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
amanzanares@csuchico.edu

***Abstract*—As applications become more complex, and the level of concurrency in systems continue to rise, developers are struggling to scale complex data models on top of a traditional byte stream interface. Middleware tailored for specific data models is a common approach to dealing with these challenges, but middleware commonly reproduces scalable services already present in many distributed file systems.**

We present DataMods, an abstraction over existing services found in large-scale storage systems that allows middleware to take advantage of existing, highly tuned services. Specifically, DataMods provides an abstraction for extending storage system services in order to implement native, domain-specific data models and interfaces throughout the storage hierarchy.

I. Introduction

From business analytics to scientific simulations, application requirements are continually pushing the limits of high-performance storage systems. However, as application complexity and concurrency grow, developers face increasing pressure from existing I/O interfaces. With next-generation, open-source storage systems currently in development, now is the right time to look for architectures and abstractions that are able to serve as a platform for the construction of domain specific interfaces.

An overwhelming majority of storage systems today are built assuming a byte-stream I/O interface. This has had a profound impact on software architecture: the inability of applications to explicitly represent domain-specific data models throughout the storage hierarchy has led to the development of middleware libraries that provide data model abstractions (e.g., HDF5, NetCDF), and I/O stack extensions that help circumvent scalability bottlenecks (e.g. MPI-IO, PLFS, IOFSL). While it is common for applications and middleware to influence the design of each other, this co-design generally ends at the level of file interfaces, and as a result, services such as metadata management, data translation, alignment, and views are forced into a one-dimensional abstraction that is difficult to scale to millions of clients.

The key insight driving the work we introduce in this paper is: many of the services implemented by middleware libraries already exist in distributed storage systems. We propose Data Model Modules (DataMods), a framework that exposes these services through high-level programming models that allow developers to avoid error-prone duplication of complex software and instead rely on generalized, robust, efficient, and scalable implementations. The framework supports the construction of native, domain-specific data models and interfaces that extend throughout the storage hierarchy. DataMods is a set of generalized abstractions over common services found in distributed storage systems that provide well-defined scalability properties. We have identified the following three components that compose the current version of the DataMods framework. First, the *file manifold* provides an abstraction encapsulating metadata management and file layout to allow middleware libraries to define complex, non-static striping strategies tailored to a particular data model. Second, *typed and active storage* provide a programming model for defining computation and interfaces at the lowest-level of the storage hierarchy. And finally, *asynchronous services* are used to coordinate deferred actions, such as indexing, compression, and de-duplication.

There are already many cases in which file systems expose limited amounts of information and control to support specialized optimizations. For instance, HPC applications using MPI-IO can define complex file layouts for optimizing parallel I/O, but many file systems only provide simple, fixed parameters for tuning data layout (e.g. object size). Other examples include the numerous storage interfaces (e.g. key-value pairs) used by the Hadoop project that build directly upon byte streams, despite being at odds with the underlying byte stream interface. One reason storage systems with support for complex data models have not emerged is a lack of consensus on what such a model would look like. It would be useful if storage systems provided a mechanism for creating and extending the set of exported data models without sacrificing scalability. Such a feature would allow applications to use domain specific data models, without the storage system supporting a single, exclusive interface.

We demonstrate the efficacy of the DataMods framework using a case study based on checkpoint/restart workloads. Specifically, we base our design off of the Parallel Log Structured File System (PLFS), and show how index maintenance and compression can be offloaded to the file system and scheduled asynchronously to a computation writing the checkpoint.

Next we discuss existing storage system services (Section II), and typical services that are duplicated in middleware (Section III). In Section IV we provide an overview of the DataMods framework, and Section V presents a use case based on checkpoint/restart workloads. We conclude with related work (Section VII) and future extensions.

II. Storage System Services

Large-scale file systems contain many scalable services that function together to implement the common byte stream interface. DataMods is a generalization of these services, and this section provides a brief overview of the services being targeted. We are using Ceph [1] as a reference, and other parallel file systems offer a similar set of services.

Scalable metadata management. Clients interact with a metadata service that manages the file system namespace, and provides cache coherency and security. A key component to Ceph's metadata scalability is its use of fixed-size inode structures that can be embedded in directory fragments allowing servers to quickly rebalance using tree partitioning as workloads change. Ceph also avoids large block lists and expensive communication to perform block location look-ups by using a compact function that calculates object locations [2].

Distributed object storage. A scalable cluster of object storage devices (OSDs) export a key/value-like interface, and persist both metadata and file data [3]. 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 a class, and allowing interfaces other than basic read and write functionality. For example, metadata updates stored within a special metadata-class object, used by Ceph to serialize directory updates at the object-level, allow the OSD to avoid locking overhead and improve scalability.

