Toward a Unified Object Storage Foundation for Scalable Storage Systems

Cengiz Karakoyunlu
University of Connecticut
Storrs, CT, USA
cengiz.k@uconn.edu

Dries Kimpe, Philip Carns
Kevin Harms, Robert Ross
Argonne National Laboratory
Argonne, IL, USA
{dkimpe,carns,rross}@mcs.anl.gov
harms@alcf.anl.gov

Lee Ward
Sandia National Laboratories
Albuquerque, NM, USA
lee@sandia.gov

***Abstract*—Distributed object-based storage models are an increasingly popular alternative to traditional block-based or file-based storage abstractions in large-scale storage systems. Object-based storage models store and access data in discrete, byte-addressable containers to simplify data management and cleanly decouple storage systems from underlying hardware resources. Although many large-scale storage systems share common goals of performance, scalability, and fault tolerance, their underlying object storage models are typically tailored to specific use cases and semantics, making it difficult to reuse them in other environments and leading to unnecessary fragmentation of datacenter storage facilities. In this paper, we investigate a number of popular data models used in cloud storage, big data, and high-performance computing (HPC) storage and describe the unique features that distinguish them. We then describe three representative use cases-a POSIX file system name space, a column-oriented key/value database, and an HPC application checkpoint-and investigate the storage functionality they require. We also describe our proposed data model and show how our approach provides a unified solution for the previously described use cases.**

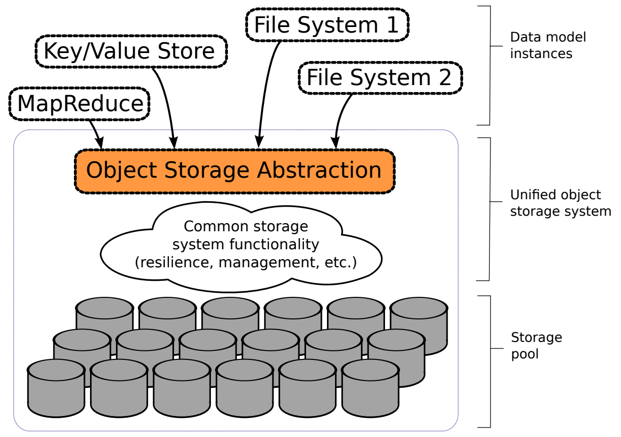
File System 1
Key/Value Store
File System 2
MapReduce
Object Storage Abstraction
Common storage system functionality (resilience, management, etc.)
Data model instances
Unified object storage system
Storage pool

Fig. 1: Example deployment scenario in which big data, cloud storage, and HPC data models share the same storage pool via a unified object storage abstraction.

I. Introduction

Storage technology has improved rapidly, particularly in terms of storage density; but storage throughput has not kept pace with advances in computational performance. This trend has led to increased demand for large-scale storage systems that aggregate and coordinate many storage devices, in turn driving the need for better abstractions to manage those storage devices. Object-based storage [1], [2] has emerged as a strong competitor for the block-based model, quickly becoming a popular underlying model for referencing and accessing data distributed over large numbers of storage devices in these systems. An *object* is an ordered logical collection of bytes with a numerical identifier. Objects consist of data, attributes describing the object, such as QoS attribute, and device-managed metadata, such as security information [1], [3]. Objects have variable sizes and can be used to store any kind of data. The object storage model abstracts away a variety of resource-specific management tasks, such as block allocation, space management, and various forms of atomicity. However, it still allows considerable flexibility for a variety of higher-level data models to be built atop it. Although object models were originally envisioned as a device-level interface [2], today's large-scale storage systems more commonly repurpose the object model as a software interface atop a variety of storage substrates [4], [5], [6], [7].

Although several object-based storage models have been implemented and used as the basis for the popular storage and file systems [8], [9], [7], [3], existing object-based storage models are typically tailored to a particular use case or data model, making them difficult to reuse in other contexts. This situation also makes it difficult to share a common storage pool for different big data, cloud storage, or HPC storage tasks, increasing management overhead and adding complexity to the task of storage provisioning for facilities with diverse storage needs. Ideally each data model would coexist using a shared object storage foundation as shown in Figure 1.

