Syndesis: Mapping Objects to Files for a Unified Data Access System

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*—The predominant data model in cloud storage is the object-based storage. Object stores follow a simpler API with *get()* and *put()* operations to interact with the data. A wide variety of data analysis software has been developed around objects using their APIs. In fact, the evolution of Big Data analytics is a major driver for highly optimized data-centric quality software. However, organizations maintain file-based storage clusters and a high volume of existing data are stored in files. This is specifically true for the scientific communities. In this paper, we present the key characteristics of object-based and file-based storage APIs, we explore several object-to-file mappings aiming to bridge the semantic gap between these data models. The evaluation of our mapping algorithms exposes various strengths and weaknesses of each strategy and frames the extended potential of a unified data access system. Results show that our solution can offer more than 3x higher performance for specific workloads while keeping minimal overhead of our library.**

***Index Terms*—Data Management, Data-Intensive Computing, Convergence of HPC and Big Data, Object Stores, Parallel File Systems, Semantic gap**

I. Introduction

In the modern era of data explosion via Web services and with the tremendous growth of Cloud environments, data are stored, accessed, shared, and processed as objects [1]. The need for extreme scales and programmatic ease of accessing data, gave birth to Object Storage. Popular examples include Microsoft's Azure DocumentDB [2], Google's collection of Cloud Storage (Cloud Storage [3], Datastore [4], BigTable [5]), Amazon's DynamoDB [6], Cassandra [7], MongoDB [8] and others. Most of these services offer simplistic APIs with basic *get()*, *put()*, and *delete()* operations. These storage systems consist the NoSQL space and support REST APIs [9], [10]. A wide variety of freely available data analysis software has been built around Object Stores. Most notably Apache BigTop collection [11] offers a comprehensive suite of highly optimized and scalable software.

On the other hand, historically in traditional computing such as cluster computing and high-performance computing [12] and specifically in fields like science, finance, banking, stock market and others, data are stored in files. File systems, both local and distributed, have been developed and highly optimized through the years. In large scale environments, parallel file systems (PFS) such as Lustre [13], GPFS [14], and PVFS [15] are in use. Distributed file systems are also in large deployments most notably GoogleFS or HDFS [16]. All of these systems use popular storage interfaces and standards to interact with the files. POSIX-IO [17], MPI-IO [18] and HDF5 [19] are some of the most widely used.

Files, objects, and the systems supporting them are not in harmony. File systems and object stores are designed with different end goals and therefore interfaces have drifted away from a uniform data access. 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 [20]. BigData analytics [21], [22] software has been very successful in Cloud environments that run on top of some kind of an object store. We understand the need to bridge the gap between file and object semantics by a) exposing existing data repositories residing in file systems to cloud data processing solutions without the need of costly data movement and transformations, and b) allowing the cloud software stack to operate on files and thus, bringing powerful data processing capabilities to more traditional computing sites without the cost of re-developing software or entirely transitioning to a cloud environment (e.g., solutions on premises).

In this paper, we explore several ways to map objects to files. More specifically, we designed and implemented four new mapping strategies of a typical object (i.e., in the form of a key-value pair regardless of the value type) to one or more files. These mapping strategies cover a wide variety of input workload characteristics and express several features such as scalability, consistency and high performance. This work paves the way towards a unified data access system and allows a cross-platform data processing. We envision a data path agnostic to the underlying data model and we aim to leverage each storage solution's 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. Our mapping strategies are transparent to the application which simply needs to connect to our library (i.e., either by recompilation or preloading).

The contributions of this paper are:

- We present key characteristics of object-based and file-based storage solutions.
- We design and implement a unified storage access system that bridges the semantic gap between object-based and file-based storage systems.
- We evaluated our solution and the results show that, in addition to providing programming convenience and effi-

ciency, our library can grant higher performance avoiding costly data movements between object-based and file-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

Performing data analysis using HPC resources can lead to performance and energy inefficiencies [23]. In [24] 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 BigData analytics systems are different in design philosophies and target different applications. D. Reed and J. Dongarra in [25] 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.