Recovery and fault-tolerance. Recovery and fault-tolerance are handled transparently by the cluster of OSDs in the background. Following the failure of an OSD, other storage devices 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 volume of data stored on the failed OSD is moved. Additional services such as scrubbing and repair are handled as asynchronous background tasks on each storage device.

File operations. A client retrieves a file inode from the metadata service when it opens a file. The inode contains standard file metadata, as well as the parameters needed to dynamically map the byte-stream to the underlying set of physical data objects. Ceph allows applications limited control over file layout by permitting the file layout parameters to be customized on a per-file basis. Unfortunately, the striping strategy of a file is applied to the entire byte-stream, despite many applications using irregular data models within one file, making the choice of a single layout difficult.

Next we present functionality commonly implemented by middleware, and then discuss how services found within storage systems can be re-purposed to support middleware requirements.

III. Middleware Services

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.

Metadata management and file layout. Middleware often manages complex abstract data models—(e.g. multi-dimensional arrays)—instances of which have real-world meaning that must be kept with the file. Generally this information is located in headers that encode free-text knowledge as well as data type and schema descriptions. In addition to application-facing metadata, middleware must coordinate the layout of multiple data model instances within the byte stream, often using sophisticated index structures that complicate file layout maintenance (e.g. shifting byte streams when expanding headers and indices).

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 is positioned above the file system client level, thus there is little to no support for intelligent, data model specific access methods. Rather, structure– and content-aware indexes, as well as layout metadata must be read, then queried by middleware [4]. Additional I/Os are necessary to retrieve the targeted data, and any optimizations based on locality must have been performed when the data was originally written.

Asynchronous services. Middleware performs data compression, implements basic workflows, and performs data management tasks such as indexing. These operations are performed online (i.e., while a file is open) because there is no support from the file system for deferred operations despite many tasks being amenable to asynchronous completion—after a file is closed.

Next we will examine how the file system services described in Section II can be generalized to subsume the common needs of middleware described in this section. It should noted that we make no formal claim toward the generality of our abstractions, but rather seek to demonstrate their applicability in the context of a variety of middleware systems, and expect that over time a common model will emerge, capable of handling all cases.

IV. Data Model Modules

This section presents DataMods, an abstraction over existing storage system services commonly reproduced in middleware. The interface consists of three abstractions. First, the *file manifold* encapsulates metadata management and data placement, allowing middleware to extend inode structures with custom schema information, rule-based placement logic, and domain-specific striping strategies. Second, *typed and active storage* provides a safe mechanism for extending the interface and behavior of the underlying object store. Finally, applications can take advantage of *asynchronous services* to schedule deferred work such as indexing, compression, de-duplication, and basic workflows.

A. File Manifold

A file manifold is a generalization of metadata storage and data placement services, and addresses the needs of middle-

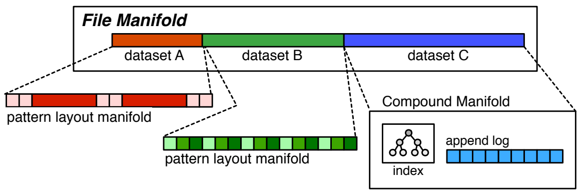


Fig. 1. Example of *file manifold* with three distinct striping strategies in a single file view.

ware 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 a high-level file manifold stitches each sub-file together forming a composite view. Figure 1 illustrates an example of such a file consisting of three datasets. The first two are represented by compact, pattern-based striping strategies with different configurations, and the third uses an index to record data element placement (e.g. the vertex list of a mesh).

While inodes in Ceph may be flexibly sized, in order to remain scalable, they must be kept small enough to be stored inline with directory entries. Thus, applications that require more state, such as a large index, cannot store all meta data within the inode. These applications may store within an inode an object namespace identifier, and rules for locating and translating auxiliary state, stored within one or more objects that are accessed indirectly by clients through the use of the manifold. In Section V we present a case study that demonstrates this technique.

B. Active and Typed Storage

Advanced interfaces that go beyond interacting with typeless binary objects are needed to efficiently support the type of intelligent data access, filtering, and manipulation that middleware libraries perform at a high-level. There has been a large amount of previous work done related to active storage at many levels of the storage system, but each tends to adopt an approach that assumes arbitrary code injection. This presents very difficult challenges related to security and quality-of-service. Rather, we propose that a programming model with well-defined performance costs be exported by the system, and used by applications to construct model-specific computations and custom interfaces. The programming model and an associated run-time environment control access to existing, generalized system services and resources, such as indexing and compression.