To address this problem, we first identify some of the most popular large-scale data models in use today. The following list divides them into four categories with representative examples:

- **Parallel file systems**: Lustre [10], GPFS [11], Panasas [3], PVFS [12], Ceph [8]
- **Cloud object storage**: Amazon S3 [13], Swift [14], Rados Gateway [6], [15]

- **MapReduce**: Google File System (GFS) [16], Hadoop HDFS [17]
- **Key/value stores**: Dynamo [18], Redis [19], Hyperdex [20], Cassandra [21], HBase [22], BigQuery [23]

Note that these data models aren't necessarily mutually exclusive. For example, several parallel file systems have been extended to support MapReduce workloads. We will refer to these classifications in the remainder of the paper for clarity, however, in order to simplify the discussion of use cases and requirements that are shared across groups of storage systems.

TABLE 1: Requirements for popular scalable storage data models

	Shared Requirements				Distinguishing Requirements				
	High Performance	Scalability	Fault Tolerance	Concurrent Read Access	Concurrent Write Access	Synchronization Primitives	Atomicity	Compute/Storage Locality	Record Oriented Access
Parallel File System	✓	✓	✓	✓	✓	✓	✓		
Cloud Object Storage	✓	✓	✓	✓			✓		
MapReduce	✓	✓	✓	✓				✓	
Key/Value Store	✓	✓	✓	✓	✓	✓	✓		✓

Table 1 breaks down the large-scale storage data models in terms of core requirements. The first four are general requirements that are shared across all such data models. Concurrent write access refers to the ability to have multiple processes write simultaneously to the same file, object, or database. Synchronization primitives are features such as file locking [24] or conditional operations [25] that allow multiple processes to explicitly coordinate concurrent writes. Atomicity is the ability to modify data such that the write is applied in its entirety or not at all. The granularity of atomicity can vary widely across data models. For example, cloud object storage systems may offer object-granular atomicity, key/value stores may offer per-key granularity, and file systems may not offer atomicity at all except in the directory name space. Locality allows applications to execute on server nodes with local copies of data relevant to computation. Record-oriented access is needed for storage systems that refer to units of data in terms of opaque keys rather than ranges of bytes.

In this paper we propose a new object-based storage API, known as the Advanced Storage Group (ASG) interface, that seeks to unify features necessary to support the data models outlined above without compromising usability or limiting implementation flexibility. The contributions of this paper are as follows:

- Identify the requirements that differentiate four key large-scale data storage models
- Propose a new object storage API that unifies the features necessary to meet those requirements
- Present a set of case studies that evaluate how the proposed API would be used as a foundation for a diverse set of storage constructs

The rest of the paper is organized as follows. Section II presents example drivers for this work. Section III describes the proposed ASG API and how it can be used to implement the use cases given in Section II. Section IV shows how our approach presents a unified solution for the use cases described in Section II. Section V reviews related work in object-based storage systems. Section VI summarizes our findings and presents potential avenues for future work.

II. Motivation

In this section, we discuss a number of common storage uses, that serve as one of the drivers for our work.

A. Implementing POSIX Directories

In a POSIX file system, data files are located by looking them up by name in a directory. POSIX directories have the following properties. First, creating or removing a file (or subdirectory) in a directory is an atomic operation, and duplicate entries are not allowed. If multiple processes try to create or remove the same entry at the same time, exactly one of them will succeed. Second, a directory entry has associated metadata, for example, the last access time or the size. In addition to create and remove, three other operations are possible on a directory: opening (lookup) of a name, updating the metadata associated with a name, and renaming an entry to a new name. These operations are atomic as well. As soon as an update completes, all processes in the system see the updated information; at any time before that, the old information is preserved. At no point will a lookup return a blend of old and new metadata. The same is true for rename. Either the old name will be in the directory, or the new name, but not both.

Existing object storage models typically do not directly support directory primitives, nor do they support operations designed to implement structures and synchronization required for implementing a POSIX directory. Consequently, most data models requiring directory like indexes implement this functionality by using additional services (for example, metadata servers), using the object storage only for the actual file data.

B. Column-Oriented Key/Value Store

A column-oriented database differs from a traditional database in that records are stored in column order rather than row order, as shown in Table 2. Data for a database entry is stored in a column, each row stores the same data field of a database entry, and shards (horizontal partitions of a database) represent a collection of rows. This organization improves the performance of analysis-oriented workloads in which ad hoc queries are performed over all values in a column. A column-oriented database will generate large, contiguous disk access patterns in this case because there is no need to skip over interleaved column data for each entry. In addition, each row typically has a large number of columns, and not every row needs to have the same set of columns. Most column-oriented databases allow the creation of new columns at any time, simply by writing to them.

Existing object storage models generally do not support column-oriented databases; since record functionality is missing and applications are forced to manage the storage space.

TABLE 2: Example organization of a column-oriented key value store.

