store it on a fast access storage device (memory, SSD, etc.) on each compute node. When a VM boot process starts, YOLO transparently loads the corresponding boot image into the memory and serve all I/O requests directly.

1.2.1. Boot Image

To create boot images, we capture all read requests generated when we boot completely a VM. Each read request has: (i) a file_descriptor with file_path and file_name, (ii) an offset which is the beginning logical address to read from, and (iii) a length that is the total length of the data to read. For each read request, we calculate the list of all block_id to be read by using the offset and length information and we record the block_id along with the data of that block. A boot image contains a dictionary of key-value pairs in which the key is the pair (file_name,block_id) and the value is the content of that block. Therefore, with every read request on the VMI, we can use the pair (file_name,block_id) to retrieve the data of that block. In a cloud system, we create these boot images for all available VMIs and store them on each compute node. Using this technique, the space needed to store boot images for the 900+ VMIs available from Google Cloud is around 40 GB, which represents less than 3% of the original size of all VMIs as detailed in Table 1.

Image	No. of images	Size of images	Size of all boot images	Reducing rate
CentOS	156	223 GB	6.3 GB	97.2 %
Debian	180	216 GB	4.7 GB	97.8 %
Ubuntu	236	272 GB	16 GB	94.1 %
CoreOS	221	173 GB	1.2 GB	99.3 %
RHEL	167	302 GB	7 GB	97.7 %
Windows	15	191 GB	5.2 GB	97.3 %
Total	983	1.34 TB	40.4 GB	97.4 %

Table 1
The statistics of 900+ Google Cloud VMIs and their boot images. We group the VMIs into image families and calculate the boot images for each image family.

To avoid generating I/O contention with other operations when accessing these boot images, we store boot images on dedicated devices which can be either a local storage device, a remote attached volume, or simply dedicated random access memories. Access to boot images are handled by yolofs as described in the next section.

1.2.2. yolofs

We developed yolofs using FUSE (Filesystem in User space) to serve all the read requests executed by VMs during the boot process via the boot images. FUSE allows to create a custom file system in userspace without changing the kernel of the host OS. Furthermore, recent analysis [a-18,a-25] confirmed that the performance overhead when using FUSE against read requests is acceptable. However, other solutions are also possible if the performance will become an issue, for example using library interposition.

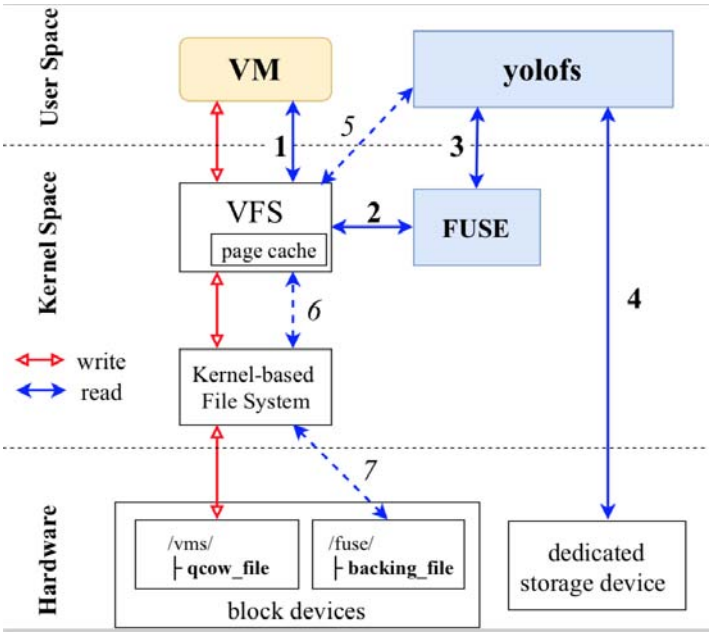


Figure 1- yolofs read/write data flow

In Figure 1, we illustrate the workflow of yolofs along with the read/write data flow for a VM created with a shared image disk. We start yolofs in a compute node before starting any VM operations. When a VM issues I/O reads on its backing file which is linked to our mounted yolofs file system, the VFS routes the operation to the FUSE's kernel module, and yolofs will process it (i.e., Step 1, 2, 3 of the reading flow). yolofs then returns the data directly from the boot image which already was in they yolofs' memory (Step 4). If not, yolofs would load that boot image from its dedicated storage device (where it stores all the boot images of this cloud system) to the memory. Whenever the VM wants to access data that is not available in the boot image, yolofs utilizes the kernel-based file system to read the data from the disk (Step 5, 6, and 7 of the reading flow). All I/O writes generated from the VM go directly to the QCOW file of that VM and they are not handled by yolofs (the writing flow in Figure 2)

1.3. Evaluation

In this document we discuss two experiments we made to evaluate Yolo gains in the presence of I/O and memory intensive workloads.

1.3.1 Experimental Setup

Experiments have been performed on top of the Grid'5000 Nantes cluster. Each physical node has 2 Intel Xeon E5-2660 CPUs (8 physical cores each) running at 2.2 GHz; 64 GB of memory, a 10 Gbit Ethernet network card and one of two kinds of storage devices: (i) HDD with 10 000 rpm Seagate Savvio 200 GB (150 MB/s throughput) and (ii) SSD with Toshiba PX02SS 186 GB (346 MB/s throughput). Regarding CEPH, we used CEPH version 10.2.5 deployed through 5 nodes (1 master and 4 data nodes, using HDD). When needed, CEPH has been used to deliver the remote-attached VM image disks to different VMs (each "compute" node mounted the remote block devices with ext4 format). Regarding the VMs' configuration, we used the Qemu/KVM hypervisor (Qemu-2.1.2 and Linux-3.2) with virtio enabled (network and disk device drivers). VMs have been created with one vCPU and 1 GB of memory and a share image disk using QCOW2 format with the write-through cache mode. During each experiment, each VM has been assigned to a single core to avoid CPU contention and prevent non-controlled side effects. The I/O scheduler of VMs and the physical node is CFQ.
Regarding the VM boot time, we assumed that a VM is ready to be used when it is possible

to log into it using SSH. This information can be retrieved by reading the system log, and it is measured in milliseconds. To avoid side effect due to the starting of other applications, SSH has been configured as the first service to be started. All experiments have been repeated at least ten times to get statistically significant results.
Last but not the least, to evaluate the benefit of YOLO w.r.t. default cache strategies, we implemented an ad-hoc prefetching script that preloads into the memory the mandatory data. By such a way, read operations could be served from memory rather than from the storage device as long as the page cache is not evicted (the prefetching script and the boot process of VMs are invoked simultaneously).

1.3.2 Boot Time analysis under I/O contention environment

Figure 2 shows the boot time of one VM on three different storage devices under an I/O-intensive scenario. YOLO delivers significant improvements in all cases. On HDD, booting only one VM lasts up to 2 minutes by using the normal boot policy. Obviously, prefetching boot and YOLO speed up boot duration much more than the normal one because the data is loaded into the cache in a more efficient way.
The same trend can be found on SSD in Figure 2b where the time to boot the eVM increased from 3 to 20 seconds for the normal strategy, 3 to 6 seconds for the prefetching boot, and from 3 to 4 seconds for YOLO. While YOLO is faster than prefetching boot by a small amount, YOLO is up to 4 times faster than all at once policy under I/O contention of 15 coVMs. An interesting point is related to the CEPH scenario. When the coVMs are stressing the I/O, they generate a bottleneck on the NIC of the host OS, which impacts the performance of the write requests that are performed by the eVM during the boot operation. This leads to worse performance for YOLO and the prefetching boot strategies than in the HDD and SSD scenarios, ranging from 3 to 58 seconds and from 3 to 61 seconds respectively. However, it is still twice faster than the boot time of all at once, which is 107 seconds at 15 coVMs.
To sum up, on a physical node which already had I/O workloads, YOLO is the best solution to boot a new VM in a small amount of time. YOLO reduces the boot duration 5 times on local storage (HDD and SSD) and 2 times on remote storage (using CEPH).