This divergence 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 [26]. 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 [27] 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. 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 JASMINE needs to support a wide range of deployment environments.

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 [25]:

1) 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,
2) difference in architecture of programming models and tools,
3) management of heterogeneous resources and,
4) management of diverse global namespaces stemming from different data pools.

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.

III. Background and Related Work

A. *What is a file-based storage*

A file system 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 an object-based storage*

An object store 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. The object store keeps the blocks of data that make up an object together. It also adds extended metadata to the object 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.

C. *Object vs file storage*

Object stores overcome 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

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

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. Finally, parallel file systems exploit high degree of parallelism to offer high bandwidth which makes them great for streaming data. Table I summarizes some of the key differences.

Another critical characteristic to understanding the nature of our mapping strategies is that data analysis software relies on existing storage interfaces. Our solution does not require any changes to the application code but instead intercepts all storage calls (e.g. CRUD operations) and redirects them to the file system.

D. Related work

Ceph [28] is a new type of distributed file system that can write in both objects and files since it's underlying storage is based on object store devices. With this design, they created APIs that can support both file operations and object operations. However, to use their system, one needs to switch the entire storage installation to a Ceph deployment, and applications need to be rewritten using the specific Ceph API to be able to use it. PanasasFS [29], [30] and OBFS [31] similarly utilize OSDs. Their design offloads some administrative tasks on the disk itself making it run faster for specific workloads. Our solution does not require any change to the application code and it can bridge the semantic gap between objects and files regardless the underlying storage system.

Intel's DAOS [32] has explored a unification of multiple namespaces into a global one where applications can access data from multiple underlying storage solutions such as simple POSIX, HDF5 or ADIOS [33]. That is a step towards a unification of interfaces and storage subsystems.

Major cloud providers have built file connectors to their object storage infrastructure. Amazon's Elastic File System [34] provides a standard file system interface and file system access semantics for data that reside in Amazon cloud storage. Even though this helps accessing data in the cloud using a file interface, it is tightly coupled with Amazon Web Services and it is not a general solution like ours. Microsoft's Azure File Storage [35] makes data migration easy. They urge users to move their data in this service in order to access and process data by utilizing the same code. Our solution can skip the moving of data and can operate even on premises. User's can simply use powerful cloud software to analyze existing data on their file-based clusters.

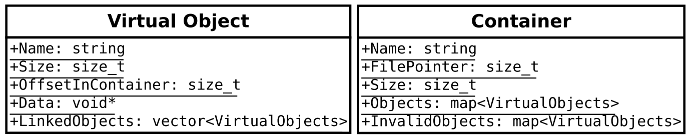

Virtual Object
+Name: string
+Size: size_t
+OffsetInContainer: size_t
+Data: void*
+LinkedObjects: vector<VirtualObjects>
Container
+Name: string
+FilePointer: size_t
+Size: size_t
+Objects: map<VirtualObjects>
+InvalidObjects: map<VirtualObjects>

Fig. 1: Virtual Object - Container

IV. Design and Implementation

A. Library implementation details

Our library is implemented in the middleware layer. A prototype version can be found online. [1] 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 [36] that performs 20% faster than the standard malloc and has a smaller memory footprint. For hashing, we selected the *CityHash* functions by Google [37], 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 [38] which is faster than the STL equivalent containers and reduces memory usage by 50-80%.

Upon initialization of the library, a namespace crawler scans the underlying file system to create the initial metadata information. Additionally, if the user provides access to a key-value store, the crawler will scan the key space and will create a unified single namespace for the user. All these metadata information are stored in a special file in the file system. The library also loads this metadata file in memory for faster queries. Deletion operations are handled by invalidation and namespace garbage collection on application exit.

