IRIS: I/O Redirection via Integrated Storage

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

ABSTRACT

There is an ocean of available storage solutions in modern high-performance and distributed systems. These solutions consist of Parallel File Systems (PFS) for the more traditional high-performance computing (HPC) systems and of Object Stores for emerging cloud environments. More often than not, these storage solutions are tied to specific APIs and data models and thus, bind developers, applications, and entire computing facilities to using certain interfaces. Each storage system is designed and optimized for certain applications but does not perform well for others. Furthermore, modern applications have become more and more complex consisting of a collection of phases with different computation and I/O requirements. In this paper, we propose a unified storage access system, called IRIS (i.e., I/O Redirection via Integrated Storage). IRIS enables unified data access and seamlessly bridges the semantic gap between file systems and object stores. With IRIS, emerging High-Performance Data Analytics software has capable and diverse I/O support. IRIS can bring us closer to the convergence of HPC and Cloud environments by combining the best storage subsystems from both worlds. Experimental results show that IRIS can grant more than **7x** improvement in performance than existing solutions.

CCS CONCEPTS

•Information systems → Mediators and data integration; Distributed storage; *Data exchange; Data federation tools; Cloud based storage; Hierarchical storage management;* Wrappers (data mining);

KEYWORDS

Parallel File Systems, Object Storage, Integrated Access, Unified Storage, I/O Convergence

ACM Reference format:
Anthony Kougkas, Hariharan Devarajan, Xian-He Sun. 2018. IRIS: I/O Redirection via Integrated Storage. In *Proceedings of ICS '18: International Conference on Supercomputing, Beijing, China, June 12–15, 2018 (ICS '18),* 10 pages.
DOI: 10.1145/3205289.3205322

1 INTRODUCTION

In the age of Big Data, scientific applications are required to process large volumes, velocities, and varieties of data, leading to an explosion of data requirements and increased complexity of use [10]. In High-Performance Computing (HPC), traditional data management consists mostly of parallel file systems (PFS), such as Lustre [4], PVFS2 [37], GPFS [39], etc. Historically, the data model of the underlying storage systems has followed the POSIX standard and PFSs have been responsible for managing it. However, while the single stream of bytes model of POSIX is needed for strong consistency, it is inconvenient for parallel access and might also lead to expensive data transformations. As we get closer to the exa-scale era, PFSs face significant challenges in performance, scalability, complexity, limited metadata services, and others [12], [17]. Modern HPC storage systems are not the best fit for Big Data applications since they were designed with traditional scientific applications in mind.

High availability of popular general purpose analysis frameworks like MapReduce [11], Spark [51], and others in Apache BigTop [2], as well as the wide variety of available Object Stores such as MongoDB [28], HyperDex [13], and Cassandra [24], have created a healthy software environment in Cloud computing and Big Data applications. However, these analysis frameworks are not designed for HPC machines and do not take advantage of any capabilities of the extremely expensive and sophisticated technologies present in existing supercomputers. They also cannot support traditional HPC workloads (i.e., MPI applications) and would most likely fail to meet the demand of High-Performance Data Analytics (HPDA) [19], the new generation of Big Data applications, which involve sufficient data volumes and algorithmic complexity to require HPC resources. International Data Corp. (IDC) forecasts that the HPDA market will grow from $3.2 billion in 2010 to $16.9 billion in 2018 [20]. Currently, approximately 70% of HPC sites around the world with at least 30% of their available compute cycles perform HPDA.

HPDA is driven by the increasing ability of powerful HPC systems to run data-intensive problems at larger scale, at higher resolution and with more elements. 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. In [43] 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. D. Reed and J. Dongarra in [35] point out that the tools and cultures of HPC and HPDA 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 [54]. 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 [7] 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 HPDA. However, such unification is extremely challenging with a wide range of issues [35]: 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, d) management of diverse global namespaces stemming from different data pools, etc. 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.

In this paper, we introduce two novel abstractions, namely Virtual Files and Virtual Objects, that help overcome the above mentioned challenges. We present the design and implementation of IRIS (I/O Redirection via Integrated Storage), a unified and integrated storage access system. IRIS is a middleware layer between applications and storage. By using virtual files and objects, IRIS can unify any data model and underlying storage framework, and thus, allow applications to use them collaboratively and interchangeably. With IRIS, an MPI application can directly access data from an Object Store avoiding the costly data movement from one system to another while providing an effective computing infrastructure for HPDA. Thus, IRIS creates a unified "storage language" to bridge the two very different compute-centric and data-centric data storage camps. By using this "language", IRIS extends HPC to HPDA; a vital need from both the HPC and the data analytic community. IRIS seamlessly enables cross-storage system data access without any change to user code. IRIS's modular design allows the support of a wide variety of applications and interfaces. From MPI simulations to HPDA analysis software and from high-level I/O libraries such as pNetCDF [26], HDF5 [14], MOAB [41], MPI-IO [42] etc., to the more Cloud-based Amazon S3 [1] and Openstack Swift [30] REST APIs, IRIS comfortably integrates the I/O requests.

The contributions of this paper are: 1) We designed and implemented a unified storage access system that integrates various underlying storage solutions such as PFSs or Object Stores. This system is called IRIS. 2) We introduced two novel abstract ideas, the Virtual File and the Virtual Object that can help map user's data structures to any data management framework. 3) We evaluated our solution and the results show that, in addition to providing programming convenience and efficiency, IRIS can grant higher performance by up to 7x than existing solutions in the intersection of HPC and HPDA.

2 BACKGROUND

Parallel file systems: PFSs are widely popular and well understood within the storage community. Therefore we will not expand much on the background and we only list some advantages and disadvantages to provide context. The main advantages a PFS offers is simultaneous access by many clients, scalability, and capability to distribute large files across multiple nodes, a hierarchical global name space, and high bandwidth via parallel data transfer. While many scientific applications following the PFS assumption of access one single large contiguous file, some applications such as those in astronomy and climatology have non-contiguous data access patterns and generate many small I/O requests. While PFSs excel at large and aligned concurrent accesses, they face significant performance degradation in case of small accesses, unaligned requests, and heavy metadata operations [8]. Some limitations of PFSs [18], originate from persistence: data access performance is dependent of the underlying files, directories, and tree structures (the higher the number of files in a PFS, the greater the risk of performance degradation). Maintaining data consistency of file systems poses an overhead on the overall system, and often creates issues such as fragmentation, journaling, and simultaneous operations on the same file system structures. Finally, the storage subsystem may pose additional limitations because of RAID, disk sizes, and other limiting factors by either hardware or software. PFS will need a major lifting for the next generation of exascale supercomputers, even without considering the newly emerged HPDA I/O demands.

