DataMods: Programmable File System Services

Noah Watkins, Carlos Maltzahn, Scott Brandt
University of California, Santa Cruz
{jayhawk,carlosm,scott}@cs.ucsc.edu

Adam Manzanares
California State University, Chico
nmtadam@gmail.com

ABSTRACT

Cloud-based services have become an attractive alternative to in-house data centers because of their flexible, on-demand availability of compute and storage resources. This is also true for scientific high-performance computing (HPC) applications that are currently being run on expensive, dedicated hardware. One important challenge of HPC applications is their need to perform periodic global checkpoints of execution state to stable storage in order to recover from failures, but the checkpoint process can dominate the total run-time of HPC applications even in the failure-free case! In HPC architectures, dedicated stable storage is highly tuned for this type of workload using locality and physical layout policies, which are generally unknown in typical cloud environments. In this paper we introduce DataMods, an extended version of the Ceph file system and associated distributed object store RADOS, which are widely used in open source cloud stacks. DataMods extends object-based storage with extended services take advantage of common cloud data center node hardware configurations (i.e. CPU and local storage resources), and that can be used to construct efficient, scalable middleware services that span the entire storage stack and utilize asynchronous services for offline data management services.

1. INTRODUCTION

The use of Infrastructure as a Service (IaaS) offerings have dramatically increased in recent years, owing their success to seemingly limitless scalability, and cost effective resource allocation. However, the architectural design of IaaS storage services are largely incompatible with the needs of high-performance computing (HPC) software. In order for the HPC community to take advantage of cost savings offered by the use of cloud services—especially important for academic and governmental institutions— specialized I/O storage services must be be offered by IaaS providers.

Many HPC applications are tightly coupled, and sensitive to failures. This is a problem for long running simulations. A common, general fault-tolerance solution is global checkpoint and restart. In order to recover from system failures applications often use a global checkpoint mechanism to periodically save their execution state to a file system, allowing the application to be restarted from the latest snapshot after a failure occurs. Because the saved state must be globally consistent, during a checkpoint applications cannot make progress. Thus improving checkpoint and restart performance is an important goal. It is common in dedicated HPC data centers to highly customize and tune the physical layout of checkpoint data for a particular file system in order to obtain good performance. Unfortunately, exposing low-level tuning parameters and data locality information is largely antithetical to common cloud architectures that favor black-box designs that can be transparently reconfigured for load balancing and consolidation.

The process of application checkpoint is difficult to scale because of the mismatch between high-level data models and the traditional POSIX file I/O interface. Data access libraries (e.g. HDF5 and MPI-IO) provide valuable domain-specific abstractions to HPC application—such as arrays, meshes, and graphs—over the standard POSIX file I/O interface, but over time have become highly complex. To understand why, consider that these middleware libraries must implement flexible, parallel services such as metadata management, data alignment, and provide specialized access such as views, all on top of the byte-stream data model that makes much of the structural information unavailable to the file system. In order to remain scalable, these middleware layers must choreograph access within a narrow stage defined by so called "magic numbers"—the alignment specifications of the underlying physical storage—and even when a file system manages to expose these hints or allow their customization, integrating this knowledge into middleware remains non-trivial. Fortunately, many of the services duplicated by middleware libraries already exist in distributed file systems with well-defined scalability properties. Exposing these services to high-level programming models as a standardized IaaS offering will allow developers to avoid duplicating complex, error prone services in favor of generalized, robust, and scalable version of such services. This paper identifies these services and surfaces them with a set of abstractions that are de-

signed to provide a new abstract storage interface that might replace or extend the existing POSIX I/O standard.

Data Model Modules (DataMods) are a new way of building middleware services that take advantage of a variety of scalable services already found within parallel file systems. We have identified four components common to middleware libraries, and propose that existing services within distributed object-based file systems be generalized to provide the following services to high-level programming models:

1. **Metadata management.** Middleware libraries manage information describing a file layout and data types, and must be able to efficiently access this data at scale.

2. **Data placement.** Domain-specific data models are forced to fit the byte-stream model by serializing model instances to offsets within a file, and indexes maintained that record the location of data elements.

