

Asynchronous Object Storage with QoS for Scientific and Commercial Big Data

Michael J. Brim, David A. Dillow, Sarp Oral, Bradley W. Settlemyer and Feiyi Wang

Oak Ridge National Laboratory

1 Bethel Valley Rd

Oak Ridge, TN 37831

{brimmj,dillowda,oralhs,settlemyerbw,fwang2}@ornl.gov

ABSTRACT

This paper presents our design for an asynchronous object storage system intended for use in scientific and commercial big data workloads. Use cases from the target workload domains are used to motivate the key abstractions used in the application programming interface (API). The architecture of the Scalable Object Store (SOS), a prototype object storage system that supports the API's facilities, is presented. The SOS serves as a vehicle for future research into scalable and resilient big data object storage. We briefly review our research into providing efficient storage servers capable of providing quality of service (QoS) contracts relevant for big data use cases.

General Terms

HPC Storage, Cloud Storage, Object Storage, Storage QoS

1. INTRODUCTION

Commercial data-intensive computing and simulation science have divergent requirements in several areas, but they share the need for a resilient and scalable infrastructure for extreme-scale I/O. Recently, computing involving extreme-scale I/O has been described as Big Data computing, which is characterized by three key drivers: data volume, data variety, and data velocity. The volume of big data is so massive that distributed storage systems are required and the cost of moving the data to a computation resource is prohibitive. The variety of big data concerns the lack of common structure to the data, which is attributable to many data sources and, to a lesser degree, data structures that evolve over time. This variety prevents the direct use of relational database technology that requires static schemas describing the structure of the data. The velocity of big data refers to an inability to analyze data at the rate at which it is produced and the compounding nature of new data generated during analysis.

Commercial big data processing and analysis systems are

in widespread use. These systems include: (1) map-reduce infrastructures [8, 27] that provide fault-tolerant, throughput-oriented batch processing and analysis of large datasets, (2) key-value stores [9] supporting low-latency storage and retrieval of relatively small data, and (3) multi-dimensional maps [6, 16] that extend the capabilities of key-value stores to support fast access to many named values (or columns) per key. Ideally, these mature commercial-grade systems could be directly leveraged for storage and analysis of scientific big data. The foremost barrier to adoption is that scientific data is used in a tightly-coupled, computationally intensive manner that is at odds with the loosely-coupled, data intensive processing provided by commercial systems.

Scientific data generation and analysis involves distributed processes that coordinate to advance a simulation according to a stateful scientific model. Such coordinated, stateful data evolution is inefficient to perform in systems such as Hadoop. Parallel and distributed file systems based on object storage [29, 31], have cleared a path to extreme-scale I/O by addressing scalability barriers related to metadata management, distributed locking, and incremental provisioning. Improvements have also been made in file system resilience through object replication. Still, these object-based file systems are often designed to perform well for a specific class of workload, either scientific or commercial, and thus are not generally applicable. Current systems are also plagued by performance variability due to competing workloads, which makes it difficult to optimize I/O in applications and limits the overall utilization of the storage system.

Our research goal is to design and evaluate object storage technologies that address the issues of scalability, resiliency, and performance for multiple big data workloads. To this end, we have identified four research objectives:

1. Examine how an asynchronous object I/O model enables performance and utilization benefits when scheduling access to storage resources,
2. Develop techniques for ensuring storage quality of service (QoS) that eliminates performance variability under competing workloads,
3. Evaluate object resilience in terms of consistency and performance on multi-tiered storage systems containing heterogeneous storage media, and
4. Develop in-transit object data transformation and processing capabilities to support tightly-coupled and loosely-coupled analysis.

(c) 2013 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the national government of United States. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. PDSW13 November 18, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2305-9/13/11 ...\$15.00.
<http://dx.doi.org/2538542.2538565>.