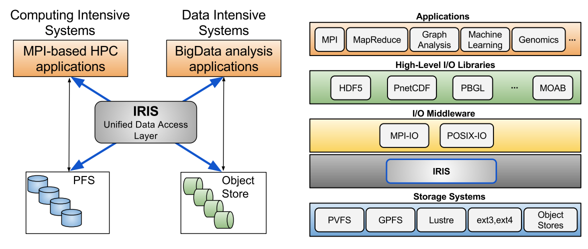
Object Storage: Object Stores encapsulate data, metadata, a globally unique identifier, and data attributes into a single immutable entity termed *object*. In an Object Store, objects are organized in a flat address space (i.e., every object exists at the same level in a large scalable pool of storage). Object Stores follow location independent addressing and are accessed via the unique identifier. Object Stores are designed primarily to manipulate data sets that do not have a predefined data model or in other words are unstructured or semi-structured. The key operations for an Object Store include retrieval (get), storage (put), and deletion (delete). Object Stores can be categorized as generic key-value stores, document-based stores, column-based stores, and graph-based stores. With a flat namespace and the ability to easily expand and store large data sets at a relatively low cost, Object Stores offer scalability, flexibility, rapid data retrieval, and distributed access. They are easily expandable and are well suited for applications requesting non contiguous data accesses and/or heavy metadata operations. Object Stores offer consistent data access throughput and a set of extensible metadata. The Object Store space (also known as NoSQL schemes) demonstrates a huge variety of different implementations each with its own strengths and weaknesses. Most of the implementations maintain the above characteristics. On the other hand, Object Stores are ill suited for access patterns with frequently changing data and ones involving complex operations since each update operation leads to the creation of a new object and the destruction of the previous one followed by an update to the metadata. Additionally, Object Stores are not POSIX-compliant. This blocks their wide adoption by the HPC community. Object stores, however, are widely used in the Cloud community and in Big Data processing engines.

With ever increasing data sizes, typical data intensive applications such as machine learning and data mining, require more and more computing power. These applications seek traditional HPC technologies to speed them up, including support for complex data structures and algorithms. There is no "one storage for all" solution and each storage system, PFS and Object Store, has its strengths and weaknesses. We explored architectural differences and performance characteristics in [23] and we found that each storage system performs best under specific conditions and they could perfectly complement each other.

3 DESIGN AND IMPLEMENTATION

3.1 IRIS Objectives

While designing IRIS we kept three major objectives in mind:
A: Enable MPI-based applications to access data in an Object Store without user intervention. This objective is designed to support two major use cases automatically. 1) An MPI application can write data directly to an Object Store environment where in turn an HPDA application will operate upon. 2) An MPI application can read data, previously created by an HPDA application, that reside in an Object Store. IRIS enables MPI applications to access data directly from Object Stores by making them accessible natively. The familiar *fread()* and *fwrite()* POSIX calls, used by MPI applications, are still the interface to access data that reside in an Object Store. Additionally, high-level I/O libraries such as MPI-IO, HDF5, and pNetCDF are also supported by IRIS.
B: Enable HPDA-based applications to access data in a PFS without user intervention. This objective is designed with two major use cases as well, similar but in an opposite direction as the first objective. 1) HPDA application needs data, previously generated by an MPI application, that reside in a PFS. 2) HPDA application can write data directly to a PFS where an MPI application will operate on it. With IRIS, HPDA applications can directly access data to and from PFSs natively. In particular, IRIS allows the *get()* and *put()* from an HPDA application to operate on files residing in a PFS. Combined with the first one, this objective gives us a powerful way to store, access, and process data from two different environments by two different computing engines, a more computing-centric processing done by MPI, and a more data-centric processing done by HPDA software. Figure 1 (a) visualizes these two objectives. The black arrows represent the native data path for each system while the blue arrows demonstrate the new data paths IRIS enables.
C: Enable a hybrid storage access layer agnostic to files or objects. This objective is designed to offer a truly hybrid access to data by abstracting the low-level storage interfaces and unifying the APIs under one system. Therefore, IRIS allows developers to interchange the storage calls independent of the underlying architecture. Via the hybrid storage layer that IRIS provides, applications can access data from both storage systems at the same time. Additionally, this allows IRIS to make intelligent decisions, use each storage system exploiting its advantages, and offer a higher I/O efficiency.



(a) Unified data access layer. (b) Software stack.

Figure 1: IRIS in high-level.

This objective basically eliminates the black arrows in Figure 1 (a) and grants IRIS the decision making responsibility about which storage subsystem to use.

3.2 Design Considerations

While developing IRIS, we faced and solved many challenges. PFSs are tightly coupled with the POSIX standard. This is a known restriction of the scalability of PFSs and a major source of performance degradation [33]. On the other hand, Object Stores are not POSIX-compliant which makes them scale very well and offer low latency for specific workloads [48]. However, they cannot replace PFSs in scientific computing due to the lack of POSIX-compliance and support of complex data structures. IRIS implements tunable consistency [44] and two modes of POSIX-compliant metadata (i.e., strict and relaxed). Therefore, IRIS can trade some POSIX-compliance to grant better performance and scalability if the application demands it. Fault tolerance is crucial in any system. IRIS adopts the fault tolerance of the underlying storage subsystems. Additionally, IRIS periodically writes the in-memory metadata information to the disk. In case of a crash, IRIS restores the metadata image from the disk and continues. In the current version of IRIS, there is still a possibility of losing metadata in between of a checkpoint and the time of the crash. We plan to extend this work and add fault tolerance features such as write logs etc.

Note that different implementations of a PFS or Object Store might possibly lead to different features and performance characteristics. However, IRIS aims to bridge the semantic gap between files and objects by abstracting the lower level storage details. With IRIS, HPC users can utilize a vast variety of data analysis software otherwise only available in Big Data environments, which in turn will increase productivity, performance, and resource utilization.