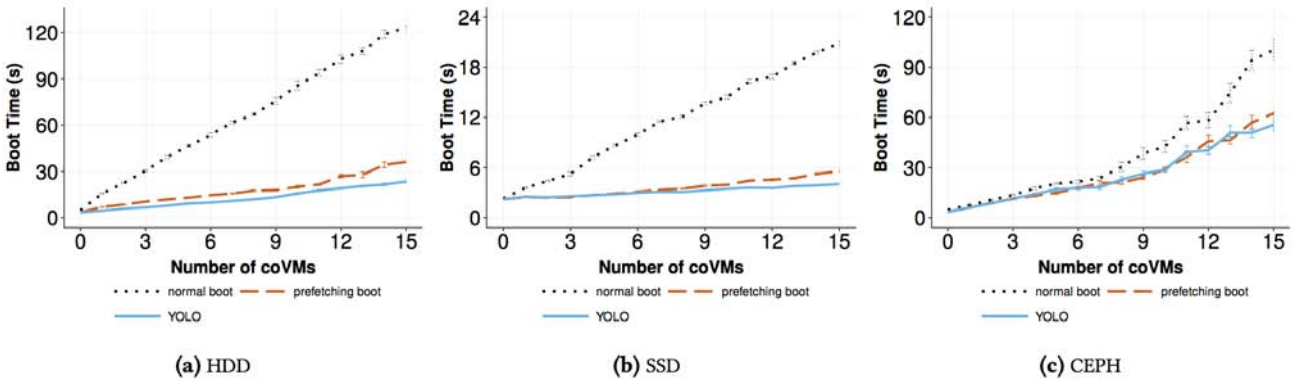


Figure 2 - Boot time of 1 VM (with shared image disk, write through cache mode) under I/O contention environment

1.3.3 Boot Time analysis under memory usage contention environment

We use this scenario to assess the influence of not having enough space to load and keep the boot images into the memory of both prefetching boot and YOLO strategies. Figure 3 gives the results we measured.

On HDD, the normal boot time can reach up to 4 times longer compared to the other two methods. With prefetching boot, the prefetched data stays in the memory to reduce the boot time until the page cache space is claimed. In this situation, the hypervisor might have to to read the prefetching data on the storage device once again. While comparing to YOLO, the boot data stays in YOLO memory space. For this reason, under memory-intensive environment, YOLO is almost 2 times faster than prefetching boot with 15 coVMs that stress the whole memory. On SSD, the different between YOLO and prefetching boot is small thanks to the performance of SSD. It is also true for CEPH, which has high read performance in general.

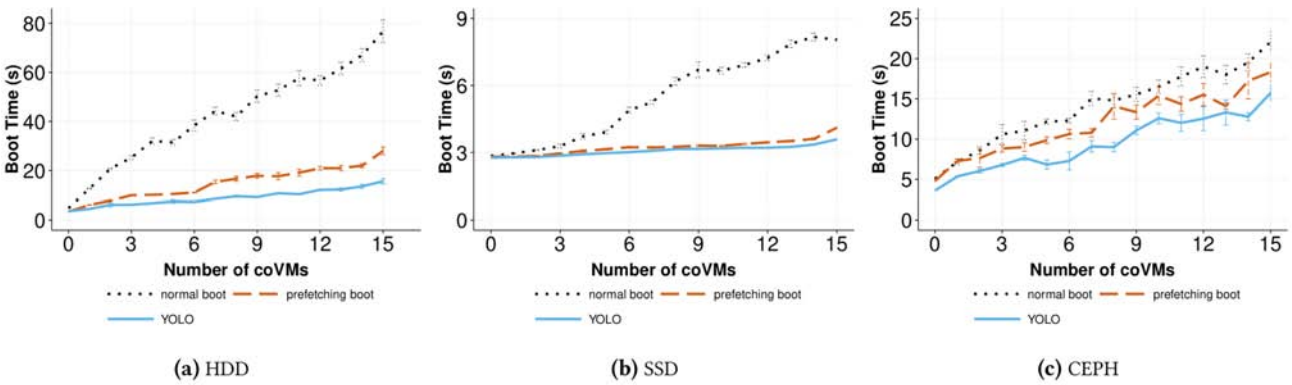


Figure 3 - Boot time of 1 VM (with shared image disk, write through cache mode) under memory usage contention environment

1.4. Summary

Starting a new VM in a cloud infrastructure is a long process. It depends on the time to transfer the VMI to the compute node and the time to perform VM boot process itself. In this work, we focus on improving the duration of VM boot process. Boot duration highly relies on the number of VM requests and the co-workloads running on the compute nodes. We identify the primary reason that lasts a VM boot time to few minutes is the I/O contention. YOLO is proposed as a new methodology to minimize the read operations on the VMI during a VM boot process. In our solution, we introduce the boot image which contains all the necessary data from a VMI to boot a VM. After that, we use yolofs to load this boot image into the memory and then yolofs serves all read requests which are executed during a VM boot duration directly from the memory. Our evaluation shows that YOLO can speed up VM boot time 2-5 times under different resources contention, and 2-10 times when booting several VMs simultaneously.

Ongoing activities focus on evaluating the YOLO concepts into the OpenStack software stack.

2. GekkoFS – A temporary distributed file system for HPC applications

2.1 Context and Problematic

High-Performance Computing (HPC) applications are significantly changing. Traditional HPC applications have been compute-bound, large-scale simulations, while today's HPC

community is additionally moving towards the generation, processing, and analysis of massive amounts of experimental data. This trend, known as data-driven science, is affecting many different scientific fields, some of which have made significant progress tackling previously unaddressable challenges thanks to newly developed techniques [b-13], [b-26].

Most data-driven workloads are based on new algorithms and data structures like graph databases which impose new requirements on HPC file systems [b-19], [b-35]. They include, e.g., large numbers of metadata operations, data synchronization, non-contiguous and random access patterns, and small I/O requests [b-7], [b-19]. Such operations differ significantly from past workloads which mostly performed sequential I/O operations on large files. They do not only slow down data-driven applications themselves but can also heavily disrupt other applications that are concurrently accessing the shared storage system [b-9], [b-31]. Consequently, traditional parallel file systems (PFS) cannot handle these workloads efficiently and data-driven applications suffer from prolonged I/O latencies, reduced throughput, and long waiting times.

Software-based approaches, e.g., application modifications or middleware and high-level libraries [b-10], [b-16], try to support data-driven applications to align the new access patterns to the capabilities of the underlying PFS. Yet, adapting such software is typically time-consuming, difficult to couple with big data and machine learning libraries, or sometimes (based on the underlying algorithms) just impossible.

Hardware-based approaches move from magnetic disks, the main backend technology for PFSs, to NAND-based solid-state drives (SSDs). Nowadays, many supercomputers deploy SSDs which can be used as dedicated burst buffers [b-15] or as node- local burst buffers. To achieve high metadata performance, they can be deployed in combination with a dynamic burst buffer file system [b-2], [b-36].

Generally, burst buffer file systems increase performance compared to a PFS without modifying an application. Therefore, they typically support POSIX which provides the standard semantics accepted by most application developers. Nevertheless, enforcing POSIX can severely reduce a PFS' peak performance [b-34]. Further, many POSIX features are not required for most scientific applications [b-14], especially if they can exclusively access the file system. Similar argumentations hold for other advanced features like fault tolerance or security.

In this second result, we present GekkoFS, a temporarily deployed, highly-scalable distributed file system for HPC applications which aims to accelerate I/O operations of common HPC workloads that are challenging for modern PFSs. GekkoFS pools together fast node-local storage resources and provides a global namespace accessible by all participating nodes. It relaxes POSIX by removing some of the semantics that most impair I/O performance in a distributed context and takes previous studies on the behavior of HPC applications into account [b-14] to optimize the most used file system operations.

For load-balancing, all data and metadata are distributed across all nodes using the HPC RPC framework Mercury [b-30]. The file system runs in user-space and can be easily deployed in under 20 seconds on a 512 node cluster by any user. Therefore, it can be used in a number of temporary scenarios, e.g., during the lifetime of a compute job or in longer-term use cases, e.g., campaigns. We demonstrate how our lightweight, yet highly distributed file system GekkoFS reaches scalable data and metadata performance with tens of millions of metadata operations per second on a 512 node cluster while still providing strong consistency for file system operations that target a specific file or directory.

2.2. Related Work

General-purpose PFSs like GPFS, Lustre, BeeGFS, or PVFS [b-a3], [b-12], [b-24], [b-27], [b-28] provide long-term storage which is mostly based on magnetic disks. GekkoFS instead builds a short-term, separate namespace from fast node-local SSDs that is only temporarily accessible during the runtime of a job or a campaign. As such, GekkoFS can be categorized into the class of node-local burst buffer file systems, while remote-shared burst buffer file systems use dedicated, centralized I/O nodes [b-36], e.g., DDN's IME [b-1].

