Object storage semantics for replicated concurrent-writer file systems

Philip Carns, Robert Ross, and Samuel Lang

Argonne National Laboratory
Argonne, IL 60439
{carns,rross,slang}@mcs.anl.gov

***Abstract*—High-performance I/O is a key requirement for many of today's critical computational science applications, and parallel file systems are being driven to progressively larger scales to keep pace with demand. One cost-effective way to meet this demand is through the deployment of commodity storage hardware in conjunction with file systems that provide software resiliency. This requires a re-evaluation of the core components of parallel file system architecture, however. In addition to interacting with resilient protocols, parallel file systems must also take into account unique HPC workloads that include bursty, highly concurrent access to large shared files. Such workloads are traditionally a challenge for software replication algorithms, in part because the underlying storage does not provide convenient semantic building blocks. In this work we isolate a common component of many parallel file systems, the object storage abstraction layer, and propose the introduction of semantic properties that will enable it to better serve as the building block for resilient HPC storage architectures. The properties that we have identified are atomicity, explicit versioning, and commutativity. We outline how these properties can be used to simplify software replication protocols for highly concurrent workloads. We also demonstrate that these properties can be implemented portably while still maintaining high performance on both commodity and enterprise-class storage platforms.**

I. Introduction

High-performance I/O is a key requirement for many of today's critical computational science applications. These applications perform simulations in fields as diverse as physics, energy, earth science, climate, chemistry, and biology. They also perform I/O for a variety of reasons, including defensive I/O (checkpointing), simulation output, out-of-core computation, and analysis. The diverse demand for high-performance I/O has driven storage systems to progressively larger scales in order to keep pace with computational performance and facilitate scientific discovery. As a result, parallel file systems are one of the most important components in the system software stack for HPC. Parallel file systems are responsible for aggregating storage hardware, providing a consistent name space for persistent storage, coordinating accesses from a large number of application processes, and tolerating component failures.

Several modern parallel file systems, including PVFS [1], Lustre [2], PanFS [3], and Ceph [4], utilize a shared-nothing architecture in which multiple servers provide parallel access to a collection of independent storage devices[1]. This approach is popular because it offers incremental scalability and is compatible with both commodity and enterprise-class storage hardware. An example of this architecture is shown in Figure 1. Clients access file servers using file-system-specific protocols, while the file servers in turn relay data to local storage. A common design feature in this architecture is the abstraction layer between each file server and its storage devices. Most shared-nothing parallel file systems elect to use an object storage model at this layer. Objects are referenced by unique integer identifiers, and each object stores a stream of byte-addressable data along with some number of named attributes. This model shares many features with the T10 OSD specification [5], but most current file systems elect to implement the model as a software abstraction on top of standard block devices or local file systems rather than using native T10 targets.

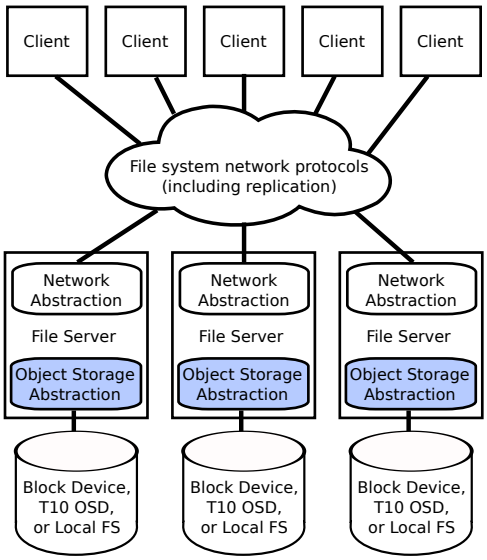


Fig. 1. Shared-nothing parallel file system architecture

Parallel file systems, and therefore their underlying object storage abstractions, must be tailored to the I/O workloads that are prevalent in HPC. Studies have consistently shown that parallel computational science applications exhibit unique I/O workload characteristics. They often access large files, write much more often than in other domains, and interleave access to files with interprocess spacial locality [6]. In addition, the

[1]PVFS and Lustre may use shared storage for failover purposes, but this configuration is not required

access patterns come in heavy bursts and the size of requests vary dramatically [7], [8].

These workload patterns have been well known for many years, but they continue to be influenced by emerging trends as systems are pushed toward exascale. Concurrency will increase as a larger number of compute cores are deployed [9], [10]. Commodity storage is also likely to play a larger role in order to reduce the cost of future HPC deployments. Failure rates will increase as a direct result of these two trends [11], [12]. We therefore expect future parallel file systems to rely on software replication algorithms in order to service unprecedented numbers of concurrent I/O requests while gracefully handling the failure of unreliable storage devices.

The semantics of existing object storage abstractions are not ideal for efficient software replication under the workloads presented by HPC, however. The most significant challenge lies in maintaining object consistency in the face of highly concurrent write workloads. In this work, we identify a concise set of new object storage semantics aimed at simplifying that task. In addition, we build a prototype object storage abstraction to demonstrate that these semantics can be implemented portably in software on any Unix file system without additional hardware or kernel modifications. We analyze the performance of the prototype in order to quantify the overhead introduced for typical HPC workloads at the object storage level. Our goal is to establish a high-performance building block for use in the construction of highly concurrent software replication protocols.

The remainder of the paper is organized as follows. Section II outlines the proposed semantics and motivates their use as building blocks for software replication in a shared-nothing, high-concurrency file system. Section III describes the implementation of a prototype object storage abstraction that incorporates those semantic properties. Section IV evaluates the performance of this abstraction layer for concurrent I/O workloads. Section V describes previous related research. Section VI summarizes our findings and outlines avenues for future work.