3.3 IRIS Architecture

IRIS is a library that sits between applications and storage systems. As such, it interacts with applications from above and issues its I/O requests to the underlying storage system. This is no different from any other middle-ware library. However, the design of IRIS makes it capable to connect to many different applications, both HPC and HPDA, and storage systems at the same time. Figure 1 (b) demonstrates the new software stack with IRIS. One important note is that IRIS is integrated transparently to the application and high-level I/O libraries. Its modular design enables IRIS to support different applications and makes it flexible for future interfaces. IRIS already supports POSIX, MPI-IO, HDF5, pNetCDF, and S3/Swift user interfaces. From the storage system side, IRIS can interact with the local

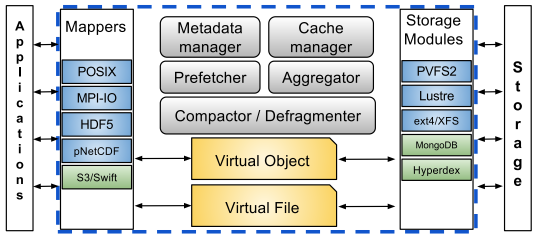


Figure 2: IRIS internal design.

Linux ext3 or ext4 file systems, PVFS2 (also known as OrangeFS), Lustre, as well as HyperDex and MongoDB Object Stores. There are several components that work together to make IRIS a two-way bridge between MPI applications/PFS and HPDA applications/Object Stores. Two such components are abstract ideas, the VirtualFile and the VirtualObject. IRIS utilizes these components internally to build the connections between incompatible standards. Figure 2 demonstrates IRIS' design. We use the following example to illustrate and explain the components one by one: an MPI application accesses an Object Store to first write some data and then to read them back.

VirtualFile is an in-memory construct that is used for mapping and thus, bridging the semantic gap between POSIX files and objects. Its goal is to provide applications the illusion of the existence of a normal file system and that all operations are performed on a "real" file. The virtual file is simply a space in memory which IRIS is using to map user's file structures to key-value pairs. The relationship between a virtual file and the objects that it includes is 1-to-N.

VirtualObject is an in-memory construct that is used for mapping objects to files and therefore bridges the semantic gap between them. Its goal is to provide applications the illusion of the existence of a normal Object Store much like the virtual file above. The relationship between virtual objects and files is N-to-1.

Mappers are responsible for creating a translation between what the applications are passing to IRIS and what the underlying storage system expects. This translation is possible with the use of virtual files and/or objects. For instance, if the application calls *fwrite()*, then IRIS intercepts it and calls the POSIX mapper to create a mapping to the underlying storage. For our example, a virtual file is created holding the keys responsible to carry out the operation. This virtual file is then passed to the storage module. When the application calls *fread()*, the mapper finds the virtual file that contains the keys that hold the data and passes it to the storage module.

Storage modules are responsible for issuing I/O requests to the underlying storage systems. IRIS aims to support many different storage systems, hence the modular design. Storage modules are tied to the specific storage subsystem they implement. Storage modules take a virtual file or virtual object (e.g., if application is MPI or S3, respectively) as an input and return the actual data. Going back to our example, the storage module takes a virtual file created from the mapper and, using the appropriate interface (e.g., MongoDB API), it calls *get()* or *put()* according to the desired operation.

Metadata manager is responsible for keeping track of metadata information about the entire library. First, it maintains any metadata required from the user interface in memory. We explored two modes for IRIS: a strict-POSIX mode where all metadata from POSIX are maintained in memory and persisted in the Object Store at *fclose()*, and a relaxed-POSIX mode where basic metadata information are kept. There is a trade-off between POSIX compatibility and performance. Metadata manager is also responsible for maintaining all memory structures that facilitate the mapping between user calls and storage system calls (i.e., virtual files, virtual objects).

Compactor/defragmenter is a background internal service. IRIS makes use of virtual namespaces, virtual files, and virtual objects to achieve a true integration of two incompatible systems. IRIS maps application's objects to virtual objects that are stored within files in the PFS. Therefore, these files can end up fragmented after a series of update operations. This component is activated periodically in the background to defrag fragmented data structures and files, or compact buffers and sets of key-value pairs. The compactor applies all object mutations asynchronously much like LevelDB [16]. The work done from this IRIS component is crucial internally, and it guarantees a successful and efficient execution. Lastly, it can help with performance optimizations such as better indexing of the active namespace.

Aggregator is a performance-driven designed component. It is equivalent to the collective I/O operation in MPI-IO. Aggregator always tries to combine requests together for better performance. IRIS interacts with the storage systems in an optimized way. That means, either issue larger requests, avoiding small accesses and excessive disk head movements, or minimizing network traffic.

Prefetcher is the second performance-driven component. It does exactly what the name suggests. It implements a few prefetching algorithms to optimize I/O operations as whole. The current version of IRIS supports prefetching for sequential data access, strided access, and random access. We plan to support user defined in future versions.

Cache manager is the component that handles all buffers inside IRIS. It is equipped with several cache replacement policies such as least recently used (LRU) and least frequently used (LFU). It works in conjunction with the prefetcher. It can be configured to hold "hot" data for better I/O latency. It acts as a normal cache layer for the entire framework.

3.4 Implementation Details

IRIS is implemented in C++ and totals more than 11000 lines of code (LOC). A prototype version can be found online. 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 that performs 20% faster than the standard malloc and has a smaller memory footprint. For hashing, we selected the *CityHash* functions by Google, the fastest collection of hashing algorithms at this moment. We specifically used the 64-bit version of the hashing functions. For structures such as maps and sets, we used Google's BTree library which is faster than the STL equivalent structures and reduces memory usage by 50-80%. Finally, all configurable parameters in IRIS are globally set per application via our *Configuration Manager*.

3.4.1 ***Mapping modes.*** IRIS maps user's data structures to an underlying storage system in two directions: files-to-objects and objects-to-files.

Files-to-objects: we designed and implemented three different mapping strategies: a) balanced, b) write-optimized, and c) read-optimized. These strategies aim to better serve respective workloads and are all configurable through IRIS's configuration settings.

For the balanced mapping, IRIS divides the virtual file into buckets which are tied to a fixed-sized object. It then maps any file request, that falls into a bucket according to the offset and the size of the operation, to the respective object or collection of objects. The mapping is the same regardless of the read or write operation since it maps file location to objects. The size of these buckets is configurable and can affect the performance. After extensive testing, we found that a bucket size of the median request size 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.