In general, node-local burst buffers are fast, intermediate storage systems that aim to reduce the PFS' load and the applications' I/O overhead [b-15]. They are typically collocated with nodes running a compute job, but they can also be dependent on the backend PFS [b-2] or in some cases even directly managed by it [b-21]. BurstFS [b-36], perhaps the most related work to ours, is a standalone burst buffer file system, but, unlike GekkoFS, is limited to write data locally. BeeOND [b-12] can create a job-temporal file system on a number of nodes similar to GekkoFS. However, in contrast to our file system, it is POSIX compliant and our measurements show a much higher metadata throughput than offered by BeeOND [b-32].

The management of inodes and related directory blocks are the main scalability limitations of file systems in a distributed environment. Typically, general-purpose PFSs distribute data across all available storage targets. As this technique works well for data, it does not achieve the same throughput when handling metadata [b-4], [b-25], although the file system community presented various techniques to tackle this challenge [b-2], [b-11], [b-22], [b-23], [b-37], [b-38]. The performance limitation can be attributed to the sequentialization enforced by underlying POSIX semantics which is particularly degrading throughput when a huge number of files is created in a single directory from multiple processes. This workload, common to HPC environments [b-2], [b-21], [b-22], [b-33], can become an even bigger challenge for upcoming data-science applications. GekkoFS is built on a new technique to handle directories and replaces directory entries by objects, stored within a strongly consistent key-value store which helps to achieve tens of millions of metadata operations for billions of files.

2.3. GekkoFS Design and Implementation

GekkoFS offers a user-space file system for the lifetime of a particular use case, e.g., within the context of an HPC job. The file system uses the available local storage of compute nodes to distribute data and metadata and combines their node-local storage into a single global namespace.

The file system's main goal focuses on scalability and consistency. It should therefore scale to an arbitrary number of nodes to benefit from current and future storage and network technologies. Further, GekkoFS provides the same consistency as POSIX for file system operations that access a specific data file. However, consistency of directory operations, for instance, can be relaxed. Finally, GekkoFS is hardware independent to efficiently use today's network technologies as well as any modern and future storage hardware that is accessible by the user.

2.3.1. POSIX relaxation

Similarly to PVFS [b-5] and OrangeFS [b-18], GekkoFS does not provide complex global locking mechanisms. In this sense, applications should be responsible to ensure that no conflicts occur, in particular, w.r.t. overlapping file regions. However, the lack of distributed locking has consequences for operations where the number of affected file system objects is unknown a priori, such as readdir() called by the ls -l command. In these indirect file system operations, GekkoFS does not guarantee to return the current state of the directory and

follows the eventual-consistency model. Furthermore, each file system operation is synchronous without any form of caching to reduce file system complexity and to allow for an evaluation of its raw performance capabilities.

GekkoFS does not support move or rename operations or linking functionality as HPC application studies have shown that these features are rarely or not used at all during the execution of a parallel job [b-14]. Finally, security management in the form of access permissions is not maintained by GekkoFS since it already implicitly follows the security protocols of the node-local file system.

2.3.2. Architecture

Figure 4 depicts the GekkoFS' architecture that consists of two main components: a client library and a server process. An application that uses GekkoFS must first preload the client interposition library which 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 independently, sending a response when finished. In the following paragraphs, we describe the client and daemon in more detail.

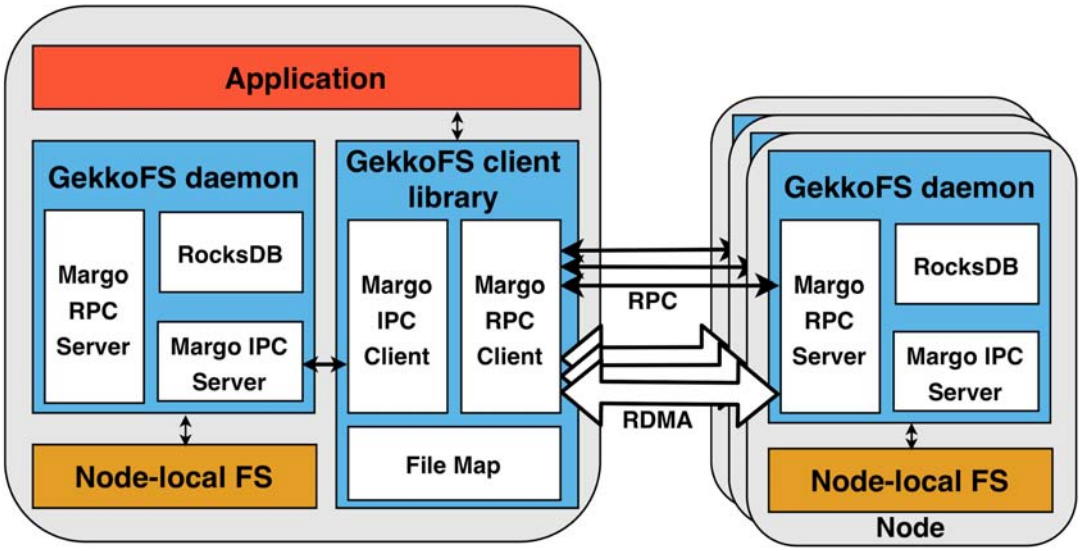


Figure 4 - GekkoFS architecture

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

Each file system operation is forwarded via an RPC message to a specific daemon (determined by hashing of the file's path) where it is directly executed. In other words, GekkoFS uses a pseudo-random distribution to spread data and metadata across all nodes, also known as wide-striping. Because each client is able to independently resolve the responsible node for a file system operation, GekkoFS does not require central data structures that keep track of where metadata or data is located. To achieve a balanced data distribution for large files, data requests are split into equally sized chunks before they are distributed across file system nodes. If supported by the underlying network fabric protocol, the client exposes the relevant chunk memory region to the daemon, accessed via remote-direct-memory-access (RDMA).

b) GekkoFS daemon: GekkoFS daemons consist of three parts: 1) A key-value store (KV store) used for storing metadata; 2) an I/O persistence layer that reads/writes data from/to the underlying local storage system (one file per chunk); and 3) an RPC-based communication layer that accepts local and remote connections to handle file system operations.

Each daemon operates a single local RocksDB KV store [b-8]. RocksDB is optimized for NAND storage technologies with low latencies and fits GekkoFS' needs as SSDs are primarily used as node-local storage in today's HPC clusters.

For the communication layer, we leverage on the Mercury RPC framework [b-30]. It allows GekkoFS to be network-independent and to efficiently transfer large data within the file system. Within GekkoFS, Mercury is interfaced indirectly through the Margo library that provides Argobots-aware wrappers to Mercury's API with the goal to provide a simple multi-threaded execution model [b-6], [b-29]. Using Margo allows GekkoFS daemons to minimize resource consumption of Margo's progress threads and handlers which accept and handle RPC requests [b-6].

2.4. Evaluation

We evaluated the performance of GekkoFS based on various unmodified microbenchmarks which catch access patterns that are common in HPC applications. Our experiments were conducted on the MOGON II supercomputer, located at the Johannes Gutenberg University Mainz in Germany. All experiments were performed on Intel 2630v4 Intel Broadwell processors (two sockets each). The main memory capacity inside the nodes ranges from 64 GiB up to 512 GiB of memory. MOGON II uses 100 Gbit/s Intel Omni-Path to establish a fat-tree network between all compute nodes. In addition, each node provides a data center Intel SATA SSD DC S3700 Series as scratch-space (XFS formatted) usable within a compute job. We used these SSDs for storing data and metadata of GekkoFS which uses an internal chunk size of 512 KiB.

Before each experiment iteration, GekkoFS daemons are restarted (requiring less than 20 seconds for 512 nodes), all SSD contents are removed, and kernel buffer, inode, and dentry caches are flushed. The GekkoFS daemon and the application under test are pinned to separate processor sockets to ensure that file system and application do not interfere with each other.

2.4.1. Metadata performance

We simulated common metadata intensive HPC workloads using the unmodified mdtest microbenchmark [b-17] to evaluate GekkoFS' metadata performance and compare it against a Lustre parallel file system. Although GekkoFS and Lustre have different goals, we point out the performance that can be gained by using GekkoFS as a burst buffer file system. In our experiments, mdtest performs create, stat, and remove operations in parallel in a single directory – an important workload in many HPC applications and among the most difficult workloads for a general-purpose PFS [b-33].