II. Shared-Nothing Object Storage Semantics

Although it is possible to export an object storage interface over the network, we are focusing on local storage abstractions in this work. We therefore assume that any network communication or replication is implemented at a higher level by the parallel file system itself, as illustrated in Figure 1. The role of the object storage abstraction in this environment is simply to serve as a building block that provides persistent storage functionality and facilitates the construction of software replication protocols. In order to serve this purpose effectively, however, it must take into account the environment outlined in Section I. Multiple clients are likely to apply bursts of writes concurrently to the same object. The writes may be assembled into a pipeline of smaller operations in order to improve network, storage, and memory utilization. However, a server node or its storage devices may fail or lose power at any time. Thus, writes must be durable; and it must be possible to construct new replicas or synchronize old replicas based on surviving objects.

Most object storage abstractions provide base semantics analogous to those defined by the T10 OSD specification. We have evaluated those semantics and identified three key enhancements for use in this environment: atomicity, explicit versioning, and commutativity. The following subsections outline the rationale for each enhancement in the context of write operations within a distributed replication environment.

A. Atomicity

The concept of atomicity is straightforward in this context: Each object write operation must be applied to an object in its entirety or not at all. This semantic eliminates the possibility of partially applied updates being visible in an object following a hardware, storage software, or application software failure. Replication is therefore simplified because all objects are always in a well-defined state. The current T10 OSD standard does not provide atomicity of this nature for arbitrary write operations. It instead provides a weaker guarantee on the minimum number of bytes that will be committed as a group at any given time.

We propose treating object write operations as transactions. In order to facilitate HPC workloads, the object storage abstraction should allow multiple transactions to be opened simultaneously. It should also support pipelined operations by presenting explicit API functions to open, close, and abort transactions so that multiple writes can be grouped together as a single atomic update. This minimizes the transaction overhead for large operations and allows the atomicity to extend over regions that are not constrained by individual buffer size. In nonpipelined cases, optional flags can be provided as a convenience to automatically encapsulate individual writes in transactions.

B. Explicit versioning

Explicit versioning means that each write transaction should be associated with an explicit, sequential, *caller-selected* version number. This semantic simplifies replication by allowing the file system to coordinate version numbers across replicas and ensure that the same modifications are applied consistently on each one. The version numbers should be unique to a given object identifier (rather than global to all objects) so that coordination is limited strictly to the servers participating in the replication of a given object. In addition, each object should maintain a history of which version numbers have been applied to an object and what regions of the current object they correspond to. This information can optimize replica reconstruction by serving as a mechanism to identify the minimum set of updates that must be applied to a replica in order to synchronize it with a peer. The standard T10 OSD specification does not include any integrated versioning functionality.

The object storage abstraction can provide this functionality by using version numbers as object transaction identifiers. It can also provide API functions to retrieve the next unused

version number as well as the current overall version state of an object. The file system should make a best effort to assign sequential version numbers, but the object storage abstraction should allow version numbers to be skipped in order to accommodate failures or stalled operations without delaying subsequent operations. Because version numbers are sequential in most cases, the overall version state of an object can be compactly expressed as the highest version number applied to the object along with a list of version number extents that are missing from the object.

C. Commutative updates

The naive approach to maintaining consistency when writing to a replicated object is to serialize all write operations and apply them in a deterministic order to all replicas. However, this approach introduces artificial delays and makes poor use of storage resources that can achieve higher performance if multiple write operations are applied simultaneously. We instead propose that write updates should be commutative, meaning that they can be applied in any order (and concurrently) as long as the end result deterministically produces the same result as if they had been applied sequentially.

In order to do this, the object storage abstraction can use the version numbers of each transaction to enforce the Thomas Write Rule [13] with byte-level granularity. The object storage abstraction can achieve this by performing write operations in two steps. The first step is to store the entire write buffer persistently and durably on disk. The second step, performed at transaction close time, is to compare the version number of the write against the existing version number for the object region that is being modified. The new data is made visible to readers only if it has a higher version number than that of the data (if any) already present. If the transaction version is too old, then it is discarded. This, in conjunction with the atomicity described earlier, implies that writes are not only commutative, but also idempotent. Duplicate stale writes are guaranteed not to change the state of an object. The T10 OSD standard does not provide commutative operations.

Commutative, idempotent updates provide a key advantage in implementing replication protocols. Writes from any client can be applied immediately to any replica, even if they arrive in a different order or are duplicated as a result of network retransmission. This allows the file system to maximize concurrency while still maintaining consistency. Replicated objects using this functionality may temporarily diverge before arriving in the same state. It is up to the file system implementer to constrain this divergence according to the semantic requirements of the file system.

D. Read semantics

The preceding subsections dealt exclusively with the semantics of write operations. For read operations we advocate the use of more relaxed semantics. Reads need not be atomic or versioned and therefore do not need to be encapsulated in transactions. Instead, we propose the semantic guarantee that the results of all closed write transactions are immediately visible to readers. If a write transaction is closed while an overlapping read is in progress, then the results of the read are undefined. The implication of this semantic decision is that read operations are never required to block on any other concurrent operations. This simplifies the implementation of file system read operations and allows for maximum concurrency. Note that this read behavior is sufficient to implement standard MPI-IO semantics as well as PVFS's nonconflicting write semantics [14], both of which have proved successful in production HPC deployment. Stricter read semantics could be implemented for other environments.