In this paper we describe our initial efforts to address the first two research objectives. Section 2 describes the use cases in scientific and commercial big data that motivate our object storage system design and the benefits of slotted QoS in reducing performance degradation due to workload interference. Section 3 introduces the Scalable Object Store (SOS) and its corresponding API (libsos), and describes our prototype implementation of the SOS. In Section 4 we briefly discuss our future research. Finally, Section 5 differentiates our research from prior work in scientific and commercial big data systems.

2. BIG DATA USE CASES

Our design for an asynchronous object storage system is motivated by the interaction between two fundamental use cases: scientific application checkpointing and statistical inferential data analysis. No existing storage system is well suited to accelerate both use cases. Our designs go beyond accelerating each use case in isolation, rather our goal is to support both Big Data use cases from within a single storage system instance. A primary problem preventing the use of a single storage system for simultaneously supporting intensive read and write scenarios is the effects of storage device interference. Even well-formed checkpointing and analysis applications will generate substantial performance degradation due to the disk heads seeking between unallocated disk regions to support writes to previously allocated disk regions to support reads. Here we briefly describe the requirements imposed by each application use case, and then present experimental results showing the performance effects of interference.

2.1 Scientific Application Checkpoints

Due to the sheer number of components in current petascale supercomputers such as Titan [21] and Sequoia [17], the expected mean-time-to-interruption (MTTI) for large-scale applications is less than one day. For long running simulations, application checkpoints are necessary to ensure progress in the presence of system failures and interruptions. Large-scale application checkpoint workloads consist processes concurrently writing data to the storage system periodically and occasionally reading checkpoint data to restart after interruptions. We estimate the maximum practical size of checkpoint data as 75% of the physical memory size of compute nodes used by the application. On systems like Titan, which contains over 18,600 compute nodes each with 32 gigabytes of memory, an application could write up to 450 terabytes of checkpoint data. Current petascale applications store checkpoints on the order of every four to eight hours. For exascale systems, it is expected that failure rates will require applications to checkpoint as often as every hour.

Modern scientific applications rely upon checkpoint libraries such as parallel NetCDF [18] and HDF5 [10] that provide portable and hierarchical organization of complex data while hiding the details of parallel I/O, as compared to older applications that use application-specific checkpoint data formats and custom distribution of checkpoint I/O among processes. Both netCDF and HDF5 use similar abstractions for managing complex data. These abstractions include datasets, groups, and attributes. Datasets, known as variables in netCDF, are multi-dimensional arrays of (possibly complex) data types. Groups are metadata abstractions for associating datasets and other groups. Attributes are key-value

pairs that provide metadata to annotate a dataset or group.

2.2 Statistical Data Analytics

Commercial Cloud Storage (i.e., SaaS, PaaS, and IaaS) storage requirements can vary greatly between customers, with application-dependent I/O patterns and data sizes ranging from a few gigabytes to hundreds of terabytes. Amazon Simple Storage Service (S3) [3], Google Cloud Storage [13], and Microsoft Windows Azure Storage [19] are commonly used virtual storage platforms. In addition to virtual storage for cloud customers, data used to manage the business operations of cloud and e-commerce providers may also reside in cloud storage. Commercial clients mine the large quantities of data in clouds using data processing techniques based on tools such as Google’s BigQuery [12] or Amazon’s DynamoDB [9] and Elastic MapReduce [2]. The analysis methods rely on statistical inference (rather than statistical description) to identify interesting phenomena within the data. These tools are capable of providing much better data mining throughput by limiting data movement, and instead serializing and transporting the code such that bottom-up data analysis is typically performed at or near the individual storage servers. Inferential analysis typically involves reading huge amounts of data, on the order of terabytes or petabytes, although the individual datums being analyzed may be comparatively small. Following each analysis phase, a data reduction is performed that summarizes the results for that piece of data analysis.