3. **Intelligent data access.** Structural and content-aware data subsetting require special purpose indexes that must be explicitly managed, and locality information is difficult to derive despite its importance in optimizations that reduce data movement.

4. **Asynchronous services.** Middleware software performs many synchronous tasks that are amenable to asynchronous processing, such as various compression techniques, indexing construction, and workflow execution.

A DataMod module contains a programmatic specification for how each generalized service should behave, and the system ensures correct behavior without sacrificing the scalability of each service. For instance, the layout of a file may be expressed with a few basic formulas and some accompanying metadata. These rules and parameters are stored within a file inode, but the system must prevent an inode from growing too large in order to retain scalability properties of the metadata and namespace service.

The remainder of this paper is structured as follows. Section 2 describes distributed object-based file systems, the services they provide, and the interfaces they export. In Section 3 this interface is contrasted with the domain-specific data models that applications actually require, and in Section 4 we show how existing services can be generalized and used within high-level programming models.

2. CEPH DISTRIBUTED FILE SYSTEM

Large-scale file systems contain many scalable services that function together to implement the common byte-stream interface. In this section we provide an overview of these services in the context of the Ceph distributed object-based file system, focusing on the scalability properties that must be maintained when exposing generalized interfaces to high-level programming models.

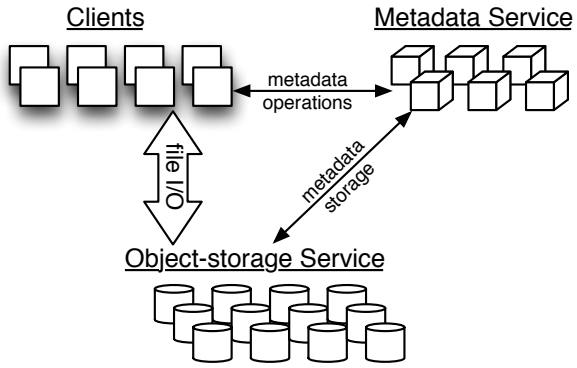

Figure 1: Architecture of object-based file systems. Clients communicate with distinct clusters for either metadata or file data operations.

An object-based file system refers to a file system architecture in which metadata (e.g. files, directories, and locks) and file data are managed by distinct distributed services within a file system cluster. A high-level view of Ceph's architecture is illustrated in Figure 1 in which clients read and write file data directly to an object-storage service, and direct metadata requests to a separate service. This separation of services is important because of the vastly different access patterns needed by file data and metadata.

Scalable metadata management. Clients interact with a dedicated metadata service (MDS) that manages the file system namespace, provides additional services such as coherency control and security, and allows the metadata cluster to scale independently of the underlying object-storage system. A key component to Ceph's metadata scalability is its use of fixed-size inode structure that can be embedded in directory fragments, allowing metadata servers to quickly rebalance using tree partitioning as workloads change. The fixed-size inode is made possible by trading large, explicit block lists for a compact generating function that calculates object locations [5].

Distributed object storage. A cluster of object storage devices (OSDs) persist both metadata and file data in flexibly sized containers called objects. Each OSD consists of local storage (e.g. HDD or SSD), a cache, multi-core CPU, and RAM. Objects managed by the cluster can belong to different *classes*, taking on the behavior defined by the class, and allowing in-

terfaces other than basic read and write functionality. For example, metadata updates stored within a special MDS-class object serialize directory updates at the object-level, avoiding locking overhead and improving scalability.

Recovery and fault-tolerance. Recovery and fault-tolerance are handled transparently by the distributed object store. At the time an OSD fails, all other OSDs in the cluster become responsible for a share of the data stored on the failed OSD. Ceph employs a scalable shuffling technique that guarantees only an amount of data proportional to the size of the failed OSD is moved. Additional services such as scrubbing and repair are handled as asynchronous background tasks on each storage device.

2.1 Permitted File Operations

Ceph exposes a standard POSIX file I/O interface to clients, including a full hierarchical namespace. When a client opens a file it retrieves the inode from the metadata service. The inode contains standard file metadata such as ownership and protection, as well as the configuration used by clients to map the file byte-stream into a set of objects contained within the object-store cluster, a process known as *striping*. Ceph allows applications and middleware limited control over physical layout by allowing customization of the per-file striping strategy configuration.