E. Potential replication protocols

The I/O semantics described in this work can be used to implement a variety of replication protocols. We believe that this design decision should be made at the file system level rather than dictated by the object semantics, however. We do not intend for underlying version numbers or semantics of the object storage devices to be exposed to end users.

The semantic choices for replication range from traditional pessimistic protocols such as two-phase commit [15], which would delay closing object transactions until the commit phase, to more relaxed protocols based on an eventual consistency model [16]. The latter is a particularly compelling model because it does not require that the concept of atomicity be extended to span replicas. Eventual consistency inherently allows state drift across replicas.

Multiple architectural choices are available for replication as well. Replication may be client driven, in which case clients are responsible for transmitting redundant data [3]. Primary-copy replication may be more appropriate in conjunction with the object semantics described here, however, because primary servers can assign sequential version numbers for an object without performing additional communication steps. It also conserves client to server bandwidth, which is a constraining resource on many large-scale systems [17]. Placement strategy is another important component of the overall replication architecture, but we have deliberately omitted it from our discussion of local storage semantics. A number of well-known replica placement algorithms are compatible with our approach [18], [19].

Object storage abstractions typically provide access to byte-addressable data streams as well as named attributes. In this work we are evaluating these semantics in the context of the former, though the same semantics could be applied to either access method.

III. Prototype Implementation

We have constructed a new object storage abstraction prototype, called the Versioned Object Storage Device (VOSD), in order to evaluate the feasibility of implementing the semantic extensions described in the previous section. The VOSD presents a user-space API for accessing data using an object storage model. It also exposes the transaction and version number extensions as described in Section II.

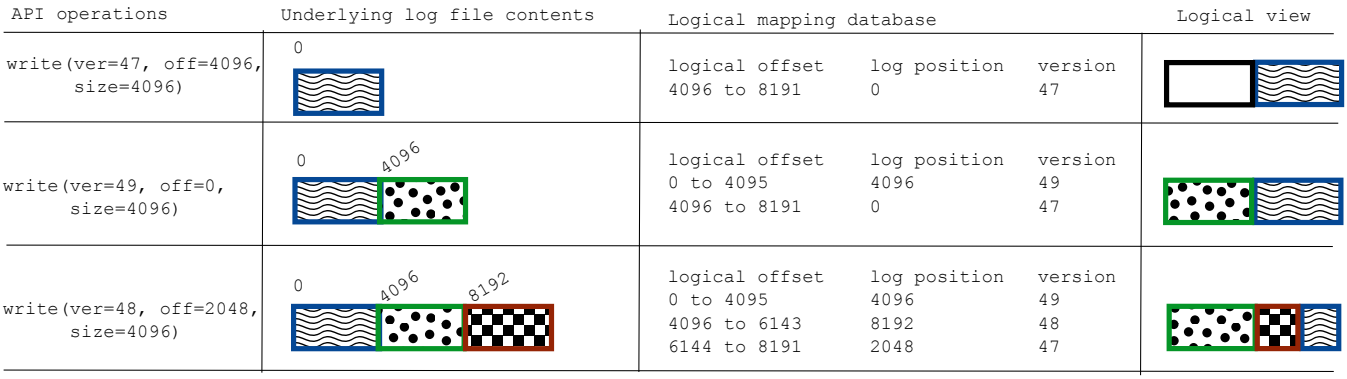


Fig. 2. Example sequence of three VOSD writes

The current VOSD implementation stores data for each object in a normal POSIX file, while metadata for all objects is stored in a Berkeley DB database. The VOSD API itself does not rely on any POSIX file or Berkeley DB conventions and could be implemented by using a variety of other libraries or resources. This approach is architecturally similar to the Trove abstraction used in the PVFS file system. The most important design distinction (which allows it to implement atomicity, versioning, and commutativity) is that the logical view of each object is decoupled from how the data is stored in the underlying POSIX file. More specifically, the data for each object is stored in a log-structured manner [20]. The Berkeley DB database is used to maintain a mapping of logical object extents to log offsets.

Each write operation is assigned the next available position in the log for the target object. The data is then written into the log. The updated data is not visible to readers, however, until the transaction closes. At that time, a read/modify/write transaction is performed on the Berkeley DB database to check the previous version numbers for the affected logical regions. If the new write has a higher version number, then the database is updated to map the logical offset to the new log position and version number. If the version is too old for a given region, then no change is made. This behavior means that an arbitrary number of concurrent writes can be transferred to disk simultaneously. Conflict resolution is delayed until transaction close time, at which point low-latency database transactions are used to enforce consistency and apply the Thomas Write Rule with byte-level granularity. Existing data is never overwritten within the log.

Reads are serviced from the same log that data is written to. Because the data is not stored in canonical order, read operations must consult the Berkeley DB database first in order to determine what portions of the log must be accessed in order to service a given read request. The database uses a custom sort order for keys to ensure that the logical mapping exhibits spacial locality based on object identifiers and logical offsets. This allows for range queries to be performed quickly using Berkeley DB cursor operations. It also improves Berkeley DB cache performance and minimizes the likelihood of extraneous disk access during concurrent read operations.