C. Asynchronous Services

Compression and indexing are common operations performed by middleware libraries, and are typically performed online while a file is open. However, these types of processes are amenable to being performed asynchronously, reducing the application-level burden, while allowing idle time within the storage system to be exploited for larger-scale, global optimizations. In addition to typical data management routines, middleware implement workflows. For example, the Climate Data Operators offer regridding services, and can be used to compute statistical summaries, a highly efficient yet data intensive function.

Asynchronous services re-use active storage abstractions to define deferred work. Deferred work should be executed based on coarse-grained relationships to middleware actions such as updating an object or closing a file. Using a coarse-grained schedule will provide flexibility to the storage system for coordinating computation to reduce resource contention, potentially taking into account long-term scheduling information from a cluster job management system.

The next section demonstrates how each of these components can be used to construct a real-world example based on optimizing throughput for checkpoint/restart workloads.

V. Use Case: Checkpoint/Restart

Large-scale, long-running computations rely on parallel file systems to save periodic checkpoints of application state. A checkpoint represents a globally consistent view of the computation and can be used to restart an application from a known point, following a failure. Since checkpoints require consistency across potentially millions of threads, computations are generally suspended during a checkpoint. Thus, decreasing the time required to complete a checkpoint can result in immediate increases in compute efficiency. However, a common I/O workload in which all processes write a single file (N-1) is notoriously difficult to optimize due to intra-file serialization.

The Parallel Log-structured File System (PLFS) was developed as a middleware layer to address throughput scalability for N-1 workloads by transparently decoupling writers, and transforming N-1 workloads into N-N workloads, In N-N workloads each process writes to a dedicated log-structured file, and avoids the need for finding specific “magic number” tuning parameters [5]. Since the logical view of the file being written—a single byte-stream—is no longer explicitly maintained, each write must be recorded in an index, and this index must be globally available to all processes when the file is opened for reading in order to identify the log containing a particular byte.

The PLFS middleware is implemented as either a FUSE-based file system, or as an ADIO plugin to MPI-IO, and is designed to sit directly above a standard POSIX file system interface. Index and log-structured files are maintained by PLFS, stored in a special container directory that represents the high-level logical file, and all index creation and compression is performed above the file system interface. The scalability of PLFS can be reduced by two primary factors. First, the size of an index can become very large resulting in increased I/O when writing, and memory pressure from the index required by each client when reading. And second, the number of individual log and index files can grow to put pressure on the underlying file system’s metadata services [6].

A DataMods-based approach to implementing PLFS makes three improvements over the middleware-based architecture.

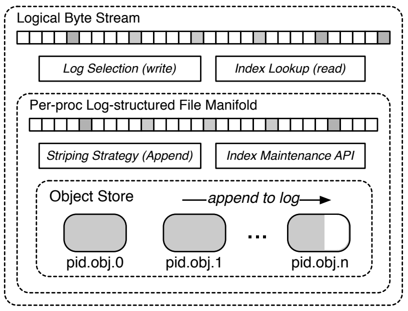
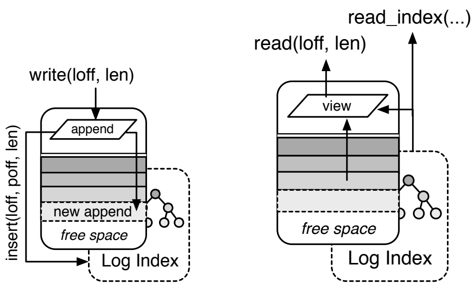


Fig. 2. The file manifold hierarchy. The top-level file view maps writes to per-process logs that are automatically indexed by low-level active objects.



(a) Object write (b) Object view

Fig. 3. Object classes used by PLFS manifold. An index is automatically generated, and logical views can be constructed.

First, index creation and maintenance are handled transparently by the base file system. Second, metadata load is reduced by avoiding the creation of multiple files per process. And third, index consolidation and compression are performed offline, allowing the computation to resume as soon as possible. An overview of the architecture is shown in Figure 2, where a three level hierarchy is illustrated.

A. *Hierarchical File Manifold*

Figure 2 illustrates the multi-level file manifold used in the DataMods implementation of PLFS. The outer manifold, labeled *logical byte stream*, represents a single logical file into which multiple processes are writing checkpoint data, and is identical to the file view presented by PLFS. This file manages metadata such as the number of processes, but does not store any file content. Rather, the outer manifold is recursively defined by a per-process log file, and routes each I/O request to the target, inner file manifold.