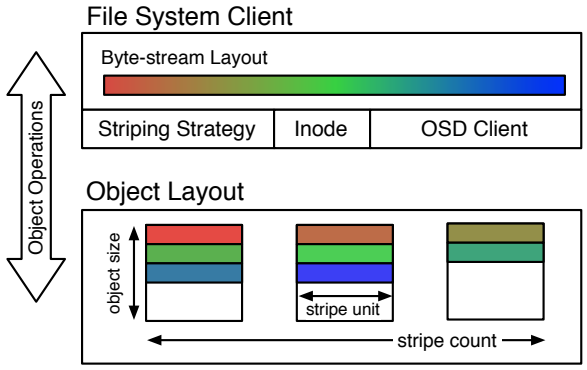

Figure 2: A file system client uses a striping strategy to distribute a byte-stream across a set of objects. The strategy is partially configurable via parameters found in the file inode.

Figure 2 illustrates how clients control physical file layout. Three parameters are used to specify the striping strategy—*object size*, *stripe unit*, and *stripe count*—allowing applications to manipulate layout along multiple dimension. Unfortunately, the striping strategy of a file is applied to the entire byte-stream, and many applications may write files with varying degrees of regularity, or simply have no regular pattern, gaining little or no benefit from the additional control mechanism.

Finally, Ceph exposes additional control over the consistency guarantees required by the POSIX interface. Locking can be relaxed among multiple writers of the same file using the O_LAZY feature when applications wish to make their own guarantees over concurrent writer behavior (such as client-side synchronization). However, even O_LAZY benefits are limited by false sharing in the case of unaligned access. Object-based file systems feature a complex mix of services designed to expose scalable byte-stream interface, but as we will see in the next section, this interface is far from ideal for middleware developers.

3. MIDDLEWARE-LEVEL INTERFACES

Applications interact with complex structured data such as arrays, meshes, relational tables, and key-value pairs, through a life cycle that involves saving, restoring, and communicating application state. However, in order to persist application data to a file system the serialization of structured data must conform to the data model exposed by the POSIX file I/O interface. For example, Figure 3 illustrates an instance of a graph that an application stores in a file system. There are many challenges that must be met when managing complex data, and these challenges are best addressed by data management middleware.

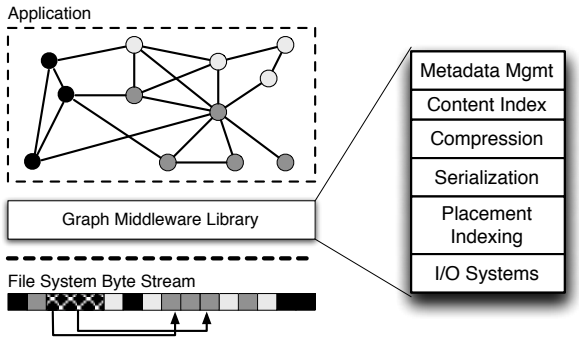

Figure 3: Middleware libraries map application-level data model instances onto low-level byte-streams. These libraries contain many sub-systems, such as indexing that records the location of data model elements, depicted by the intra-stream references in the byte-stream.

Middleware bridges the division between complex application data models, and the type-less byte-stream interface by providing structured access to data. The challenge faced by middleware is how the translation between the two abstractions can be made efficient and scalable. Returning to Figure 3, consider the functionality that must be implemented within this middleware. Subsets of the graph managed by potentially thousands of clients in a distributed application are serialized, compressed, placed within the byte-stream, and

the placement location recorded in an index; all while trying to reduce intra-file contention due to unaligned accesses. It should not be surprising then that what middleware builds within a byte-stream begins to resemble a file system.

Metadata management. Middleware places metadata within a byte-stream that may take on many roles. For example, headers in the HDF5 file format are placed at well-known positions and act as superblocks, allowing clients to bootstrap themselves with btree-based indexing and file layout information. Multiple distinct data model instances (e.g. arrays) can be kept in a single HDF5 file and organized using a heriarchical namespace, and clients must be aware of the distinction using namespace metadata.