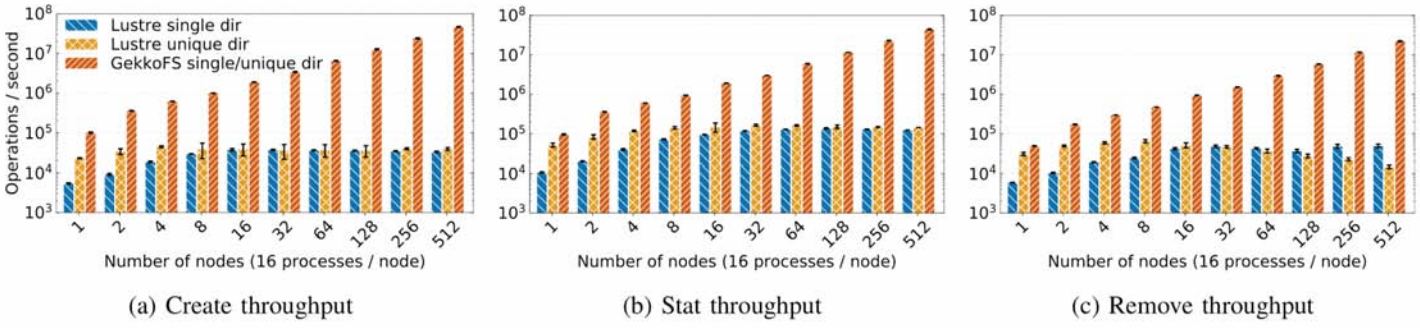


Figure 5 - GekkoFS' file create, stat, and remove throughput for an increasing number of nodes compared to a Lustre file system.

Each operation on GekkoFS was performed using 100,000 zero-byte files per process (16 processes per node). From the user application's perspective, all created files are stored within a single directory. However, due to GekkoFS' internally kept flat namespace, there is conceptually no difference in which directory files are created. This is in contrast to a traditional PFS that may perform better if the workload is distributed among many directories instead of in a single directory. Figure 5 compares GekkoFS with Lustre in three scenarios with up to 512 nodes: file creation, file stat, and file removal. The y-axis depicts the corresponding operations per second that were achieved for a particular workload on a logarithmic scale. Each experiment was run at least five times with each data point representing the mean of all iterations. GekkoFS' workload scaled with 100,000 files per process, while Lustre's workload was fixed to four million files for all experiments. We fixed the number of files for Lustre's metadata experiments because Lustre was otherwise detecting hanging nodes when scaling to too many files.

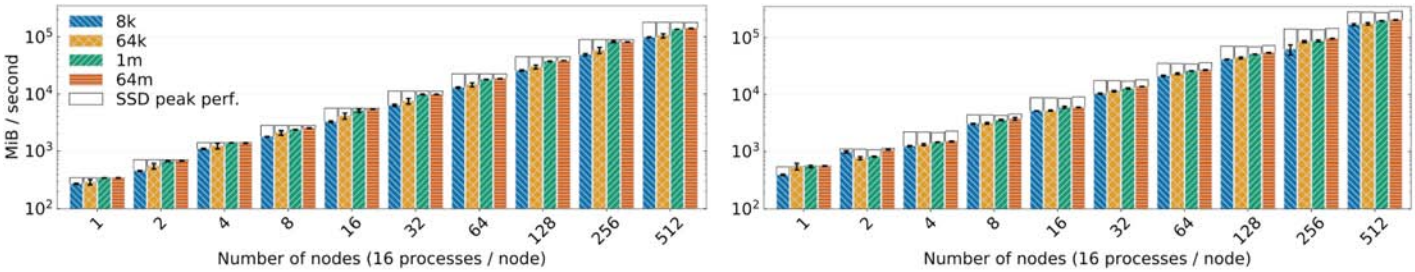
Lustre experiments were run in two configurations: All processes operated in a single directory (single dir) or each process worked in its own directory (unique dir). Moreover, Lustre's metadata performance was evaluated while the system was accessible by other applications as well.

As seen in Figure 5, GekkoFS outperforms Lustre by a large margin in all scenarios and shows close to linear scaling, regardless of whether Lustre processes operated in a single or in an isolated directory. Compared to Lustre, GekkoFS achieved around 46 million creates/s (~1,405x), 44 million stats/s (~359x), and 22 million removes/s (~453x) at 512 nodes. The standard deviation was less than 3.5% which was computed as the percentage of the mean.

2.4.2. Data performance

We used the unmodified IOR [b-17] microbenchmark to evaluate GekkoFS' I/O performance for sequential and random access patterns in two scenarios: Each process is accessing its own file (file-per-process) and all processes access a single file (shared file). We used 8 KiB, 64 KiB, 1 MiB, and 64 MiB transfer sizes to assess the performances for many small I/O accesses and for few large I/O requests. We ran 16 processes on each client, each process writing and reading 4 GiB in total.

GekkoFS data performance is not compared with the Lustre scratch file system as the peak performance of the used Lustre partition, around 12 GiB/s, is already reached for ≤ 10 nodes for sequential I/O patterns. Moreover, Lustre has shown to scale linearly in larger deployments with more OSSs and OSTs being available [b-20].



(a) Write throughput (b) Read throughput

Figure 6 - GekkoFS' sequential throughput for each process operating on its own file compared to the plain SSD peak throughput.

Figure 6 shows GekkoFS' sequential I/O throughput in MiB/s, representing the mean of at least five iterations, for an increasing number of nodes for different transfer sizes. In addition, each data point is compared to the peak performance that all aggregated SSDs could deliver for a given node configuration, visualized as a white rectangle, indicating GekkoFS' SSD usage efficiency. In general, every result demonstrates GekkoFS' close to linear scalability, achieving about 141 GiB/s (~80% of the aggregated SSD peak bandwidth) and 204 GiB/s (~70% of the aggregated SSD peak bandwidth) for write and read operations for 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 is for file system operations with a transfer size of 8 KiB.

For the file-per-process cases, sequential and random access I/O throughput are similar for transfer sizes larger than the file system's chunk size. This is due to transfer sizes larger than the chunk size internally access whole chunk files while smaller transfer sizes access one chunk at a random offset. Consequently, random accesses for large transfer sizes are conceptually the same as sequential accesses. For smaller transfer sizes, e.g., 8 KiB, random write and read throughput decreased by approximately 33% and 60%, respectively, for 512 nodes owing to the resulting random access to positions within the chunks. For the shared file cases, a drawback of GekkoFS' synchronous and cache-less design becomes visible. No more than approximately 150K write operations per second were achieved. This was due to network contention on the daemon which maintains the shared file's metadata whose size needs to be constantly updated. To overcome this limitation, we added a rudimentary client cache to locally buffer size updates of a number of write operations before they are send to the node that manages the file's metadata. As a result, shared file I/O throughput for sequential and random access were similar to file-per-process performances since chunk management on the daemon is then conceptually indifferent in both cases.

2.5. Summary

In this section, we gave an overview of GekkoFS, a new burst buffer file system for High Performance Computing applications with relaxed POSIX- semantics, allowing it to achieve millions of metadata operations even for a small number of nodes and close to linear scalability in various data and metadata use cases. As future work, it would be interesting to investigate GekkoFS' with various chunk sizes, to evaluate benefits of caching, and to explore different data distribution patterns.

3. Týr: Transactional Object Storage for HPC and Big Data Convergence

3.1. Context and Problematic

The digitalization of the everyday life drives the expansion of processes which not only produce growing amounts of real-time data, but also produce theoretical data through large-scale simulations. This fosters unprecedented opportunities to increase the knowledge that can be extracted from a system by submitting it to various hypothetical conditions that are hard or costly to observe in practice. Such perspective is especially relevant for the autonomous car industry, complementing sensor data with large-scale simulations to train the underlying algorithms, or to continuously improve the software without endangering real people on the streets. Unfortunately, such correlation of various computational models (i.e. bulk processing, stream processing, simulations) is difficult because the HPC and Big Data Analytics (BDA) stacks remain mostly separated today. A few works have started to tackle the convergence between HPC and BDA through a top-down approach. However, the staggering divergence at the lower levels of the software stack (i.e., storage) prevents application portability or data sharing between platforms.

