MarFS, a Near-POSIX Interface to Cloud Objects

JEFF INMAN, WILL VINING, GARRETT RANSOM, AND GARY GRIDER

Jeff Inman is a Software Developer in LANL's High-Performance Computing Division, with surprisingly many decades of research experience in areas including parallelism, bioinformatics, GPUs, compilers, embedded computing, and scalable storage. jti@lanl.gov

Will Vining graduated from the University of New Mexico with a bachelor's degree in computer science in 2016. He is currently a graduate student at LANL and is one of the primary developers for MarFS. wfvining@lanl.gov

Garrett Ransom is a recent employee of LANL's High Performance Computing (HPC) Division. As part of the Infrastructure Team, Garrett performs system administration and assists with the development of storage technologies. gransom@lanl.gov

Gary Grider currently is the Division Leader of the High Performance Computing (HPC) Division at Los Alamos National Laboratory. Gary is responsible for all aspects of High Performance Computing technologies at Los Alamos. ggrider@lanl.gov

The engineering forces driving development of "cloud" storage have produced resilient, cost-effective storage systems that can scale to 100s of petabytes, with good parallel access and bandwidth. These features would make a good match for the vast storage needs of High-Performance Computing datacenters, but cloud storage gains some of its capability from its use of HTTP-style Representational State Transfer (REST) semantics, whereas most large datacenters have legacy applications that rely on POSIX file-system semantics. MarFS is an open-source project at Los Alamos National Laboratory that allows us to present cloud-style object-storage as a scalable near-POSIX file system. We have also developed a new storage architecture to improve bandwidth and scalability beyond what's available in commodity object stores, while retaining their resilience and economy. In addition, we present a scheme for scaling the POSIX interface to allow billions of files in a single directory and trillions of files in total.

HPC Storage Challenges

The issues faced by extreme-scale HPC sites are daunting. We use Parallel File Systems to store data sets for weeks to months, with sizes in the 100s of terabytes, and bandwidth on the order of 1 TB/sec. On the other hand, our parallel archives are used to store data forever, but can only support speeds of 10s of GB/sec. MarFS was designed to provide an economical middle-ground between the expensive capacity of PFS and the expensive bandwidth of tape, storing data sets for years, with speeds of 100s of GB/sec.

The supercomputers generating the data that is ultimately stored in MarFS are currently in the millions of cores, and multiple PBs of memory, and are expected to grow to a billion cores and 10s of PBs of memory beyond 2020. Applications that produce one file per process on such machines could produce billions of files, which a user may want to keep in a single directory. Furthermore, as we push to add value to the data we store, we expect file-oriented metadata to grow by perhaps orders of magnitude. The goal is for MarFS to easily handle up to multi-PB-sized data sets, as well as metadata for billions of files in a single directory, and 10s of trillions of files in aggregate.

Modern "cloud" storage systems provide a way to scale data storage well beyond previous approaches, using sophisticated, highly scalable erasure-coded protection schemes. These systems would allow us to build very reliable storage systems out of very unreliable (and therefore inexpensive) disk technologies. The metadata underlying cloud storage is basically a flat metadata space, which also scales very well. Reliability, economics, and scalability combine to make this technology appealing to many large-data sites. For HPC, the problem with these storage systems is that they only provide simple get/put/delete interfaces using object-names, rather than POSIX file-and-directory semantics (files, directories, ownership, open/read/write/close, etc.), and most HPC datacenters need to support legacy applications that rely on POSIX semantics. It became clear from a market survey that other products that provide POSIX-like access to scalable cloud objects were not designed to handle PB-sized files, or billions to trillions of files.