Data placement. As an HDF5 file grows, indexes, metadata, and multiple data model instances must each expand within the byte stream. Since each of these are fundamentally different types of data, regions of the file will naturally be subject to different types of workload and growth patterns. For example, a btree placement index will have a high-degree of concurrent access from many readers, while locality among regions of a graph may be exploited when reading and writing model data. A challenge for middleware designers is to position data within a file in such a way that reading and writing metadata and other content is efficient and scalable.

Data access methods. Applications make structured requests against instances of a data model implemented by middleware, and middleware may also support predicate-based filtering. However, middleware occupies a position in the I/O stack above the file system client level, thus there is little to no support for intelligent, data model specific access methods. Rather, structural indexes and layout metadata are first read and queried, then located content is read, which may be a granularity that results in increased network traffic. When content-based indexes are not supported significant amount of communication may be wasted on moving data that does not match a desired predicate.

Asynchronous services. Middleware offers data compression, implements basic workflows, and performs data management tasks such as indexing. These operations are performed online—while a file is opened—but many are amenable to being executed asynchronously after a file is closed, thereby reducing the amount of time applications spend performing I/O.

Next we will examine how the file system services within distributed object-based file systems, as described in Section 2, can be generalized to subsume the common needs of middleware described in this section.

4. DATA MODEL MODULES

The traditional byte-stream interface is an important system component that has survived because of its simplicity, and will always serve an important role. In this section we provide an overview of the abstractions we propose to augment the existing byte stream interface with rich functionality for new middleware and application designers.

The interface consists three abstractions. First, the *File Manifold* encapsulates metadata management and data placement by allowing middleware to extend inode structures with custom state and rule-based placement logic. Second, *Typed and Active Objects* provide a safe mechanism for extending the interface and behavior of the underlying object store to include application-specific abstractions. Finally, applications can take advantage of *Asynchronous Services* to implement offline processes that can be used to perform indexing, compression, de-duplication, and other management activities such as basic workflows, after a file has been closed.

Tying each of these abstractions together, and to underlying file system services, is a run-time environment and language. The run-time limits the exposure of file system internals to application-specific extensions while still providing access to a rich array of built-in utilities, as well as other application-defined abstractions. The language and run-time together enforced basic scalability invariants in the underlying file system service.

4.1 File Manifolds

A file manifold is a generalization of metadata storage and data placement services, and addresses the needs of middleware to support heterogeneous byte-streams in which multiple types of data are combined in arbitrary patterns. For example, an HDF5 file may store several multi-dimensional arrays in distinct files, each with a layout tailored for a particular array, while the high-level file manifold stitches each sub-file together forming a composite view. Figure 4 illustrates an example of such a file consisting of three datasets. The first two are represented by pattern-based striping strategies with different configurations, and the third uses an index to record data element placement (e.g. a vertex list).

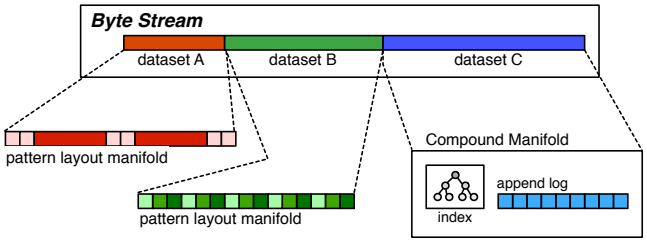

Figure 4: A file manifold is an abstract striping strategy that allows separate regions of a file to be stitched together in arbitrary patterns.

Scalability. All manifold data is stored within a

file inode, and thus must adhere to the scalability constraints of the Ceph metadata service. An inode in Ceph may be flexibly sized, but must remain to be small enough to be stored inline with directory entries. Figure 5 illustrates the extensions made to the inode structure to implement file manifolds.

