Enosis: Bridging the Semantic Gap between File-based and Object-based Data Models

Anthony Kougkas, Hariharan Devarajan, Xian-He Sun
Illinois Institute of Technology, Department of Computer Science, Chicago, IL
{akougkas, hdevarajan}@hawk.iit.edu, sun@iit.edu

Abstract—**File and block storage are well-defined concepts in computing and have been used as common components of computer systems for decades. Big data has led to new types of storage. The predominant data model in cloud storage is the object-based storage and it is highly successful. Object stores follow a simpler API with *get()* and *put()* operations to interact with the data. A wide variety of data analysis software have been developed around objects using their APIs. However, object storage and traditional file storage are designed for different purpose and for different applications. Organizations maintain file-based storage clusters and a high volume of existing data are stored in files. Moreover, many new applications need to access data from both types of storage. In this paper, we first explore the key differences between object-based and the more traditional file-based storage systems. We have designed and implemented several file-to-object mapping algorithms to bridge the semantic gap between these data models. Our evaluation shows that by achieving an efficient such mapping, our library can grant 2x-27x higher performance against a naive one-to-one mapping and with minimal overheads. Our study exposes various strengths and weaknesses of each mapping strategy and frames the extended potential of a unified data access system.**

Keywords—Data models, Parallel File Systems, Object Storage, Integrated access, Unified storage, Mapping files to objects

I. Introduction

Historically, data are stored and accessed as files, blocks or objects in equivalent storage systems. File and block storage have been around for considerably longer than object storage, and are something most people are familiar with. These systems have been developed and highly optimized through the years. Popular interfaces and standards such as POSIX I/O [1], MPI-IO [2], and HDF5 [3] expose data to the applications and allow users to interact with the underlying file system through extensive APIs. In a large scale environment the underlying file system is usually a parallel file system (PFS) with Lustre [4], GPFS [5], PVFS2 [6] being some popular examples or a distributed file system such as GoogleFS or HDFS [7]. However, applications are increasingly dealing with high volume, velocity, and variety of data which leads to an explosion of storage requirements and increased data management complexity [8]. Most of these file systems face significant challenges in performance, scalability, complexity, and limited metadata services [9], [10].

On the other hand, object storage was born from the need to increase the scalability and programmatic accessibility of storing data. It is widely popular in the cloud community and there are a lot of different implementations freely available. It offers simplistic APIs with basic *get()*, *put()*, and *delete()* operations. Most notable examples include the Amazon S3 [11] and the OpenStack Swift [12] API. So far, object storage has been used widely for stale, archival data, which fits nicely with the fact that changes are accommodated by creation of new versions of data, rather than modifying existing data. However, this seems to be changing and we see more high-performance and low latency solutions. Few examples include Cassandra [13], MongoDB [14], and HyperDex [15]. The flat name space organization of the data in object storage, in combination with its expandable metadata functionality, facilitate its ease of use, its scalability, and its resiliency via replicated objects. Lastly, object stores are the best option to store, access, and process unstructured and semi-structured data making them a widely used storage solution for Big Data problems.

D. Reed and J. Dongarra in [16] point out that the tools and cultures of HPC and BigData analytics have diverged, to the detriment of both; unification is essential to address a spectrum of major research domains. There is an increasingly important need of a unified storage access system which will support complex applications in a cost-effective way leading to the convergence of HPC and BigData analytics. However, such unification is extremely challenging with a wide range of issues such as: a) gap between traditional storage solutions with semantics-rich data formats and high-level specifications, and modern scalable data frameworks with simple abstractions such as key-value stores and MapReduce, b) difference in architecture of programming models and tools, c) management of heterogeneous resources and, d) management of diverse global namespaces stemming from different data pools.

In this paper, we explore several ways to map files to objects. Specifically, we designed and implemented three new mappings of a typical POSIX file to one or more objects. These mappings can cover MPI-IO as well since it is using POSIX files at its core. We also implemented a novel mapping of an HDF5 file to one or more objects. Our mappings pave the way towards a unified storage access system. Using our mappings one can efficiently utilize a file interface to access and process data that reside to a totally different storage subsystem such as an object store. With this extra functionality, we can leverage strengths of each storage solution and complement each other for known limitations. This is a powerful abstraction for the user who, under our solution, still uses the familiar file interface while a scalable and efficient object store supports all I/O operations.

The contributions of this paper are:

- We present key characteristics of file-based, block-based, and object-based data models.
- We design and implement a unified storage access system that bridges the semantic gap between file-based and object-based storage systems.
- We evaluated our solution and the results show that, in addition to providing programming convenience and efficiency, our library can grant higher performance avoiding costly data movements between file-based and object-based storage systems.