B. Mapping Strategies

We designed and implemented four new strategies to map user's objects to underlying files. Relative to object-to-file mapping those are: 1-to-1, N-to-1, N-to-M simple, and N-to-M optimized. Each of these mappings demonstrate strengths and weaknesses and were designed to offer greater flexibility to various input workloads. To achieve those mapping strategies we introduce two new terms. *Virtual Object*, which encapsulates the application's object along with other metadata information necessary to our library. Virtual objects are the unit of mapping to the underlying file system. *Container*, which represents a file in the file system that holds virtual objects and other metadata information useful to our library such as indexing and update logs. Containers may exist either in memory or on disk. The data consistency is guaranteed by the POSIX-compliant file system. Figure 1 shows their respective classes and their public members.

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

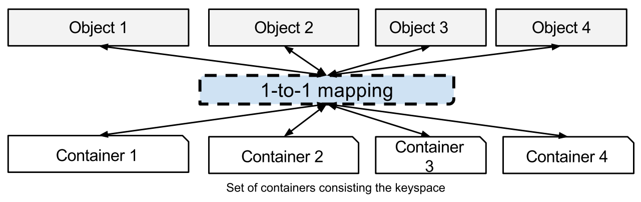


Fig. 2: Mapping strategy: 1-to-1

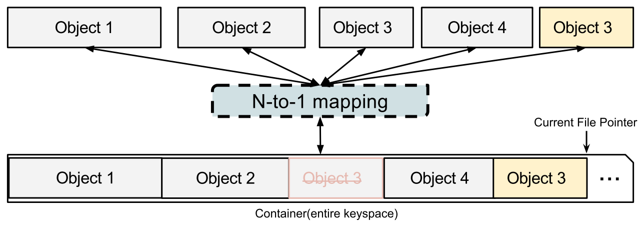


Fig. 3: Mapping strategy: N-to-1

1) **One object to one file (1-1)**

Description: In this mapping strategy, each application's object is mapped to a unique container (i.e., file). The goal is to enable processing of existing collections of files. For example, assuming there is a collection of pictures stored in a file system, this strategy will map each file to an object. Therefore, one can access and process this dataset by simply using a *get()* and *put()* interface. This strategy is also quite fast for a relative small number of objects. Figure 2 demonstrates this strategy.

Challenges: Since each object is mapped to a unique container, the underlying file system needs to manage many, often small, files. This is a known performance bottleneck of file systems because of excessive metadata operations. Every time a file is accessed, the file system needs to check permissions, update the timestamps (i.e., time accessed, time modified) and other related structures. This excessive metadata operations is also the main reason of scalability issues with file systems.

Benefits: The benefits of using such mapping strategy are several. The mapping semantics are the simplest. The overhead of this mapping and the memory footprint are kept at minimum. Update operations simply mutate the respective file. Finally, for existing data residing in a file system, this is the only strategy that could allow such access to the data. In order to achieve this, there needs to be a bootstrapping sequence where the entire file system namespace is subscribed into a key space.

Limitations: The maximum number of files in a file system has a limit (e.g, for NTFS and ext4 this limit is 4 billion). Also, the number of concurrently opened files is limited (e.g., default in ext4 is 64K). Therefore, first the I/O performance with large number of objects will be low due to increased metadata operations, and second the scalability suffers from the limitations a file system tree hierarchy imposes.

2) **Multiple objects to one file (N-1)**

Description: In this mapping strategy, the entire keyspace of application's objects is mapped to one container. Figure 3 demonstrates the details of this strategy. The goal is to maintain the simplicity of the mapping while attacking the limitations of the previous strategy. Virtual objects are written sequentially in the container. Any updates are simply appended at the end of the file. This strategy is good for smaller dataset sizes.

Challenges: Since each object resides in one big container, indexing is very important to facilitate faster *get()* operations. For this reason each virtual object maintains the container offset where the actual object resides. The container also maintains a map of existing virtual objects. Update operations need to update all of these memory structures before data are written in the disk.

