

TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication

Pierre Matri, María Pérez, Alexandru Costan, Gabriel Antoniu

▶ **To cite this version:**

Pierre Matri, María Pérez, Alexandru Costan, Gabriel Antoniu. TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication. CCGRID 2018 - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2018, Washington, United States. pp.452-461, 10.1109/CCGRID.2018.00072 . hal-01892691

HAL Id: hal-01892691
https://hal.science/hal-01892691

Submitted on 10 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication

Pierre Matri*, María S. Pérez*, Alexandru Costan†‡, Gabriel Antoniu‡

*Universidad Politécnica de Madrid, Madrid, Spain, {pmatri, mperez}@fi.upm.es
†IRISA / INSA Rennes, Rennes, France, alexandru.costan@irisa.fr
‡Inria, Rennes, France, {alexandru.costan, gabriel.antoniu}@inria.fr

***Abstract*—Small files are known to pose major performance challenges for file systems. Yet, such workloads are increasingly common in a number of Big Data Analytics workflows or large-scale HPC simulations. These challenges are mainly caused by the common architecture of most state-of-the-art file systems needing one or multiple metadata requests before being able to read from a file. Small input file size causes the overhead of this metadata management to gain relative importance as the size of each file decreases. In this paper we propose a set of techniques leveraging consistent hashing and dynamic metadata replication to significantly reduce this metadata overhead. We implement such techniques inside a new file system named TýrFS, built as a thin layer above the Týr object store. We prove that TýrFS increases small file access performance up to one order of magnitude compared to other state-of-the-art file systems, while only causing a minimal impact on file write throughput.**

I. Introduction

A large portion of research in data storage, management and retrieval focuses on optimizing access performance for large files [1]–[5]. Yet, handling a large number of small files raises other difficult challenges, that are partly related to the very architecture of current file systems. Such small files, with a size inferior to a few megabytes, are very common in large-scale facilities, as shown by multiple studies [6], [7]. They can be generated by data-intensive applications such as CM1 [8] or HACC [9], Internet of Things or Stream Processing applications, as well as large scale workflows such as Montage [10], CyberShake [11] or LIGO [12]. Improving file access performance for these applications is critical for scalability in order to handle ever-growing data sets on large-scale systems [13].

As the amount of data to be transferred for storage operations on any single small file is intuitively small, the key to optimizing access performance for such files lies in improving the efficiency of the associated metadata management. Actually, as the data size for each file decreases, the relative overhead of opening a file is increasingly significant. In our experiments, with small enough files, opening a file may take up to an order of magnitude more time than reading the data it contains. One key cause of this behavior is the separation of data and metadata inherent to the architecture of current file systems. Indeed, to read a file, a client must first retrieve the metadata for all folders in its access path, that may be located on one or more metadata servers, to check that the user has the correct access rights or to pinpoint the location of the data in the system. The high cost of network communication significantly exceeds the cost of reading the data itself.

We advocate that a different file system architecture is necessary to reduce the cost of metadata management for such workloads involving many small files. While one could think of co-locating data and metadata, a naive implementation would pose significant performance or consistency issues [14]. For example, the metadata for a directory containing large numbers of files would potentially be replicated to an important number of servers, both wasting precious memory resources and causing low metadata update performance. While others worked on optimizing metadata management [15]–[19], their contributions iterate on existing architectures, improving the internal handling of operations at the metadata server by means of caching or compression, without fixing the root cause of the issue, that is, the metadata distribution.

In this paper we propose a completely different approach and design a file system from the bottom up for small files without sacrificing performance for other workloads. This enables us to leverage some design principles that address the metadata distribution issues: consistent hashing [20] and dynamic data replication [21]. Consistent hashing enables a client to locate the data it seeks without requiring access to a metadata server, while dynamic replication adapts to the workload and replicates the metadata on the nodes from which the associated data is accessed. The former is often found in key-value stores [22], [23], while the latter is mostly used in geo-distributed systems [24]–[26].

We briefly motivate this research leveraging a short experiment (Section II) and review the related work (Section III). Our contributions are threefold:

- **We propose a novel file system architecture optimized for small files** based on consistent hashing and dynamic metadata replication while not sacrificing performance for other workloads (Section IV).
- We detail the design and implementation of **TýrFS, a file system implementing the aforementioned principles** (Section V), built atop the Týr object storage system [27].
- After introducing our experimental platform (Section VI), **we leverage TýrFS in various applicative contexts and factually prove its applicability to both HPC and Big Data workloads** (Section VII). We prove that it significantly outperforms state-of-the-art file systems, increasing throughput by up to one order of magnitude.

We finally discuss the limitations of our approach (Section VIII) and conclude on the future work that could further enhance our proposal (Section IX).

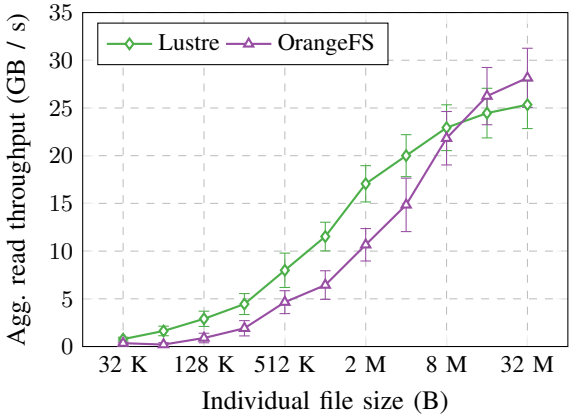


Fig. 1. Lustre and OrangeFS read throughput for a fixed-size data set when varying the size of the individual files it is composed of.

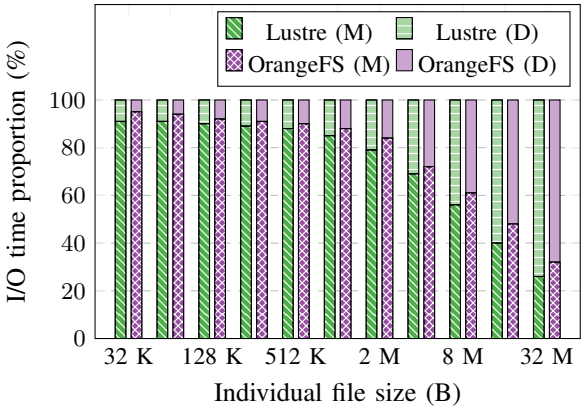


Fig. 2. Breakdown of reads between metadata (M) and data (D) access for a fixed-size data set varying the size of the individual files it is composed of.