		Column 0	Column 1	Column 2	Column 3
Shard 1	Row 0	Alice	Bob	Brad	Charles
	Row 1	Smith			Springfield
Shard 2	Row 0	111-1111		144-1144	321-4321

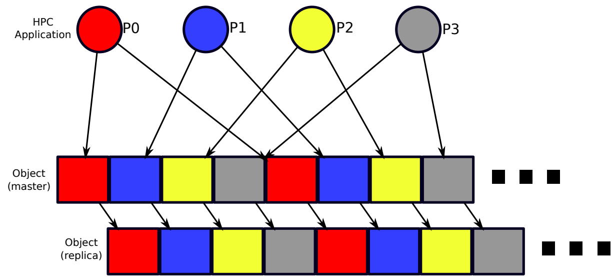

Fig. 2: Example of an HPC application writing in parallel to a replicated object.

As an example, in T10 [2], if each attribute represents a cell of the column-oriented database, grouping certain attributes to represent the rows and columns of a column-oriented database is not an easy task, since mapping attributes to rows and columns and keeping track of mapping information are challenging.

C. HPC Application Checkpoint

HPC application workloads are characterized by bursty, highly concurrent, write-intensive I/O patterns [26]. In particular, many scientific simulations periodically write checkpoint data for application resilience. In these scenarios, all application processes typically write simultaneously to the same shared data set, as shown in Figure 2. Although the application processes are coordinated and do not generally write to overlapping byte ranges in the file, the access patterns may be highly interleaved and are not necessarily block aligned. Optimizations such as two-phase I/O [27] and I/O forwarding [28] can be used to mitigate the level of concurrency observed by the storage system, but data must still be written by many processes in order to leverage enough I/O paths to meet bandwidth requirements.

Metadata overhead and high concurrency are the key challanges for this type of concurrent write access pattern. Existing data models tried $N-N$ and $N-1$ checkpointing strategies [29]. $N-N$ checkpointing is the case where each process writes to a separate checkpoint file, and $N-1$ checkpointing is the case where all the processes write to a shared file. Both checkpoint patterns pose challenges. $N-1$ checkpointing suffers from limited bandwidth, since all processes are trying to concurrently write to the same file. On the other hand, $N-N$ checkpointing creates a lot of files, increasing the metadata overhead.

III. Proposed API and Storage Model

In this section we describe the ASG storage model, its fundamental building blocks and basic operations.

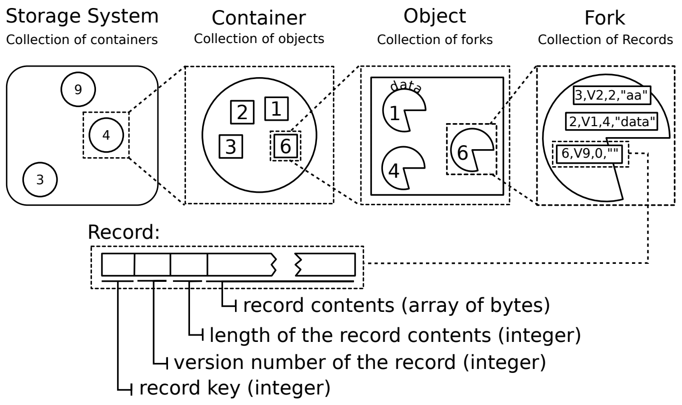

Fig. 3: Architecture of the ASG Storage Model

A. Architecture

The main architecture of the ASG storage model is shown in Figure 3. The core concepts are described as follows.

- The basic building block of the ASG storage model is a *record*. Each record consists of a key, a version number, data, and length of data. The key, version number, and length of data are represented with integers, whereas data is an array of bytes of variable length. Key is the numerical identifier of a record. Version numbers are used to order the write operations to a record. The data field can be empty; a record at its *initial condition* will have version number *zero* and will contain no data. The records are not explicitly created in the ASG storage model; they already exist in the system at their initial conditions before they are manipulated by the ASG storage model operations.
- A *fork* is a collection of records forming a distinct namespace for the records it contains. Each fork is identified by an integer. Forks allow related collections of data (for example, indexes, metadata, or header inforation) to be stored alongside the primary data stream for an object with the same security, locality, and atomicity [30].
- An *object* is a collection of forks. It provides a distinct namespace for the forks it contains. Each object is identified by an integer.
- A *container* is a collection of objects. It provides a distinct namespace for the objects it contains. Each container is identified by an integer. Containers partition the storage system into logical units with different security domains; as an example each container could contain a distinct file system.