Benefits: The benefits of using this strategy are extracted mostly by solving the inefficiencies of the 1-to-1 mapping. Only one file is created and maintained in the underlying file system which makes the metadata operations lightweight. Searching is offloaded from the file system to our library and structures are kept in memory for faster operations. Mapping cost is relatively low. This strategy works well in a multi-application environment as well. Data consistency is guaranteed by the file system. Concurrent reads are allowed.

Limitations: The limitation of this strategy is that the container size can and will grow arbitrarily large. Most of the file systems impose a limit to the maximum file size (i.e., ext4 is 32TB, XFS is 8EB). While this may seem a difficult number to reach, moving forward to exascale, this strategy will be limited by this.

3) **Multiple objects to multiple files simple (N-M [S])**

Description: In this mapping strategy, a collection of application's objects is mapped to a collection of containers. The constraint for the creation of new containers is the container size. After each container reaches the maximum container size (i.e., default in our solution is about 128MB) it will trigger the creation of the next container. This strategy tries to overcome the previous strategies' limitations. The number of files is controllable by the strategy and containers' size is predefined (i.e., user can tune this). Update operations mutate the virtual object that resides in the container. If the updated object's data is larger, then a special update-container linked with the original container is created to hold the extra data. A background compaction thread crawls the file system periodically or by demand to concatenate container's with their respective update-container. Figure 4 demonstrates this mapping strategy.

Challenges: While this mapping strategy overcomes many of the scalability limitations of the previous strategies, searching for an object is quite challenging. Each container maintains a map of existing objects and each object is linked to possible updated objects in the update-container. However, searching the entire key space is burdensome since one must query all the containers. A possible fix for this is to apply bloom filters on the containers much like Google's LevelDB [39]. This filter

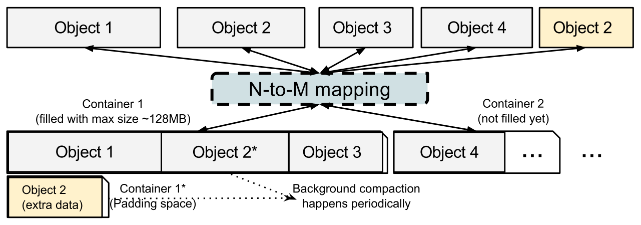

Object 1
Object 2
Object 3
Object 4
Object 2
N-to-M mapping
Container 1 (filled with max size ~128MB)
Container 2 (not filled yet)
Object 1
Object 2*
Object 3
Object 4
...
...
Object 2 (extra data)
Container 1* (Padding space)
Background compaction happens periodically

Fig. 4: Mapping strategy: N-to-M simple

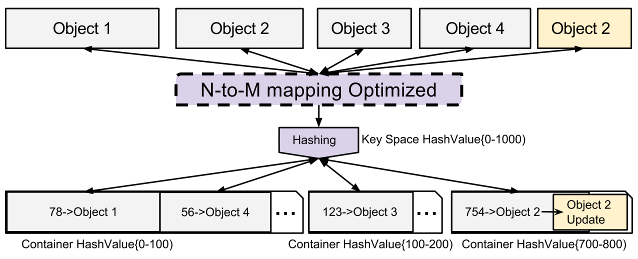


Fig. 5: Mapping strategy: N-to-M optimized

will reduce the number of unnecessary disk reads needed for a *get()* call by a factor of approximately a 100 according to the documentation of LevelDB.

Benefits: By controlling the number and the size of created containers the mapping algorithm creates data access patterns to the underlying file system that are favorable to its I/O performance. Our library performs *fwrite()*s and *fread()*s to the file system in chunks which significantly helps the performance specifically write operations. We also implemented caching in memory for hot keys in a separate container to facilitate faster *get()*s. Finally, the scalability of this strategy is definitely higher than the previous two mappings.