2.3 Experimental Measurement of Interference

Our construction of a distributed QoS reservation scheme is still rudimentary, however we have performed several experiments demonstrating the destructive interference caused by storage servers simultaneously servicing read and write requests. We configured a disk profile tool, XDD [14] to generate file read accesses and then added multiple file writers to simulate the costs of performing scientific checkpoints while a data analysis job is ongoing. We then leveraged XDD’s lockstep command mode to service each of the file access streams in isolation. We divided each second into 16 access slots, and the files were serviced only during designated slots in round robin order. In every configuration we are using 4 I/O threads per file, direct I/O to bypass Linux block caching (common on systems with fast attached storage), 4MiB request sizes, and file sizes of at least 64GiB of randomly generated data.

In Figure 1 we demonstrate the effects of slotted access using a single SUSE 11 Linux server connected via Fibrechannel to an Infortrend EonStore S16F-R1430 (configured at RAID level 5 with 64KiB stripes). The storage network fabric was provided by a Brocade Silksworm 4100 switch. The local file system used on the storage server was XFS, configured to match the storage hardware. Along the X-axis we fix a single reading client (4 threads) and then add writers (4 threads per file). We feel there is significant opportunity for performance improvement via tuning. To that end we included our best tuned single file read and write performance numbers; however, until we include the network components of our storage server for our targeted use cases we avoided speculative storage optimizations [25].

In Figure 2 we show the data collected for a high end Hewlett-Packard DL585 G7 storage server using a local 5TB FusionIO PCIe Flash device. The host provides 48 cores,

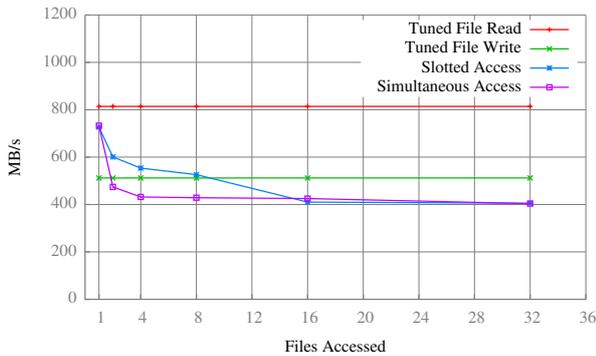


Figure 1: QoS Access Schemes with a Disk Array

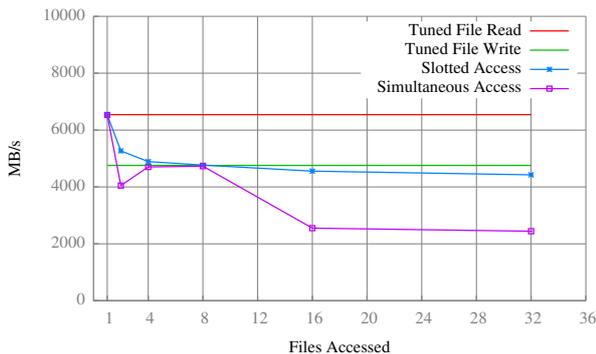


Figure 2: QoS Access Schemes for PCIe Flash

384GB of memory, runs RHEL 6, and we used a local XFS file system to access the storage device. We also used the *numactl* tool to pin XDD worker threads to the NUMA node closest to the storage device. Because the FLASH device imposed no seek penalty, as we added simultaneous access clients, performance decreased primarily due to excess I/O threads. By dividing each second of service time into approximately 16 slots, our QoS access scheme was able to generate higher, more stable performance by controlling the number of I/O threads. We have included our best tuned single file access for this device, but have not employed any advanced tuning in our multiple file access runs.

The experiment results demonstrate that on modern storage media, whether traditional disk drives or high-end solid state devices, simultaneous access to multiple files (including a mix of read and write jobs) reduces storage server performance, even when the access size is large. Further, a slotted access scheme offers the opportunity to maintain high levels of I/O performance. Our slotting scheme goes further and demonstrates that an I/O slotting scheme may successfully use multiple I/O threads, which improves disk access performance but also increases the slot switching time as it waits for multiple threads to complete. There is no technical reason that requires waiting for all threads to complete their I/O before servicing the next slot; however we felt that this level of synchronization presented the most conservative view of the performance data.