The record, fork, and object identifiers in the ASG storage model are not global. For example, two different containers can have objects with the same identifiers. Similarly, two different objects or forks can have forks or records with the same identifiers. There can be a maximum of 2^{64} objects in a container, 2^{64} forks in an object, and 2^{64} records in a fork in the ASG storage model, giving many options to translate the applications described in Section II to the storage model.

B. Operations

In this section we describe the ASG storage model operations a client can use to interact with the storage system. All of the ASG storage model operations are atomic; meaning that they either fully complete or have no effect at all, which satisfies the atomicity requirement in Table 1.

1) write: The *write* operation stores data in a sequential range of records. The input arguments to this function are location information (container, object, fork, and starting record identifiers), local buffer that stores the data to be written, number of records to be modified (range of the write operation), conditional flag, and user-specified version number. The conditional flag can be set to one of the following four values in order to control the semantics of the write operation with respect to per-record version numbers:

- *NONE*: Write should succeed without checking any version number.
- *ALL*: Write should succeed only if the user-specified version number is greater than all the version numbers in the user-specified range.
- *UNTIL*: Write should *continue* until it comes across a record that has a version number greater than or equal to the user-specified version number.
- *AUTO*: In this case, the user-specified version number can be ignored. The biggest version number existing in the user-specified range is found and incremented; and the new data is written with this incremented version number.

The input data of the write operation is divided into the same number of chunks as the number of records in the user-specified range. The same length of data is written to each record in this range.

When the write operation is completed successfully, it returns the size of the written data and the newly assigned version number.

2) read: The *read* operation retrieves data from a sequential range of records. The input arguments to the read operation are location information (container, object, fork, and starting record identifiers), local buffer to store the read data, number of records to be read, conditional flag, and user-specified version number. The range in the read operation is identical to the range defined in the write operation. The conditional flag of the read operation can be set to one of the following three values:

- *NONE*: Read should succeed without checking any version number or conditional flags.
- *ALL*: Read should succeed only if the user-specified version number is greater than all the version numbers existing in the user-specified range.
- *UNTIL*: Read should *continue* until it comes across a record that has a version number greater than or equal to the user-specified version number.

When the read operation is completed successfully, it returns the number of the records read in addition to the version number of these records.

3) reset: The *reset* operation returns an entity (container, object, fork, or record) back to its *initial condition*. In an entity at its original condition, all the records will have version number *zero* and will contain no data. The reset operation can work on any entity of the ASG storage model. The reset operation takes in the identifier information of the entity to be reset as an input argument, and it also supports conditional execution based on the existing version number and given conditional flag. The conditional flags that can be used with the reset operation are the same as the conditional flags used in the read operation. When the reset operation is completed successfully, it returns the number of entities reset.

4) probe: The *probe* operation can be used to iteratively enumerate containers within a storage system, objects within a container, forks within an object, or records within a fork. The probe operation can work on any entity of the ASG storage model. The probe operation takes in the identifier information of the entity (container, object, or fork) to be probed as an input argument, entity id to start with, local buffer to store the retrieved information, and maximum number of entities for which the information will be retrieved. The probe operation returns various information about an ASG entity, such as the range of existing records in that entity, their version numbers, and sizes.

C. Relation to data model requirements

In this section, we show how the features provided by the ASG storage model make it possible to meet the requirements of the common data models listed in Table 1. We note that none of these features are new; the ASG storage model just presents a reusable unified API bringing these features together while minimizing complexity. The features provided by the ASG storage model and how they meet the requirements of common data models can be summarized as follows:

- *Unified byte stream and key/value storage*: The ASG storage model supports both byte-stream [3] and key/value-based storage [18]. Each byte is stored as a one-byte record. With the numerical identifiers and record contents, each record can be also used as a key/value store. As a result, the ASG storage model supports both file-based and key/value-based access models and also enables **record-oriented access** for both of these models.
- *Eliminating object attributes*: Object-based storage models, such as T10 [2], use attributes to describe the objects; meaning that object attributes are used to store metadata. In the ASG storage model, we still have metadata describing the records; however, we do not store the metadata in separate attributes as is normally done in object-based storage models. Metadata can be stored in dedicated forks, giving the opportunity to store data and metadata together, to treat metadata in the same way as data, and to have simple metadata management by alleviating the need for a separate metadata API. Simplified metadata management reduces the metadata overhead and improves **scalability**.