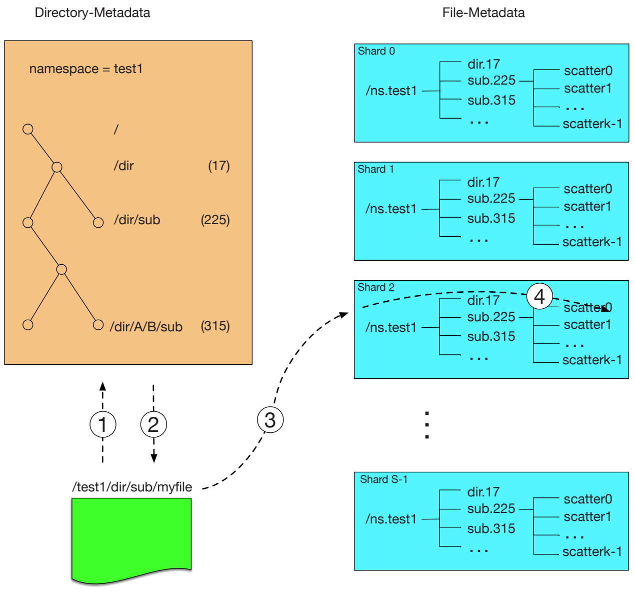

Figure 1: Storing metadata (MD) for a new file having path `/test1/dir/sub/myfile`. A directory-MD Server (dMDS) holds directory MD, and a set of file-MD Servers (fMDS) hold parts of the file MD. (1) The dMDS is consulted for access-permissions (if not in cache). (2) The dMDS also returns the inode of the leaf directory (e.g., 225). (3) A hash of the file-path, modulo the number of fMDS shards, selects the shard to hold this file MD. (4) The file-path hash, modulo number of internal "scatter" directories, identifies the internal subdirectory for the MD.

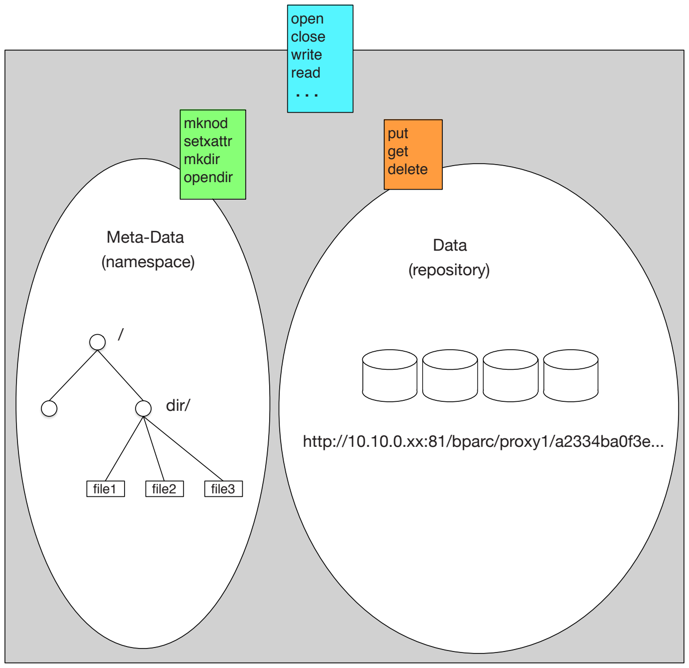

Figure 2: The default metadata scheme uses a regular POSIX file system to represent files, with object-storage holding file contents. The file system must support sparse files and extended attributes. Data and metadata schemes are installed as modular DAL and MDAL implementations, respectively.

MarFS is an open-source software technology developed at LANL to bridge this gap, putting a highly scalable POSIX metadata interface on top of highly scalable cloud object systems, making object storage systems usable by legacy applications. MarFS scales data capacity and bandwidth by splitting data across many objects, or even many object systems. For metadata, MarFS is designed to scale capacity and bandwidth in two dimensions. Currently, directory-metadata is scaled by simple directory decomposition high in the tree. We've developed a prototype file-metadata service, where we've demonstrated scaling metadata by sharding it across many file systems, as illustrated in Figure 1. This metadata sharding is not yet in use in the production version of MarFS.