Figure 2 illustrates an example in which a sequence of three write transactions is applied to a VOSD object. The first write arrives for offset 4096, but it is applied to position zero in the log. The logical view of the object reflects that it is a sparse object at this point. The next write then arrives for offset 0. It is applied to the next available log position (4096). This write does not conflict with any previous write operations. Therefore all data is now made visible in the logical view of the object. The third write operation spans from offset 2048 to 6143, it partially overlaps with both of the previous write operations. However, its version number (48) is only high enough to obsolete one of the two previous write operations (versions 47 and 49). This is reflected in the final logical view, in which only a portion of the data from the third write operation becomes visible in the logical view of the object. Note that the logical mapping and log position of the first write (version 47) has been updated to reflect that the first half of that segment should no longer be visible to readers.

The final logical view of the object in Figure 2 is identical to the outcome that would have been produced by a strict sequential ordering based on version number. The sequential approach would have required the second write operation to delay until the third one had been applied. In contrast, the VOSD allowed all writes to proceed immediately by leveraging its commutative property. The versioning history is implemented by storing the version number for each extent alongside the mapping information in the database. The atomicity semantic is implemented by relying on Berkeley DB atomicity and requiring that all data be written to disk before updating the logical mapping.

The VOSD must also provide the ability to abort transactions as part of its API. Recall that we do not commit mapping information to the internal database until a transaction is closed. Thus, a transaction can be discarded at any time by simply releasing an in-memory data structure that accumulates pending updates to the Berkeley DB mapping database. Any data that has been written to the log is simply left in place. Because this data is not visible to readers, no explicit “undo” step is required.

The log-structured layout was chosen specifically to aid in implementing the proposed semantic features, because it

allows the VOSD to store an arbitrary number of concurrent write updates before deciding whether to make them visible or not. We elected to store the log for each object in a separate Unix file in order to simplify the architecture. Objects can be deleted by simply unlinking the underlying file and removing its corresponding keys from the Berkeley database. Additional Berkeley databases are maintained alongside the logical mapping database in order to speed retrieval of the current version number and the list of missing versions for any given object.

An important consideration for any log-ordered data storage system is when (or whether) to reorder the contents of the log. There are two primary motivations for reordering data after it has been written. The first is to harvest obsolete log entries in order to reclaim disk space. An example of this scenario can be seen in Figure 2. The final logical view of the object is only 8 KiB, but the underlying log for the object is 12 KiB. The log could be reordered in this case to free up the 4 KiB of unused data. The second motivation for reordering is to place the data in a more optimal layout for subsequent read operations. A canonical order, for example, would be more efficient if the data is expected to be read later by a serial application.

Previous workload studies have indicated that HPC jobs rarely write to files created by other jobs [6]. Hence, reordering in order to reclaim overwritten data may not often be required. In contrast, however, we expect reordering for the purpose of read optimization to be a significant concern. This is true particularly for analysis applications that use data access patterns that are unrelated to the access pattern used to write simulation output. In this scenario we plan to take advantage of idle time on the file system to automatically reorder data in the background. Other workload studies have shown that HPC I/O traffic tends to occur in intense bursts, with significant idle periods between bursts [7]. We will investigate log reordering in the context of storage systems that can identify idle periods and inform the VOSD of appropriate times to perform reordering operations.

IV. Evaluation

Our goal in evaluating the VOSD prototype is to determine whether the new semantic features can be provided efficiently in the presence of HPC I/O workloads. To do this, we compare the VOSD to two existing object storage abstraction implementations: the Trove component of the PVFS file system and the ObjectStore component of the Ceph file system. Both of those implementations are used by user-space file servers to abstract access to local storage devices. Trove does not offer atomicity, versioning, or commutativity, but it has been successfully deployed in a variety of production environments and is also the basis for several research efforts [17], [21], [22]. The Ceph ObjectStore improves upon Trove semantics by offering atomicity. It also serves as the foundation for software replication in Ceph. Ceph is still in development, however, and is not yet recommended for production use.

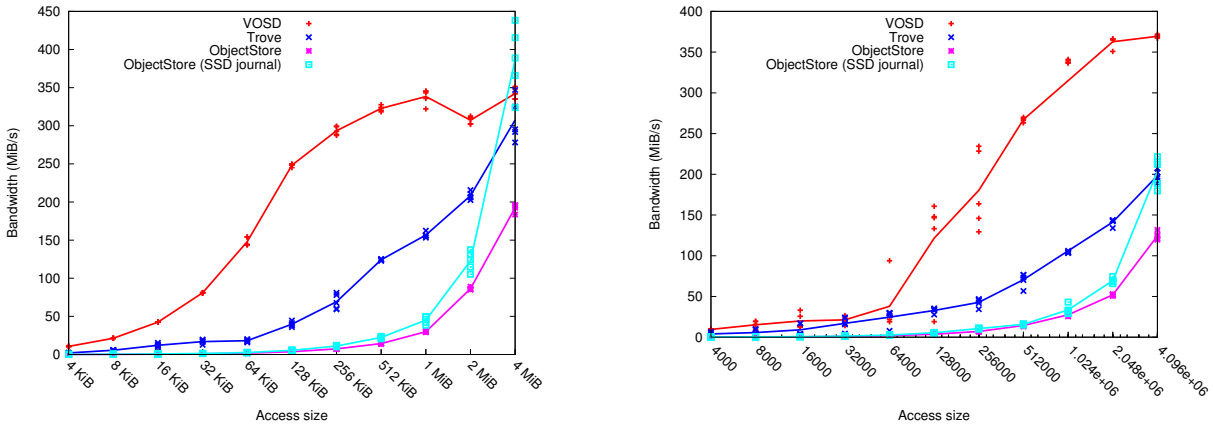
Although the VOSD, Trove, and ObjectStore all offer similar base object functionality, they differ significantly in terms of how they provide interfaces for concurrent operations. Trove offers an asynchronous interface for reads and writes, in which operations are posted with one function and tested for completion in another. The VOSD prototype uses post functions as well, but completion is handled via callbacks that are intended to drive continuation in the calling program. The ObjectStore queues write transactions asynchronously via a C++ class interface, with multiple callbacks used to differentiate between when the data is durable and when it is visible to readers. The ObjectStore read method is a blocking function call, however, and it automatically allocates a buffer to contain the data being read rather than utilizing existing buffers. These API differences make it difficult to interchange the three abstractions within existing file system implementations. We will therefore compare them using a microbenchmark that is customized for each API. In the case of ObjectStore reads, the microbenchmark uses explicit threads to provide concurrency.