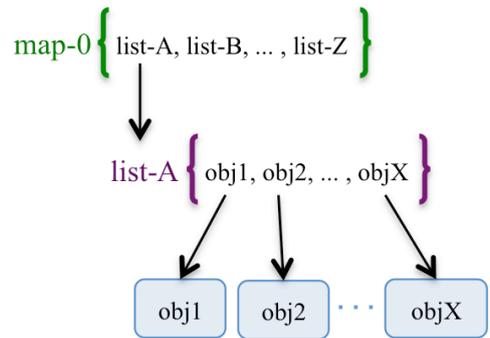


Figure 3: Object Collection Namespace

3. SCALABLE OBJECT STORAGE

In this section we introduce the Scalable Object Store (SOS). A fundamental goal of the object service is to provide all storage service interactions asynchronously. An earlier asynchronous storage system, the lightweight file system, leveraged asynchrony in order to overlap computation and storage system access [22]. For our targeted use cases, checkpointing and data mining, the system memory constraints typically do not allow enough available excess memory to overlap computation and storage access. When the application processes require all of the system memory, there simply is no buffer space available to perform concurrent data modification; this is particularly true for scientific applications where the next memory state is dependent upon the current memory state. Instead, we leverage client asynchrony to allow better storage resource scheduling at the servers [15, 24], allowing the storage servers to initiate and control storage system access. Further, the semantics of object-based storage access remove the potential requirement that distributed clients coordinate storage system access to eliminate inter-application interference.

3.1 Object Storage Abstractions

Our design for an object storage system that supports Big Data relies on three key abstractions: objects, object collections, and object lockers. We define an object as a named data buffer. Objects are expected to range in size from small (e.g., a few kilobytes to store the value associated with some key) to very large (e.g., a multi-gigabyte dataset). The amount of data that can be stored in a single object is unbounded by design, though practical implementations may impose limits due to storage device capabilities. Objects may contain holes, which permits the storage of sparse data.

Organization of objects is provided by two forms of object collections: lists and maps. An object list is a named collection of objects. An object map is a named associative collection of object lists, where each list is associated with its unique name. Both forms of object collections support direct lookup by name and provide iterators for enumerating the collection. As shown in Figure 3, object collections provide a three-level name space. Additional levels of hierarchy can be achieved by storing the names of other objects, lists, or maps in an object.

The final and most important abstraction in our object

storage design is the object locker, a dynamically-provisioned, named allocation of storage resources that provides three critical services: (1) an isolated name space for objects and object collections, (2) data resiliency via customizable object replication, and (3) storage quality of service (QoS). Object lockers provide name spaces that are private to a particular user or group; only applications run by the user/group can store and retrieve objects and manage object collections. Private name spaces for file systems [11, 20, 23] have long been seen as a way to simplify user access to data and improve productivity, and we believe these benefits hold true for object storage. For cloud storage platforms that service many users, name space isolation is a mandatory requirement to meet privacy expectations.

Data resiliency in our object storage system is provided in terms of replicating individual objects. Replication parameters, such as number of replicas and object data striping, are specified for all objects in a locker when the locker is created. Object placement is also determined by the locker rather than the client, a natural extension of the intrinsic asynchrony of the proposed interfaces. Object lockers also provide storage QoS guarantees that are negotiated when a locker is created. Storage QoS, broadly defined, concerns the management of contracts negotiated between clients and shared storage servers. These contracts specify expected levels of I/O performance and availability of resources, and are commonly known as service-level agreements (SLAs).

3.2 libsos API Types

Each of the primary abstractions in the API (i.e., storage lockers, objects, object lists, and object maps) is represented by an opaque handle struct. A handle contains a field `id` of type `sos_id_t` that serves to uniquely identify the target to the libsos client. Note that the same target may have a different identifier (`id`) when accessed by a different client. Using client-unique ids avoids the common scalability barriers associated with distributed consensus in the management of globally unique ids. The object list and map collections also provide iterators. Synchronous request functions return an integer enumeration that indicates success or a failure condition. Asynchronous request functions return a globally unique id of type `sos_req_t`. Each client maintains a counter to generate local request ids. The client's unique id, a combination of host and process ids, is then prepended to the counter value to derive the global request id.