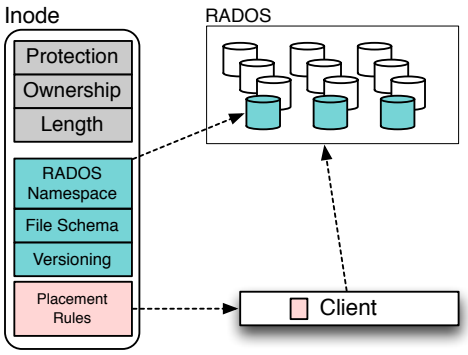

Figure 5: FIXME: this graph needs to be expanded

In addition to the common fields stored within an inode—access times, object namespace, modes—file manifolds expand an inode by incorporating custom metadata and striping rules. The interface through which inode extensions are defined enforce limits on the size of an extended inode. The scalability constraints of an inode do not prevent file manifolds from being dependent on more data than is allowed, and may instead choose to store metadata within auxiliary objects with references maintained in the inode itself.

Byte-stream compatibility. Middleware need not be constrained to exporting a byte-stream, and may opt to work directly with objects. In such cases middleware have maximum freedom over data layout, while still being able to take advantage of the namespace management features offered by the metadata service. There are interesting implications that result from not supporting a byte stream interface. First, measures must be taken to ensure that general binary utilities (e.g. cp, mv) do not succeed, as there may be not real meaning to a byte-stream. Interestingly, it may be possible to automatically export an archive format as a byte stream, for any file manifold, by serializing objects and metadata.

4.2 Active and Typed Objects

Advanced interfaces that go beyond interacting with type-less binary objects are needed to efficiently support the type of intelligent data access, filtering, and manipulation that middleware libraries perform at a high-level. This is accomplished through an object-level abstraction that combines type and interface metadata with a rule-based specification of behavior.

Figure 6 illustrates the active object abstraction. There are two ways that new behavior can be added to objects. First, hooks can be registered with the system

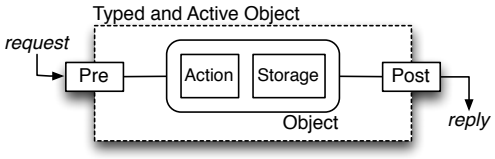

Figure 6: Typed and active objects combine rules and actions for handling requests, interacting with object data, and may generate transitive actions that affect other objects or system services.

that wrap existing object behavior. For example, a preprocessing hook could be registered around the existing read and write behavior to implement an application-specific compression codec. Second, entirely new behavior can be defined by implementing named methods associated with an object type. For example, an application may implement an object method that provides model-specific views using an automatically maintained index of object contents.

Applications implement new object types and behavior in a similar way to defining file manifolds, by specifying state information and rules that the system automatically maintains and isolates on behalf of an application. As shown in Figure 7, applications begin by registering a new object type that includes state information and rules defining behavior. Object types do not change, and therefore can be aggressively cached throughout the system. Following registration, the system will return to the application a handle used when referencing objects, such as when new objects are created. Native object types built into the system have well-known handles, and can be used in the same way as application-defined types. Finally, the underlying object store is responsible for evaluating object rules on behalf of applications or asynchronous services within a run-time environment.

4.3 Asynchronous Services

Middleware libraries perform compression, indexing, and are involved in high-level workflows, among other data management tasks. Each of these tasks is performed online while files are opened, but are amenable to being processed asynchronously. For example, aggressive compression can be applied offline to reduce data volumes that would otherwise introduce unacceptable overhead to the write fast path. High-level libraries such as the Climate Data Operators can be used to implement workflows such as regridding or computing statistical summaries. Providing middleware designers flexibility in when and where data-intensive operations such as these are scheduled can help increase utilization by spreading work throughout the system.