The three abstractions also differ significantly in terms of underlying data organization. The Trove abstraction is the most traditional. It reads and writes data to underlying POSIX files in strictly canonical order. It supports standard I/O access as well as direct I/O. In the case of direct I/O, underlying reads and writes must be block aligned. Trove uses intermediate buffers and read/modify/write operations to maintain this requirement. It also explicitly tracks object sizes in a Berkeley DB database.

The ObjectStore abstraction is somewhat more complex than Trove because of its atomicity requirement. It uses a unified, write-ahead data journal for all objects. Writes are first applied (in log order) to the journal and then applied to the underlying POSIX file for the object. The journal can be replayed following a crash in order to preserve consistency. The journal can be stored in a normal file or on a raw block device, and it can be configured to use direct I/O or standard I/O. The ObjectStore does not support direct I/O to the object files themselves. The journal can also be disabled, in which case the ObjectStore relies exclusively on BTRFS transaction ioctls to preserve atomicity. This configuration resulted in lower write performance for our test system, however.

The VOSD prototype data layout strategy was described in the preceding section. It supports both direct I/O and standard I/O to its underlying files. In the case of direct I/O writes, all log entries are rounded up to the next aligned log position in order to avoid the need to perform read/modify/write operations. This approach achieves high performance at the cost of wasted capacity for unaligned writes. This space could be reclaimed via reodering as discussed in Section III, though read/modify/write may still be desirable for degenerate cases.

The microbenchmark used in these experiments is designed to measure concurrent read and write bandwidth for workloads similar to those seen in HPC servers. The microbenchmark first reads in a workload description file, then issues up to 32 concurrent operations to service the workload in parallel. The



(a) block aligned linear access

(b) unaligned random access

Fig. 3. Commodity storage: write bandwidth with 32 concurrent operations

benchmark was time limited to 60 seconds in all cases. All file systems and storage directories were removed between write experiments. The Linux buffer cache was also flushed between both read and write experiments. Object creation times were not included in the measurements. Ceph recommends the use of BTRFS as the underlying file system, so all ObjectStore tests were performed using BTRFS. Ideally we would use the same file system for Trove and the VOSD as well, but at this time BTRFS does not properly support direct I/O read operations. We therefore used EXT4 for those experiments unless otherwise noted. The VOSD microbenchmark assigns version numbers sequentially and uses automatic transaction flags for all write operations.

We used the Trove component from PVFS 2.8.2 and the ObjectStore component from Ceph 0.20.2 for all experiments. Trove was unmodified and used default configuration parameters. Ceph was unmodified except for a minor configuration parameter patch.[2] All ObjectStore parameters were left at their default value with two exceptions: the journal was set to always be larger than the total amount of data written in order to prevent interference from journal trimming activity, and the number of operation threads was set to 64 to ensure that sufficient threads were always available for the microbenchmark workload. All three abstractions were configured to use direct I/O to the extent possible.

A. Commodity storage performance

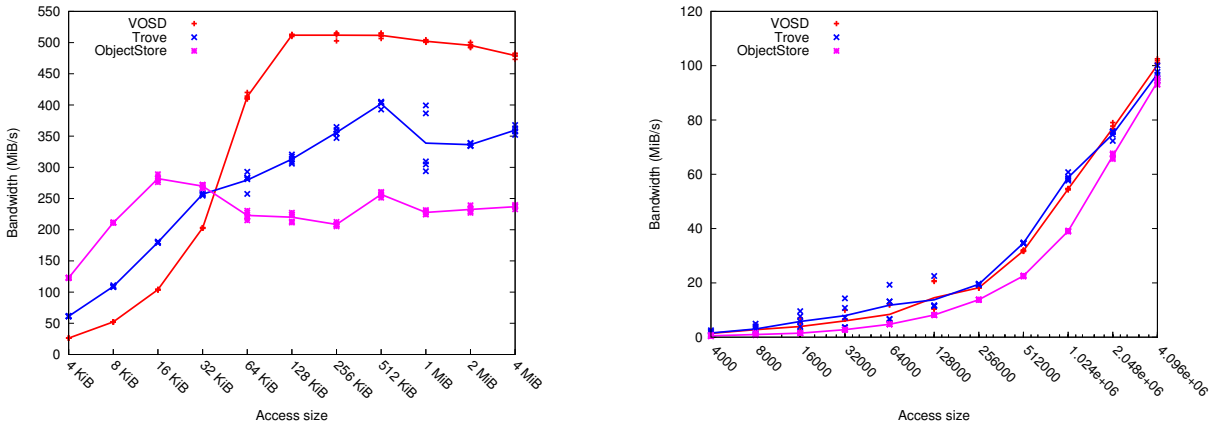
To investigate behavior on commodity storage, we first executed the microbenchmarks on a Linux server equipped with a hardware RAID controller. The server contained four quad-core Intel Xeon ES5520 CPUs, 48 GiB of RAM, and two Areca ARC 1231 RAID controllers. The first controller utilized 8 Western Digital RE4-GP SATA drives organized into a RAID 0 configuration. The second controller utilized 8 Intel X-25M V2 solid state disks, also organized into a RAID 0. All experiments were performed on the Western Digital drives unless otherwise noted. The peak write and read throughput of the Western Digital array was found to be 593 MiB/s and 300 MiB/s respectively. This was measured by using dd to transfer 4 MiB blocks to and from the raw device using direct I/O and synchronized I/O flags. The software consisted of Ubuntu 10.04 with an Ubuntu-packaged 2.6.32 Linux kernel.