The rest of this paper is organized as follows. Section II provides the motivation of this work. In Section III we describe the background and the related work. In Section IV we present the design and implementation details of our mapping strategies. Results of our library's evaluation are presented and analyzed in Section V. Finally, conclusions and future work are laid out in Section VI.

II. Motivation

The increasing ability of powerful HPC systems to run data-intensive problems at larger scale, at higher resolution and with more elements gave birth to a new category of computing, namely High-performance Data Analytics (HPDA) [17]. In addition, the proliferation of larger, more complex scientific instruments and sensor networks to collect extreme amounts of data pushes for more capable analysis platforms. Performing data analysis using HPC resources can lead to performance and energy inefficiencies [18]. In [19] the authors point out that traditional offline analysis results in excessive data movement which in turn causes unnecessary energy costs. Alternatively, performing data analysis inside the compute nodes can eliminate the above mentioned redundant I/O, but can lead to wastage of expensive compute resources and will slow down the simulation job due to interference. Therefore, modern scientific workflows require both high-performance computing and high-performance data processing power. However, HPC and HPDA systems are different in design philosophies and target different applications.

The divergence of tools and cultures between HPC and HPDA has led HPC sites to employ separate computing and data analysis clusters. For example, NASA's Goddard Space Flight Center uses one cluster to conduct climate simulation, and another one for the data analysis of the observation data [20]. Periodically, simulation data are compared with observation data, and are used in data analysis. Similarly, observation data and analysis results are used in simulation to increase accuracy and efficiency. Due to the data copying between the two clusters, the data analysis is currently conducted off-line, not at runtime. However, runtime simulation/analysis will lead to more accurate and faster solutions. The data transfer between storage systems along with any necessary data transformations are a serious performance bottleneck and cripples the productivity of those systems. Additionally, it increases the wastage of energy and the complexity of the workflow. Another example is the JASMIN platform [21] run by the Center of Environmental Data Analysis (CEDA) in the UK. It is designed as a ”super-data-cluster”, which supports the data analysis requirements of the UK and European climate and earth system modeling community. It consists of multi-Petabyte fast storage co-located with data analysis computing facilities, with satellite installations at several locations in the UK. A major challenge they face is the variety of different storage subsystems and the plethora of different interfaces that their teams are using to access and process data. They claim that PFSs alone cannot support their mission as JASMIN needs to support a wide range of deployment environments.

A radical departure from the existing software stack for both communities is not realistic. Instead, future software design and architectures will have to raise the abstraction level, and therefore, bridge the semantic and architectural gaps. We envision a data path agnostic to the underlying data model and we aim to leverage each storage solutions strengths while complementing each other for known limitations. With our mapping strategies we strive for maximizing productivity and minimizing data movement which leads to higher performance and resource utilization.

III. Background and Related Work

A. What is a file-based storage

A file storage stores data in a hierarchical structure. Data are saved in files and directories and presented in the same format. Data can be accessed via the Network File System (NFS) or the Server Message Block (SMB) protocols. Files are basically a stream of bytes and they are part of a namespace that describes the entire collection in a certain file system. The file system maintains certain attributes for each file such as its owner, who can access the file, its size, the last time it was modified and others. All this information follows the file and it is called metadata.

B. What is a block-based storage

A block storage stores data in fixed-length volumes, also referred to as blocks. It is typically used in storage-area networks (SAN). Each block acts as an individual hard drive and is configured as such. These blocks are controlled by a server-based operating system. Data are accessed via a Fibre Channel over Ethernet or a SCSI controller using the appropriate protocol respectively. A collection of blocks (i.e., chunk of data) can form a file. Each block has an address and data are accessed by a SCSI call to that address. There are no metadata associated with the blocks except the address. Applications control how blocks are combined or accessed. That makes block storage a good candidate for storing file systems or database.

C. File vs block storage

While block storage writes and retrieves data to and from certain blocks, file storage requests data through user-level

TABLE I: Object vs File Storage

Category	**Object Storage**	**File Storage**
Data unit	Objects	Files
Update	Create new object	In-place updates
Protocols	REST and SOAP	NSF with POSIX
Metadata	Custom	Fixed attributes
Strengths	Scalability	Simplified access
Limitations	Frequent updates	Heavy metadata
Performance	High throughput	Streaming of data

data representation interfaces such as POSIX and HDF5. This client-server method of communication occurs when the client uses the data's file name, directory location, URL and other information. With block storage, the server receives the request, looks up the data storage locations where the data is stored and retrieves it using storage-level functions. The server does not send the file to the client as blocks, but as bytes of the file. File-level protocols cannot understand block commands, and block protocols cannot convey file access requests and responses. Because the main usage of block storage is implementing file systems or databases, we will focus for the rest of the paper on files and objects.