For the write-optimized mapping, IRIS creates objects for each request and inserts them in a B+ tree hierarchy for the subsequent get operations. In this mapping, *fwrite()* and *fread()* have different mapping functions since we prioritize the write operation speed. Therefore, for write operations we simply create objects as fast as possible and we update our map of available ranges of file offsets and keys. For read operations, the mapper first finds the correct keys within the range of offsets passed, and performs one or multiple *get()* operations from the Object Store. It then concatenates the correct data according to a timestamp (i.e., the latest data for each key are kept) and it returns to the caller.

For the read-optimized mapping, IRIS first creates a plethora of various-sized keys for each put request and updates the map of available keys and ranges of file offsets. The goal here is to speed up the *fread()* so most of the work is done by the write operation. For example, assuming an *fwrite()* of 2 MB request size at 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. Therefore, a subsequent *fread()* will access the best combination of these keys minimizing the calls to the underlying Object Store while maintaining a strong data consistency. This mapping strategy significantly speeds up read operations by sacrificing extra storage capacity, method generally acceptable due to the low cost of disk space. More on the design, implementation, and evaluation results of these three mapping strategies are explored and presented in details in [21].

Objects-to-files: we designed and implemented four mapping strategies regarding object-to-file mapping: a) 1-to-1, b) N-to-1, c) N-to-M simple, and d) N-to-M optimized. These strategies aim to maintain data consistency while being general enough to support a variety of workloads. These mapping modes are configurable through IRIS's configuration settings.

In the 1-to-1 mapping strategy, each application's object is mapped to a unique file. The goal is to enable processing of existing collections of files and one can access and process data by simply using a *get()* and *put()* interface. The overhead of this mapping and the memory footprint are kept at minimum. Update operations simply mutate the respective file. The mapping semantics are also the simplest making this strategy quite fast for a relatively small number of objects.

In the N-to-1 mapping strategy, the entire keyspace of the application's objects is mapped to one big file. The goal is to maintain the simplicity of the mapping. Virtual objects are written sequentially in the file. Any updates are simply appended at the end of the file while marking the previous object as invalid. This strategy is good for smaller dataset sizes. Since each object resides in one big file, indexing is very important to facilitate faster *get()* operations. For this reason each virtual object maintains the file offset where the actual object resides. Under this strategy, metadata operations of the underlying file system are lightweight. Searching is offloaded from the file system to IRIS with in-memory structures for faster operations. The mapping cost is relatively low. Data consistency is guaranteed by the file system. Concurrent reads are allowed.

For the N-to-M simple mapping strategy, we first introduce a new structure, called *container*, which represents a file that holds virtual objects and other metadata information useful to IRIS such as indexing and updating logs. 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 IRIS is 128MB) it will trigger the creation of the next container. 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.

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. 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. IRIS' defragmenter periodically runs in the background 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. The goal of this strategy is to be able to scale and to support fast writes, reads, and updates. More on the design, implementation, and evaluation results of these four mapping strategies are explored and presented in details in [22].

3.4.2 ***Prefetching modes***.
Traditional PFSs and Object Stores implement optimizations such as read-ahead and prefetching to offer better read performance. These optimizations rely on the data access pattern created by the applications. However, when IRIS maps a file call over an Object Store, it loses this capability since an *fread()* will be transformed to one or more *get()* operations with keys that may not have a relation between them, and thus, any prefetching the underlying Object Store tries to perform will not work. Similarly, a *get()* in IRIS may be transformed in a sequence of *fread()* operations that may not demonstrate locality or sequentiality, and thus, the underlying file system prefetching will not help. Therefore, we implemented these optimizations within IRIS. Specifically, prefetching in IRIS has two modes: synchronous (i.e., read-ahead) and asynchronous. We have also implemented three prefetching algorithms: sequential, random, and user-defined. For the *sync* mode, prefetching is performed synchronously. Each *fread()* triggers the prefetcher component and, depending on the

Table 1: IRIS' metadata modes (time in ms)

#files	IRIS_Strict	IRIS_Relaxed	Local-Ext4	Remote-OrangeFS
1000	3.398	2.941	149.439	4,513.024
10000	33.712	28.928	1,257.540	50,058.524
100000	345.769	287.198	13,152.400	434,528.242
1000000	3,644.250	2,989.640	143,828.000	4,934,578.340

access pattern, it uses one of the prefetching algorithms, sequential or random, to fetch the next piece of data. It passes the fetched data to the cache manager. Any subsequent *fread()* will check if the requested data are already in cache before reaching to the disk. The difference with the *async* mode, as the name suggests, is that the *fread()* will return to the caller after it triggers an asynchronous fetch. IRIS maintains a map of outstanding asynchronous operations, and thus, every *fread()* first checks for any pending fetch before it actually performs any other reading. This asynchronicity will boost the performance for certain workloads where I/O and computation are periodically switching. The sequential algorithm takes the current read arguments, such as the offset, the size, and the count, and tries to calculate the next piece of data that the application will need. IRIS offers a configurable parameter about the prefetching unit. It can be exactly the size of the previous *fread()* or a predefined value (e.g., 2 MB). For instance, an *fread()* of 1 MB will trigger a fetching of the next 1 MB or if the predefined value is set, the next 2 MB. For all the asynchronous calls, we used the built-in standard library std::future with the *"async"* scheduling flag on.

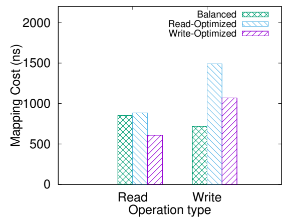
3.4.3 ***POSIX-compliant metadata modes.*** While the POSIX standard has been around for a very long time and has served us well, there are certain features in the standard that may have less value as we move to the exa-scale era. One of the much debated characteristics of the POSIX file is its metadata information. These are expressed by a structure named *Stat* and they include the following: the device ID, the file serial number, the mode of the file, the number of hard links, the user ID, the group ID, the size of the file, the time of last access, the time of last data modification, the time of last status change, the block size, and lastly, the number of blocks. All of this information is involved in all POSIX calls. Maintaining these structure updated (e.g., updating the time accessed or checking the file permissions) can be a performance bottleneck. IRIS offers a fully POSIX compliant mode, we call *strict-POSIX* that obeys the standard. However, since the files are mapped into an Object Store, it is not always needed to maintain all above information. Hence, we created a second mode called *relaxed-POSIX* where we only update crucial information about the virtual file such as the file size, and the mode checking. By skipping the rest of the metadata operations, IRIS can offer higher performance. The test comprises of opening and closing up to a million files. Table 1 compares each IRIS' metadata mode with POSIX and reports the time. The *relaxed-POSIX* offers about 18% higher performance when compared with the *strict-POSIX* mode. This test was conducted on our development machine with an Interl i7, 16GB RAM and an SSD drive of 480MB read and 350MB write speed. The actual POSIX calls were tested on a local Linux ext4 file system and on a remote OrangeFS file system and are presented as a reference. Moreover, IRIS maintains metadata information in-memory until *fclose()* is called when IRIS persists the memory structures to disk. This way, IRIS demonstrates orders of magnitude better performance compared to a traditional file system that updates metadata on disk.