3.3 libsos API Functions

The first action taken by a client is to connect to the SOS (a companion function is used to cease interactions with the SOS). Once a connection has been made, a client proceeds to create or discover object lockers. Because all objects and object collections are stored in lockers, a valid locker handle must be obtained that can be passed to operations on objects or collections. Lockers are created using the `sos_locker_create` function and destroyed using `sos_locker_destroy`. The latter function implicitly deletes all objects and collections stored in the locker. Existing lockers can be found using `sos_locker_find`, which searches for a locker having a specified name, or by listing all existing lockers with `sos_locker_list`. The latter function lists only lockers for which the client has access permissions. Currently the functions for managing lockers are defined as synchronous requests. Our intention is to enable asynchronous locker

requests in the future.

Given a valid locker handle, clients can begin operating on objects and object collections in that locker. The API provides optimistic retrieval functions that search for a target object or collection by name and either return the existing target or create a new target with that name. Optimistic retrieval allows groups of clients operating on the same target to concurrently submit asynchronous get requests without coordinated locking. For a new target, the first request processed by the SOS will create the target and subsequent get requests will return the new target. To query existing targets without implicit creation, the API also provides lookup functions that simply check for existence. Objects can be explicitly created using `sos_obj_create`, which is passed an initial data buffer to use as the object's contents. Destruction of objects and collections is provided by the functions `sos_obj_destroy`, `sos_list_destroy`, and `sos_map_destroy`. Objects provide asynchronous I/O operations including `sos_obj_read`, `sos_obj_write`, `sos_obj_append`, and `sos_obj_truncate`. In addition to the standard data buffer and byte count, the read and write operations require an explicit offset, since object I/O is not byte-stream oriented. The size of an object is defined as the offset of the last valid byte, and can be retrieved using `sos_obj_get_size`.

Object collections are managed using asynchronous insert and remove operations. Object lists permit multiple objects to be inserted or removed at once using `sos_list_insert` and `sos_list_remove`. Conversely, the object map functions `sos_map_insert` and `sos_map_remove` operate on a single object list associated with a given key. Clients are informed of asynchronous request completion in one of two ways. A client can poll for the status of a particular request or can register a callback function that will be executed when the request completes. Additionally, clients may use the function `sos_req_group` to generate a `sos_req_t` that represents a group of outstanding requests. This group request id can then be polled to wait for completion of all requests in the group. Outstanding requests can be explicitly canceled using `sos_req_cancel`. Cancellation does not interrupt requests that are actively being serviced, which prevents partial I/O operations by ensuring a canceled request is either never serviced or runs to completion. Once a request completes or is canceled, a client must finalize the asynchronous request by calling `sos_req_retire`. When passed a group `sos_req_t`, `sos_req_retire` will finalize all requests in the group. Attempts to disconnect from the SOS will fail until all requests have been retired.

3.4 SOS Architecture

We have developed a prototype version of the SOS in support of the libsos API. This prototype serves two purposes. First, it allows for direct experimentation with a variety of scientific and commercial big data workloads, which helps to ensure the abstractions provided by the API are suitable in terms of both usability and scalability. Second, it provides a vehicle for further research in scalable, resilient, and high-performance techniques for big data object storage, particularly in our current focus areas of storage QoS and in-situ data processing.