II. Motivation: overhead of reading small files

In this section we seek to describe in more detail the issue by studying the concrete performance effects of reducing file size with a large data set. For that matter, we plot the time taken and file system CPU usage when reading a constant-size data set while decreasing the size of the files it contains.

We perform these experiments using both the Lustre [28] (2.10.0) and OrangeFS [29] (2.9.6) distributed file systems, that we setup on 24 nodes of the Paravance cluster in Rennes (France) of the Grid'5000 testbed [30]. On 24 other machines of the same cluster (384 cores) we launch a simple ad-hoc MPI benchmark reading in parallel the contents of a pre-loaded set of files. The set size varies between 38,400 $\times$ 32 MB files (100 files per rank) to 3.84×10^7 $\times$ 32 KB files (100,000 files per rank), for a total combined size of 1.22 TB.

In Figure 1 we plot the aggregate throughput achieved by Lustre and OrangeFS. For both systems we note a decrease of the read throughput as we reduce the size of the files. The decrease is as significant as one order of magnitude when replacing 32 MB by 32 KB files.

In Figure 2 we detail these results. We calculate the total execution time of the benchmark with Lustre and OrangeFS. We highlight the proportion of this time spent on the `open` operation fetching the metadata (M) and on the `read` operation actually reading the data (D). Interestingly, with smaller file sizes, the `open` operation takes a significant portion of the total I/O time. This proportion is above 90% for 32 KB files for both systems, while it is less than 35% for larger file sizes.

These results clearly demonstrate that reducing the size of the files puts a very high pressure on the file system metadata. This is caused by the very architecture of most file systems, in which metadata and data are separated from each other. This implies that metadata needs to be accessed before performing any read operation. Consequently, we argue that reducing the cost of metadata management without changing the file system architecture cannot possibly address the root cause of the performance drop.

These observations guide the design and implementation of TýrFS, which we detail in Sections IV and V.

III. Related work: metadata optimisations

While improving I/O performance when accessing small files in a distributed file system is a known problem, most of the work to date focuses on optimizing metadata management and assumes constant file system architecture. Two main solutions appear: reducing resource consumption required for metadata management, or completely eluding the problem with combining a set of small files into a smaller set of larger files.

Optimizing metadata management can be done in a number of ways. In [16], Carns *et al.* study with PVFS [31] the impact of various strategies such as metadata commit coalescing, storing data right inside inodes or pre-creating files. While highly effective, those optimizations are orientated towards improving small file *write* performance and have little to no effect on *read* operations. Creating archives from a large number of small files is the second frequent option found in the literature. For instance in HDFS [32], the HAR (Hadoop Archive) format [33] is leveraged by [19], [34] to transform an input dataset composed of many small files into a set of large files that can be handled efficiently by the metadata node. Building on this work, Vorapongkitipun *et al.* [35] propose a new archive format for Hadoop that further improves performance. While this approach is arguably extremely effective, it requires a pre-processing step to transform the input data set. Furthermore, this requires modifying the application so it is able to take such archives as input. As such, this approach is not applicable for workloads where small files are generated as intermediate data between processing steps. These challenges are partly addressed by Zhang *et al.* in [18], by delegating the merge task to a middleware process. While highly interesting, this approach retains the main drawback of archiving, that is, complexifying operations other than simple reads.

Overall, we note that the optimizations proposed in the literature having a significant impact on read performance rely on archiving, and optionally indexing files. While such approaches are indisputably effective for analyzing large sets of data, they invariably result in relaxing or dropping part of the POSIX I/O API. Notably, unlike this paper, no related work considers modifying a file after it has been written. While this may not be an issue for Big Data Analytics, this certainly is for HPC use cases relying on I/O frameworks such as HDF5 [36].

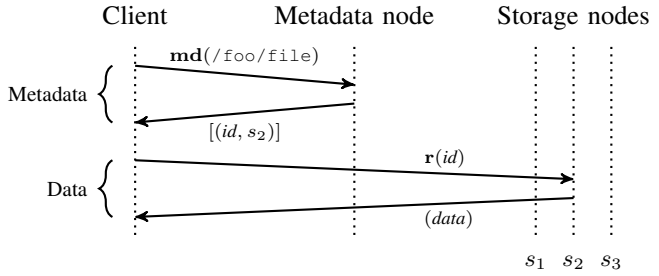


Fig. 3. Typical read protocol for a small file with centralized metadata management (*e.g.* Lustre or HDFS).

IV. Designing a file system for small files

In this section we detail the main architectural and design goals that drive the design of a file system optimized for small files. We consider as small files those for which it does not make sense to split them into multiple pieces (*chunking*). Since this notion is variable and cannot be defined precisely, we will consider as small - files with a size under a few megabytes.

A. Detailing the metadata distribution issue

First, let us consider the behavior of a typical distributed file system for reading a file, whose path is `/foo/file`. We consider the case of centralized metadata management, which is the base architecture of Lustre [28] or HDFS [32]. As shown in Figure 3, the request is performed in two steps. First, the client interrogates the metadata server to get the layout information of the file to read, including the actual node on which the data is located. Only with this information can the client access the data, in a second step. While the cost of such metadata fetching is amortized when reading large files, it cannot be with small ones. Should the user seek to read many files, the cost of this communication with the metadata node is to be paid for each file access. This largely explains the results observed in Section II.

The previous results also noticeably show a further performance degradation with OrangeFS [29] compared to Lustre. This is caused by the distribution of the metadata across the entire cluster, which brings better horizontal scalability properties at the cost of additional metadata requests, as illustrated in Figure 4. Note that the location of the metadata for any given file in the cluster is independent from the location of its data.

Intuitively, we understand that two problems are to be tackled when dealing with small files at scale:

- First, the client should be able to locate by itself the pieces of data to be read without requiring communication with a dedicated metadata server.
- Second, all the metadata required to read a file should be colocated with the file data whenever possible.

Those two steps would allow the client to bypass metadata requests entirely, consequently offering significantly better performance when dealing with many small files. While these principles are seemingly easy, they are challenging to implement for distributed file systems because of the hierarchical nature of their namespace.