D. What is an object-based storage

An object storage manipulates data as discrete units called objects. These objects are kept inside a single expandable pool of data (e.g. repository) and are not nested as files inside folders and directories. Object storage keeps the blocks of data that make up a file together and adds all of its associated metadata to that file. It also adds extended metadata to the file which eliminates the hierarchical structure used in a file storage, placing everything into a flat address space, called a storage pool, or key space, or object space. A unique identifier is assigned to each object and is used by the application to retrieve this object.

E. Object vs file storage

Object storage overcomes many of the limitations that file systems face (especially in scale) sometimes with a hit in performance. As more and more data is generated, storage systems have to grow at the same pace. As file systems grow we may run into durability issues, hardware limitations and management overheads. The flat name space organization of the data, in combination with its expandable metadata functionality, make object store a better choice for the Big Data era. However, object storage is not the answer to all storage-related problems. File systems provide guarantees for strong consistency which object stores cannot (most of the implementations offer eventual consistency). Strong consistency is needed for real-time systems where data are frequently being mutated. Table I summarizes some of the key differences.

All these data models are supported by the respective storage systems. Each system exposes data to the application via a different API. Traditional applications written based on files (i.e., POSIX interface) cannot access data residing in an object store unless they either change their code to use the get/put calls or by first moving the data in a file system and then use the typical fread/fwrite calls. In this paper, we aim to solve this limitation without affecting the application code. Our solution is transparent to the user which simply connects the application to our library, which then, is responsible to intercept the I/O calls and map them to an object store.

F. Related work

Few distributed file systems replaced the way they store data internally. Conventional PFSs split a file into smaller pieces or stripes and store them separately on local file systems of different storage nodes. This new category of distributed file systems replaces the local file system with Object Storage Devices (OSD) to distribute the smaller pieces of data. CephFS [22] is a new type of distributed file system that promotes the separation of data and metadata management. CephFS is a POSIX-compliant filesystem that uses a Ceph Storage Cluster to store its data. The Ceph filesystem uses the same Ceph Storage Cluster system as Ceph Block Devices. With this design, Ceph offers APIs that can support both file operations and object operations. PanFS [23], [24] also utilizes OSDs internally. Unlike other storage systems that loosely couple parallel file system software with legacy block storage arrays, PanFS combines the functions of a parallel file system, volume manager, and scalable RAID engine into a single holistic platform. Similarly, OBFS [25] utilizes OSDs internally to create a distributed file system. Its design offloads some administrative tasks on the disk itself making it run faster for specific workloads.

For all the above solutions, however, one needs to switch the entire storage installation to their proposed system, and applications need to be rewritten to use their APIs. Existing installations of file systems and object stores, that one might have, need to migrate all data to the above platforms. Thus, the above systems do not aim to integrate file-based and object-based data models but rather provide an enhancement of supported features. Finally, the authors in [26], exposed the limitations of a POSIX file interface and they proposed an alternative interface to directly access the underlying storage objects. Basically, they presented an extension to the POSIX API that exposes a different abstraction for storing data. However, it cannot be used on top of object stores whereas our solution can bridge those two storage systems.

IV. Design and Implementation

In this section we first provide some implementation details and then present our new mapping strategies. For POSIX files we propose three new mapping strategies: a) balanced, b) write-optimized, and c) read-optimized. Finally, we present our novel mapping strategy of HDF5 files to objects (i.e., key-value pairs). HDF5 is a data model and file format for storing and managing data. It is designed for flexible and efficient I/O and for high volume and complex data. It is also very popular among the scientific community.

A. Library implementation details

Our solution consists of a middle-ware library that applications can link to (i.e., either compile with or preload it). We intercept the file reads and writes and map them to gets and puts transparently to the application. Our library is writen in C++. We carefully optimized the code to run as fast as possible and minimize the overhead of the library. The code is optimized with state-of-the-art helper libraries. A few examples include the following. For memory management we chose Google's *tcmalloc* library [27] that performs 20% faster than the standard malloc and has a smaller memory footprint. For hashing, we selected the *CityHash* functions by Google [28], the fastest collection of hashing algorithms at this moment. We specifically used the 64-bit version of the hashing functions. For containers such as maps and sets, we used Google's BTree library [29] which is faster than the STL equivalent containers and reduces memory usage by 50-80%. A prototype version can be found online.[1]

Upon initialization of the library, a key-space crawler scans the underlying key-value store to create the initial metadata information. Additionally, if the user provides access to a file system, the crawler will scan the file namespace and will create a unified single key space for the user. All these library metadata information are stored in a special object in the key-value store. The library also loads this metadata object in memory on bootstrap for faster queries. Our library does not support directory operations. Deletion operations are handled by invalidation and key-space garbage collection on application exit. Note that our library does not aim to implement a file system on top of a key-value store but rather bridge the two systems under the file-based data model. Our mapping algorithms aim to leverage, whenever possible, the strengths of the underlying storage solution and applications might experience a boost in performance.