This highlights the need for a new storage model that bridges the gap between HPC and BDA applications. It should handle ever-increasing amounts of data that are challenging for legacy systems to manage while also providing these applications with the variety of semantics they expect (e.g., parallel file systems, key-value stores, wide column databases or graph databases). The latter requirement can only be met by decoupling the data storage layer from the data presentation layer.

In this third result, we demonstrated that a simple-enough (*e.g.*, object storage) storage layer could cross the boundary between HPC and BDA platforms while serving as a base building block for a wide variety of storage abstractions the applications expect. The architecture we proposed is highlighted on Figure 7.

Object storage has proven to be a strong storage model both for extreme and cloud computing. In the former, it serves as support for Lustre [c-1] or DeltaFS [c-11] while in the latter it is the base building block for the Ceph [c-9] file system, or is exposed directly to the user in the form of key-value or document stores [c-3, c-6, c-8, c-2].

Despite this similarity in approaches leveraged, several critical divergences remain between the HPC and Cloud communities. Among the most important ones are the choices made with respect to storage consistency. This can include synchronizing storage operations or ensuring that the storage system remains in a consistent state in case of failures, and is critical for a range of applications that include real-time indexing or data aggregation systems. While the HPC community tends to control such consistency at the application level, the Cloud community typically delegates this task to the storage system through the use of storage transactions [c-4].

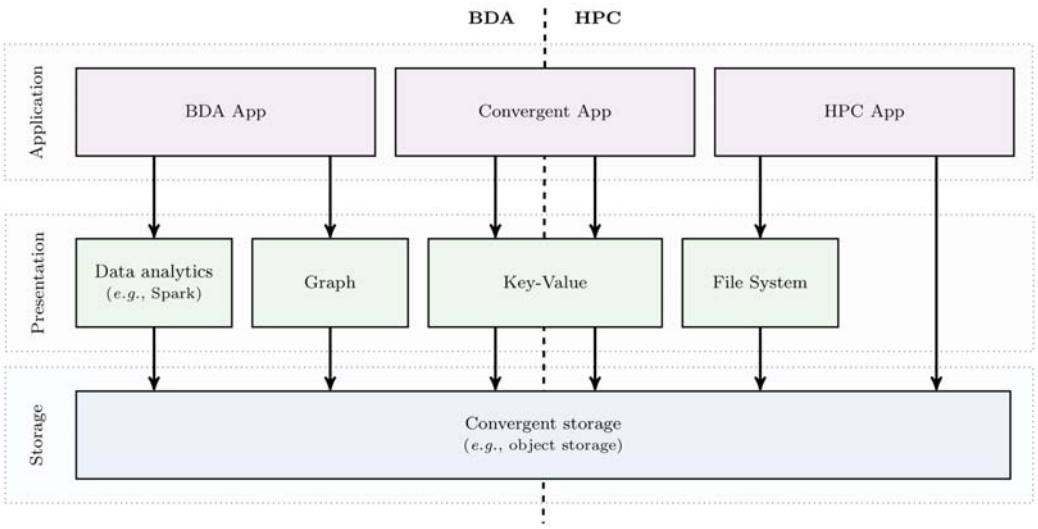


Figure 7 - Týr Storage Architecture Vision

Transactions essentially group multiple storage operations as a single task that is applied consistently and atomically to the storage system. Yet, no blob storage system today offer transaction support, which is difficult due to the challenges associated with the potentially large size of the objects.

3.2 Týr Design and Implementation

3.2.1. Design Principles

To address the aforementioned challenges, we propose a series of design principles for building a converging storage system based on the observations mentioned in the previous paragraph. In particular, we defend that this system should provide transactions as a way to enforce strong consistency at the storage level in order to offer support for a variety of use-cases such as real-time data aggregation. The key design principles that lay the foundation of Týr, in support of HPC and BDA convergence, are:

Predictable data distribution. Similarly with Rados [c-10] as well as a number of key-value stores [c-3, c-8, c-5], Týr enables clients to access any piece of data without any prior communication with any metadata node. This ensures low read and write latency by eliminating remote fetches as happens in the case of BlobSeer [c-7].

Transparent multi-version concurrency control. Multi-version concurrency

control isolates readers from concurrent writers. Essentially, it enables writers to modify a copy of the original data while enabling concurrent readers to access the most recent version. Each user connected to the database sees a snapshot of the database at a particular instant in time. Any changes made by a writer will not be seen by other clients until the changes have been completed. This approach proves critical to the performance of BlobSeer under high contention. Contrary to BlobSeer, Týr does not expose multiple versions to the reader, keeping this complexity hidden from the clients.

ACID transactional semantics. Týr offers built-in ACID transactions as a way to ensure the consistency and correctness of multi-object operations. The lightweight transactional protocol is core to the internals of Týr. It notably ensures Týr linearizability, and supports its replication-based fault- tolerance as well as the internal version garbage collection.

Atomic transform operations. Atomic transform operations enable users to perform simple binary data manipulation in-place, with a single round-trip between the client and the storage cluster. This includes binary additions, multiplications or bitwise operations. Such operations are key to efficient data aggregation. They are supported by the internal transactional protocol of Týr.

3.2.1. Implementation

We implement the design principles of Týr in a software prototype. This includes the Týr server, an asynchronous C client library, as well as partial C++ and Java bindings. The server itself is approximately 25,000 lines of Rust and GNU C code. This section describes key aspects of the implementation.

Týr is internally structured around multiple, lightweight and specialized event-driven loops, backed by the LibUV library [c-12]. When a request is received, it is forwarded to one of the relevant event loops for further asynchronous processing. No request queueing is done in order to avoid communication delays, and thus reduce the overall latency of the server. On-disk data and metadata storage uses Google's LevelDB key-value store [c-13], a state-of-the-art log-structured merge tree [c-14] based library optimized for high-throughput.

The intra-cluster and client-server request/response messages are serialized with Google's FlatBuffers [c-15] library. It allows message serialization and deserialization without parsing, unpacking, or any memory allocation. These messages are transmitted using the UDT protocol [c-16]. UDT is a reliable UDP-based application level data transport protocol. UDT uses UDP to transfer bulk data with its own reliability control and congestion control mechanisms. This enables transferring data at a much higher speed than TCP.

3.3 Evaluation

We evaluated the Týr proposal through several experiments and use-cases [c-17, c-18, c-19]. In this document, we limit the discussion to three evaluations. First, we discuss the transactional write performance of a Týr cluster with a heavily-concurrent usage pattern, Second, we analyze the raw read performance of the system. Finally, we gauge the reader/writer isolation in Týr.

3.3.1. Experimental Protocol

Experimental setup: We deployed Týr on the Microsoft Azure Compute platform on up to 256 nodes. For all experiments, we used D2 v2 general-purpose instances, located in the East US region (Virginia). Each virtual machine has access to 2 CPU cores, 7 GB RAM and 60 GB SSD storage. The host server is based on 2.4 GHz Intel Xeon E5-2673 v3 [c-20] processors and is equipped with 10 Gigabit ethernet.

Evaluated systems: To the best of our knowledge, no distributed storage systems with a comparable low-level data model and built-in transactions are available today. Throughout these experiments, we compared Týr with RADOS [c-10], a distributed blob storage system developed as part as Ceph [c-9]. RADOS is based on a decentralized architecture and does not make use of Multiversion Concurrency Control. We also compared Týr with BlobSeer [c-7], an open-source, in-memory distributed storage system which shares the same data model and a similar API. BlobSeer has been designed to support a high-throughput for highly-concurrent accesses to shared distributed blobs. Unlike Týr, BlobSeer distributes the meta- data over the cluster by means of a distributed tree. Finally, we compared Týr with Microsoft Azure Storage blobs, a fully-managed blob storage system provided as part of the Microsoft Azure cloud platform. This system comes in three flavors: append blobs, block

blobs and page blobs. Append blobs are optimized for append operations, block blobs are optimized for large uploads, and page blobs are optimized for random reads and writes [c-21]. For our experiments, we used RADOS 0.94.4 (Hammer), BlobSeer 1.2.1, and the version of Azure Storage. Blobs available at the time these experiments were run.

Dataset and workload: In order to run these experiments, we used a dump of real data obtained from the MonALISA system [c-22]. This data set is composed of ~4.5 million individual measurement events, each one being associated to a specific monitored site. We used multiple clients to replay these events, each holding a different portion of the data. Clients are configured to loop over the data set to generate more events when the size of the data is not sufficient. The read operations were performed by querying ranges of data, simulating a realistic usage of the MonALISA interface. In order to further increase read concurrency, the data was queried following a power-law distribution.