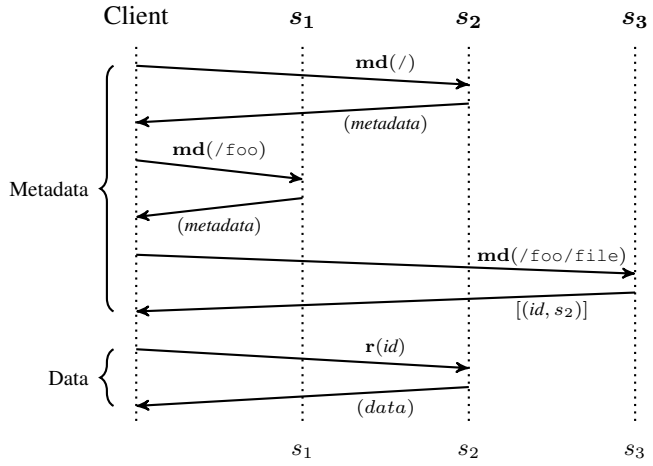


Fig. 4. Typical read protocol for a small file with decentralized metadata management on three storage nodes s_1, s_2 and s_3 (*e.g.* OrangeFS).

B. Predictably distributing data across the cluster

In typical file systems, the data is distributed across the cluster in a non-predictable fashion. A metadata process is responsible for assigning a set of nodes to each file based on some system-specific constraints [37]. This location is saved in the file metadata, which therefore needs to be queried in order to later pinpoint the location of the data in the cluster. While this allows a great latitude in finely tweaking data location, it inherently imposes at least two successive requests in order to retrieve data in the cluster. As the data is small, this request cost is non-amortizable in the case of small files.

This challenge has long been solved for state-of-the-art distributed object stores such as Dynamo [23] or Riak [38]. Indeed, instead of relying on metadata to locate information in a cluster, these systems distribute the data in a predictable fashion, using a common algorithm shared across the cluster and the clients. This effectively allows the clients to infer the location of any data they want to access without prior communication with any metadata node.

Such predictable data distribution is however challenging for file systems due to their hierarchical nature. For accessing any single file, a client needs to read the metadata of all the folder hierarchy for this file in order to check, for example, that the user has the correct permissions for reading it. Let us illustrate this with a simple example. A folder at `/foo` contains many files to be processed by a data analytics tool. To distribute the load across the storage nodes, these files should arguably be spread across the cluster. As such, while clients would be able to locate each file in the cluster because of the predictable location algorithm, they would still need to access the metadata for `/`, `/foo` and for the file to be read. All these metadata pieces are potentially located on a different set of servers, and would hence require additional communication that we are precisely seeking to eliminate.

Although such predictably data distribution is a step towards the solution, it cannot be considered independently of metadata management. Some file systems such as Ceph [39] partially leverage predictable distribution for reducing the metadata size without addressing the issue of its distribution.

C. Towards data and metadata colocation

To significantly reduce the overhead of metadata fetching we propose to collocate it with the data it concerns. Let us keep our example of the `/foo` folder containing many files. This implies that the node storing the data for the file `/foo/a.txt` would also hold the metadata for `/`, `/foo` and `/foo/a.txt`. We do so by leveraging an architecture roughly similar to that of OrangeFS. Each node composing the file system cluster integrates two services. First, a metadata service manages meta-information relative to folders and files. Second, a data service persists the data relative to the objects (files or folders) stored on this node.

As a first principle, we propose to always store the metadata for any given file on the same nodes the first chunk of the file is stored. As small files are by definition composed of a single chunk, this effectively eliminates the need for remotely fetching the file metadata before accessing its data. This contrasts with the design of OrangeFS, which does not guarantee such collocation. Such design is conceptually similar to the decision of embedding the contents of small-enough file right inside inodes on the Ext4 local file system.

Collocating the file data with the metadata of its folder hierarchy is a significantly harder problem. Indeed, a folder may contain many other folders and files potentially stored on different nodes. Specifically, the root folder itself is by definition ancestor of all folders and files. One could think of statically replicating this data to every relevant node. Yet, this solution would have huge implications in terms of storage overhead because of the potentially large amount of metadata replicated, and in terms of write performance because of the need to maintain consistency across all metadata replicas.

To alleviate this overhead, we replicate the metadata dynamically. The metadata agent on each node determines which of the metadata to replicate locally based on the current workload it serves. This solution is largely tackled in geo-distributed storage systems, in which dynamic replication is used to selectively replicate data objects to various discrete regions while minimizing write and storage overhead by adapting this replication based on user requests [24]–[26]. Our solution is largely inspired by their proved design and best practices.

We further reduce the storage overhead by replicating the metadata that is strictly necessary for accessing a file. Indeed, only a tiny subset of the whole metadata for the folder hierarchy of a file has to be accessed before reading it: permissions. In addition, we argue that permission changes are a very infrequent operation that is performed either when a folder is first created or as a manual management task which is unlikely to be time-sensitive ; we could not find any application in which permission change is in the critical path. As such, replicating the permissions is unlikely to have a significant negative effect on the performance of real-world applications. Intuitively, this solution implies that the closer a folder is to the root of the hierarchy, the more likely its permissions are to be replicated to a large number of nodes. As an extreme case, the permissions for the root folder itself would be replicated to every node in the cluster.

In Figure 5 we show a high-level overview of the proposed architecture, highlighting the different modules on each node composing the file system cluster.

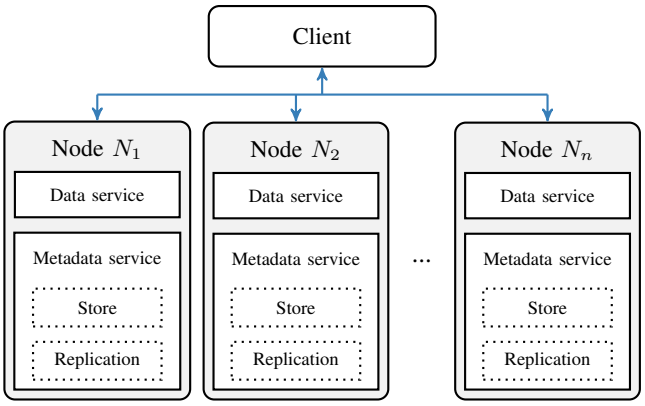


Fig. 5. High-level overview of the proposed architecture. Each node integrates both a data and a metadata service. The metadata service itself is composed of a local metadata store and the dynamic metadata replication service.

V. TýrFS: A file system optimized for small files

In this section we detail the implementation of the above principles in a prototype file system optimized for small files, that we name TýrFS. We implement it as a thin layer atop the Týr transactional, distributed object storage system [27]. The system is composed of a data service described in Section V-A, directly mapped to Týr native semantics. A lightweight metadata management module runs on each server node. We describe in detail its implementation in Section V-B. Finally, a client library exposes both a mountable and a native interface optimized for TýrFS. We detail it in Section V-C.