3.4.4 ***Caching.*** IRIS utilizes many buffers and caches. The cache manager offers a memory space to the prefetcher to cache data and also to the application itself to cache user's data. The size of all buffers are configurable via IRIS's configuration settings. We have implemented two cache replacement policies, least recently used (LRU) and least frequently used (LFU). These algorithms are no different than any other common LRU and LFU implementations. IRIS also offers the ability to cache write operations for subsequent reads. This write-caching, along with prefetching, can boost the reading performance, and is spatial and temporal tunable, giving more control to the user. Caching plays a big role in the performance optimization, and good memory management is crucial. IRIS has successfully incorporated such technologies. Note that IRIS' caching is on top of any caching mechanisms inherent by the underlying storage solution.

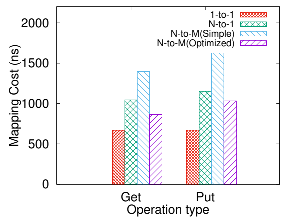
4 EVALUATION

4.1 Hardware and software used

Testbed: All experiments were conducted on Chameleon systems [9]. 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. The total experimental cluster consists of 1536 client MPI ranks (i.e., 32 nodes), and 16 server nodes for PFS and Object Store each.
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. Our choice of those specific storage systems as representatives from each category (e.g., PFS and Object Stores) was made for several reasons. OrangeFS (formerly known as PVFS2) is widely understood by the HPC community, and it is mature enough in terms of development and research to be the representative for the PFS. MongoDB has established itself as one of the most popular Object Stores. It offers competitive performance and several API bindings. We acknowledge that different representatives of each storage subsystem may have differences in their implementations. However, the focus of this study is not benchmarking OrangeFS and MongoDB, but the evaluation of the unified data access layer and how IRIS performs against the native workload of a PFS and of an Object Store. Thus, we believe this issue does not hurt the conclusions and contributions of this study.
Applications: The applications we used span over a wide range of scientific simulations and data analysis kernels. These applications are real-world code representative of applications running on current supercomputers. They have been used on NCSI's Kraken and NCSA's Blue Waters, ORNL's Titan, and ANL's Intrepid and Mira. Specifically we used: *CM1* [6, 15], a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model designed for idealized studies of atmospheric phenomena, *LAMMPS* [34, 38], a classical molecular dynamics code and an acronym for



(a) Files-to-objects



(b) Objects-to-files

Figure 3: IRIS mapping overhead.

Large-scale Atomic/Molecular Massively Parallel Simulator, *Montage* [29], an astronomical image mosaic engine, *WRF* [46], a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs, *LANL_App1* [27], an anonymous scientific application running in Los Alamos National Lab, and *K-Means clustering*, a typical HPDA data analysis kernel. All test results are the average of five repetitions to eliminate OS noise.

4.2 Evaluation Results

4.2.1 ***IRIS library overhead.*** In this test, we measure the overhead of I/O calls using IRIS expressed in time (nanoseconds). The reported time refers to the difference between an I/O call to the native storage solution and the same call over IRIS which will redirect it to a different storage system. For example, a normal *fread()* will read data from a file system whereas in IRIS will be mapped to a *get()* operation, and data will be fetched from an Object Store. The I/O time is excluded since we wanted to isolate the overhead added by the translation of the I/O calls. Figure 3 shows the average overhead in time for both mapping files-to-objects and objects-to-files. In the first case, the input is 128K file operations (i.e., 131,072 POSIX calls) of 64KB size with mixed reads and writes. The overhead per call on average in this case is about 1100 nanoseconds or 0.00025%. In objects-to-files case, the input is similarly 128K object operations (i.e., 131,072) of 64KB size with mixed gets and puts. In this case, the extra time needed by IRIS to map an object call to a file is on average 1300 nanoseconds, or a 0.00030% relative to the native system, depending on the mapping strategy used. Thus, the mapping overhead of IRIS is minimal.

4.2.2 ***IRIS I/O performance.*** In this series of tests, we evaluate IRIS in real world scenarios. We first run the applications and we isolate the I/O phases since we only want to study the storage performance and not the time spent in computations. Using the IOSIG tracing tool [50] we collect all I/O traces. Each process operates on its own file (i.e., file-per-process) and performs I/O of about 100MB. The total dataset is 150GB for the largest scale.

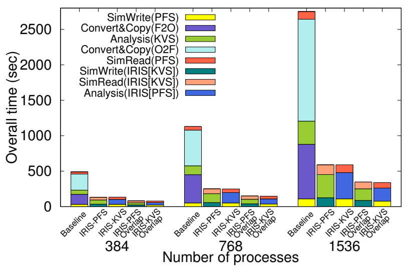
Each test consists of several phases. We first run the simulation part of the application on top of a PFS. In the figures, this is noted *SimWrite* followed by the storage system in parenthesis. We then convert and copy all data (i.e., output of the simulation phase) to a data-intensive cluster equipped with an Object Store (e.g., MongoDB in our case). Note that in this section we refer to Object Stores as KVS (i.e., Key-Value Stores) for short. In the figures, this is noted as *Convert&Copy* followed by the direction of the data transfer (i.e., files-to-objects as F2O and objects-to-files as O2F). We then execute the data analysis using the respective analysis kernel, and it is presented in the figures as *Analysis* followed by the storage system in parenthesis. The results of the analysis are written in the KVS and then are converted and copied back to the PFS for the next phase of the simulation which is referred as *SimRead* followed by the storage system in parenthesis. This execution flow is similar to how NASA's Goddard Space Flight Center first conducts climate simulation on their supercomputer facilities and then the data analysis of the observation data on a different cluster designed for data-intensive computations. We refer to this flow as *Baseline* in the following figures. The reported time is a compound of the time needed by all phases and is calculated as:

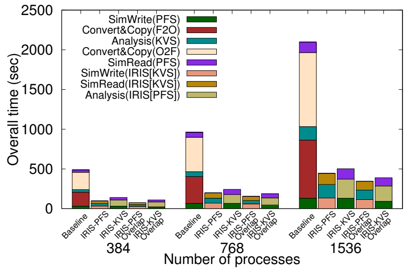
Total time $= SimulationWrite + CopyDatafromPFStoKVS + DataAnalysis + CopyDatafromKVStoPFS + SimulationRead$ (1).