B. Mapping Strategies

1) **Naive mapping strategy**

As the baseline mapping we consider the one where each separate file is mapped into one object. This is a 1-to-1 mapping and each filename becomes the unique identifier for the object store. While this naive mapping works, there are many limitations with it. First, the file size is an arbitrary number that could violate the limitations of the object store for maximum object size (e.g. MongoDB has a 32MB limit, Cassandra a 64MB). Second, this mapping exhibits extremely poor performance. Each request returns the whole object (i.e., having size equal to the original file) which means excessive unnecessary I/O. Third, with this mapping we do not take advantage of any of the nice architectural advantages that an object store offers. Therefore, we carefully designed and implemented three new strategies for mapping POSIX files to objects: a) balanced, b) write-optimized, and c) read-optimized. These strategies aim to better serve respective workloads.

[1]Code will become available upon acceptance of this paper.

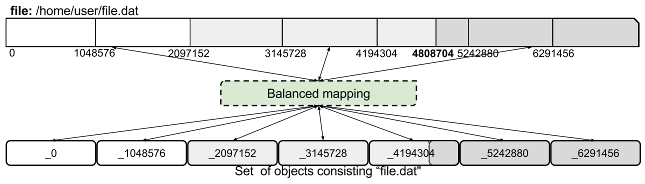
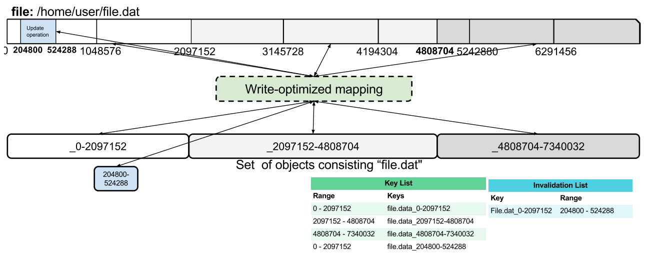


Fig. 1: Balanced mapping strategy.

2) **Balanced mapping strategy**

In this mapping strategy a file is divided into predefined, fixed-size, smaller units of data we call *buckets*. The buckets represent different locations in a file and they are based on the offset. Therefore any *fread()* or *fwrite()* naturally maps into one or more buckets according to a starting offset and a total size of the operation. Each bucket becomes an object and gets persisted in the underlying object store for any subsequent reads or updates. Figure 1 demonstrates an example of this mapping strategy. For this example, we assume that the file was created earlier and therefore the objects too. We also assume that the bucket size is 1MB. Finally, our object creation follows a certain naming schema: *filename_base-offset*. Base offset here is the bucket size (i.e., 0 - 1MB - 2MB etc). In this example, we have three read requests. First, `fread(buf,2097152,1,fh)` gets two objects since it is clearly within the boundaries of the object size. The next `fread(buf,2711552,1,fh)` will get three objects. If we look closer to the third object, our solution will fetch the entire object but only deliver to the application the portion it needs. In this example, it will only deliver from offset 4194304 to offset 4808704. Finally, the last `fread(buf,2531328,1,fh)` will similarly fetch three objects and only deliver part of the first object to the application.

This mapping strategy is the same for both fread() and fwrite() operations leading to a balanced and stable performance for a mixed workload. The abstraction of buckets helps map location in the file onto one or more objects. This could possibly create extra reading of data for requests that are not aligned to the bucket boundaries. However, this happens only for the corner buckets and the maximum extra reading is less than two bucket sizes. Therefore, the pre-configured bucket size is really important and could potentially affect the performance. For instance, if a very small bucket size such as 20 KB is selected then for a simple *fwrite()* of 20 MB, the mapping will create more than 1000 keys which will stress the underlying object store. A more reasonable 2 MB bucket size will only create 10 keys leading to better performance and latency. After extensive testing, we found that *a bucket size equal to the median of all request sizes* is the best and more balanced choice. If the user does not have access to an I/O trace file of his/her application then a default bucket size of 512 KB is suggested. Lastly, in this strategy any update operation (i.e., fwrite() over existing data) might need to first *get()* an existing object and update only portion of its value before it can *put()* the updated object down to the object store. This extra get()



Key List	
Range	**Keys**
0 - 2097152	file.data_0-2097152
2097152 - 4808704	file.data_2097152-4808704
4808704 - 7340032	file.data_4808704-7340032
0 - 2097152	file.data_204800-524288

Invalidation List	
Key	**Range**
File.dat_0-2097152	204800 - 524288

Fig. 2: Write-optimized mapping strategy.