A. Implementation of the data service

The data service directly leverages the capabilities natively offered by Týr to efficiently distribute and store data in the cluster. A file in TýrFS is directly mapped to a blob in Týr, with the blob key being the absolute file path.

The predictable distribution of the data across the cluster that we describe in Section IV-B is inherited from Týr. The base system leverages consistent hashing of an object absolute path to distribute data across the cluster. Such method has been demonstrated to fairly distribute the load across the whole cluster in distributed systems, and to be applicable to local file systems [40]. The stored data can thus be located and accessed by TýrFS clients directly. Read and write operations as per the POSIX I/O standard are mapped directly to the blob read and write operations offered by Týr. Folders are also stored as a blob holding a list of all objects contained in it. A folder list operation only requires reading the blob ; adding a file to a folder is implemented as a blob append. A background process compacts the file list when objects are deleted from a folder.

A side effect of using consistent hashing for data distribution is the need to move the physical location of the data when a file is moved in the folder hierarchy. This operation is implemented by writing "staple" records at the new location of the object and of its descendents, referencing their previous location. The data transfer occurs in the background. Moving a folder containing many descendents is consequently a costly operation. We argue that this trade-off is justified by the scarcity of such operations compared to the shorter critical path and thus higher performance of file reads and writes.

B. Popularity-based metadata service

The metadata service is implemented as a lightweight plugin running inside each Týr server process. Its responsibilities are threefold. First, it serves as a persistence layer for the files whose first chunk of data is stored on this process. Second, it integrates a dynamic replication agent, which analyzes the received requests to determine for which folders to replicate and store the metadata information locally. Finally, it serves as a resolver agent for the metadata requests, dispatching the requested information directly if available locally, or forwarding the request to the appropriate node determined using consistent hashing if not. The most interesting part of this metadata service is arguably the dynamic replication agent, which is the focus of this section.

We use *popularity* as the main metric for dynamic replication to determine which folders to locally replicate metadata for. We define popularity as the number of metadata requests for any given folder in a defined time period. These popularity measurements are performed independently for each node. Consequently, on any given node, the more frequently metadata for a specific folder is requested the more likely it is to be considered for dynamic replication to that node. This metric is particularly suited for workloads accessing vast numbers of small files. Considering that at least a metadata request is needed for reading each file, a data set composed of many files will be significantly more likely to have the metadata for its ancestor folders replicated locally than a data set composed of a single large file.

To measure the popularity for each folder, we record the number of times its metadata is requested to the local resolver agent. Keeping exact counters for each possible folder is memory-intensive, so we only focus on the k most frequently requested folders, k being user-configurable. This problem is known as top-k counting. A variety of memory-efficient, approximate solutions to this problem exist in the literature. For this use-case, such an approximate algorithm is tolerable as popularity estimation errors cannot jeopardize the correctness of the system. Our implementation uses Space-Saving [41] as top-k estimator. It guarantees strict error bounds for approximate request counts, while exhibiting a small, predictable memory complexity of $O(k)$. The output of the algorithm is an approximate list of the k folders for which the metadata is the most frequently requested since the data structure has first been initialized. We perform these measurements over a sliding time window in order to favor immediate popularity over historical measurements.

Whenever a metadata request arrives at the metadata resolver for a folder whose metadata is not locally replicated, the resolver agent determines whether or not it should be replicated. It does so by adding the requested folder path in the current time window for the top-k estimator. If the folder path is present in the output of the estimator, this path should be considered as popular and the associated metadata should be replicated locally. The agent determines which node holds the metadata for this folder, and remotely fetches the metadata from this node. Should the metadata be kept locally, it is stored in a temporary local store. In any case, the fetched metadata is returned to the client. This process is highlighted by the pseudo-code of Algorithm 1.

Algorithm 1 High-level algorithm of the metadata resolver.

```
function RESOLVEFOLDERMETADATA(path)
    let current_window be the current top-k window
    INSERT(path, current_window)
    if path is available locally then
        return the locally-available metadata for path
    end if
    md ← REMOTEFETCHMETADATA(path)
    if path in output of current_window then
        SUBSCRIBEMETADATAUPDATES(path)
        keep md in the local metadata store
    end if
    return md
end function
```

Although folder permission modifications are a very rare occurrence in real-world applications, such metadata updates should nonetheless be consistently forwarded to metadata replicas. We integrate inside the TýrFS metadata agent a synchronous metadata notification service. When the permissions for any given folder are modified, this service synchronously notifies all nodes locally storing a dynamic replica of this metadata. Subscription to updates is performed atomically by a remote metadata agent during the metadata fetch if the remote agent has determined that this metadata should be dynamically replicated to it. Acknowledgement is sent to the client requesting the metadata change only after the update has been applied to all dynamic replicas.

A background bookkeeping service monitors changes in the current time window for the top-k estimator in order to identify the metadata for previously-popular objects that are not popular anymore. The locally available metadata is marked for deletion. It will be kept locally as long as the locally available storage permits. When space needs to be reclaimed, the node deletes the locally-replicated metadata marked for deletion. It also synchronously cancels its subscription for metadata update notifications.

C. TýrFS client

The TýrFS client provides two different interfaces. A native interface exposes all POSIX I/O functions. In addition, a mountable interface, itself based on the native library, allows TýrFS to be mounted as any other file system. The mountable interface leverages the FUSE library. The native interface is implemented as a thin wrapper around the Týr native client library, which handles all the message serialization and communication with the storage cluster.

As an optimization for the TýrFS design, the native interface adds a function to those of the POSIX I/O standard. In common file systems, reading a file is split between the metadata fetching (`open`) and the data read operation (`read`). Withi the design of TýrFS embracing collocation of data and metadata, this can be further optimized by combining the two operations into a single one. Doing so saves the cost of an additional client request to the file system cluster, therefore reducing read latency and increasing throughput. The TýrFS client proposes a native method exposing such behavior, named `fetch`, which functionally opens and reads a file given by its absolute path.