The typical I/O workload of simulations is mostly checkpointing and it is repeated periodically [5]. Thus, we focus our evaluation to one such checkpoint phase in which application's data access patterns are the same.

When executing the tests on top of IRIS, applications can use new data paths to read or write data. As Figure 1 (a) shows, IRIS can store simulation output from an MPI application directly to a KVS, and analysis results directly to a PFS eliminating any data transfers between KVS and PFS. Therefore, during the execution flow we examine, IRIS offers two new directions of performing the I/O. First, the MPI simulation can directly write data to the KVS. In the figures, this is noted as SimWrite or SimRead followed by IRIS in parenthesis with the underlying storage system in brackets (e.g., IRIS[KVS] means IRIS runs on top of KVS). This way, the data analysis application makes native I/O calls to the KVS in order to write the analysis results. After that, the simulation can read the data directly from the KVS (i.e., fread() via IRIS). Second I/O direction via IRIS is the case where data analysis application reads simulation data and writes analysis results directly from/to PFS, noted as Analysis followed by IRIS in parenthesis with the underlying storage system in brackets (e.g., IRIS[PFS] means IRIS runs on top of PFS). Besides the elimination of data movements between storage subsystems, IRIS offers the opportunity to further optimize the entire workflow by overlapping the phases in Equation 1. When the HPC and data analytic environments are separated, as in the baseline, all phases are also separated and must execute serially one after the other. With IRIS, these phases can be overlapped (e.g., data analysis can start as soon as the first simulation results are available). IRIS' overlapping mode, noted as *IRIS-Overlap*, can significantly reduce the total execution time and transform the workflow from a pure serialized process to a concurrent one. Figure 4 shows all performance results.

CM1: CM1's workload demonstrates a sequential write pattern. In this test, every process first writes the checkpoint data (e.g., atmospheric points with a set of features), then data are combined with observation data residing on the KVS and are analyzed with a Kmeans clustering kernel. Finally the analysis results are fed back to the simulation as an input for the next phase. As it can be seen in Figure 4(a), the fastest simulation time is on top of the PFS and the fastest analysis time is on top of the KVS. However, the transfer of data between PFS and KVS dominates the overall execution time. On the other hand, IRIS eliminates the need for copying data and redirects the calls to the appropriate storage


(a) CM1 performance with IRIS


(b) Montage performance with IRIS

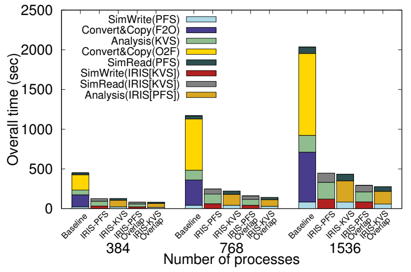

(c) WRF performance with IRIS.

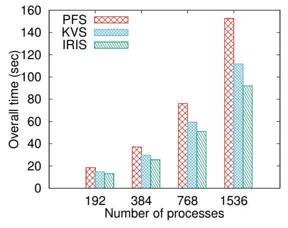
Figure 4: IRIS' I/O performance.

system. The performance gain is more than **7x** for the IRIS-overlap. The scalability of our solution is characterized as linear. Note that running Kmeans clustering on top of PFS is 1.7x slower on average when compared to running on top of the KVS due to the small data accesses that this kernel demonstrates.

Montage: Montage's workload demonstrates a diverse pool of tasks. As an image mosaic builder, it creates a mosaic with 10 astronomy images. It uses 8 analysis kernels, and is composed of a total of 36 tasks. The first phase is performed using MPI on top of the PFS. Then, the analysis tasks are offloaded to the data analysis cluster. There are task dependencies and therefore the analysis results are written back to the PFS as input to the simulation tasks. Figure 4(b) demonstrates the results of the evaluation. Similarly to the previous application, the copying takes much of the overall time for the base case. Since IRIS avoids the data transfer, it can speed up the overall time by more than **4.5x** and scales linearly. Furthermore, IRIS-overlap outperforms the baseline by **6x**. Note that running the Montage Analysis kernels on top of PFS is 2.2x slower on average when compared to running on top of the KVS.

WRF: WRF's workload characteristics demonstrates two distinct phases. As a weather forecast model, it first performs a simulation using MPI. Data are (x,y,z) data points with several fields such as temperature, wind, humidity, etc. The simulation output is copied to the KVS and data are merged with other data collected from sensor networks, and are then analyzed. The analysis results are copied back to the PFS for the next simulation phase. This process is repeated until the model converges. As it can be seen in Figure 4(c), IRIS significantly speeds up by **7x** the overall execution time. Note that running the WRF Analysis kernels on top of PFS is 50% slower against running them on top of the KVS.

4.2.3 ***IRIS in Hybrid mode.*** One of the objectives of IRIS' design is to present a unified access to storage as mentioned in Subsection 3.1. Throughout the above tests, we observed that a PFS is sensitive to frequent small data accesses whereas a KVS demonstrates stable performance. We implemented a hybrid mode in IRIS where requests are being redirected according to their size towards the appropriate storage subsystem. Our hypothesis is that each storage system can grant higher I/O performance when faced with a favorable workload. In other words, larger data accesses can leverage the parallelism of PFS, and smaller ones can be placed on the KVS due to the vertical data distribution. We collected and


(a) LAMMPS I/O performance

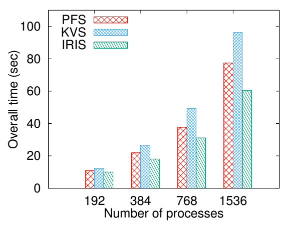

(b) LANL_App1 I/O performance

Figure 5: IRIS in Hybrid mode.

examined the I/O traces of LAMMPS and LANL_App1 to understand the I/O behavior of these two scientific applications. We found that both of them have a repetitive data access pattern. For LAMMPS, each process first writes a few requests in the order of KBs followed by one large request of several MBs. For LANL_App1 the pattern is similar but it performs read operations instead of write. To test our hypothesis, we run both applications on top of a PFS, a KVS, and IRIS in hybrid mode, and measured the time needed to complete all I/O requests. Each process performs 32MB of I/O, with 1536 MPI ranks the total I/O is 48GB. IRIS redirects small data access to the KVS and larger than the threshold to the PFS. For LAMMPS, the threshold was set to 64KB and there were twice as many small data accesses than large ones (i.e., favoring the KVS). For LANL_App1, the threshold was set to 128KB and the ratio of small to large data accesses was roughly 1/4 (i.e., favoring the PFS). In Figure 5, it can be seen that based on the workload with the above ratios, each storage subsystem performs better than the other. IRIS in hybrid mode is able to adapt to different workloads and leverage the appropriate storage solution. In our test, IRIS shows a **40-60%** improvement in performance since it avoids hurtful access patterns trying to leverage the best of both storage systems.