Limitations: There is no apparent scalability limitation. However, as discussed above, for workloads with heavy object mutations, the performance of this mapping can be limited when compared with the previous two strategies. Additionally, the mapping cost is higher than before and the overheads from compactions can be considerable even though it happens asynchronously.

4) **Multiple objects to multiple files optimized (N-M [O])**

Description: In the optimized version of N-to-M objects to files mapping strategy, application's objects are first hashed into a key space and then mapped to the container responsible for that range of hash values. For hashing we used Google's CityHash [37] collection that is proven to be very fast (i.e., in the order of few nanoseconds). Figure 5 shows how this strategy works. Specifically, keys go through the hashing function and get a 128 bit hash value. Containers are created according to a range of hash values. This strategy is extremely scalable since containers represent a range of keys regardless of their size. The container size is relative to the overall size of the keys it holds. Update operations simply write at the end of the container while invalidating the previous object. A background thread periodically performs defragmentation of the containers to save storage space. Searching is performed in constant time. To achieve this, we associated a truth array with each container. If an object exists in the container then the index of that object's hash will be true. We implemented a similar logic to quickly check if a container already exists. The goal of this strategy is to be able to scale and to support fast writes, reads and updates. To further optimize the performance, we perform all I/O operations in a non-blocking fashion. Virtual objects are first put in memory in a *MemTable* whose size is configurable. Once the memtable is filled, a backup memtable becomes active in order to accept incoming objects. The virtual objects in the filled memtable are then written in their respective container on disk. A separate thread is spawned in the background to perform the file system call. With this asynchronism, we managed to overlap I/O calls with computation and boost the performance of the overall mapping. A blocking version of this mapping strategy is also available, but we highly recommend users to use the non-blocking.

Challenges: Frequent updates could create fragmented containers. We solved this with a background defragmentation thread. Another challenge comes from the mapping based on the hash value. Containers responsible for a range of keys could end up with only few keys stored in them resulting to a lot of file system calls. Therefore, for small number of objects this mapping strategy could be overwhelming. In this case, we recommend using a 32bit or 64bit hashing to limit the number of containers (i.e., configurable withing our library through Google's CityHash algorithms).

Benefits: This strategy is the most scalable. It is the best for large number of objects. Writes and updates are fast. Reads and searches are also very fast using the indexing we implemented. We also kept the mapping cost at its minimum.

Limitations: The only limitation of this strategy is the higher memory usage with memtables consuming space otherwise allocated to the application. If memory space is of importance, users can use the blocking version of this strategy and set the memtable size to few MBs.

V. Evaluation Results

A. Experimental Setup

Testbed: All experiments were conducted on the Chameleon testbed [40]. More specifically, we used the bare metal configuration offered by the system. Our node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz, 128 GB RAM, 10Gbit Ethernet, and a local 240GB SSD drive. The performance of the disk drive is about 520MB/s read and 350MB/s write.

Software and applications: The operating system of the cluster is CentOS 7.0 and the PFS we used is OrangeFS 2.9.6 [41] (formerly known as PVFS2 [15]). As a key-value store (KVS) we used Redis 4.0.1 [42]. We used our own synthetic workload generator that given an I/O trace file it can "replay" all I/O operations. As an input, we used multiple workload characteristics such as only-get, only-put, mixed, and get - put with fixed or variable request size. All test

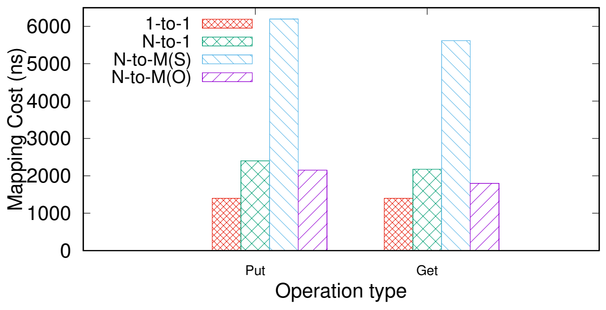


Fig. 6: Mapping cost per operation in nanoseconds

results are the average of ten repetitions. For real application workloads, we used Yahoo Cloud Serving Benchmark [43], a widely used framework with variety of workloads for evaluating the performance of different key-value and cloud serving stores. We also used a K-means clustering application [44] to further examine our solution.