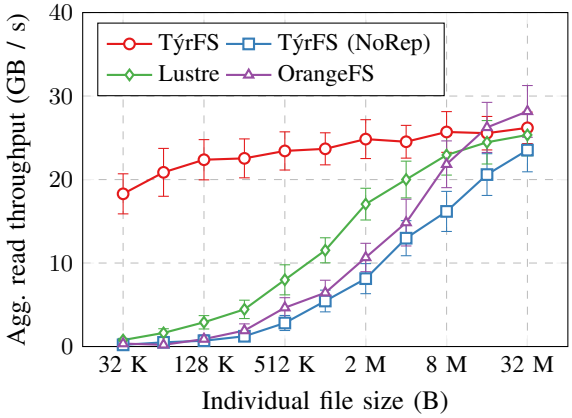


Fig. 6. TýrFS, Lustre and OrangeFS read throughput for a fixed-size data set, varying the size of the individual files it is composed of.

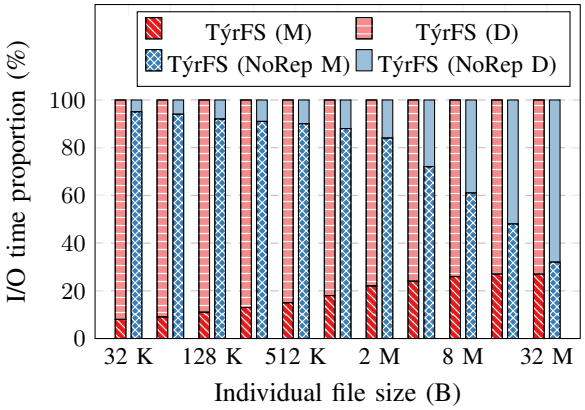


Fig. 7. Breakdown of reads between metadata (M) and data (D) access for a fixed-size data set varying the size of the individual files it is composed of.

VI. Experimental configuration

We execute all following experiments on the Grid'5000 testbed we introduced in Section II. The machines we use are part of the Paravance cluster [30], located in Rennes. Each node embeds 16 cores (2 x 8-core Intel Xeon E5-2630v3 processor) as well as 128 GB RAM. Connectivity between nodes is done using low-latency 10 Gigabit ethernet. All nodes are connected to the same switch. Node-local storage is provided by two 600 GB spinning drives.

We compare the performance of TýrFS with that of multiple file systems, from both the HPC and the Big Data communities. Lustre 2.10.0 [28] and OrangeFS 2.9.6 [28] are deployed according to best practices. With Lustre we dedicate one node for metadata. For Big Data applications we compare TýrFS with Ceph 12.2.0 [39] and HDFS 2.8.0 [32]. All systems are configured not to replicate data, consequently not providing fault-tolerance which is beyond the scope of this paper. We discuss fault tolerance in Section VIII-C.

We benchmark the performance of the file system using two different application profiles. For HPC file systems we leverage the same benchmark as in Section II, which accesses the file system through the TýrFS native client and leverages the `fetch` operation it provides. The benchmark reads through the whole pre-loaded random dataset from all ranks, evenly distributing the last across all ranks. No actual computation is performed on the data. Big Data Analytics applications are running atop Spark 2.2.0 [42] and are extracted from SparkBench [43]. All these applications have the common characteristic to be mostly read-intensive. We connect Spark applications to Ceph by modifying Hadoop to use the POSIX I/O interface and to TýrFS using the native interface. We expose to Spark the data layout of both systems to ensure data locality. TýrFS is configured with a measurement time window of 1 minute and a k of 20 that we determined to be optimal for our configuration. We discuss this setup in Section VIII-D.

Unless specified otherwise, for Lustre and OrangeFS we use 24 machines to deploy the storage systems, plus 8 machines to deploy the client applications. With Lustre we dedicate one node for metadata management. Big Data applications are deployed alongside storage on the whole 48 nodes of the cluster. All results are the average of 50 experimental runs.

VII. Experimental validation

In this section we prove that TýrFS delivers its promise of drastically improving read performance for small files. We also demonstrate that this benefit applies to applicative contexts as diverse as HPC frameworks and Big Data applications compared to state-of-the-art, industry standard file systems.

A. Drawing the baseline performance in HPC contexts

We first compare the performance results obtained with TýrFS to those of Lustre and OrangeFS in the exact same conditions as in Section II. The curves for Lustre and GPFS are reproduced for reference. The size of the input data set is constant. We vary the number of input files between 3,840 $\times$ 32 MB files (1 file / rank) to 3.84×10^6 $\times$ 32 KB files (10.000 files / rank). The data is generated randomly.

In Figure 6 we plot the total execution time of the benchmark for TýrFS, Lustre and OrangeFS. To evaluate the baseline performance of the metadata replication system, we also plot the performance achieved by TýrFS with dynamic replication disabled. The performance for large files is on par with that of Lustre or OrangeFS due to the possible amortization of the metadata requests. However, TýrFS shows a significant performance improvement over those two systems for small files. Disabling metadata replication causes TýrFS performance to drop at the level of other distributed file systems. The similarity of the performance of TýrFS without metadata replication with that of OrangeFS is due to the fact that both embrace akin distributed metadata management principles. This unequivocally demonstrates the effectiveness of dynamic metadata replication with small files.

Figure 7 shows the relative breakdown of the I/O time between the metadata (M) and data (D) access inside the `fetch` operation. We note that the relative time spent on metadata access as the data file increases tends to decrease with TýrFS. This clearly contrasts with other file systems, in which the throughput drop is mainly due to the high cost of metadata management. In TýrFS, the main factor impacting read time is the added cost of issuing additional read requests. The efficiency of the dynamic metadata replication is highlighted by the performance of TýrFS with this service disabled, which here also is smilar to that of other file systems.

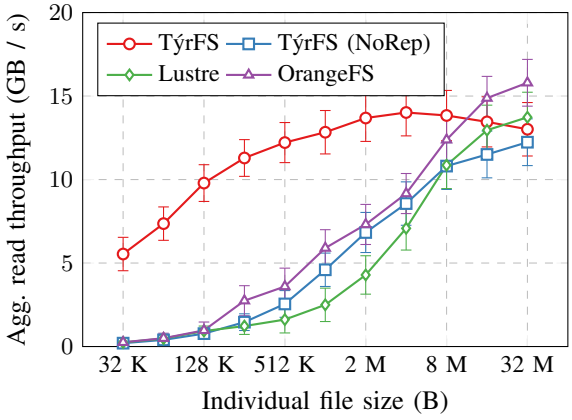


Fig. 8. Shared TýrFS, Lustre and OrangeFS read throughput for a fixed-size data set, varying the size of the individual files is is composed of.