- *Record versioning*: Versioning in the ASG storage model enables sorting writes to a record as shown in previous studies [31], [32]. As the version number changes with each write operation, highly **concurrent write operations** will be consistent, and the **performance** of the system will increase.
- *Conditional operations*: The conditional read and write operation flags provide **synchronization primitives** and **atomicity** in a data model, as has been shown in a few storage models [25], [33], [18], [19], [20]. Using the conditional flags with the ASG storage model operations, multiple processes can coordinate *concurrent writes* without using any explicit locking method. Using conditional flags also ensures that each ASG operation either fully completes or has no effect, making sure the system does not end up in an inconsistent state and that **fault tolerance** is achieved.
- *Independently addressable records*: ASG is a **record-oriented** storage model similar to some other data models [34], [35]. Each entity in the ASG storage model has a numerical identifier; when ASG primitives access a record, they explicitly use the identifiers of the enclosing container, object, and fork along with the identifier of that record. As a result, each record in ASG has a distinct location and is independently accessable. Records are the smallest units of storage an operation can access in the ASG storage model. Having access to independently addressable records also makes it possible to support **concurrent read and write access** on them.
- *Fork structure*: Forks can be used to store metadata, as discussed previously. In addition, they can be used to group records that store related data together [36], [30]. This approach improves **performance** by simplifying data management and enables collecting provenance from related records in an efficient way to support **fault tolerance**.
- *Server location*: The ASG storage model exposes location information of its entities to higher-level applications. Applications either can take control of the server location of ASG entities by using this information, or they can let the storage system to handle localization and choose ASG entities randomly without worrying about server locations. When the applications decide which ASG entities to use, they can move **computation closer to storage**.

IV. Example Use Cases

In this section we show how ASG storage model can be used as a foundation for three example use cases.

A. Implementing POSIX Directories

In our first use case, we show how the ASG storage model features can be used to implement POSIX directory operations. Entries in a directory can be represented with the ASG records as shown in Figure 4. In order to map a directory entry to an ASG record, the name of the directory entry can be hashed into a record key. The name, inode information, and data of the directory entry can be stored together in the ASG record. Since each ASG record is independently addressable, the uniqueness requirement of each entry in a directory can be satisfied. Indeed, one can implement POSIX directory operations using the ASG storage model. We note that other namespace and directory implementations also can be supported by using the ASG storage model, even though we show a POSIX directory implementation in this section.

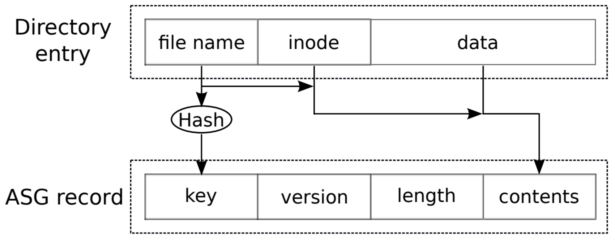

Fig. 4: Mapping directory entries to ASG entities.

In order to create a new file (or subdirectory) in a directory, an underlying ASG *write* operation is called with conditional flags. Similarly, in order to remove a file (or subdirectory) from a directory, underlying ASG *read* and *reset* operations are called with conditional flags. The ASG *read* operation returns the version number of the ASG record representing a directory entry. Checking the version number returned by the ASG *read* operation, ASG *write* does not create a directory entry if it already has been created (nonzero version number), and ASG *reset* does not remove a directory entry if it has not been created yet (zero version number). As a result, the ASG storage model ensures the atomicity of the POSIX create and remove operations, and it prevents duplicate directory entries. If multiple processes try to create or remove a directory entry at the same time, only one of them succeeds.

Other POSIX directory operations, such as updating the metadata of a directory entry and renaming an entry, can be also supported by using the ASG storage model. In order to update the metadata of a directory entry or rename a directory entry, ASG *read* and *write* operations are called with conditional flags. Again, the ASG *read* operation returns the version number of the ASG record representing a directory entry. The ASG *write* does not update the metadata of a directory entry if it has not been created yet (zero version number). While renaming a directory entry, ASG *write* creates the new directory entry with no conditional flags, meaning that it overwrites the new entry if it already exists; and ASG *reset* removes the old entry as soon as the new entry is successfully created. As a result, the ASG storage model ensures the atomicity of the POSIX update and rename operations. All processes in the system see the updated metadata information as soon as update is done, and they see the old metadata information at any time before update completes. No process sees old and new metadata at the same time. For the rename operation, either the old or the new directory entry exists, not both.

In order to lookup a directory entry or to stat a directory, ASG *read* and *probe* operations are called. Similar to the previous POSIX operation implementations, the ASG *read* operation returns the version number of the ASG record representing a directory entry. This version number is not important for the lookup operation, which returns any data available in the entry at time it was called. For the stat operation, however, ASG *probe* keeps track of this version number; hence, if the directory is modified while the stat operation is not done yet, ASG *probe* identifies modified entries and returns updated information about them as a result of the stat operation. Using conditional operations, the ASG storage model ensures the atomicity of the lookup and stat operations by returning updated information with them; at no point is old and new information for an entry returned together.