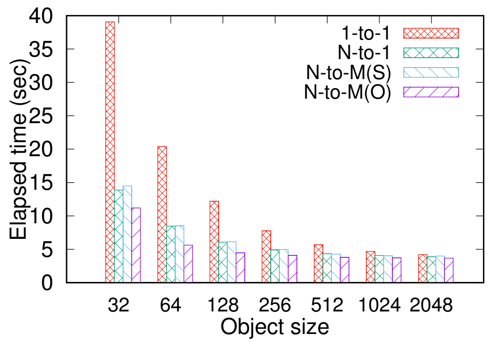
B. Experimental Results

1) **Mapping cost**

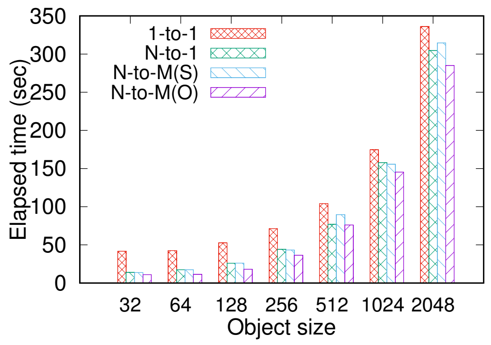
Figure 6 shows the average mapping cost expressed in time (ns). In this test, the input is 128K objects (i.e., 131072) of 64KB size with mixed get-put calls. We report the average time spend in mapping. As it can be seen, the 1-to-1 strategy has the smallest cost since it naturally maps each object to a file. Our N-to-M simple has the highest since each object mapping involves more tasks such as checking the container's size (i.e., create a new container if needed).

2) **How the object size affects the I/O performance**

The object size is very important in terms of performance of the underlying storage system. In this test we change the object size from 32KB to 2MB and measure the overall time to perform the I/O over the parallel file system. In Figure 7 (a), the input data size is 1.2GB which means that the number of objects to map is changing. For instance, for 32KB object size the total number of objects are roughly 40000 whereas for 2MB object size the number drops to 600. The best performance is coming from the N-to-M optimized strategy which is 3.5x faster than 1-to-1 for small objects and 1.2x faster for large ones. The N-to-1 and N-to-M simple perform almost the same. In Figure 7 (b) we keep the number of objects the same while we change the object size. Specifically, the total number of objects to be mapped is 40000 which gives a total data size of 1.2GB for the 32KB object size and roughly 80GB for the 2MB object size. We can observe that all mapping strategies scale well for variable object size with the N-to-M optimized being the best. It is clear that with larger object size the file system grants higher performance in all mapping strategies.



(a) Weak scaling (decreasing number of objects)



(b) Strong scaling (same number of objects)

Fig. 7: Variable object size overall performance.

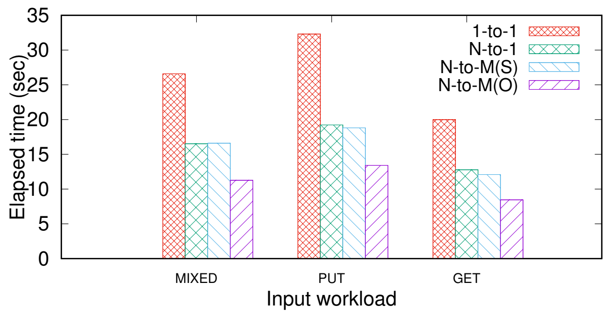


Fig. 8: Performance of mapping strategies.