Figure 3 shows the write bandwidth achieved by all three object storage abstractions for two workloads: a linear workload, in which all accesses were perfectly block aligned, and a random workload, in which all accesses were unaligned. The latter is the more likely scenario in HPC, in which a large number of processes access data according to an application data model that has no relation to the underlying block device. Five samples are plotted for each data point, with lines connecting the average value for each one. In both the linear and random workloads, the ObjectStore performance is bounded by its journaling architecture, which requires writing each buffer to disk twice. We also tested an additional configuration that used the SSD RAID as a dedicated journal device. This configuration improved ObjectStore write performance by avoiding journal contention on the primary RAID array. The most significant improvement appears in the linear write case with large buffer sizes, in which not only journal access but also object access occurs in an ideal order.

The Trove abstraction achieved higher performance than did the ObjectStore in most cases, because it performs I/O directly to canonical objects without preserving atomicity. It also utilizes direct I/O to eliminate any interference from the Linux buffer cache. The Trove version used in these tests also does not issue fsync() operations to its files, which is in contrast to the approach taken by the ObjectStore and the VOSD. Although fsync() has no effect on the Linux buffer cache for direct I/O, on modern Linux kernels it performs an additional step of issuing a barrier operation to the block device. This ensures that the hard drive cache is flushed in case of power failure; it would introduce an additional minor overhead in the Trove direct I/O implementation.

The VOSD achieves the highest write bandwidth for most

[2]http://comments.gmane.org/gmane.comp.file-systems.ceph.devel/551.



(a) block aligned linear access

(b) unaligned random access

Fig. 4. Commodity storage: read bandwidth with 32 concurrent operations

access sizes by virtue of its log-structured layout, which ensures well-ordered access to the disk regardless of the logical write access pattern. It also includes an efficient coalescing implementation that reduces write overhead further. Berkeley DB operations are issued with the DB_TXN_WRITE_NOSYNC flag set, which prevents the DB log from being flushed to disk immediately upon the close of a transaction. Instead, the VOSD will detect concurrent operations and delay completion of those operations until they can collectively use the log_flush() function to force the transaction log to disk. This approach improves concurrent performance without sacrificing atomicity or durability; no VOSD operation is allowed to complete until the database is flushed. The VOSD also uses a similar mechanism to coordinate fsync() calls in order to minimize the number of barrier operations that must be submitted to the block device. Trove implements coalescing as well [23], but only for relatively brief metadata and database operations.

Figure 4 shows the concurrent bandwidth for read operations using the same workload patterns as in the write case. In all examples, the data was first written using aligned 4 MiB buffers. The original write order makes no difference for Trove or the ObjectStore, but it is a factor in the VOSD performance. We investigate that issue further in Section IV-B. All three abstractions performed similarly for the random read workload, because they are all constrained by the same disk bottlenecks in accessing unaligned data.

The linear read workload shows more differentiation between the implementations. The ObjectStore offers the best performance for small I/O operations, because the Linux buffer cache automatically reads ahead and prevents the disk from having to service small I/O operations. Trove and the VOSD prototype perform better for access sizes in which the cost of buffer copying in the Linux kernel becomes too high to be offset by read-ahead. At small sizes Trove performs better than the VOSD because it does not need to perform a DB lookup to locate object data. For larger sizes the VOSD performs better than Trove, though additional experimentation is necessary

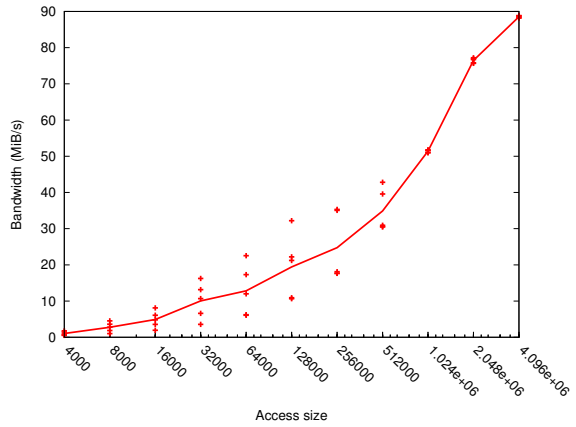
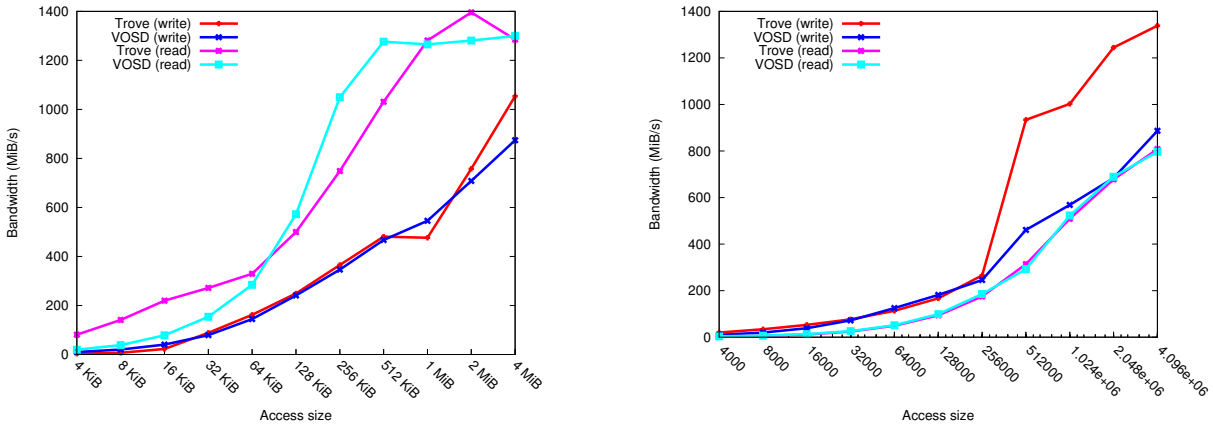