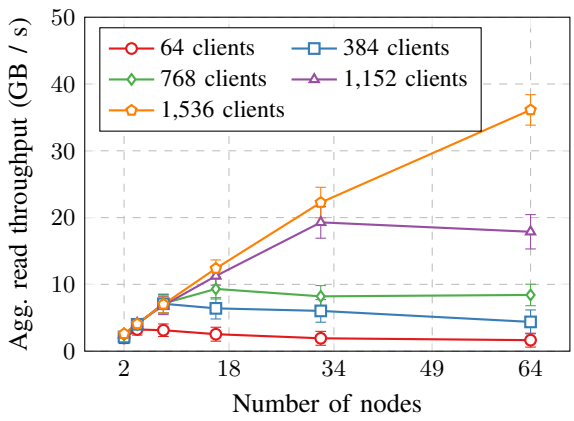


Fig. 9. TýrFS horizontal scalability, varying the number of clients and the number of cluster nodes with 32 KB input files.

B. Evaluating performance in a multi-user, shared context

In the previous experiment we did simulate a single-user use-case, where a single application is free to use all system resources. Accessing the file system in these conditions is especially favorable to TýrFS, as the metadata replication is dedicated to a single user and is able to quickly adapt to this application. Yet, it is typical for HPC platforms to offer a single, centralized file system in which all users share a common pool of resources. This is a significantly more complex problem.

We simulate a multi-user setup by replaying on the file system real-world I/O traces from the Second Animation Dataset [44], representative of a data-intensive I/O workload from multiple clients. We dedicate 24 more clients to this task, each replaying a share of the 3.2 TB trace file at maximum rate. We concurrently execute on 24 other nodes the same benchmark as in the previous experiment, and measure the achieved throughput across all systems varying the size of the input files between 32 KB and 32 MB each. We limit the dynamic replication storage on each node to a low number: 20 replication slots shared across all concurrent users.

In Figure 8 we plot the achieved throughput of our benchmark in these conditions. Similarly to what we observe with single-user experiments, TýrFS significantly outperforms Lustre and OrangeFS as the size of the input files is reduced. Compared to the previous experiment, the shared nature of the file system results in a lower achieved throughput across all systems. TýrFS in particular shows a higher performance reduction at smallest file sizes due to the saturation of the file system nodes. Interestingly, we see that reducing the file size of the input data set from 32 MB to 8 MB in Týr causes a noticeable 8% throughput *increase*. This is because smaller file sizes involve a higher number of metadata requests for the input file set, which in turn increase the likelihood of the metadata for the data set access path to be dynamically replicated across the whole cluster. This effect is amplified by the low limit we put on the replication store size for each node. These results demonstrate the applicability of TýrFS for shared, highly-concurrent file systems, while additionally confirming that the dynamic metadata replication we propose especially benefits small files use-cases.

C. Ensuring horizontal scalability

Horizontal scalability is a critical feature of large-scale systems targeted at data-intensive applications. It allows the system to scale out, increasing its total performance by adding commodity machines to the cluster. Týr, the base building block of TýrFS, has a demonstrated near-linear horizontal scalability that makes it particularly suitable for large-scale use-cases. TýrFS does not introduce any centralization that could jeopardize these features. In particular, it inherits from OrangeFS the distributed metadata management principles, which support the horizontal scalability of this system. As such, we argue that TýrFS should be able to inherit this desirable feature.

In this section we formally demonstrate the horizontal scalability of TýrFS. We adopt the same setup as in Section VII-A. We vary the number of storage nodes from 2 to 64 on the Paravance cluster. We setup clients on up to 24 nodes of a second Grid'5000 cluster: Parasilo. Both clusters are co-located in the same facility, and are interconnected with low-latency 2 x 40G Ethernet links aggregated using the Link Aggregaton Control Protocol (LACP). Each client node runs 64 ranks of the MPI read benchmark that concurrently reads files from our random, pre-loaded 1.22 TB data set. We run the benchmark for 5 minutes, reading the dataset from within an endless loop. The dataset we use in this experiment is composed of $3.84 \times 10^7 \times$ 32 KB files.

In Figure 9 we plot the measured aggregated throughput of TýrFS across all clients, varying the number of nodes from 2 to 64 and the number of concurrent clients from 64 to 1,536. The results demonstrate that TýrFS shows a near-linear horizontal scalability when adding nodes to the storage cluster, being able to serve more clients with higher aggregate throughput. The decrease in throughput observed with the smallest cluster sizes when increasing the number of clients is due to the saturation of the storage servers. In this configuration, and despite working on a data set composed of very small files, we measured the throughput of TýrFS as high as 36 GB / s across all clients with 64 storage servers and 1,536 concurrent client ranks.

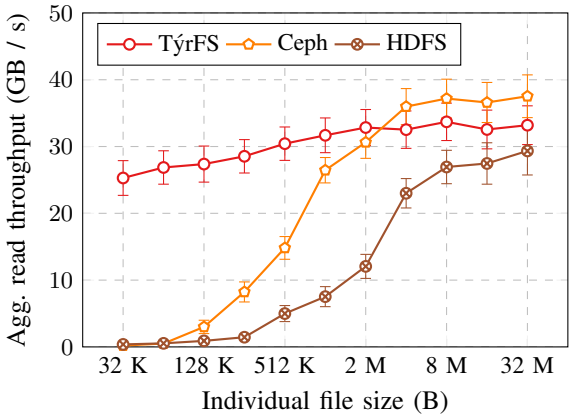


Fig. 10. TýrFS, Ceph and HDFS read throughput for a fixed-size data set, varying the size of the individual files it is composed of.

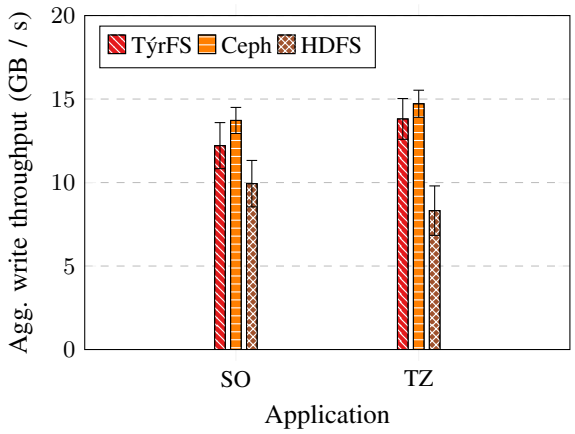


Fig. 11. TýrFS, Ceph and HDFS write throughput with Sort (balanced I/O) and Tokenizer (write-intensive) applications.

D. Experimenting with Big Data applications

For these experiments we setup a configuration according to the best practices for Big Data Analytics. We deploy both the storage system and the computation on all 48 nodes of the cluster. We compare the performance of Týr with that of Ceph and HDFS. The experiments are based on a series of different Spark applications.