3) **Library mapping and I/O performance**

Synthetic benchmark: In this test, we created a synthetic workload that consists of 4GB total size of I/O. The object size is kept at 64KB. For the mixed case in figure 8, the benchmark first writes 1GB, reads the data back, then it updates the 1GB and reads it again. The put and get cases are simply writing and reading the entire 4GB respectively. The 1-to-1 mapping suffers from the excessive number of files created (i.e, about 65536). The N-to-M optimized performed the best since it takes advantage of the memtables and writes to the disk with better granularity (i.e., relative to the memtable size and not the 64KB objects). It completed the test in 11 seconds for the mixed case, 13 seconds for the put and 8.5 for the get case. Note the difference in read and write performance because of the specifications of our drive.

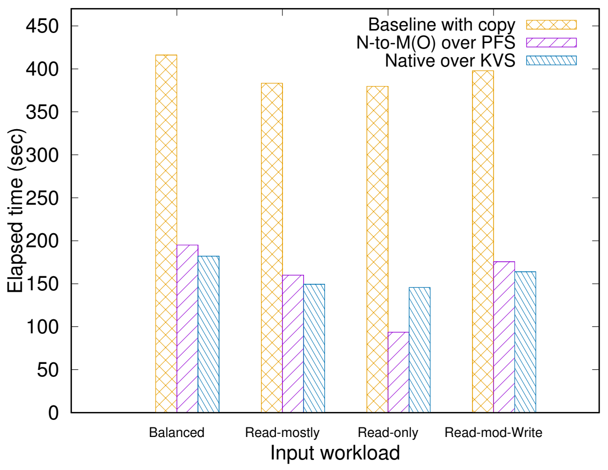


Fig. 9: Performance comparison using YCSB.

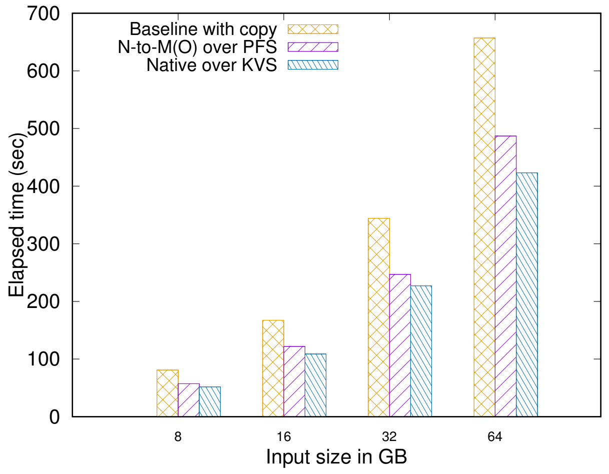


Fig. 10: Performance comparison using K-means clustering.

Real applications: To test our mappings under real workloads we used 2 applications: Yahoo Cloud Serving Benchmark (YCSB) and K-means clustering. We used all 24 cores on our testbed and all storage systems used are deployed locally. We also preloaded data (i.e., offline preparation) that will be used as inputs to the applications. We tested the following workloads from YCSB: a) Balanced: 50% reads and 50% writes. b) Read-mostly: 90% read and 10% writes c) Read-only: 100% read d) Read-modify-write: the client will read a record, modify it, and write back the changes. The total I/O in this test is 64 GB. It is performed with a 64 KB object size in the form of records (i.e., 64 fields, 1000 bytes each including the key). For the K-means clustering algorithm aims to find the evenly spaced sets of points in subsets of euclidean space and partition these subsets into well-shaped and uniformly sized convex cells. The algorithm starts with initial placement of some number k of points in each cell. It then repeatedly computes the centroid for each cell and moves the k points till it converges. This algorithm is a mixture of computation and I/O intensive phases. It reads points from the disk, performs the k-means algorithm and write back the final output. We use four point data sets with 8, 16, 32, and 64 GB total size respectively to test the scaling. Under these tests, all get() and put() operations are mapped via our library to a PFS and we measure the overall time to execute the workload and the achieved throughput in operations per second.