B. Column-Oriented Key/Value Store

Table 3 shows an example of how a column-oriented key/value database might be expressed using ASG primitives. Rows are represented as ASG records, columns are represented as ASG forks, and shards are represented as ASG objects. ASG records are variable-sized, and any value in the database can be referenced by a unique {object ID, fork ID, record ID} triple. Since ASG *write* operation can take zero-length data as input, rows can have empty columns in the database.

TABLE 3: Example organization of a column-oriented key value store using the ASG storage model.

		Column:fork 0	Column:fork 1	Column:fork 2	Column:fork 3
Shard:object 1	Row:record 0	Alice	Bob	Brad	Charles
	Row:record 1	Smith			Springfield
Shard:object 2	Row:record 0	111-1111		144-1144	321-4321

Columns map well to forks in this example because the fork construct allows each column to be addressed independently while still ensuring that all records within a row are stored in the same object. An entire row can therefore be accessed (or added or removed) atomically. The ASG object storage model does not dictate on-disk layout; but it is expected that the storage system would organize data on disk such that forks are contiguous in this case. Shards map naturally to objects because they partition the data set into discrete chunks that can be used to parallelize column-oriented queries, if objects are distributed across different servers.

Forks and variable-sized records provided by the ASG interface are critical to expressing this use case. Without these features, a column-oriented key/value storage system would be forced to maintain an additional mapping index to translate between row, column nomenclature and offset, and size nomenclature. This translation layer not only would add complexity for the data model implementor, it would also prevent critical semantic information from being expressed to the storage system. An ASG-based storage system, for example, may recognize a linear column-oriented access pattern and adapt its underlying storage layout accordingly, while a traditional storage system making the same optimization would have to do so based on assumptions derived from generic byte range access patterns. The ASG *probe* function also leverages the additional structured data semantic information provided by fork and record-oriented access to enable efficient enumeration of both the rows and columns of a table.

C. HPC Application Checkpoint

Implementing HPC checkpointing strategies directly on top of the ASG storage model is straightforward because of its structure and explicit location control feature. One can implement both $N-N$ and $N-1$ checkpointing strategies using the ASG storage model and thus overcome the limitations of these methods as explained in Section II-C.

In the $N-N$ checkpointing method, each process writes to a separate checkpoint file. As explained in Section III-C, the ASG storage model exposes location information of its entities to higher-level applications. Therefore, any application trying to implement $N-N$ checkpointing method can take advantage of this information to pick ASG entities that will store checkpoint data, in a manner to balance the metadata load across the system without dealing with any dedicated location servers. Additionally, as explained in Section III-C, object attributes are eliminated in the ASG storage model, and as a result metadata management is simplified. Having simple metadata management and explicit location control reduces the metadata overhead in the $N-N$ checkpointing method.

If the $N-1$ checkpointing method is implemented, each process writes to a shared checkpoint file. As explained in Section III-C, the ASG storage model has record versioning and conditional operation features. Therefore, processes trying to write to a checkpoint file concurrently can take advantage of record versioning and conditional operations to order their writes to the checkpoint file. This strategy alleviates the need to set locks on the checkpoint file, since it is possible to update the file atomically using the ASG storage model operations. As a result, having record versioning and conditional operations makes it possible to have highly concurrent writes to the checkpoint file in the $N-1$ checkpointing method.

V. Related Work

Network-Attached Secure Disk (NASD) [37] is the primary work on object-based storage, and it led to specifications of standards [2] for object-based storage. NASD introduces variable-length objects with attributes, rather than fixed-length traditional blocks, to enable self-management and to obviate the need to know about the host operating system. Moving data management to the storage disks increases the networking, security, and space management capabilities.