Experimental configuration: Because of the lack of native transaction support in Týr competitors, we used ZooKeeper 3.4.8 [c-23] (ZK), an industry-standard, high-performance distributed synchronization service, which is part of the Hadoop [c-24] stack. Zookeeper allows us to synchronize writes to the data stores with a set of distributed locks. ZooKeeper locks are handled at the lowest-possible granularity: one lock is used for each aggregate offset (8-byte granularity), except for Azure in which we had to use coarse-grained locks (512- byte granularity). This is because Azure page blobs, which we used for storing the aggregates, requires writes to be aligned on a non-configurable 512-byte page size [c-21]. We have used page blobs because of their random write capabilities. Furthermore, append blobs are not suited for operating on small data objects such as MonALISA events: they are limited to 50.000 appends.

3.3.2. Transactional Write Performance

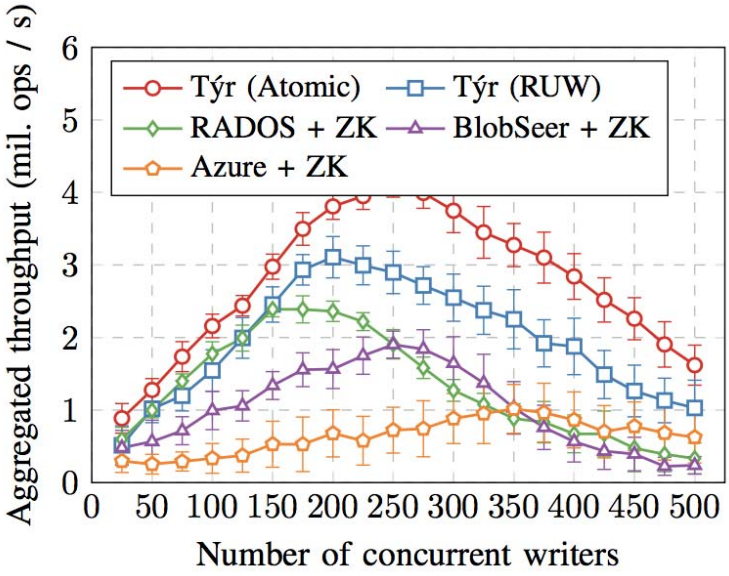


Figure 8 - Write Performance of Týr, RADOS, BlobSeer and Azure Storage varying the number of concurrent clients, with 95% confidence interval.

High transactional write performance is the key requirement that guided the design of Týr. To benchmark the different systems in this context, we measured the transactional write performance of Týr, RADOS, BlobSeer with the MonALISA workload. We also measured the performance of the same workload on the Azure Storage platform. Týr uses atomic operations. However, being the only system to support such semantics, we also tested the Týr behavior with regular read- update-write (RUW) operations as a baseline. Týr

transactions are required to synchronize the storage of the events and their indexing in the context of a concurrent setup. We used ZooKeeper to synchronize writes on the other systems. All systems were deployed on a 32-node cluster, except for Azure Storage which does not offer the possibility to tune the number of machines in the cluster.
The results, depicted in Figure 8, show that the Týr peak throughput outperforms its competitors by 78% while supporting higher concurrency levels. Atomic updates allowed Týr to further increase performance by saving the cost of read operations for simple updates. The significant drop of performance in the case of RADOS, Azure Storage and BlobSeer at higher concurrency levels is due to the increasing lock contention. This issue appears most frequently on the global aggregate blob, which is written to for each event indexed. In contrast, our measurements show that Týr's performance drop is due to CPU resource exhaustion. Under lower concurrency, however, we can see that the transaction protocol incurs a slight processing overhead, resulting in a comparable performance for Týr and RADOS when the update concurrency is low. BlobSeer is penalized by its tree-based metadata management which incurs a non-negligible overhead compared to Týr and RADOS. Overall, Azure shows a lower performance and higher variability than all systems. At higher concurrency levels however, Azure performs better than both RADOS and BlobSeer. This could be explained by a higher number of nodes in the Azure Storage cluster, although the lack of visibility over its internals doesn't allow to draw any conclusive explanation. We can observe the added value of in-place atomic operations in the context of this experiment, which enables Týr to increase its performance by 33% by avoiding the cost of read operations for simple updates.

3.3.3. Read performance

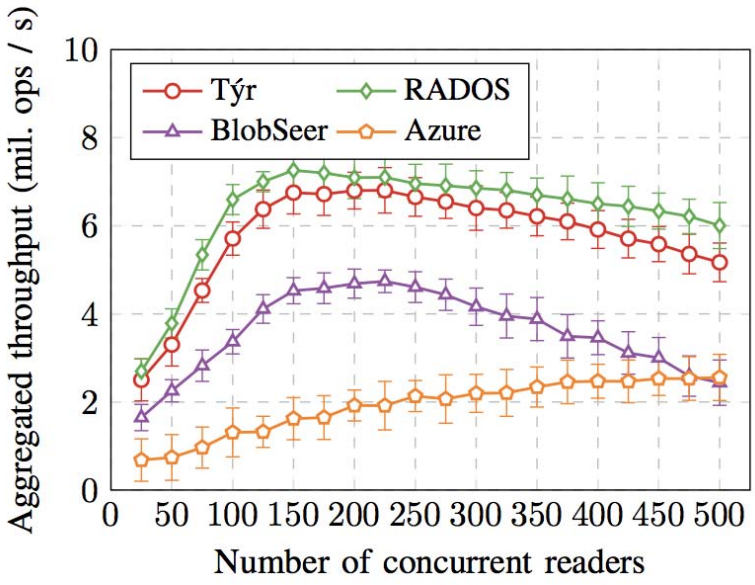


Figure 9 - Read Performance of Týr, RADOS, BlobSeer, and Azure Storage, varying the number of concurrent clients, with 95% confidence interval.

We evaluated the read performance of a 32-node Týr cluster and compared it with the results obtained with RADOS and BlobSeer on a similar setup. As a baseline, we measured the same workload on the Azure Storage platform. We preloaded in each of these systems the whole MonALISA dataset, for a total of around 100 Gigabytes of uncompressed data. We then performed random reads of 800 Byte size each from both the raw data and the aggregates, following a power- law distribution to increase read concurrency. This read size corresponds to a typical 100-minute average of MonALISA aggregated data. To prevent memory overflows, servers throttle the number of concurrent requests in the system. We

configured Týr to have a maximum of 1,000 total concurrent requests in process. We plotted the results in Figure 9.
The lightweight read protocol of both Týr and RADOS allows them to exhaust the CPU resources quickly and to outperform both BlobSeer and Azure Storage peak throughput by 44%. On the other hand, BlobSeer requires multiple hops to fetch the data in the distributed metadata tree. This incurs an additional networking cost that limits the total performance of the cluster. Under higher concurrency, we observe a slow drop in throughput for all the compared systems except for Azure Storage due to the involved CPU in the cluster getting overloaded. Once again, linear scalability properties of Azure could be explained by the higher number of nodes in the cluster, although this can't be verified because of the lack of visibility over Azure internals.
Týr and RADOS show a similar performance pattern. Measurements show RADOS outperforming Týr by a margin of approximately 7%. This performance penalty can partly be explained by the overhead of the multiversion concurrency control in Týr, enabling it to support transactional operations.

3.3.4 Reader/writer isolation

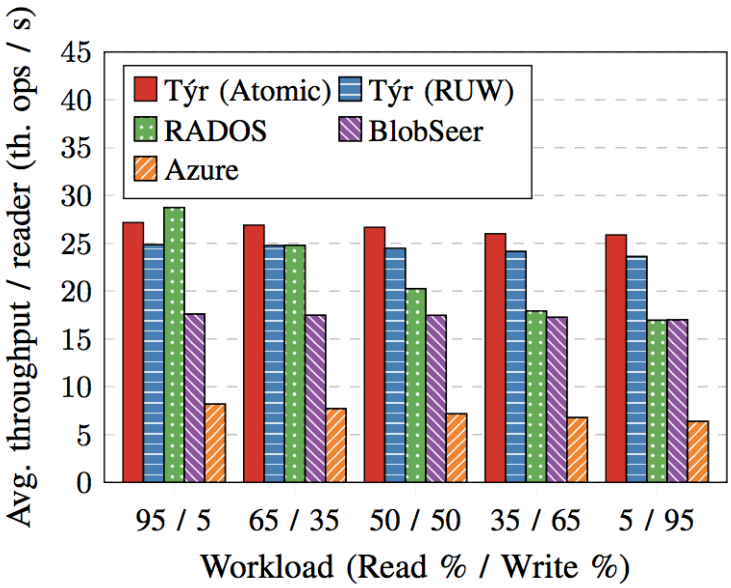


Figure 10 - Throughput of Týr, RADOS, BlobSeer, and Azure Storage for workloads with varying read to write ratio. Each bar represents the average read throughput with 200 concurrent clients averaged over a one-minute window of read-to-write ratio.