In figure 9, we compare our N-to-M optimized mapping strategy (i.e., the fastest from the previous test) to the baseline which consists of first copying and transforming the data to an Object Store and then perform the computations using native I/O calls (i.e., in this case over Redis). Finally, as a reference we include the time to run the application on top of the native storage system (i.e., blue bar in the figure). As it can be seen, our solution provides more than a 2x boost in performance since it avoids the costly data movement and allows the application to utilize the object store APIs over a PFS. As it is expected, the copying dominates the overall time. However, even without the copying, our solution provides competitive performance when compared to the native I/O calls over the KVS. Especially in the read-only test case, our mapping strategy creates a sequential access pattern which favors PFSs and provides higher performance due to read-ahead and prefetching.

Figure 10 shows the results for K-means clustering application. In this test, baseline consists of first copying the input data from PFS to the KVS and then executing the computations. Since this application is more computation-intensive than the pure I/O of YCSB, we can observe that copying data is a smaller fraction of the overall execution time. Even so, our solution offers more than 40% performance improvement. When compared to the native calls over the KVS, our mapping only adds 9% overhead to perform all necessary operations. Specifically, for the 8 GB input our solution took 57 seconds to complete whereas the baseline needed 80 and the native calls over KVS 52 seconds.

VI. Conclusions and Future Work

In this paper, we designed and implemented several object-to-file mapping strategies. We evaluate all mapping strategies and we report strengths and weaknesses of each one of them. Our evaluation shows that with better design of the mapping algorithm we can get almost 4x higher performance compared to a naive mapping of objects to files. Our N-to-M objects-to-files mapping strategy demonstrated higher performance for *get()*, *put()*, and *update()* operations consistently. Additionally, we showed that our library can perform more than 2x faster than existing solutions for specific workloads.

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] R. Cattell, "Scalable sql and nosql data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[2] R. Becker, *Learning Azure DocumentDB*. Packt Publishing Ltd, 2015.

[3] Google, "Google Cloud Storage," 2017. [Online]. Available: https://cloud.google.com/storage/docs/

[4] ——, "Google DataStore," 2017. [Online]. Available: https://cloud.google.com/datastore/docs//

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[7] 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

[8] MongoDB, "MongoDB," 2017. [Online]. Available: https://www.mongodb.com/white-papers

[9] Amazon, "Amazon S3," 2017. [Online]. Available: http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

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

[11] Apached Software Foundation, "Bigtop - Apache Bigtop," Apache, 2016. [Online]. Available: http://bigtop.apache.org/

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

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

[14] 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.

[15] 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.

[16] 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.

[17] OpenGroup, "POSIX standard," 2017. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[18] 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.

[19] 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.

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

[21] E. Joseph and S. Conway, "IDC Update on How Big Data Is Redefining High Performance Computing," IDC, Tech. Rep., 2014.

[22] 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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: a proposal for an exascale storage system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 50.

[33] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[34] Amazon, "Amazon Elastic File System," 2017. [Online]. Available: https://aws.amazon.com/efs//

[35] Microsoft, "Microsoft Azure File Storeage," 2017. [Online]. Available: https://azure.microsoft.com/en-us/services/storage/files/

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

[37] ——, "CityHash library," 2017. [Online]. Available: https://github.com/google/cityhash

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

[39] G. Inc, "LevelDB," 2017. [Online]. Available: https://github.com/google/leveldb

[40] Chameleon.org, "Chameleon system," 2017. [Online]. Available: https://www.chameleoncloud.org/about/chameleon/

[41] "OrangeFS parallel file system." [Online]. Available: http://www.orangefs.org/

[42] "Redis key-value store." [Online]. Available: https://redis.io/

[43] "Yahoo! Cloud Serving Benchmark." [Online]. Available: https://github.com/brianfrankcooper/YCSB

[44] "K-means Clustering application." [Online]. Available: https://github.com/genbattle/dkm