5 RELATED WORK

Object Stores in HPC: Object Stores have been capturing the HPC community's attention for a while now due to the advantages they can offer. Scalability and lower latency for small accesses are the most important benefits of using Object Stores. However, current usage of Object Stores is very limited, mostly as a supplemental component, where PFS remain as the storage backend.

IndexFS [36] and BatchFS [53] have proposed removing the metadata management responsibility, which is proven to be one of the major performance bottlenecks from the PFS, and offloading it to an Object Store. Similarly, FusionFS [52] implements an analogous technique in which it uses a distributed hash table to store and query metadata information. While this approach boosts the performance of PFS by optimizing the metadata workload, it does not introduce Object Stores as a general storage to the scientific community. Applications still use the PFS as a storage solution. On the contrary, MarFS [3] utilizes an Object Store as the storage pool and maintains the POSIX semantics by exposing a file system to the applications. In this case, the Object Store replaces the PFS as the back-end storage solution. While there are benefits in doing so, PFS and Object Store are competing instead of peacefully coexisting and complementing each other. All the above use cases of Object Stores in HPC do not solve what we aim to achieve: a unification of the storage subsystems and the liberation of the interfaces to allow applications to transition from a compute-intensive phase to a data-intensive one. Both storage systems are treated equally inside IRIS and data can be accessed from any data interface.

Object Storage Devices inside PFS: 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 [47] is a new type of distributed file system that promotes the separation of data and metadata management. It does so by replacing the allocation tables, which PFSs usually use, with a pseudo-random data distribution function. With this design, they created APIs that can support both file operations and object operations. However, it does not support the integration of PFS and Object Stores. PanasasFS [31, 49] uses parallel and redundant access to OSD, per-file RAID, distributed metadata management, and other internal technologies to offer a high performance distributed file system. Since they use OSDs, their design offloads some administrative tasks on the disk itself making it run faster for specific workloads. Similarly, OBFS [45] utilizes OSDs internally to create a distributed file system. Again, it is not an integration. It is an enhancement of PFS with some object operations. To use their system, HPC systems need to switch the entire storage installation to their proposed solution, and applications need to be rewritten to be able to use their solution. In contrast, IRIS aims to bridge any existing file system with any object store, and users can utilize both subsystems without modifying their code. With IRIS even CephFS or Panasas can be bridged with any other storage solution by adding the appropriate storage module.

MapReduce on top of PFS: There has been some work about bringing MapReduce to the HPC community. In [40], the authors created a layer on top of PVFS2 to support MapReduce workloads. Its limitations involve limited scalability and while it allows MapReduce applications to access data in PFS, it does not enable the other direction of an MPI application accessing data on an Object Store. It is also specific to PVFS2. In [32] the authors demonstrated the potential of BlobSeer in substituting HDFS to enable efficient MapReduce applications. BlobSeer adopts versioning instead of locking protocols to handle the concurrency issue. Both of the above works assume the existence of one type of file system to support both HPC and MapReduce applications. IRIS is developed to hide the complexity of underlying storage systems, and does not require the modification of existing file systems. Alluxio [25] (formerly known as Tachyon), is a distributed system enabling reliable data sharing at memory speed across cluster computing frameworks. It supports various existing frameworks, such as Spark, MapReduce, and Flink. Alluxio, in a way, is the closest system to our proposal in terms of its goals and objectives to integrate multiple programming environments with several storage pools. However, it relies heavily on main memory which is a valuable resource especially in HPC systems. Additionally, its file support (i.e., MPI-IO, HDF5, pNetCDF etc.) is basic to simple POSIX calls without any of the optimizations IRIS has.

6 CONCLUSIONS AND FUTURE WORK

Parallel file systems have been the defacto storage solution in the HPC community. On the other hand Object Stores have emerged in recent years to serve the increasingly important data-intensive computation paradigm. In this paper we designed and implemented a novel I/O system, named IRIS, which can redirect I/O requests to an integrated storage layer. By abstracting the lower level storage system details, we managed to enable new data paths agnostic to the underlying storage system and offer a truly unified data access layer. The new potential is valuable to application developers who are now free to use any storage interface interchangeably. Experimental evaluations show that, in addition to providing programming convenience and efficiency, IRIS can grant more than 7x higher performance for certain workflows. IRIS aims to bridge the best storage solutions of both worlds (i.e., PFS from HPC and Object Stores from Cloud) and bring us closer to the convergence of the HPC and Cloud ecosystems.

As a future step, we plan to incorporate a prediction model we already have built into IRIS. This model takes as an input an I/O trace file (i.e., a log that describes the I/O behavior of the application), the system configuration along with the application arguments (total number of processes, size of input data), and predicts which storage system between PFS and Object Store will lead to better performance. Second, if the user provides a workflow description in a form of a directed acyclic graph (DAG), then IRIS will be able to adjust to the workflow and utilize the available resources accordingly. We also plan to test IRIS on a *burst buffer* deployment. We believe there is plenty of work left towards a truly agnostic, unified data access model for the exa-scale era to come.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grants no. CCF-1744317, CNS-1526887, and CNS-0751200.

REFERENCES

[1] Amazon Inc. 2017. Amazon S3. (2017). http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.

[2] Apache Software Foundation. 2017. Bigtop software collection. (2017). http://bigtop.apache.org/.

[3] David John Bonnie. 2015. *MarFS-Scalable POSIX on Object File System.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[4] Peter J Braam et al. 2014. The Lustre storage architecture. (2014). ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf.