We performed simultaneously reads and writes in a 32-node cluster, using the same setup and methodology as with the two previous experiments. To that end, we preloaded half of the MonALISA dataset in the cluster and measured read performance while concurrently writing the remaining half of the data. We ran the experiments using 200 concurrent clients. With this configuration, all three systems proved to perform above 85% of their peak performance for both reads and writes, thus giving comparable results and a fair comparison between the systems. Among these clients, we varied the ratio of readers to writers in order to measure the performance impact of different usage scenarios. For each of these experiments, we were interested in the average throughput per reader.
The results are depicted in Figure 10. They illustrate the added value of multiversion concurrency control, on which both Týr and BlobSeer are based. For these two systems, we observe a near stable average read performance per client despite the varying number of concurrent writers. In contrast, RADOS, which outperforms Týr for a 95/5 read-to-write ratio,

shows a clear drop in performance as this ratio decreases. Similarly, Azure shows a slight drop in performance as the number of concurrent writers increases.

3.4 Summary

We propose a set of key design principles that enable building a storage system targeted at HPC and BDA convergence by overcoming most limitations of the existing approaches. First, we advocate the relevance of predictable data and metadata distribution using consistent hashing techniques to enable direct read and write access to data without requiring centralized metadata management, while also uniformly distributing the load across the cluster. Transparent multi-version concurrency control supports high write performance for highly-contended I/O patterns that are increasingly common for a variety of applications (i.e. append logs, data aggregation). These features are leveraged to provide ACID transactional semantics to ensure data consistency while letting users orchestrate multiple storage calls without requiring complex application- level coordination such as locking. Finally, these semantics enable atomic transform operations which bring significant throughput increase to applications performing data aggregation by reducing the amount of communication required for simple update patterns.

From the performance viewpoint, we evaluated our prototype Týr using a real-world use case from the CERN LHC on the Microsoft Azure cloud: its throughput outperforms that of state-of-the-art systems by more than 75%. Additional experiments (not presented in this document) demonstrated that Týr scales on large clusters of commodity machines to support millions of concurrent read and write operations per second.

Ongoing works focus on the use of Týr in a multiple data centers context.

REFERENCES

[a-1] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In Cloud Computing and Services Science , IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer International Publishing, 3–20. https://doi.org/10.1007/978-3-319-04519-1_1

[a-2] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal De Lara. 2011. Kaleidoscope: cloud micro-elasticity via VM state coloring. In Proceedings of the sixth conference on Computer systems. ACM, 273–286.

[a-3] Pradipta De, Manish Gupta, Manoj Soni, and Aditya Thatte. 2012. Caching VM instances for fast VM provisioning: a comparative evaluation. In European Conference on Parallel Processing . Springer, 325–336.

[a-4] Debian. 2011. Initial ramdisk. https://wiki.debian.org/initramfs

[a-5] Hoyte Doug. 2012. vmtouch: the Virtual Memory Toucher. https://hoytech.com/vmtouch/

[a-6] Alberto Garcia. 2017. Improving the performance of the qcow2 format. https://events.static.linuxfound.org/sites/events/files/slides/kvm-forum-2017-slides.pdf.

[a-7] The PostgreSQL Global Development Group. 2014. Pgbench benchmark. https://www.postgresql.org/docs/9.4/static/pgbench.html

[a-8] Red Hat. 2012. libvirt: The virtualization API.

[a-9] Keren Jin and Ethan L Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In Proceedings of International Conference on Systems and Storage (ACM SYSTOR 2009). ACM, 7.

[a-10] Thomas Knauth and Christof Fetzer. 2014. DreamServer: Truly on-demand cloud services. In Proceedings of International Conference on Systems and Storage (ACM SYSTOR). ACM.

[a-11] KVM. 2012. The QCOW2 Image. https://kashyapc.fedorapeople.org/virt/lc-2012/snapshots-handout.html

[a-12] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: rapid virtual machine cloning for cloud computing. InProceedings of the 4th ACM European conference on Computer systems. ACM, 1–12.

[a-13] Linux. 1995. mincore.https://linux.die.net/man/2/mincore

[a-14] Ming Mao and Marty Humphrey. 2012. A performance study on the VM startup time in the cloud. In Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE, 423–430.

[a-15] Thuy Linh Nguyen and Adrien Lèbre. 2017. Virtual Machine Boot Time Model. In Parallel, Distributed and Network-based Processing (PDP), 201725th Euromicro International Conference on. IEEE, 430–437.

[a-16] Bogdan Nicolae, Franck Cappello, and Gabriel Antoniu. 2011. Optimizing multideployment on clouds by means of self-adaptive prefetching.In European Conference on Parallel Processing. Springer, 503–513.

[a-17] Bogdan Nicolae and M Mustafa Rafique. 2013. Leveraging collaborative content exchange for on-demand VM multi-deployments in iaas clouds.In European Conference on Parallel Processing. Springer, 305–316.

[a-18] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and extension of user space file systems. InProceedings of the 2010 ACM Symposium on Applied Computing. ACM, 206–213.

[a-19] Kaveh Razavi and Thilo Kielmann. 2013. Scalable virtual machine deployment using VM image caches. InProceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, 65.

[a-20] Kaveh Razavi, Liviu Mihai Razorea, and Thilo Kielmann. 2013. Reducing VM startup time and storage costs by VM image content consolidation. In Euro-Par 2013: Parallel Processing Workshops. Springer,75–84.

[a-21] Kaveh Razavi, Gerrit Van Der Kolk, and Thilo Kielmann. 2015. Preb-akedµvms: Scalable, instant VM startup for IAAS clouds. In Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on. IEEE, 245–255.

[a-22] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices.ACM SIGOPS Operating Systems Review 42, 5 (2008).

[a-23] SEAS. 2004. Stress.http://people.seas.harvard.edu/~apw/stress/

[a-24] SUSE. 2010.Description of Cache Modes. https://www.suse.com/documentation/sles11/book_kvm/data/sect1_1_chapter_book_kvm.html

[a-25] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017.To FUSE or Not to FUSE: Performance of User-Space File Systems.. InFAST. 59–72.

[a-26] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. 2005. Scalability, fidelity, and containment in the potemkin virtual honey farm. In ACM SIGOPS Operating Systems Review, Vol. 39. ACM, 148–162.

[a-27] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. InProceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 307–320.12

[b-1] Infinite Memory Engine. https://www.ddn.com/products/ime-flash-native-data-cache.

[b-2] J. Bent, G. A. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, “PLFS: a checkpoint file system for parallel applications,” in Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA, 2009.

[b-3] P. J. Braam and P. Schwan, “Lustre: The intergalactic file system,” in Ottawa Linux Symposium, 2002, p. 50.

[b-4] P. Carns, Y. Yao, K. Harms, R. Latham, R. Ross, and K. Antypas, “Production i/o characterization on the cray xe6,” in Proceedings of the Cray User Group meeting, vol. 2013, 2013.

[b-5] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, “PVFS: A parallel file system for linux clusters,” in 4th Annual Linux Showcase & Conference 2000, Atlanta, Georgia, USA, October 10-14, 2000, 2000.

[b-6] P. H. Carns, J. Jenkins, C. D. Cranor, S. Atchley, S. Seo, S. Snyder, and R. B. Ross, “Enabling NVM for data-intensive scientific services,” in 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW@OSDI 2016, Savannah, GA, USA, November 1, 2016., 2016.

[b-7] P. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, “Input/output characteristics of scalable parallel applications,” in Proceedings Supercomputing ’95, San Diego, CA, USA, December 4-8, 1995, 1995, p. 59.

[b-8] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, “Optimizing space amplification in rocksdb,” in CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings, 2017.

[b-9] M. Dorier, G. Antoniu, R. B. Ross, D. Kimpe, and S. Ibrahim, “Calciom: Mitigating I/O interference in HPC systems through cross-application coordination,” in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 155– 164.