Fig. 5. Commodity storage: linear VOSD read performance with 4 MiB access size. The original write order was random, with access size shown on the x axis.

to confirm the reason. It may be due to differences in the threading model used internally by the two implementations. Note that both Trove and the VOSD prototype achieved higher read performance than the dd baseline of 300 MiB/s. This is because the disk array produces higher throughput for concurrent read operations than it does for serialized read operations.

B. Read sensitivity to log ordering

In Section IV-A we noted that the VOSD read performance is sensitive to the original write order of the data. Figure 5 illustrates that issue. In this case we execute a linear read workload with 32 concurrent operations but hold the access size fixed at 4 MiB. We instead vary the manner in which the data is written before the read benchmark is executed. Neither the order nor the alignment nor the access size match between the write and read workloads. The VOSD read performance therefore more closely resembles the random unaligned read case of Figure 4(b), because it must access the log out of order and assemble fragments from multiple log entries in



(a) block aligned linear access

(b) unaligned random access

Fig. 6. Enterprise storage: bandwidth with 32 concurrent operations

most cases.

This behavior would not show up in typical synthetic benchmarks because they are often executed using the same access pattern for both read and write test phases. It also is not likely to be a problem for checkpoint restarts, because the application is likely to reload data in the same manner that it was written. The most plausible scenario to illustrate this problem is when an independent analysis tool is used to process data written by a parallel simulation. In that case the write and read workloads could diverge dramatically. As outlined in Section III we intend to address this in future work by taking advantage of file system idle time to reorder data before it is analyzed.

C. Enterprise storage

Section IV-A evaluated object storage performance using commodity hardware. Enterprise storage may have different characteristics, however. We therefore continued our evaluation using one of the 128 file servers connected to the Intrepid Blue Gene/P system at Argonne National Laboratory. This server contains two dual-core AMD Opteron 2216 processors and 8 GiB of RAM and uses the 2.6.16 Linux kernel. It is attached via Infiniband 4X DDR to a Data Direct Network 9900 SAN containing 480 disk drives. The DDN was configured with both read and write caching enabled, using two controllers with mirrored caches and redundant UPS-backed power supplies. The LUN that we used for testing was formatted with an XFS file system. This is a production SAN volume, so we were allowed only a limited time window to perform experiments. As a result we collected only one sample per data point, rather than five as in Section IV-A. We were also unable to reformat the LUN used for these experiments. We limited our experiments to the VOSD and Trove, since XFS is not supported by Ceph's ObjectStore.

Figures 6(a) and 6(b) show the results of repeating the earlier microbenchmarks on a DDN 9900 LUN. Because DDN incorporates extensive resiliency features, we disabled barrier support in XFS and disabled fsync() of object files in the VOSD. It is sufficient to simply flush the Linux buffer cache in order to provide durability. Both the VOSD and Trove utilized direct I/O as in the previous experiments, and both utilized the same Berkeley DB coalescing strategies as before.

These results illustrate differences in the performance characteristics of commodity and enterprise storage hardware that go beyond simple throughput scaling. Notably, the log ordering performed by the VOSD offered no performance advantage for writes. The DDN 9900 uses a large sophisticated cache that minimizes the effect of poorly ordered write operations. The sync coalescing also had limited impact because barrier operations were not a factor in this configuration.

In fact, the Trove implementation outperformed the VOSD by a wide margin for large access sizes in unaligned random write workloads. The reason is that for random workloads the Trove implementation seldom has to update its Berkeley DB database. Trove used Berkeley DB during writes only to track the current object size. In a random workload, the size changes only when an operation happens to extend past the current end of file. Therefore many of the Trove random write operations did not access Berkeley DB at all. The VOSD, in contrast, has to update the DB for every write operation. On commodity storage this was offset by the benefits of log ordering and coalescing. In this environment, however, it is more important to simply minimize the number of IOPs than to apply writes in an optimal order.

V. Related Work

Devulapalli et al. developed a software-based T10 OSD target that stores data using POSIX files and an SQLite database [24]. As a standards-compliant implementation, this project focuses on iSCSI transport compatibility rather than direct use by local software components. They have also explored atomic primitives in the context of attribute operations [25].