Numerous studies in the literature have similar scope to our work. OSD+ [38], [39] presents a model similar to the one specified by the OSD standard [2] except for the addition of dedicated directory objects. The directory objects in OSD+ store file names and attributes and support metadata-related operations. The Panasas File System [3] is built on object-based storage devices. The OSD wire protocol of Panasas uses the operations from the OSD standard [2] to enable byte-oriented access to data, to manipulate attributes and to create or delete objects. Lustre [7], [40] is a distributed file system based on object-based storage. The object storage server in Lustre is responsible for providing access to file data stored in objects on object storage targets. Ursa Minor [31] is a parallel file system that supports versioned writes. It keeps the existing object-storage interface [2] mostly intact except for introducing *slices* (i.e. fragments of object data), and it uses timestamps to distinguish different versions of data. Datamods [41] is a framework that exploits existing large-scale storage system services to support complex data models and interfaces. Datamods avoids duplicating services already provided in distributed storage systems in middleware and improves scalability since it is not limited to a single dimension at the file level. Rados Gateway [6], [15] is an object storage service forming the foundation of Ceph [8]. It provides the clients a single logical object store and offloads object replication, failure detection, and data management tasks to the underlying object store daemons. The Ohio Supercomputing Center looked at mapping Parallel Virtual File System [12] on top of an existing object-based storage emulation [4], [5]. This mapping moved the functionality of the common components of a traditional storage system, such as I/O, directory, or metadata servers, to OSDs and improved the performance of the overall system thanks to the capabilities of the object-based storage devices [42], [43], [44]. VSAM [35] supports both fixed-sized and variable-sized records depending on the application. Forks in NTFS [36] are similar to records in the ASG storage model; they are byte streams storing file data and auxiliary information such as metadata and security settings. Conditional operations are used in Amazon SimpleDB [33], Amazon DynamoDB [18], Redis [19], and Hyperdex [20].

A number of studies form the technical basis of our work. Transactional Object Storage Device (TOSD) [32] shows that object-based storage is a common component of many parallel file systems, and it introduces three optimizations to the object-based storage model in order to serve highly concurrent workloads better: atomicity, versioning, and commutativity. Goodell et al. [45] extended the POSIX API by organizing the storage around data objects in order to map complex data structures to these data objects and have direct access between the data objects and applications. Carns et al. [25] investigated conditional update operations as an alternative to distributed pessimistic locking operations in object-based storage systems.

VI. Conclusions and Future Work

In this paper, we presented a new object-based storage model, ASG; introduced its architecture and primitives; and described a couple of use cases based on this model. As the use cases clearly show, the ASG storage model is flexible and can act as a starting point for building complex storage applications. Features supported by the ASG storage model make it possible to support requirements of common data models.

Acknowledgments

The authors thank the following people: Matthew Curry, Geoff Danielson, Ruth Klundt, and Justin Wozniak. This material is based on work supported by the U.S. Department of Energy's Oak Ridge National Laboratory and included the Extreme Scale Systems Center, located at ORNL and funded by the DoD in part by contract number 4000111689 "Novel Software Storage Architectures". This work also was supported by U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

[1] M. Mesnier, G. Ganger, and E. Riedel, "Object-based Storage," *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84 – 90, aug. 2003.

[2] T10 Technical Committee of the InterNational Committee on Information Technology Standards, "Object-based Storage Devices - 3 (OSD-3)." [Online]. Available: http://www.t10.org/

[3] 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 *In FAST-2008: 6th Usenix Conference on File and Storage Technologies*, 2008, pp. 17–33.

[4] A. Devulapalli, D. Dalessandro, P. Wyckoff, and N. Ali, "Attribute Storage Design for Object-based Storage Devices," in *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*. Washington, DC: IEEE Computer Society, 2007, pp. 263–268.

[5] A. Devulapalli and N. Ali, "Integrating Parallel File Systems with Object-based Storage Devices," in *Proceedings of Supercomputing*, 2007.

[6] "Librados API documentation." [Online]. Available: http://ceph.com/docs/master/api/librados/

[7] Oracle Corporation, "Lustre File System." [Online]. Available: http://wiki.lustre.org/

[8] 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*, 2006, pp. 307–320.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924947

[10] "Lustre: A Scalable, High-Performance File System," Cluster File Systems Inc. white paper, version 1.0, November 2002. [Online]. Available: http://www.lustre.org/docs/whitepaper.pdf

[11] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083349

[12] 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 and Conference*. Atlanta, GA: USENIX Association, October 2000, pp. 317–327. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/pvfs.ps

[13] "Amazon Simple Storage System (Amazon S3)." [Online]. Available: http://aws.amazon.com/s3/

[14] L.-F. Cabrera and D. D. E. Long, "Swift: A Storage Architecture for Large Objects," U.C. Santa Cruz, Tech. Rep. UCSC-CRL-89-04, 1990. [Online]. Available: ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-89-04.tar.Z

[15] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Fast, Scalable, and Reliable Storage Service for Petabyte-scale Storage Clusters," in *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW 07)*, Reno, NV, Nov. 2007.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Bolton Landing, NY: ACM Press, October 2003, pp. 96–108. [Online]. Available: http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[18] "Amazon DynamoDB." [Online]. Available: http://aws.amazon.com/dynamodb/