[b-10] M. Folk, A. Cheng, and K. Yates, “Hdf5: A file format and i/o library for high performance computing applications,” in Proceedings of supercomputing, vol. 99, 1999, pp. 5–33.

[b-11] W. Frings, F. Wolf, and V. Petkov, “Scalable massively parallel I/O to task-local files,” in Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA, 2009.

[b-12] F. Herold, S. Breuner, and J. Heichler, “An introduction to beegfs,” 2014, https://www.beegfs.io/docs/whitepapers/Introduction to BeeGFS by ThinkParQ.pdf.

[b-13] T. Hey, S. Tansley, and K. M. Tolle, Eds., The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, 2009.

[b-14] P. H. Lensing, T. Cortes, J. Hughes, and A. Brinkmann, “File system scalability with highly decentralized metadata on independent storage devices,” in IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, May 16-19, 2016, pp. 366–375.

[b-15] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, “On the role of burst buffers in leadership-class storage systems,” in IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA, 2012, pp. 1–11.

[b-16] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, “Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS),” in 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE@HPDC 2008, Boston, MA, USA, June 23, 2008, 2008, pp. 15–24.

[b-17] “Mdtest metadata benchmark and ior data benchmark,” 2018, https://github.com/hpc/ior.

[b-18] M. Moore, D. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Quarles, S. Sampson, S. Yang, and B. Wilson, “Orangefs: Advancing pvfs,” FAST poster session, 2011.

[b-19] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, “File-access characteristics of parallel scientific workloads,” IEEE Trans. Parallel Distrib. Syst., vol. 7, no. 10, pp. 1075–1089, 1996.

[b-20] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. K. , J. Rogers, J. James Simmons, and R. Miller, “Olcfs 1 tb/s, next-generation lustre file system,” in Proceedings of Cray User Group Conference (CUG 2013), 2013.

[b-21] S. Oral and G. Shah, “Spectrum scale enhancements for coral. presentation slides at supercomputing’16.” 2016, http://files.gpfsug.org/presentations/2016/SC16/11 Sarp Oral Gautam Shah Spectrum Scale Enhancements for CORAL v2.pdf.

[b-22] S. Patil and G. A. Gibson, “Scale and concurrency of GIGA+: file system directories with millions of files,” in 9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011, 2011, pp. 177–190.

[b-23] S. Patil, K. Ren, and G. Gibson, “A case for scaling HPC metadata performance through de-specialization,” in 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012, 2012, pp. 30–35.

[b-24] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. S¨uß, and A. Brinkmann, “A configurable rule based classful token bucket filter network request scheduler for the lustre file system,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, November 12 - 17, 2017, pp. 6:1–6:12.

[b-25] K. Ren, Q. Zheng, S. Patil, and G. A. Gibson, “Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion,” in International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014, 2014, pp. 237–248.

[b-26] R. Ross, R. Thakur, and A. Choudhary, “Achievements and challenges for i/o in computational science,” in Journal of Physics: Conference Series, vol. 16, no. 1, 2005, p. 501.

[b-27] R. B. Ross and R. Latham, “PVFS - PVFS: a parallel file system,” in Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA, 2006, p. 34.

[b-28] F. B. Schmuck and R. L. Haskin, “GPFS: A shared-disk file system for large computing clusters,” in Proceedings of the FAST ’02 Conference on File and Storage Technologies, January 28-30, Monterey, California, USA, 2002, pp. 231–244.

[b-29] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. H. Carns, A. Castell´o, D. Genet, T. Hérault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. H. Beckman, “Argobots: A lightweight low-level threading and tasking framework,” IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 3, pp. 512–526, 2018.

[b-30] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross, “Mercury: Enabling remote procedure call for high- performance computing,” in 2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013, 2013, pp. 1–8.

[b-31] S. Thapaliya, P. Bangalore, J. F. Lofstead, K. Mohror, and A. Moody, “Managing I/O interference in a shared burst buffer system,” in 45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016, 2016, pp. 416–425.

[b-32] Thinkparq and BeeGFS, “Beegfs the leading parallel cluster file system,” 2018, https://www.beegfs.io/docs/BeeGFS Flyer.pdf.

[b-33] M.-A. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann, “Challenges and solutions for tracing storage systems: A case study with spectrum scale,” ACM Trans. Storage, vol. 14, no. 2, pp. 18:1–18:24, 2018.

[b-34] M. Vilayannur, P. Nath, and A. Sivasubramaniam, “Providing tunable consistency for a parallel file store,” in Proceedings of the FAST ’05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA, 2005. [b-35] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty, “File system workload analysis for large scale scientific computing applications,” Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2004.

[b-36] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, “An ephemeral burst-buffer file system for scientific applications,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016, 2016, pp. 807–818.

[b-37] J. Xing, J. Xiong, N. Sun, and J. Ma, “Adaptive and scalable metadata management to support a trillion files,” in Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA, 2009.

[b-38] S. Yang, W. B. Ligon III, and E. C. Quarles, “Scalable distributed directory implementation on orange file system,” Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI), 2011.

[c-1] P. J. Braam and Others. The Lustre storage architecture. White Paper, Cluster File Systems, Inc., Oct, 23, 2003.

[c-2] Kristina Chodorow. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. ” O’Reilly Media, Inc.”, 2013.

[c-3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. ACM SIGOPS operating systems review, 41(6):205– 220, 2007.

[c-4] Jim Gray and Andreas Reuter. Transaction processing: concepts and techniques. Elsevier, 1992.

[c-5] Rusty Klophaus. Riak core: building distributed applications without shared state. In ACM SIGPLAN Commercial Users of Functional Programming, page 14. ACM, 2010.

[c-6] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35– 40, 2010.

[c-7] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. Blobseer: Next-generation data management for large scale infrastructures. Journal of Parallel and distributed computing, 71(2):169–184, 2011.

[c-8] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: architecture of a real-time operational DBMS. Proceedings of the VLDB Endowment, 9(13):1389–1400, 2016.

[c-9] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 307–320. USENIX Association, 2006.

[c-10] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing’07, pages 35–44. ACM, 2007.

[c-11] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale file systems scale better without dedicated servers. In Proceedings of the 10th Parallel Data Storage Workshop, pages 1–6. ACM, 2015.

[c-12] LibUV, 2015, https://github.com/libuv/libuv

[c-13] LevelDB: A Fast Persistent Key-Value Store, 2015, https://github.com/ google/leveldb.

[c-14] P. O’Neil, E. Cheng et al., “The log-structured merge-tree (lsm-tree),” Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996.

[c-15] FlatBuffers, 2015, https://google.github.io/flatbuffers/index.html.

[c-16] PanFS, 2015, http://www.panasas.com/products/panfs.

[c-17] Pierre Matri, Yevhen Alforov, Alvaro Brandon, Michael Kuhn, Philip Carns, Thomas Ludwig, Could Blobs Fuel Storage-Based Convergence Between HPC and Big Data? In Proceedings of the IEEE International Conference on Cluster Computing, Sep 2017, Honolulu, USA. pp.81 – 86, 2017.

[c-18] Pierre Matri, María S. Pérez, Alexandru Costan, Gabriel Antoniu, TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication. In Proceedings of the

18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2018, Washington, United States. pp.452-461, 2018.

[c-19] Pierre Matri, Yevhen Alforov, Alvaro Brandon, María S. Pérez, Alexandru Costan, Gabriel Antoniu, Michael Kuhn, Philip Carns, Thomas Ludwig, Mission Possible: Unify HPC and Big Data Stacks Towards Application-Defined Blobs at the Storage Layer. Future Generation Computer Systems, Elsevier, pp.1-10, 2018.

[c-20] Intel Xeon E5-2673v2 processor, 2015, https://ark.intel.com/products/79930/Intel-20Xeon-Processor-20E5-2673-v2-25M-Cache-330-GHz

[c-21] Understanding Block Blobs, Append Blobs, and Page Blobs, 2015, https://msdn.microsoft.com/en- us/library/azure/ee691964.aspx.

[c-22] I. Legrand, H. Newman et al., “MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems,” Computer Physics Communications, vol. 180, no. 12, pp. 2472 – 2498, 2009

[c-23] P. Hunt, M. Konar et al., Zookeeper: Wait-free coordination for internet- scale systems. in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, ser. USENIXATC’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[c-24] Apache Hadoop, 2015, https://hadoop.apache.org.