Log-structured File Manifold. The core abstraction in PLFS is an auto-indexed, log-structured file that transforms writes at logical offsets into efficient object append operations. The inner manifold implements this abstraction directly on top of intelligent objects that perform automatic indexing, and exports a log-oriented interface and index maintenance routines. In its simplest form, the manifold stripes data across a set of append-only objects that it maintains in a dedicated namespace, using a basic naming scheme to preserve the append order (e.g. O_0, O_1, $\ldots$). Writes at logical offsets are received from the outer manifold and routed to the *current object*. Once a size threshold has been met a new, empty object is allocated to extend the append sequence.

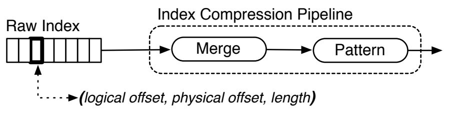
Active and typed storage. At the lowest-level is a custom object class that acts as a building block for the log-structured file manifold, and is responsible for managing the logical-to-physical mapping using an internal indexing facility. Figure 3a illustrates the functionality of the object class when a write is received. A position for the write is determined, and the logical-to-physical mapping is recorded directly in an index structure. In a similar manner, a logical offset is read by using the index to form a view over the data contained in the object, and can be used to return multiple extents that overlap a requested region, potentially useful for reducing I/O round-trips.

B. *Offline Index Compression and Consolidation*

Immediately following the completion of a checkpoint using the DataMods system, opening a file for reading can be expensive: the global index is uncompressed and fragmented across all of the objects composing the file data. To address this issue DataMods asynchronously performs two types of index compression, as well as index consolidation. The top-level logical file manifold signals the underlying asynchronous service when the file is closed by all processes, and the storage system independently schedules the deferred work offline.

Index compression by merging. The set of index entries corresponding to the objects of a process log are compressed using two strategies implemented as a pipeline, shown in Figure 4. The first strategy is *merging*: A PLFS index fundamentally consists of a set of 3-tuples (logical offset, physical offset, length) that map the logical offset of an extent to its physical location. The current version of PLFS performs basic compression by *merging* adjacent entries that correspond to a contiguous logical extent. For example, the index entries for two 100-byte writes at offsets 0 and 100 can be replaced by a single 200-byte index entry at offset 0. This is implemented in PLFS by buffering index file updates, merging when possible, and periodically flushing the buffer. The first stage of the DataMods compression pipeline performs the same merge-based compression, and achieves the same compression ratio as if PLFS used an infinite buffer. However, in practice the periodic buffer flushing will introduce only a small amount of inefficiency to the resulting compression ratio.

Index compression by pattern recognition. The second type of compression is not performed by the current version of PLFS, and utilizes pattern recognition to identify regularity in the I/O pattern, replacing a series of index entries by a compact representation when a series of entries matches the pattern. The pattern that a series of index entries must match is the same as a standard strided I/O pattern, (logical offset, length, stride, count), plus a starting physical offset. Thus, the pattern can be expanded using the formula, $O_l + i * S, i \in [1, count)$, where O_l is a starting logical offset, S is a constant stride size, and $count$ is the number of extents represented by the pattern.



Raw Index
Index Compression Pipeline
Merge
Pattern
(logical offset, physical offset, length)

Fig. 4. Index compression pipeline.

Index consolidation. While index compression reduces and consolidates the index entries of a single log file, I/O costs when opening the logical file can be further reduced by consolidating all indexes into a single file. This option can be turned on in current versions of PLFS when using the ADIO plugin for MPI-IO. By performing the same consolidation offline we avoid the additional overhead of consolidation introduced in the compute cluster. Asynchronous index creation and compression, now decoupled from the original computation, can be scheduled independently by the storage system or integrated into a higher-level cluster scheduling strategy. Next we present results we have obtained from an initial implementation of the log-structured file manifold, and the pattern-based index compression technique.

VI. Evaluation

The preliminary results of using the DataMods abstraction are promising. We implemented the log-structured object class in less than 300 lines of C++, and as re-usable functionality is transformed into native services, this size is expected to shrink considerably. Initial micro-benchmarks using a single OSD show between 11% and 17% throughput overheads for an append workload generated by the low-level object class I/O transformation. However, the workload is generated by a single client using a closed-loop workload, and thus the source of the overhead is likely to be latency. Increasing the load and optimizing physical layout of the index to reduce the number of additional I/Os is expected to reduce the throughput overhead.

We have performed an analytical evaluation of the reduction factors obtained using index compression techniques discussed in the previous section. We used PLFS traces [1] from a mix of 8 applications and I/O benchmarks using between 8 and 512 processes. The reduction factors for both techniques are shown for 82 PLFS traces in Figure 5. For each trace the effect of merging was reported, and the effect of applying the pattern recognition technique after merging. The two curves show the distribution of reduction factors obtained by each technique.