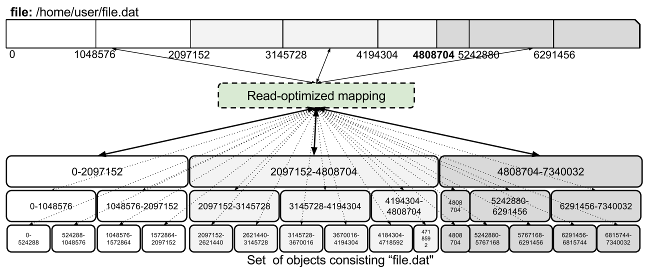


Fig. 3: Read-optimized mapping strategy.

guarantees the data consistency for any future fread(). In this strategy we always strive for strong consistency.

3) **Write-optimized**

In this mapping strategy we aim for very fast write operations. To achieve this, we always create a new object for each request and send it down to the object store. We also insert this newly created object to a B+ tree structure that maps file offset ranges to available keys within this range. This structure will help future read operations to retrieve the correct data. If the write operation updates existing data, we still create a new object but we also invalidate the data in the previously inserted object by adding the offset range in the *invalidation list*. This list guarantees the data consistency for read operations. Therefore, every *fwrite()*'s mapping could create overlapping keys or even duplicate keys. The real work is done by the read operation where the mapping needs to first find which are the correct keys corresponding to the request. It then performs one or multiple *get()* operations from the object store, concatenates the correct data according to the invalidation list, and returns to the caller of *fread()*. Figure 2 describes this mapping strategy. Let us assume that we have three write operations. The first `fwrite(buf,2097152,1,fh)` will create one object. Similarly, the other two write operations will similarly create one object each. The naming convention for this mapping strategy is *filename_startingOffset-endingOffset*. The interesting thing happens when we get an update operation. The `fwrite(buf,319488,1,fh)` at starting offset 204800 overwrites existing data. In the previous mapping, a *get()* operation would be triggered, a concatenate operation and then a new *put()*. In this mapping, we simply create a new object

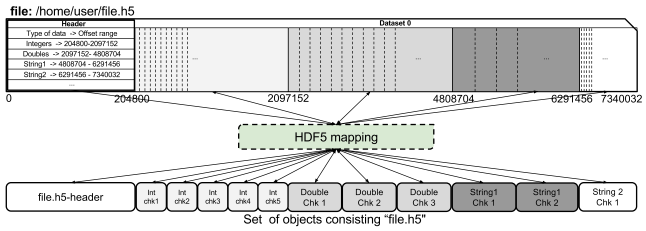


Fig. 4: HDF5 mapping strategy.

and we invalidate the range of data this new write operation updates. This slows down the read operations where multiple objects are first retrieved and then, a concatenation of only the valid data is performed. However, write operations are faster since this mapping was designed for write-heavy workloads.

4) **Read-optimized**

In this mapping strategy we prioritize read operations over write operations. To achieve a higher read performance, we sacrifice write performance and also extra storage space. Specifically, for every write request we create a plethora of various-sized objects. The granularity of this object creation is configurable but it is recommended to split objects down to 512KB. This strategy also maintains a map of all available keys for a given range of file offsets. Figure 3 demonstrates this mapping. Let us follow an example to better explain this mapping strategy. Assuming an *fwrite()* of 2 MB request size and offset 0, and a granularity of object sizes of 512 KB, the mapper will create the following keys: 1 key of 2MB, 2 keys of 1 MB and 4 keys of 512 KB. All of the created keys correspond to the same data. Any update operation of existing keys, needs to update all keys of the same data. However, a subsequent *fread()* will access the best combination of these keys trying to minimize the calls to the underlying object store while maintaining a strong data consistency. For instance, let us assume an *fread()* of 1 MB request size starting at the beginning of the file. That will trigger the mapper to return one of the 1 MB keys which best fits the data. This strategy is recommended for read-intensive workloads.

5) **HDF5 files**

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. Unlike a POSIX file, which is just a stream of bytes, a HDF5 file offers rich metadata information on the file. Additionally, it allows storing and retrieving of portions of data in a logical way while abstracting the complexity of the underlined file. This potentially lets us access data or just a portion of it more efficiently. This self-descriptive nature of HDF5 file allows us to design our mapping strategies that could take advantage of this rich metadata and intelligent sub-setting features and therefore, efficiently mapping them to objects.