MarFS Implementation Overview

Figure 2 shows the basic components of MarFS. There is a metadata implementation that handles file and directory structure, and a data implementation that stores file contents. In the default metadata implementation, user directories are implemented as regular directories, and user files are represented as sparse files truncated to the size of the corresponding data, with hidden extended attributes that hold system metadata (e.g., object-ID). This gives us basic POSIX access-control "for free."

Object-storage systems typically have a range of object-sizes for which internal storage and/or bandwidth is optimal. When storing data for files larger than this, we break the data up into distinct objects ("chunks"), transparent to the user. We refer to such multi-object files as "multi-files." Allowing data to be inserted or deleted in the middle of a multi-file (or to create sparse files) would require metadata machinery that would compromise the performance and scalability of parallel accesses. Therefore, we don't allow it. This makes us "not quite POSIX," but we gain trivial stateless computation of the object-ID and offset corresponding to any logical offset in a file, maintaining efficiency for parallel reads and writes.

Millions of small files pose another kind of metadata hazard in that they may invisibly consume significant resources from the object-store. We work around this by transparently packing many small files together into a single object, although they appear to users as distinct files. The packing is done dynamically, during data-ingest, by `pftool` (discussed below), so the packed files will typically be found together in a directory traversal, and are likely to be deleted together, avoiding packed files with many "holes." Nevertheless, we are also developing a repacker, so that multiple "Swiss cheese" packed files can be repackaged into fewer objects.

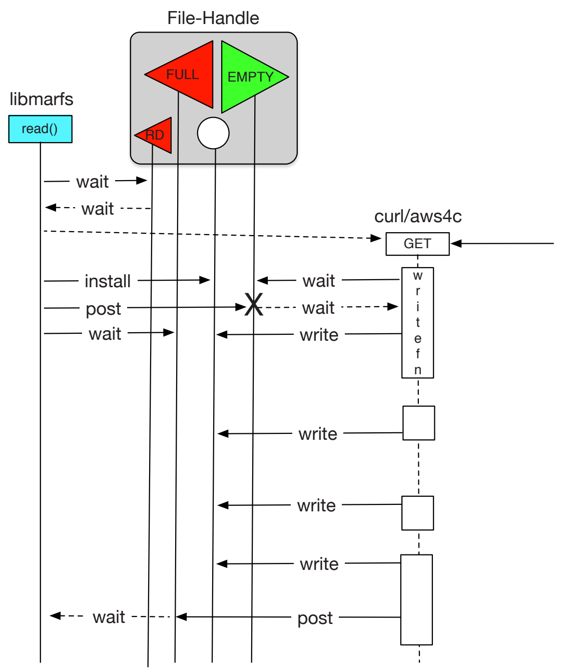

Figure 3: Sequence diagram showing interactions between a user performing a read, a file-handle containing locks, and a thread performing a GET operation on an object. The GET thread receives callbacks from `libcurl` and uses locking to coordinate across multiple `read()` calls. The colors of the locks (red/dark gray = FULL and RD, green/light gray = EMPTY) in the file-handle are represented at the moment marked "X". The circle in the file-handle is a pointer to the caller's read-buffer.

The internal MarFS data-interface must translate between the POSIX file-system semantics seen by applications (open/read /write/close) and the RESTful semantics of an object-store (get /put/delete). We do this by assigning an ongoing GET or PUT transaction to a thread at "open"-time (or at the time when data is first read-from/written-to an object). This thread can block in the `libcurl` callbacks that move data on behalf of the transaction. MarFS read or write requests then provide buffers that allow the callbacks to unblock for long enough to write data from a caller's write-buffer to a PUT, or receive GET data into a caller's read-buffer, before blocking again. When object-boundaries are crossed in a multi-file, MarFS transparently ends one transaction to the old object and begins a new transaction to the corresponding second object. This is depicted in Figure 3.

MarFS is driven by a configuration-file, allowing specification of details like the layout of namespaces and repositories, object chunk-sizes, resource quotas, types of access that are enabled, file systems used for metadata, etc.