[5] Gorda Brent. 2015. DAOS: An Architecture for Exascale Storage. (2015). http://storageconference.us/2015/Presentations/Gorda.pdf.

[6] George H Bryan and J Michael Fritsch. 2002. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review* 130, 12 (2002), 2917–2928.

[7] Lawrence Bryan. 2017. The UK JASMIN Environmental Data Commons. (2017). https://wr.informatik.uni-hamburg.de/_media/events/2017/iodc-17-lawerence.pdf.

[8] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. 2009. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE Symposium on.* IEEE, Rome, Italy, 1–11.

[9] Chameleon.org. 2017. Chameleon system. (2017). https://www.chameleoncloud.org/about/chameleon/.

[10] Steve Conway and Chirag Dekate. 2014. *High-Performance Data Analysis: HPC Meets Big Data.* Technical Report. IDC.

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[12] Jack Dongarra et al. 2011. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications* 25, 1 (2011), 3–60. https://doi.org/10.1177/1094342010391989

[13] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. *Acm sigcomm computer communication review* 42, 4 (2012), 25–36.

[14] Mike Folk, Albert Cheng, and Kim Yates. 1999. 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. ACM, Portland, OR, 5–33.

[15] Bryan George. 2017. UCAR CM1 atmospheric simulation. (2017). http://www2.mmm.ucar.edu/people/bryan/cm1/.

[16] Google Inc. 2017. LevelDB. (2017). https://github.com/google/leveldb.

[17] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier, New York, NY.

[18] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. 2009. *The fourth paradigm: data-intensive scientific discovery.* Vol. 1. Microsoft Research, Redmond, WA.

[19] High Performance Data Division Intel® Enterprise Edition for Lustre* Software. 2014. *WHITE PAPER Big Data Meets High Performance Computing.* Technical Report. Intel. https://goo.gl/GLZrRH.

[20] Earl Joseph and Steve Conway. 2014. *IDC Update on How Big Data Is Redefining High Performance Computing.* Technical Report. IDC.

[21] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2017. Enosis: Bridging the Semantic Gap between File-based and Object-based Data Models. In *Data-Intensive Computing in the Clouds(Datacloud'17), 8th International Workshop on.* ACM SIGHPC, Denver, CO.

[22] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2017. Syndesis: Mapping Objects to Files for a Unified Data Access System. In *Many-Task Computing on Clouds, Grids, and Supercomputers(MTAGS'17), 9th International Workshop on.* ACM SIGHPC, Denver, CO.

[23] Anthony Kougkas, Hassan Eslami, Xian-He Sun, Rajeev Thakur, and William Gropp. 2017. Rethinking key–value store for parallel I/O optimization. *The International Journal of High Performance Computing Applications* 31, 4 (2017), 335–356.

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35. https://doi.org/10.1145/1773912.1773922

[25] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing.* ACM, Seattle, WA, 1–15.

[26] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference.* ACM/IEEE, Phoenix, AZ, 39–39.

[27] Los Alamos National Laboratory. 2017. Anonymous Scientific Application. (2017). http://institute.lanl.gov/data/tdata/.

[28] MongoDB Inc. 2017. MongoDB. (2017). https://www.mongodb.com/white-papers.

[29] Montage. 2017. An Astronomical Image Mosaic Engine. (2017). http://montage.ipac.caltech.edu/docs/m101tutorial.html.

[30] Monty, Taylor. 2017. OpenStack Object Storage (swift). (2017). https://launchpad.net/swift.

[31] David Nagle, Denis Serenyi, and Abbie Matthews. 2004. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, Pittsburgh, PA, 53.

[32] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. 2010. BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map-Reduce applications. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, Atlanta, GA, 1–11.

[33] Swapnil Patil and Garth A Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files.. In *FAST*, Vol. 11. ACM/Usenix, San Jose, CA, 13.

[34] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics* 117, 1 (1995), 1–19.

[35] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (2015), 56–68.

[36] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: scaling file system metadata performance with stateless caching and bulk insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, New Orleans, LA, 237–248.

[37] Robert B Ross, Rajeev Thakur, et al. 2000. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th annual Linux Showcase and Conference.* Usenix, Atlanta, GA, 391–430.

[38] Sandia National Laboratories. 2017. LAMMPS. (2017). http://lammps.sandia.gov/.

[39] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Vol. 2. Usenix, Monterey, CA, 231–244.

[40] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J Lang, Garth Gibson, and Robert B Ross. 2011. On the duality of data-intensive file system design: reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, Seattle, WA, 67.

[41] Timothy James Tautges, Corey Ernst, Clint Stimpson, Ray J Meyers, and Karl Merkley. 2004. *MOAB: a mesh-oriented database.* Technical Report. Sandia National Laboratories.

[42] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. 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, 182–189.

[43] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.. In *FAST.* Usenix, San Jose, CA, 119–132.

[44] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. 2005. Providing Tunable Consistency for a Parallel File Store.. In *FAST*, Vol. 5. Usenix, San Francisco, CA, 2–2.

[45] Feng Wang, Scott A Brandt, Ethan L Miller, and Darrell DE Long. 2004. OBFS: A File System for Object-Based Storage Devices.. In *MSST*, Vol. 4. IEEE Computer Society, Adelphi, MD, 283–300.

[46] Weather Research and Forecasting Model. 2017. WRF. (2017). http://www.wrf-model.org/index.php.

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

[48] Brent Welch and Garth A Gibson. 2004. Managing Scalability in Object Storage Systems for HPC Linux Clusters.. In *MSST.* IEEE, Adelphi, Maryland, USA, 433–445.

[49] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System.. In *FAST*, Vol. 8. USENIX Association, San Jose, CA, 1–17.

[50] Yanlong Yin, Surendra Byna, Huaiming Song, Xian-He Sun, and Rajeev Thakur. 2012. Boosting application-specific parallel i/o optimization using IOSIG. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on.* IEEE, Ottawa, Canada, 196–203.

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. 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, San Jose, CA, 2–2.

[52] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. 2014. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Big Data (Big Data), 2014 IEEE International Conference on.* IEEE, Washington DC, 61–70.

[53] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop.* IEEE Press, New Orleans, LA, 1–6.

[54] Shujia Zhou, Bruce H Van Aartsen, and Thomas L Clune. 2008. A lightweight scalable I/O utility for optimizing High-End Computing applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, Miami, FL, USA, 1–7.