The SOS prototype architecture is shown in Figure 4. Client applications are linked with the libsos library. The library manages interactions with one or more storage lockers in the SOS. Lockers are virtual allocations of SOS server

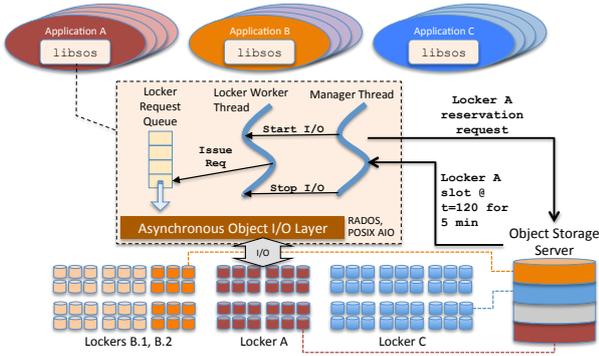


Figure 4: Scalable Object Storage Architecture

resources, and thus each server may manage data from one or more lockers. For each locker in active use, the client maintains a locker request queue (LRQ) that holds the outstanding asynchronous requests for objects and collections stored in the locker. Object collections are stored as objects that contain appropriate meta-data (i.e., the names of collection members, etc.); we refer to these objects as meta-objects. Associated with each LRQ is a worker thread that is responsible for issuing and notifying completion of requests. Worker threads within a client are scheduled by a manager thread. Currently, the manager thread uses round-robin scheduling of workers for active lockers. In our future work on QoS, the manager thread will be updated to receive instructions from the SOS as to when each worker thread should be activated based on the negotiated locker service level.

While a worker thread is active, it issues the requests in its LRQ and monitors their progress. Requests are issued by calling methods in a generic interface layer for asynchronous object I/O. In the current prototype, we have two implementations of the generic object I/O interface. The first implementation leverages the RADOS object storage system and API that is developed as part of the Ceph distributed file system. The second implementation uses the POSIX asynchronous I/O interface for interactions with existing parallel or distributed file systems.

Due to the asynchronous API provided by SOS, storage servers are able to offer dedicated access for an interval (called a slot) to active lockers. Each server services multiple lockers, but only one locker during each interval. When a client requests the creation of an object locker, the storage servers use the request as a reservation request to achieve a quality of service for the specified time interval. For example, a scientific checkpoint application that wishes to checkpoint once an hour will create a new object locker per checkpoint, which will result in a storage reservation for that checkpoint. Within that locker request the application will describe the size and desired deadline time of the checkpoint data, and the storage server will generate a locker allocated with sufficient time slots on the storage servers to perform the requested service.

4. CONCLUSION

The libsos API has been designed to support asynchronous operation from the outset. To support this asynchrony, we

have developed two novel storage system abstractions for improved isolation: object collections and lockers. Object collections offer a limited hierarchical name space that is useful for organizing and associating data objects. Lockers provide redundancy and isolation for dynamically provisioned storage resources. With these basic abstractions we are able to employ a distributed QoS scheme based on reservations and packet-based media access control (which is similar to using time slots with a distributed coordination function).

As we continue developing our prototype we are focusing on supporting additional scientific and commercial cloud usage scenarios. In the scientific realm, we are interested in accelerating the I/O patterns associated with data visualization and distributed application performance traces. The small, unaligned accesses that frequently result under both use cases are not good candidates for locality-based prefetching, and are instead most likely to benefit from a “thinner” I/O system that provides efficient access to storage. In the commercial storage scenarios we are most interested in improving the dynamism in our existing system design. Cloud providers have created scalable key-value datastores and multi-dimensional maps to serve as highly-available storage for mission-critical application data and customer data. Compared to files in virtual storage systems, the datastore entries per key are relatively small, typically less than one megabyte. This challenging scenario again results in small storage system accesses that may generate large amounts of interference with competing workloads.