Flexibility

Our initial development utilized an object store supporting the S3 protocol, but we are now in production with a Scality RING, using Scality's `sproxyd`. This protocol eliminates the need for maintenance of some internal S3 metadata, improving bandwidth. However, in our relentless quest for economical capacity and bandwidth, we have developed an alternative to cloud-style object-storage, doing our own erasure coding and storing the coded parts in distinct ZFS pools, which themselves are also erasure protected, forming a two-tier erasure arrangement.

Intel's Intelligent Storage Acceleration Library (ISA-L) provides an efficient implementation of Galois Field erasure code generation, allowing an arbitrary number of erasure blocks to be generated for a set of data blocks. Up to that number of corrupted blocks can then be regenerated from the surviving data and erasure blocks. We wrapped ISA-L functionality within a utility library (`libne`) to provide POSIX-like manipulation of sets of data and erasure blocks through higher-level `open`, `close`, `read`, and `write` functions. For example, data provided to the high-level `write` function is subdivided into N blocks. The functions of ISA-L are applied across the N data blocks to produce E additional erasure-code blocks, making a "stripe" of N+E blocks. The stripe is then written across N+E internal files, with one block per file.

The failure tolerance of the system depends on the number of erasure blocks produced. Given (N+E) blocks written with `libne`, we can survive the complete loss of up to E blocks of any stripe. If desired, checksums are also calculated across each block, providing a means of identifying corrupted blocks while reading, and are stored within either the parts themselves or in their extended attributes. Both N and E are configurable, allowing for a customized balancing of the tradeoffs between computation overhead and reliability.

Should a problem be detected, whether that be in the form of a corrupted block, offline server, failed disk, or a failed checksum verification, the erasure utilities will continue to service read requests by automatically performing regeneration on the fly. Such reads will also return an error code, indicating the blocks that are corrupt or missing, but will not attempt to repair the stored data itself. This approach preserves information about failures while avoiding interference with other ongoing accesses.

The Data and Metadata Abstraction Layers (DAL/MDAL)

The desire to experiment with swapping out storage-protocols leads us to the idea of a Data Abstraction Layer (DAL). This is an abstract interface to internal RESTful storage functions (e.g., GET, PUT, and DELETE), which can be implemented and installed in a modular way, swapping out the storage component of Figure 2. We have used this approach to provide a new kind of scalable data-store based on `libne`, where erasure-coded blocks are written across a set of independent file systems. This should allow us to overcome the overhead of the internal communication and metadata management required of an object-store, improving our overall storage throughput without compromising reliability. We refer to this architecture as multi-component storage.

We refer to a storage-server and its associated JBODs as a Disk Scalable Unit (DSU). A DSU holds one or more *capacity units*, and each capacity unit hosts an independent ZFS pool. All DSUs have an identical configuration of capacity units. So, to expand capacity, one would add an identical new capacity unit to every DSU. ZFS provides its own erasure encoding and checksum protection for each data and erasure block, but it remains vulnerable to large-scale failures. To maximize resilience and bandwidth, each of the N+E files of a stripe is written to a different DSU, all on the same-numbered capacity unit. Thus, we can survive the complete loss of any E DSUs in the set of N+E that hold an object.

The parallel nature of this design allows for independent read /write operations across each of the ZFS systems, without the opacity and overhead of an object store. Our expectation is that this architecture will provide improved bandwidth, with more than sufficient reliability.

Multi-component (MC) storage is realized as an implementation of the Data Abstraction Layer, utilizing `libne` to perform low-level accesses. The MC DAL depends on a directory tree of NFS mounts, which groups capacity units (hosting ZFS pools) into DSUs, and DSUs into *pods*, as shown in Figure 4. A pod is just a set of N+E DSUs, where N and E are the parameters of the erasure coding used in the repository. The blocks of a stripe are written across a pod, starting at some DSU and wrapping within the pod.