[19] "Redis." [Online]. Available: http://redis.io/

[20] "Hyperdex: A Searchable Distributed Key-Value Store." [Online]. Available: http://hyperdex.org/

[21] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[22] L. George, *HBase: The Definitive Guide*, 2011. [Online]. Available: http://proquest.safaribooksonline.com/9781449314682

[23] "BigQuery." [Online]. Available: https://developers.google.com/bigquery/

[24] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 195–202, May 1979. [Online]. Available: http://dx.doi.org/10.1109/TSE.1979.234181

[25] P. Carns, K. Harms, D. Kimpe, J. M. Wozniak, R. Ross, L. Ward, M. Curry, R. Klundt, G. Danielson, C. Karakoyunlu, J. Chandy, B. Settlemyer, and W. Gropp, "A Case for Optimistic Coordination in HPC Storage Systems," in *Proceedings of 2012 Parallel Data Storage Workshop (PDSW 2012)*. IEEE, 2012.

[26] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY: ACM, 2012, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254766

[27] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," in *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, Newport Beach, CA, 1993, pp. 56–70, also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[28] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *Proceedings of IEEE Conference on Cluster Computing*, New Orleans, LA, September 2009.

[29] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York: ACM, 2009, pp. 1–12.

[30] N. Nieuwejaar and D. Kotz, "The Galley Parallel File System," in *Proceedings of the 10th ACM International Conference on Supercomputing*. Philadelphia: ACM Press, May 1996, pp. 374–381. [Online]. Available: http://www.cs.dartmouth.edu/~dfk/papers/nieuwejaar:galley.ps.gz

[31] 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 *FAST'05: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, 2005, pp. 5–5.

[32] P. Carns, R. Ross, and S. Lang, "Object Storage Semantics for Replicated Concurrent-Writer File Systems," in *Proceedings of 2010 Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS 2010)*. IEEE, 2010.

[33] "Amazon SimpleDB." [Online]. Available: http://aws.amazon.com/simpledb/

[34] "HP OpenVMS Systems Documentation, Files-11 On Disk Structure Concepts." [Online]. Available: http://h71000.www7.hp.com/doc/731final/4506/4506pro_001.html

[35] D. Lovelace, R. Ayyar, A. Sala, and V. Sokal, *Vsam Demystified*, 1st ed. Riverton, NJ, USA: IBM Corp., 2003.

[36] R. Russon and Y. Fledel, "NTFS Documentation," 2004.

[37] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A Cost-Effective High-Bandwidth Storage Architecture," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 1998, pp. 92–104. [Online]. Available: http://www.acm.org/pubs/citations/proceedings/asplos/291069/p92-gibson/

[38] A. Avilés-González, J. Piernas, and P. González-Férez, "Scalable Metadata Management through OSD+ Devices," *International Journal of Parallel Programming*, pp. 1–26, 10.1007/s10766-012-0207-8. [Online]. Available: http://dx.doi.org/10.1007/s10766-012-0207-8

[39] ——, "A Metadata Cluster Based on OSD+ Devices," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, pp. 64–71, 2011.

[40] J. Lombardi and L. Zhen, "DAOS Changes to Lustre." Presented by Intel High Performance Data Division, April 2013.

[41] N. Watkins, C. Maltzahn, S. Brandt, and A. Manzanares, "DataMods: Programmable File System Services," in *Proceedings of 2012 Parallel Data Storage Workshop (PDSW 2012)*. IEEE, 2012.

[42] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan, "An OSD-based Approach to Managing Directory Operations in Parallel File Systems," in *IEEE International Conference on Cluster Computing*, September 2008. [Online]. Available: http://www.cse.ohio-state.edu/~alin/papers/cluster2008.pdf

[43] ——, "Revisiting the Metadata Architecture of Parallel File Systems," in *Third Petascale Data Storage Workshop, Supercomputing*, November 2008. [Online]. Available: http://www.cse.ohio-state.edu/~alin/papers/pdsw2008.pdf

[44] A. Devulapalli, D. Dalessandro, and P. Wyckoff, "Data Structure Consistency Using Atomic Operations in Storage Devices," *IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, vol. 0, pp. 65–73, 2008.

[45] D. Goodell, S. Kim, R. Latham, M. Kandemir, and R. Ross, "An Evolutionary Path to Object Storage Access," in *Proceedings of 2012 Parallel Data Storage Workshop (PDSW 2012)*. IEEE, 2012.