We leverage SparkBench [43] to confirm that the design of TýrFS yields significant performance improvements with small files for applications showing a wide range of I/O profiles. We use exactly the same setup as in the previous experiment, and execute three read-intensive benchmarks extracted from the benchmark suite: Connected Component (CC), Grep (GR) and Decision Tree (DT). These benchmarks are intended to evaluate the I/O performance of TýrFS when faced with different workloads types, respectively graph processing, text processing and machine learning. In Figure 10 we plot the aggregate throughput obtained across all nodes when varying the size of the input data set files. Overall, the results are very similar to those obtained with Lustre and OrangeFS. This is not surprising, and is mostly due to the very similar architecture of all these file systems, in which the relative cost of metadata management increases as the file size decreases. For these experiments as well, the throughput drop we observe with TýrFS when using 32 KB files instead of 32 MB files is only of about 25%. In similar conditions, other systems show a much more significant throughput drop of more than 90%. This results in a 3.8× increase of total computation time with OrangeFS, wasting computing resources.

Not all applications are read-intensive. Although write performance is not a main target of this paper, it is critical to understand precisely the impact of dynamic metadata replication for balanced and write-intensive workloads as well. To do so, we add two more applications to the experiment: Sort (SO) from SparkBench, which illustrates a balanced I/O workload, and Tokenizer (TZ), which provides a write-intensive one. Tokenizer reads a text file, tokenizes each line into words, and calculates the `NGrams=2` for each line. Ngrams are saved as text. This is a common preprocessing step in topic modeling for documents where word patterns are extracted for feeding a machine learning model.

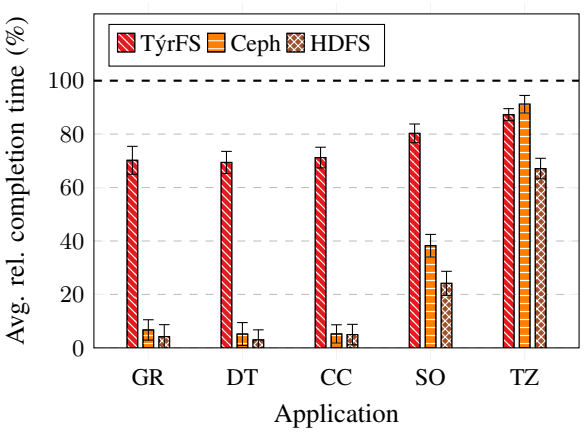


Fig. 12. Application completion time difference with 32 MB input files relative to that obtained with 32 KB input files, with TýrFS, Ceph and HDFS. 100% represents the baseline performance for 32 MB files.

In Figure 11 we plot the measured write throughput for Sort and Tokenizer with TýrFS, Ceph and HDFS. The performance of the write operations are similar across the two experiments. In terms of write performance, TýrFS exhibits a write performance consistent with that of Ceph. Ceph outperforms both TýrFS and HDFS for write throughput by a margin or respectively 8% and 35%.

Considering the application as a whole, Figure 12 summarizes our results for Big Data applications. We measure the application completion time difference for 32 MB files relative to that obtained with 32 KB files. The throughput advantage of TýrFS over Ceph and HDFS with small files is clearly visible on the application completion time, allowing it to retain at least 73% of its performance when using 32KB instead of 32MB files. Ceph and HDFS in similar conditions retain less than 5% of their performance, hence wasting costly resources. Overall, the impact of small input files is significantly lower for all three systems with Sort and Tokenizer applications because of the higer influence on writes, for which all three systems show comparable performance. The write-intensive Tokenizer application gives a 4% advantage to Ceph because of its slightly higher write throughput compared to TýrFS.

VIII. Discussion

In this section we discuss the main features and design choices of TýrFS. We detail which parts of its design impose limitations that makes it unfit for several use cases.

A. Is TýrFS also suitable for large files?

TýrFS design is specifically designed to benefit use cases involving large numbers of small files. Yet, while not the scope of this paper, no fundamental part of its design make TýrFS unfit for large files. It inherits from Týr the ability to handle very large files efficiently, by leveraging chunking. The metadata distribution in TýrFS will however be less effective with large files. The main reason is that the metadata for any given file is co-located only with the *first* chunk of the file. Reading a different chunk will trigger a remote metadata fetch for the file, which is unnecessary for small files.

B. Which use-cases is TýrFS unfit for?

Centralized metadata management offers many possibilities for managing the data saved in the system. For example, data reduction using deduplication is a well known technique to reduce storage resources. Efficient file copy at the metadata level is another technique often used to substantially increase performance of copy-on-write data sets compared to traditional file systems. Automatic data tiering could allow to balance data over multiple storage mediums depending on a given set of rules. The predictable data distribution based on consistent hashing makes all these techniques significantly more challenging to implement in TýrFS compared to other file systems, in which the metadata fetch happens prior to the data fetch.

C. What about fault tolerance?

None of the experiments presented in this paper did leverage fault-tolerance, which is outside the scope of this paper. TýrFS has been designed with this requirement in mind. In particular, data is fault-tolerant natively thanks to the data replication built inside Týr. Although not implemented in our prototype, the replication of the file metadata could be performed easily. In that context, we would replicate the metadata for any given file on all nodes storing a replica of the first chunk for that file. The replicated metadata do not need to be fault-tolerant.

D. How does one configure TýrFS?

The two configuration parameters specific to TýrFS are relative to the metadata replication module.

The size of the measurement sliding window needs to be carefully chosen. Indeed, it poses a tradeoff between responsiveness and accuracy of the measurement system. Setting this value too high would cause a relatively high replication latency because of the higher number of requests needed for a value to be part of the top-k values on each node. Setting it too low causes the popularity estimation to be performed on little data, hence reducing accuracy of the measurement. In practice, this value should be low compared to the average lifetime of applications, but high enough to capture a sufficient amount of requests. In our experiments, we measured 1 minute to provide a correct balance.

The second important parameter is the size of the output top-k estimator. Specifically, a larger k will enable more values to be monitored at the cost of memory. A smaller k would reduce the amount of monitored values at the expense of accuracy. In our experiments and in all traces we could study, most of the I/O is concentrated on a very small portion of a file system at a time. This benefits the metadata replication system, which is able to provide high metadata replication efficiency with relatively low values of k. In all our experiments, a value superior to 20 did never yield significant performance improvements. For larger deployments shared between many users, setting a higher k value could lead to increased efficiency of the metadata replication. This should be determined empirically based on the specifics and estimated load of each deployment.