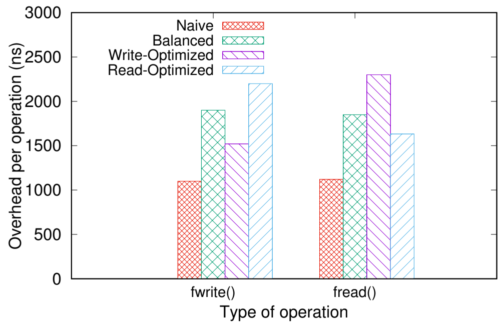


Fig. 5: Mapping overhead expressed in nanoseconds(ns).

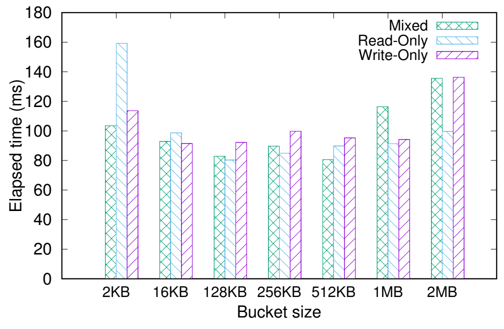


Fig. 6: Balanced mapping with various bucket size.

In this strategy, we first create an object that contains all the metadata information. Specifically, it contains the header and the organization schema of the entire file. This metadata is kept in-memory for the entire operation on the HDF5 file. This is done to enable applications to access the metadata information as fast as possible. It is persisted upon closing the file onto the object store. As datasets and attributes are added to the file, this object can expand its size. Then, since each dataset can contain multiple dimensions (i.e., each having a type of data such as integers etc.), our strategy maps these dimensions into variable-sized objects. This granularity is configurable and it depends on the type of data it contains (e.g., every 20 units of the type of data the dimension holds). The naming schema for dimensions is: *filename_dataset_dimension_base-offset*. Figure 4 shows this mapping strategy. The benefit of creating variable-sized objects according to the data type is clear. Within dimensions we follow the balanced mapping which gives us a balance performance for reads as well as writes.

V. Evaluation Results

A. Experimental Setup

Testbed: All experiments were conducted on Chameleon systems [30]. More specifically, we used the bare metal configuration offered by Chameleon. Each client node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (i.e., a total of 48 cores per node), 128 GB RAM, 10Gbit Ethernet, and a local 200GB HDD. Each server node has the same internal components but, instead of Ethernet network, we used Infiniband 56Gbit/s to avoid possible network throttling.

Software: The operating system of the cluster is CentOS 7.0, the MPI version is Mpich 3.2, the PFS we used is OrangeFS 2.9.6, and the Object Store MongoDB 3.4.3. We used our own synthetic benchmark (i.e., a workload generator) and as an input, we used multiple workload characteristics such as only-read, only-write, and mixed read-write. Additionally, we run IOR [31], a famous I/O benchmark which measures I/O performance at both MPIIO and POSIX level. Finally, we use *Montage* [32], an astronomical image mosaic engine, to test our mappings with a real application workload. All test results are the average of five repetitions.

B. Experimental Results

1) **Mapping overhead**

In this test we evaluate the mapping cost of our strategies. The input is 65536 POSIX calls of 128KB size. We evaluate the overhead per operation (i.e., fwrite() and fread()) and we report the average time spend in mapping (I/O time is excluded) in nanoseconds. As it can be seen in Figure 5, the naive mapping shows overhead of 1200 ns since it simply maps each file to an object. The balanced mapping demonstrates an overhead close to 2000 ns per operation and lies between the write-optimized and read-optimized which had less overhead for their respective input. Note here that these numbers refer only to the time spent in mapping each request.

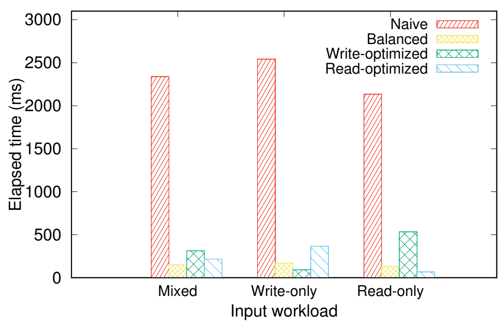
2) **Balanced mapping: performance with variable buckets**

In this test we have as an input various workloads with median request size 128 KB. We used a variable bucket size and captured the overall completion time for the balanced mapping. Bucket size is the unit of data that our library interacts with the underlying object store therefore it is one of the optimization knobs in our disposal. As it can be seen in Figure 6, a very small bucket size (e.g., 2 KB) hurts the performance since many objects are created and accessed. Similarly, for big buckets such as 1 or 2 MB there is excessive reading and splicing of the objects that contain the desired data and thus, lead to some performance degradation. The best performance comes by bucket sizes close to the *median request size*, in this test with 128, 256 and 512 KB.