To reduce the number of files in any one of the internal directories of the individual storage systems, we add another layer of k sub-directories (scatter0, scatter1, etc.) inside each ZFS pool. For a repository that has 3+1 erasure coding, two pods of four DSUs, and two capacity units per DSU, the directory *scaffolding* might look like this:

```
/repo3+1/pod[0..1]/block[0..3]/cap[0..1]/scatter[0..k-1]/
```

To determine the location of the blocks for an object, we compute a hash of the object-ID and use that to fill in the pod and scatter-directory, in a path-template provided by the MarFS configuration. For new data, computation of the capacity-unit may follow from policy guidance (e.g., favor newly added capacity, or spread load in a given ratio) rather than a simple hash. Filling-in these fields of the scaffolding template produces a new template (shown below), which is used by `libne`, along with a starting block (also computed from the hash), to write the object across the N+E independent storage systems in the selected pod:

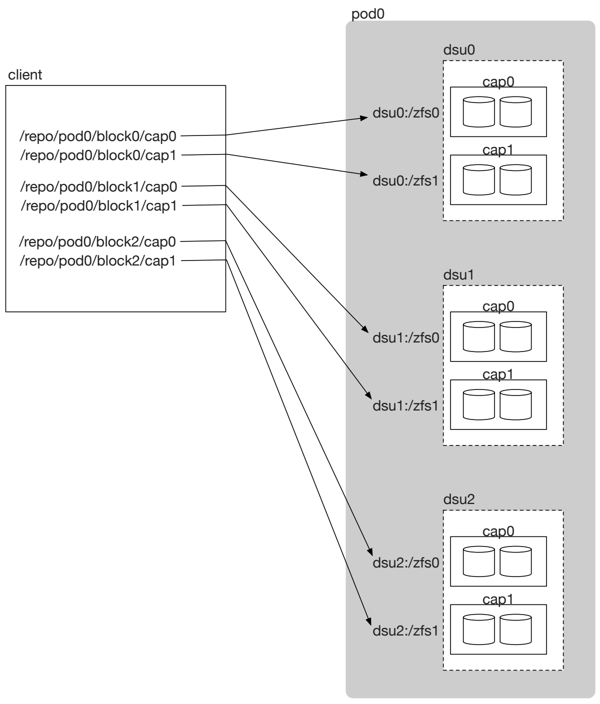

Figure 4: NFS mounts and exports supporting the multi-component DAL. This example shows a single "pod" of 3 DSUs (e.g., N=2, E=1), each having two capacity units. The capacity units each host a single ZFS file system which is exported via NFS. On the client, NFS mounts are made to each of the exports. A stripe of three blocks would be written across the DSUs. The *scatter* directories are internal to the ZFS file systems and are not shown here.

```
/repo3+1/pod1/block%d/cap1/scatter7/object-id
```

In stripes where some blocks are all-zero, ZFS can store the zero blocks much more compactly. By computing the starting block from the hash, we can ensure that capacity is utilized at roughly the same rate in each ZFS pool; otherwise, the capacity in block0 might be used up more quickly if a large number of small objects are created. For access to existing data for which the capacity unit can't be predicted from metadata (e.g., from the creation-date), we will generate a set of paths covering the available capacity units and issue *stat* requests to all of them in parallel.

The MC DAL is configurable on per-MarFS-repository basis, allowing for different storage configurations to be used simultaneously. The configurable parameters are the path template, the number of pods, the erasure parameters (N and E), the number of capacity units per DSU, and the number of scatter directories in each capacity unit.

Multi-component storage provides a high level of data integrity through two layers of erasure coding; data on any individual disk is recoverable in two decoupled erasure regimes. ZFS allows recovery of individual blocks, and data-blocks are stored along with erasure-blocks across ZFS pools. Even moderately sized multi-object files will tend to have objects in all pods. Because the pods are independent, we could lose E pools from each of the pods without data loss.