The PanFS and Lustre file systems also implement object storage abstractions. PanFS object servers use a specialized local file system to store objects. These objects are presented

over the network using a protocol that closely conforms to the T10 OSD specification [3]. The Lustre object servers use either ldiskfs (a modified version of EXT3) or ZFS to store objects on block devices [2]. The Ceph and PVFS object storage abstractions were described in detail in Section IV. Devulapalli and Ali have also investigated integration of T10 compliant object storage in the PVFS file system [26].

Log structured storage has been used for a wide variety of purposes, particularly in the database community [27]. It is also a well-known technique in file system optimization [20]. More recent work by Kimpe et al. and Bent et al. have applied log-based storage to parallel I/O at the application and middleware level [28], [29]. Polte et al. explored the use of log-structured storage in PVFS's Trove storage component [21], though their investigation was motivated by checkpoint performance rather than semantic requirements. They observed that read performance can be hindered by poor indexing of log structured data and proposed potential solutions to the problem.

Abd-El-Malek et al. utilized fine-grained, explicit write versioning in the Ursa Minor parallel file system [30]. They used global version numbers generated by clients, however, which requires either a synchronous clock or additional communication steps to implement. Konishi et al. implemented a local Linux file system that uses log-structured layouts in conjunction with automatic versioning snapshots, but there is no mechanism to assign explicit version numbers to write operations [31].

Many mechanisms have been investigated for resolving conflicts and preserving consistency in optimistic replication protocols. Saito and Shapiro have surveyed popular techniques for a variety of use cases [32], though their work does not specifically address parallel file systems.

VI. Conclusion

In this work we identified three semantic extensions to the traditional object storage model that will help enable more efficient software replication in distributed, shared-nothing, concurrent I/O environments. These semantics are atomicity; explicit versioning; and commutative, idempotent writes. We implemented a prototype, the Versioned Object Storage Device, that demonstrates that these semantics can be achieved portably in user-space using standard Unix file systems. We evaluated the prototype's performance and demonstrated that the proposed semantics can be achieved with concurrent throughput that is competitive with existing well-known object storage abstractions. Our findings suggest that commodity storage platforms benefit greatly from I/O ordering and coalescing optimizations, whereas enterprise-class storage platforms benefit more from maximizing the amount of data transferred per I/O operation.

In future work we will evaluate the cost of the log ordering approach in the context of real-world applications. We also intend to investigate the implementation of additional VOSD features that build on the decoupling of logical object view from underlying data layout. These features may include transparent data reordering, data forks, embedded checksums, snapshots, and marshaling of objects for network transmission. Our immediate goal, however, is to demonstrate the use of the semantics described in this work to construct efficient replication protocols. We will investigate both pessimistic protocols, such as two-phase commit, and more relaxed protocols based on eventual consistency.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory.

References

[1] "The Parallel Virtual File System," http://www.pvfs.org/.

[2] Oracle Corporation, "Lustre file system," http://wiki.lustre.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] 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.

[5] T10 Technical Committee of the InterNational Committee on Information Technology Standards, "Object-based storage devices - 3 (OSD-3)," http://www.t10.org/.

[6] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, October 1996. [Online]. Available: http://www.computer.org/tpds/td1996/l1075abs.htm.

[7] E. Smirni and D. Reed, "Workload characterization of input/output intensive parallel applications," in *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, ser. Lecture Notes in Computer Science, vol. 1245. Springer-Verlag, June 1997, pp. 169–180. [Online]. Available: http://vibes.cs.uiuc.edu/Publications/Papers/Tools97.ps.gz

[8] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. Mclarty, "File system workload analysis for large scale scientific computing applications," in *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004, pp. 139–152.

[9] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA, Tech. Rep., 2008.

[10] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling, "Exascale software study: Software challenges in extreme scale systems," DARPA, Tech. Rep., 2009.

[11] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[12] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007, pp. 1–16.

[13] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, 1979.

[14] R. Latham, R. Ross, and R. Thakur, "The impact of file systems on MPI-IO scalability," *Lecture Notes in Computer Science*, vol. 3241, pp. 87–96, November 2004. [Online]. Available: http://www.springerlink.com/link.asp?id=m31px2lt90296b62

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, January 2009.

[17] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York: ACM, 2009, pp. 1–12.

[18] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001, pp. 329–350.

[19] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York: ACM, 2006, p. 122.

[20] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1991.

[21] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. K. Uppugandla, "Fast log-based concurrent writing of checkpoints," in *Proceedings of the 3rd Petascale Data Storage Workshop*, 2008.

[22] P. Gu, J. Wang, and R. Ross, "Bridging the gap between parallel file systems and local file systems: A case study with PVFS," *Parallel Processing, International Conference on*, vol. 0, pp. 554–561, 2008.

[23] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, April 2009, pp. 1–11.

[24] 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.

[25] A. Devulapalli, D. Dalessandro, and P. Wyckoff, "Data structure consistency using atomic operations in storage devices," *Storage Network Architecture and Parallel I/Os, IEEE International Workshop on*, vol. 0, pp. 65–73, 2008.

[26] A. Devulapalli and N. Ali, "Integrating parallel file systems with object-based storage devices," in *Proceedings of Supercomputing*, 2007.

[27] M. Ruffin, "A survey of logging uses," *INRIA (France), Tech. Rep. BROADCAST-36. Also available as Univ. of Glasgow (Scotland), Tech. Rep*, vol. 2, pp. 94–82, 1994.

[28] D. Kimpe, R. Ross, S. Vandewalle, and S. Poedts, "Transparent log-based data storage in MPI-IO applications," in *Proceedings of the 14th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2007)*, September 2007.

[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] 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.

[31] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 102–107, 2006.

[32] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, 2005.