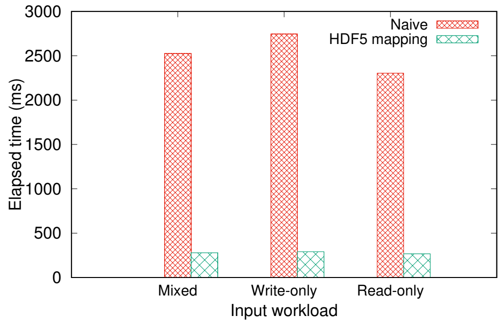
3) **Performance with synthetic benchmark**

For the following tests, we deployed MongoDB on one server node and we run the tests on a separate client machine. The tests are single-threaded. The maximum disk speeds are 480 MB/s for read and 350 MB/s for write operations. We report results in milliseconds (ms). Finally, We use three different workloads as input: mixed, read-only and write-only.

POSIX files: In this test, each I/O call is of 1MB size and the aggregated I/O is 32MB in total. As it can be seen in



(a) POSIX files.



(b) HDF5 files.

Fig. 7: Mapping performance.

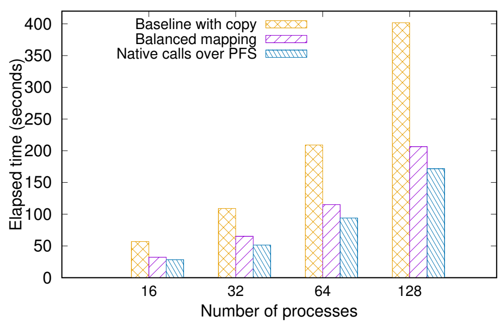


Fig. 8: I/O performance with IOR mixed workload.

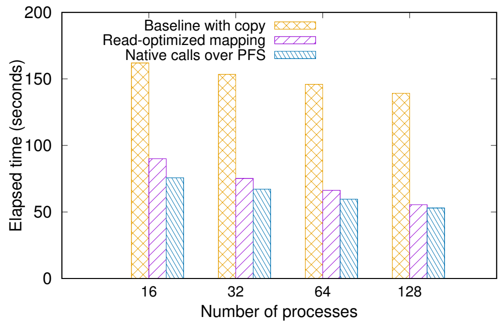


Fig. 9: I/O performance with Montage.

Figure 7 (a), each mapping performs best for the workload it was designed. When compared to the naive mapping, our balanced mapping strategy performs 15x faster for the mixed input. The read-optimized mapping strategy is 32x faster than the naive for the read-only input but performed 4x slower than the write-optimized for the write-only case since it writes more objects on the underlying object store. Finally, the write-optimized mapping strategy is 27x faster than the naive for the write-only input and almost 2x faster than the balanced mapping strategy.

HDF5 files: In this test, we use an HDF5 file with one dataset of integers. Each I/O call is of 1MB size with an aggregated total I/O of 32MB. As a baseline, we use a simple one to one mapping where each HDF5 file is mapped into an object. As it can be seen in Figure 7 (b), our mapping strategy outperforms the naive mapping by 9x for write and 8x for read operations.

4) **Performance with real applications**

For the following tests, we deployed MongoDB and OrangeFS on 4 separate server machines for each storage system and we used up to 4 client machines. The scales are 16, 32, 64, and 128 processes.

IOR: In this test, we run IOR in MPI-IO mode with 2MB block size and 512KB transfer size. Each process is performing 512MB of total I/O in its own separate file. We disable all caching and IOR performs direct I/O. This test involves a mixed workload of both read and write operations, and therefore, we only evaluate our balanced mapping strategy. The baseline workflow refers to first copying the input data from the object store into the parallel file system and then running IOR over the PFS. As a reference, we also include the performance of executing IOR over PFS with native I/O calls and without the copying time. As it can be seen in Figure 8, our library performs almost **2x** faster than the baseline since it redirects all I/O over the object store without copying the input data over to PFS. Furthermore, our solution scales well under this strong scaling scenario. Note that the time reported for our solution includes the bootstrapping time of the library that happens only once in the beginning.

Montage: In this test, we run Montage to evaluate our read-optimized mapping strategy since the application is mostly performing read operations to create a mosaic from multiple input images. The total input size is 24GB and it remains the same to create a weak scaling scenario as we increase the number of processes. For the baseline, the input images reside in the object store and therefore, they need to be first copied to the PFS before the application starts. We also compare our solution, which allows the input to be accessed directly over the object store, with the native POSIX calls when the input data already reside on the PFS. As it can be seen in Figure 9, our library performs more than **2x** faster than the baseline since it redirects all I/O over the object store without copying the input data over to PFS. Note that copying time dominates the overall execution time as we scale it. The added overhead of our library when compared with the native calls over the PFS is considered minimal (i.e., only 4% for the 128 processes).