Scalability. Middleware can schedule asynchronous work to be performed continuously in the background,

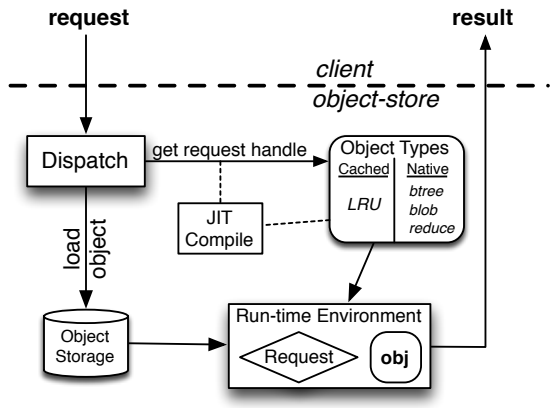

Figure 7: Object requests trigger specific actions within the object store. Native actions are built-in, and dynamic behavior defined by middleware is JIT compiled, and LRU cached.

scheduled in the future, or triggered at specific file events (e.g. close). Work is defined in a similar way to that of active and typed objects. An interface with specific behavior is defined and registered within the system, and may be accessed from within the run-time environment to register work.

Existing, threaded workqueue abstractions within the file systems are generalized to run within the context of the DataMod run-time, and their interface (e.g. addwork()) is exported to be available within the run-time environment.

4.4 Run-time Environment

The generalization of services requires the ability to synthesize new behaviors within the file system, such as formulas used to define a striping strategy, or lightweight routines that can traverse a data model instance. DataMods accomplishes this using a well-defined, safe run-time environment that includes just-in-time compilation of code, or alternatively the ability to communicate with external processes that implement required logic. The run-time environment provides middleware developers access to built-in functionality such as existing data structures and metadata, as well as the metadata and interfaces defined within and exported by an existing DataMods module. A full discussion of the requirements of the run-time environment are beyond the scope of this paper, and include addressing both resource usage and scalability isolation.

5. RELATED WORK

The Lustre file system contains a mechanism for joining files together, each with a potentially distinct striping pattern. Yu et al. used this feature to decouple MPI-IO collective writers to avoid a performance problem with large stripe widths [6]. Lustre file joining is similar to file manifolds in that multiple data sources are merged to construct a view, but Lustre is restricted to joining with concatenation semantics, rather than the flexible, rule-based joining offered by file manifolds.

There has been a wide variety of research related to active storage [2, 4] John et al. explores active storage in the context of object storage devices by allowing functionality to be defined in Java and activated remotely by clients using an extended version of the iSCSI OSD protocol. The user-defined Java class files are downloaded to the storage devices, and executed in a process distinct from the storage device OSD server. It is difficult to integrate with caching policies, data must be moved across process boundaries, and QoS is challenging with arbitrary code allowed to run in a memory hungry virtual machine.

Piernas et al. takes a princepled approach to adding active storage facilities to the Lustre file system, in which clients create empty files that act as sinks, and specify source files to be processed by arbitrary processing components. The processing components accept data streams, and several configurations of source and sink are possible. Striped files are processed in parallel by instantiating a processing component on each storage device that processes local data chunks.

In [3], Bogdan et al. introduced BlobCR that performs efficient, incremental checkpoints in a virtual machine cloud environment using VM snapshots saved to local disk (and migrated later to stable storage). An integrated API is available so that applications can synchronize checkpointing, but BlobCR only targets applications that can checkpoint local state, avoiding writing to a global file.

[1] found that several cloud providers supported HPC application with acceptable performance, and that increased network performance would bring the greatest increase in applicability of the cloud to HPC. They did not however address any I/O issues such as checkpointing.

6. CONCLUSION AND FUTURE WORK

This paper presents DataMods, a plugin mechanism allowing new file formats to be constructed in and supported directly by parallel file systems. The DataMods system provides middleware developers enhanced expression of their requirements and helps developers avoid duplicated common system services. The next steps for this work is to explore several case studies in order demonstrate the performance improvements and complexity reduction that can result from building middleware systems across the entire storage system stack.

7. REFERENCES

[1] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running hpc

applications in public clouds. In *ScienceCloud '10*, 2012.

[2] Tina Miriam John, Anuradharthi Thiruvenkata Ramani, and John A. Chandy. Active storage using object-based devices. In *HiperIO '08*, 2008.

[3] Bogdan Nicolae and Frank Cappello. Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *SC '11*, 2011.

[4] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *SC '07*, 2007.

[5] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI '06*, 2006.

[6] Weikuan Yu, Jeffrey Vetter, R. Shane Canon, and Song Jiang. Exploiting lustre file joining for effective collective io. In *CCGrid '07*, 2007.