In conjunction with `libne`, the MC DAL can read through missing blocks or corrupted data. Errors are detected when an object is read. When that happens, the object-ID is flagged as degraded and logged to a file so the object can be rebuilt, either by an offline program run by an administrator, or by a daemon that is notified when there is rebuild work to be done.

We also support a Metadata Abstraction Layer (MDAL), allowing modular replacement of the metadata system. This is how we would swap-in something like the scalable MD system of Figure 1, replacing the metadata implementation in Figure 2.

Metadata Performance

MarFS teammates wrote an MPI application to measure pure metadata (MD) performance and scalability in the forward-looking scheme of Figure 1. The goal was to benchmark only internal MD activity, ignoring any overhead associated with the persisting of data or metadata. Thus, we installed a "no-op" DAL that does nothing for data-write operations, and an MDAL that integrates with the application. Specific MPI ranks acted as clients, file-MD shards, a directory-MD shard (one instance only), or as the master. File and directory MD were stored in tmpfs. Clients performed scripted MD operations, organized by the master rank.

Using 8800 * 16 cores, and one MPI rank/core, we were able to create approximately 820M files/sec, and we stored 915 billion files in a single directory. Because the MD is distributed, and resides in a broad directory-tree per shard, a *stat* of any one of these files can return quickly. We are exploring semantics for parallel *readdir* and *stat* in this model.

Data Performance

Our production hardware uses SMR drives everywhere, and there has been concern about sustained throughput in this technology. On an object-storage testbed with 48 DSUs, we were able to achieve 28.5 GB/sec, for sustained low-level writes. With production workloads on similar hardware (but with incomplete JBODs), we are typically seeing less than 15 GB/sec. To support the pre-tape tier of the storage hierarchy for the new Trinity supercomputer, this is less-than-hoped-for performance. The multi-component architecture was developed to boost bandwidth, while also increasing reliability.

We are building a new testbed with 12 DSUs. There, we will debug and benchmark the MC DAL back end in a 10+2 configuration to prepare for a transition to production, where the 48 DSUs will be treated as four pods of 10+2.

Parallel Data-Movement with `pftool`

`pftool` is an open-source tool for moving data in parallel from one mounted file system to another and is the de facto production workhorse for performing data-movement at scale between storage systems at LANL. Moving data is coordinated by a scheduler which distributes subtasks to worker processes scattered across a cluster. As workers become idle they are given new subtasks, including performing one portion of the parallel traversal of the source-directory tree (returning sets of source-files for copy/compare as new subtasks) or executing one such copy/compare subtask. For large files, a copy/compare subtask can refer to a set of offset+size "chunks" of the large file to be copied, allowing large individual files to be copied in parallel, as well. `pftool` coordinates with file systems to choose this chunk-size. For MarFS, this means large files are broken into chunks that match up with back-end objects in a multi-file, and a special exemption from our sequential-writes-only rule is granted.

The subtasks are executed independently of each other and are asynchronous with respect to the scheduler. If the overall operation fails or is cancelled, it can be restarted and will efficiently resume with any portions of the work that were not previously performed. The duties of the scheduler are light (dispatching subtasks from a work-queue), so the scheduler doesn't become a bottleneck even at very large scales. The result is a self-balancing parallel data-movement application.

Future Work

We are exploring several new development paths, including the MD scalability of Figure 1, `pftool` extensions to allow cross-site transport, custom-RDMA protocols to improve storage bandwidth, and power management schemes for cold storage.

Acknowledgments

We thank our hard-working teammates. Thanks to Rik Farrow for helpful comments. This work was done at the Los Alamos National Laboratory, supported by the US Department of Energy contract DE-FC02-06ER25750. This publication has been assigned the LANL identifier LA-UR-16-28952. MarFS can be downloaded at https://github.com/mar-file-system/marfs. A commercial product using MarFS has been announced, with Lustre for metadata and SpectraLogic Black Pearl for data storage.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Become a USENIX Supporter and Reach Your Target Audience

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

Learn more at:
www.usenix.org/supporter