There are three interesting modes present in the graph. At the high end pattern-based compression offers little to no benefit over merging, due to the I/O patterns that have a high degree of sequentiality per process. At the low-end, strided workloads with small writes are incompressible by merging, but the I/O pattern is discovered and pattern-based compression is applied. In the center the distributions are parallel, indicating workloads with large-scale patterns that can be detected, and per-process small sequential writes that can be merged.

[1]http://institutes.lanl.gov/plfs/maps/

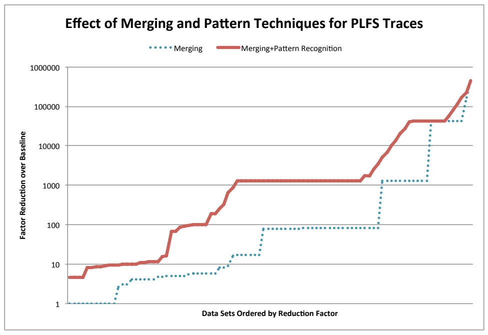


Fig. 5. Index compression ratios for 82 PLFS traces.

VII. Related Work

Previous work has looked at exposing a rich interface to developers for expressing file layout. Yu et al. [7] decoupled MPI-IO writers, and joined the separate data sets using a file concatenation feature found in Lustre, and the Vesta parallel file system has a rich mechanism for specifying file layouts [8], but is restricted to regularly strided I/O patterns.

Recent work has addressed the possibility of alternative storage system abstractions. Lofstead et al. introduce a new transaction metadata service, and suggest that applications using the service may find it more natural to use an object interface [9]. Dries et al. introduce a container abstraction for managing checkpoint data on persistent storage co-located with compute nodes, but the abstraction refers only to the local view of data [10].

There are many efforts underway to introduce computation throughout the I/O stack. Abbasi et al. introduce Data Services that allow data flows to be intercepted and manipulated [11]. Recent work [12], [13], [14] integrates active storage with object-based storage systems, and protocols such as iSCSI and T10, but are low-level facilities for arbitrary code injection. Son et al. raise the abstraction level of active storage up to that of MPI-IO [15], allowing new MPI calls to trigger co-located computation.

VIII. Conclusion and Future Work

We presented DataMods, an abstraction over existing storage system services that allow domain-specific data models to be constructed natively within the storage system. The *file manifold* encapsulates metadata and data distribution, *active and typed storage* allow new low-level behavior to be defined using existing services, and *asynchronous services* support offline computation such as compression and indexing. We showed how the abstraction could be used to duplicate the behavior of PLFS while offloading index construction and pattern-based compression to the storage system.

The current status of the project includes a preliminary implementation of the PLFS use case that we are using to evaluate end-to-end performance. We are currently designing new modules for abstractions found in Hadoop and VTK, and then plan to identify common patterns and formalize the DataMods abstractions.

References

[1] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI '06*, 2006.

[2] S. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC '06*, 2006.

[3] S. Weil, A. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters," in *PDSW '07*, 2007.

[4] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel, "Hdf5-fastquery: Accelerating complex queries on hdf datasets using fast bitmap indices," in *SSDBM '06*, 2006.

[5] 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*, 2009.

[6] A. Manzanares, J. Bent, M. Wingate, and G. Gibson, "The power and challenges of transformative i/o," in *CLUSTER '12*, 2012.

[7] W. Yu, J. Vetter, R. S. Canon, and S. Jiang, "Exploiting lustre file joining for effective collective io," in *CCGrid '07*, 2007.

[8] P. F. Corbett and D. G. Feitelson, "The vesta parallel file system," in *TOCS '96*, 2006.

[9] J. Lofstead and J. Dayal, "Transactional parallel metadata services for integrated application workflows," in *HPCDB '12*, 2012.

[10] D. Kimpe, K. Mohror, A. Moody, B. V. Essen, M. Gokhale, R. Ross, and B. R. de Supinski, "Integrated in-system storage architecture for high performance computing," in *ROSS '12*, Venice, Italy, June 29 2012.

[11] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky, "Extending i/o through high performance data services," in *CLUSTER '09*, 2009.

[12] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *SC '07*, 2007.

[13] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy, "An active storage framework for object storage devices," in *MSST '12*, 2012.

[14] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D. D. E. Long, Y. Kang, Z. Niu, and Z. Tan, "Design and evaluation of oasis: An active storage framework based on t10 osd standard," in *MSST '11*, 2011.

[15] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary, "Enabling active storage on parallel i/o software stacks," in *MSST '10*, 2010.