5. RELATED WORK

Several distributed and parallel file systems [5, 31], have adopted an underlying object storage model while still exposing a POSIX file system to clients. In these systems, objects are used to store contiguous portions of a file’s linear byte stream, but clients are given no direct methods for accessing and managing objects. In contrast, systems such as Intel’s Distributed Application Object Storage (DAOS) [4], Ceph’s RADOS [30], Sirocco [7], and Ursa Minor [1] eschew the traditional file interface in favor of direct operations on objects. The purely object-based SOS system and API is most similar to and draws inspiration from both DAOS and RADOS. DAOS containers and RADOS pools are very similar in concept to our object storage lockers, as both provide isolated name spaces and support configurable object redundancy. Similar to DAOS, all operations on objects in SOS are asynchronous. One of the key differences in our concept of storage lockers is support for QoS to guarantee performance among competing workloads. Initial work to support QoS in Ceph [32] focused on designing a new object-aware disk scheduler for use in object storage servers. Our QoS scheme is derived from existing work [28, 26, 33] however we add a distributed reservation and coordination scheme to support distributed clients.

6. ACKNOWLEDGMENTS

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy. This work was also supported by the Department of Defense (DoD) and used resources at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory and supported by DoD.

7. REFERENCES

- [1] M. Abd-El-Malek, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. *Ursa minor: versatile cluster-based storage*. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.
- [2] Amazon Web Services, Inc. Amazon Elastic MapReduce (Amazon EMR). <http://aws.amazon.com/elasticmapreduce/>, 2013.
- [3] Amazon Web Services, Inc. Amazon S3, cloud computing storage for files, images, videos. <http://aws.amazon.com/s3/>, 2013.
- [4] E. Barton. Fast forward i/o and storage. <http://www.pdsw.org/pdsw12/slides/keynote-FF-I0-Storage.pdf>, 2012.
- [5] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. Pvfs: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, ALS'00, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.
- [6] 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 Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
- [7] M. Curry, R. Klundt, and H. Ward. Using the Sirocco file system for high-bandwidth checkpoints. Technical Report Technical Report SAND2012-1087, Sandia National Laboratory, February 2012.
- [8] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, Jan. 2010.
- [9] 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. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.
- [10] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.
- [11] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The cedar file system. *Commun. ACM*, 31(3):288–298, Mar. 1988.
- [12] Google, Inc. Google BigQuery - cloud platform. <https://cloud.google.com/products/big-query/>, retrieved June 2013.
- [13] Google, Inc. Google Cloud Storage - cloud platform. <https://cloud.google.com/products/cloud-storage/>, retrieved June 2013.
- [14] I/O Performance, Inc. Xdd: The extreme dd toolset. <https://github.com/bws/xdx>, 2013.
- [15] D. Kotz. Disk-directed i/o for mimd multiprocessors. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.
- [16] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
- [17] Lawrence Livermore National Laboratory. ASC Sequoia. https://asc.llnl.gov/computing_resources/sequoia/, October 2012.
- [18] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.
- [19] Microsoft, Inc. Storage - windows azure service management. <http://www.windowsazure.com/en-us/manage/services/storage/>, 2013.
- [20] B. C. Neuman. The Prospero file system: A global file system based on the virtual model. *Computing Systems*, 5:407–432, 1992.
- [21] Oak Ridge National Laboratory. Introducing Titan - the world's no. 1 open science supercomputer. <http://www.olcf.ornl.gov/titan/>, 2012.
- [22] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.
- [23] H. C. Rao and L. L. Peterson. Accessing files in an internet: The Jade file system. *IEEE Trans. Softw. Eng.*, 19(6):613–624, June 1993.
- [24] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.
- [25] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. M. Ruwart. A technique for moving large data sets over high-performance long distance networks. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society.
- [26] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.
- [27] The Apache Software Foundation. Apache Hadoop. <http://hadoop.apache.org>, 2013.
- [28] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.
- [29] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance

- distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
- [30] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM.
- [31] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.
- [32] J. C. Wu and S. A. Brandt. Providing quality of service support in object-based file system. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, MSST '07, pages 157–170, Washington, DC, USA, 2007. IEEE Computer Society.
- [33] X. Zhang, K. Davis, and S. Jiang. QoS support for end users of i/o-intensive applications using shared storage systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 18:1–18:12, New York, NY, USA, 2011. ACM.