IX. Conclusion

Small files are known to be a significant challenge for most distributed file systems. Indeed, the hierarchical nature of this storage model often commands separating the metadata from data in the cluster. With small files, the cost of fetching metadata easily dominates the cost of actually fetching the data. This is especially problematic when dealing with very large amounts of small files, in which over 90% of the I/O time can be dedicated to fetching metadata. For use-cases reading many small files at large scale, it is critical to ensure efficient metadata management. In this paper we argue that this cannot be done without rethinking data and metadata management in a distributed file system.

We propose TýrFS as a solution to the problem. TýrFS enables clients to locate any piece of data in the cluster independently of any metadata server. A dynamic metadata replication module adapts to the workload to efficiently replicate the necessary metadata on nodes from which the associated data is frequently accessed. Clients can then read frequently accessed data directly, without involving additional servers. We did implement a prototype of TýrFS over the Týr storage system. Experiments on the Grid'5000 testbed showed a performance improvement of more than an order of magnitude with smaller file sizes compared to state-of-the-art storage systems. These improvements are applicable both to HPC and Big Data Analytics use cases.

Future work includes further improving the design of TýrFS for enabling writes to small files at high speed to provide a complete solution for managing small files at scale. We plan to evaluate TýrFS on large-scale platforms, facing real-world users and applications.

Acknowledgements

This work is part of the "BigStorage: Storage-based Convergence between HPC and Cloud to handle Big Data" project (H2020-MSCA-ITN-2014-642963), funded by the European Commission within the Marie Skłodowska-Curie Actions framework. It is also supported by the ANR OverFlow project (ANR-15-CE25-0003). The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, several universities as well as other organizations. The authors would like to thank Álvaro Brandon (Universidad Politécnica de Madrid) for his precious expertise with Spark experiments.

REFERENCES

[1] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 1, pp. 97–107, 2014.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15.

[3] A. Moniruzzaman and S. A. Hossain, "NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[4] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath, "Locality and availability in distributed storage," *IEEE Transactions on Information Theory*, vol. 62, no. 8, pp. 4481–4493, 2016.

[5] D. R. Cutting, D. R. Karger *et al.*, "Scatter/Gather: A cluster-based approach to browsing large document collections," in *ACM SIGIR Forum*, vol. 51, no. 2. ACM, 2017, pp. 148–159.

[6] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[7] "NERSC storage trends and summaries," *Accessed on*, 2017.

[8] G. H. Bryan and J. M. Fritsch, "A benchmark simulation for moist nonhydrostatic numerical models," *Monthly Weather Review*, vol. 130, no. 12, pp. 2917–2928, 2002.

[9] S. Habib, V. Morozov, H. Finkel *et al.*, "The universe at extreme scale: multi-petaflop sky simulation on the BG/Q," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 4.

[10] G. Berriman, J. Good, D. Curkendall *et al.*, "Montage: An on-demand image mosaic service for the nvo," in *Astronomical Data Analysis Software and Systems XII*, vol. 295, 2003, p. 343.

[11] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman *et al.*, "CyberShake: A physics-based seismic hazard model for southern california," *Pure and Applied Geophysics*, vol. 168, no. 3-4, pp. 367–381, 2011.

[12] A. Abramovici, W. E. Althouse *et al.*, "LIGO: The laser interferometer gravitational-wave observatory," *science*, pp. 325–333, 1992.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 155–163.

[14] L. Pineda-Morales, A. Costan, and G. Antoniu, "Towards multi-site metadata management for geographically distributed cloud workflows," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 294–303.

[15] B. Dong, Q. Zheng *et al.*, "An optimized approach for storing and accessing small files on cloud storage," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1847–1862, 2012.

[16] P. Carns, S. Lang, R. Ross *et al.*, "Small-file access in parallel file systems," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–11.

[17] T. White, "The small files problem," *Cloudera Blog, http://blog.cloudera.com/blog/2009/02/the-small-filesproblem*, 2009.

[18] Y. Zhang and D. Liu, "Improving the efficiency of storing for small files in HDFS," in *Computer Science & Service System (CSSS), 2012 International Conference on*. IEEE, 2012, pp. 2239–2242.

[19] G. Mackey *et al.*, "Improving metadata management for small files in HDFS," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–4.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[21] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman, "Simulation of dynamic data replication strategies in data grids," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 10–pp.

[22] I. Stoica, R. Morris *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[24] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. S. Pérez, "Towards efficient location and placement of dynamic replicas for geo-distributed data stores," in *Proceedings of the ACM 7th Workshop on Scientific Cloud Computing*. ACM, 2016, pp. 3–9.

[25] P. Matri, M. S. Pérez, A. Costan, L. Bougé, and G. Antoniu, "Keeping up with storage: decentralized, write-enabled dynamic geo-replication," *Future Generation Computer Systems*, 2017.

[26] D. Jayalakshmi, T. R. Ranjana, and S. Ramaswamy, "Dynamic data replication across geo-distributed cloud data centres," in *International Conference on Distributed Computing and Internet Technology*. Springer, 2016, pp. 182–187.

[27] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. S. Pérez, "Týr: blob storage meets built-in transactions," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 573–584.

[28] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[29] M. Moore, D. Bonnie *et al.*, "OrangeFS: Advancing PVFS," *FAST poster session*, 2011.

[30] "Grid'5000 – Rennes Hardware (Paravance)," *Accessed on*, 2017.

[31] R. B. Ross, R. Thakur *et al.*, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[33] "Hadoop Documentation – Archives," *Accessed on*, 2017.

[34] V. G. Korat and K. S. Pamu, "Reduction of data at namenode in HDFS using harballing technique," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, vol. 1, no. 4, pp. pp–635, 2012.

[35] C. Vorapongkitipun and N. Nupairoj, "Improving performance of small-file accessing in Hadoop," in *Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on*. IEEE, 2014, pp. 200–205.

[36] 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.

[37] J. Laitala, "Metadata management in distributed file systems," 2017.

[38] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[40] P. H. Lensing, T. Cortes, and A. Brinkmann, "Direct lookup and hash-based metadata placement for local file systems," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 5.

[41] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.

[42] M. Zaharia, M. Chowdhury *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[43] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 53.

[44] E. Anderson, "Capture, conversion, and analysis of an intense NFS workload." in *FAST*, vol. 9, 2009, pp. 139–152.