VI. Conclusions and Future Work

In this study we first explored file, block and object data models and the characteristics of their respective storage systems. We then presented several novel strategies to map a file to one or more objects. We implemented four techniques that take into consideration the input workload to offer the best possible performance with minimum overheads. Our evaluation shows that with better design of the mapping algorithms we can get 2x-30x higher performance compared to a naive mapping of files to objects. Additionally, we showed that our library can perform more than 2x faster when running real applications since we avoid the costly data movements and transformations.

As a future step we plan to incorporate these mapping strategies into a bigger I/O framework that integrates different storage subsystems and thus, come closer to a true convergence of parallel and distributed architectures. We believe these mappings are a fundamental step towards this goal. We envision a system that offers universal data access regardless of the storage interface and our mappings are a fundamental step towards this goal.

References

[1] OpenGroup, "POSIX standard," 2016. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[2] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, IEEE. Annapolis, Maryland: IEEE, 1999, pp. 182–189.

[3] M. Folk, A. Cheng, and K. Yates, "Hdf5: A file format and i/o library for high performance computing applications," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, vol. 99. Portland, OR: ACM, 1999, pp. 5–33.

[4] P. J. Braam *et al.*, "The Lustre storage architecture," 2014. [Online]. Available: ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf

[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, vol. 2. Monterey, CA: Usenix, 2002, pp. 231–244.

[6] 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*. Atlanta, GA: Usenix, 2000, pp. 391–430.

[7] 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.

[8] S. Conway and C. Dekate, "High-Performance Data Analysis: HPC Meets Big Data," 2014. [Online]. Available: https://goo.gl/k8wN9U

[9] J. Dongarra *et al.*, "The International Exascale Software Project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[10] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. New York, NY: Elsevier, 2011.

[11] Amazon, "Amazon S3," 2016. [Online]. Available: http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

[12] Monty Taylor, "OpenStack Object Storage (swift)," 2016. [Online]. Available: https://launchpad.net/swift

[13] A. Lakshman and P. Malik, "Cassandra," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 35, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1773912.1773922

[14] MongoDB, "MongoDB," 2016. [Online]. Available: https://www.mongodb.com/white-papers

[15] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," *Acm sigcomm computer communication review*, vol. 42, no. 4, pp. 25–36, 2012.

[16] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[17] H. P. D. D. Intel® Enterprise Edition for Lustre* Software, "WHITE PAPER Big Data Meets High Performance Computing," Intel, Tech. Rep., 2014. [Online]. Available: http://www.intel.com/content/www/us/en/lustre/intel-lustre-big-data-meets-high-performance-computing.html

[18] H. Devarajan, A. Kougkas, X.-H. Sun and H. B. Chen, "Open ethernet drive: Evolution of energy-efcient storage technology," in *Proceedings of the ACM SIGHPC Datacloud'17, 8th International Workshop on Data-Intensive Computing in the Clouds in conjunction with SC'17*. Denver, CO: ACM, 2017.

[19] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines." in *FAST*, 2013, pp. 119–132.

[20] S. Zhou, B. H. Van Aartsen, and T. L. Clune, "A lightweight scalable i/o utility for optimizing high-end computing applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Miami, FL, USA: IEEE, 2008, pp. 1–7.

[21] L. Bryan, "The UK JASMIN Environmental Data Commons," 2017. [Online]. Available: https://wr.informatik.uni-hamburg.de/_media/events/2017/iodc-17-lawerence.pdf

[22] 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.

[23] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 53.

[24] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system." in *FAST*, vol. 8, 2008, pp. 1–17.

[25] F. Wang, S. A. Brandt, E. L. Miller, and D. D. Long, "Obfs: A file system for object-based storage devices." in *MSST*, vol. 4, 2004, pp. 283–300.

[26] D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross, "An evolutionary path to object storage access," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 36–41.

[27] Google, "TCMalloc library," 2016. [Online]. Available: https://github.com/gperftools/gperftools

[28] ——, "CityHash library," 2016. [Online]. Available: https://github.com/google/cityhash

[29] ——, "B-Tree library," 2016. [Online]. Available: https://code.google.com/archive/p/cpp-btree/

[30] Chameleon.org, "Chameleon system," 2016. [Online]. Available: https://www.chameleoncloud.org/about/chameleon/

[31] "Ior benchmark," https://goo.gl/YtW4NV.

[32] Montage, "An Astronomical Image Mosaic Engine." 2016. [Online]. Available: http://montage.ipac.caltech.edu/docs/m101tutorial.html