Integrating Parallel File Systems with Object-Based Storage Devices

Ananth Devulapalli
Ohio Supercomputer Center
ananth@osc.edu

Dennis Dalessandro
Ohio Supercomputer Center
dennis@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
pw@osc.edu

Nawab Ali
The Ohio State University
alin@cse.ohio-state.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

ABSTRACT

As storage systems evolve, the block-based design of today's disks is becoming inadequate. As an alternative, object-based storage devices (OSDs) offer a view where the disk manages data layout and keeps track of various attributes about data objects. By moving functionality that is traditionally the responsibility of the host OS to the disk, it is possible to improve overall performance and simplify management of a storage system. The capabilities of OSDs will also permit performance improvements in parallel file systems, such as further decoupling metadata operations and thus reducing metadata server bottlenecks.

In this work we present an implementation of the Parallel Virtual File System (PVFS) integrated with a software emulator of an OSD and describe an infrastructure for client access. Even with the overhead of emulation, performance is comparable to a traditional server-fronted implementation, demonstrating that serverless parallel file systems using OSDs are an achievable goal.

1. INTRODUCTION

The ability of current storage systems to supply the I/O data rates needed by high-end computing applications has been insufficient for many years. While Moore's Law shows how processing elements are becoming faster over time due to increased chip densities, performance improvements in magnetic storage occur at a much slower rate. To meet the throughput and reliability demands of applications, parallel storage systems composed of commodity disks are used. The use of commodity components in these systems lowers the cost, but adds the expense of more complex implementation, management and client overhead.

It is an opportune moment to consider redefining the way in which storage is treated in a computing system. Currently, a disk drive is treated as a "dumb" peripheral. The operating system instructs the disk what data to write, and in what location. Further, the operating system does not expose any information about how the data is related. However, modern disk drives are actually quite complex. They perform write buffering, block remapping, command reordering, selective read-ahead and other operations on their own. It is the historical mode of interaction with storage that inhibits major improvements in performance, scalability and manageability.

With the recent introduction of an ANSI standard for a new interface to storage devices [34], the semantic level of communication with a disk drive becomes significantly higher. The standard specifies an object-based interface to storage rather than a block-based interface, among other important features. Unlike block-based devices, an object-based storage device (OSD) is aware of the logical data organization as defined by users. It manages all the internal layout decisions for data and keeps a variable set of metadata for each object. These features radically change the role of a storage element in a computing system. Rather than being relatively passive, as with block-based devices, an OSD can take a more active role in managing all aspects of storage, including data, metadata, security and reliability.

High-performance computing environments stress storage systems more heavily due to the specialized workloads seen there. Data is often streamed in large blocks unlike the small random accesses used in desktop environments. Parallel applications also tend to access storage cooperatively, allowing for better overall throughput. However, the metadata load in parallel file systems can sometimes be a major bottleneck [32, 31, 23]. Thus parallel file system designers are faced with a new and unique set of challenges for deploying OSDs. Overcoming these challenges will result in improved performance and reduced component count. While the operation set offered by an OSD is richer than that of traditional block-based devices, it does not provide all of the functionality desired by a parallel file system.

Many parallel file systems [4, 20, 35] already represent file data as objects. However, the storage devices themselves still maintain a simple block-based view of the storage medium. All decisions related to data layout and organization are the responsibility of the file system. As such these implementations are unable to leverage the capabilities of OSDs. Our work instead aims to integrate parallel file systems with true object-based storage devices. However, since the OSD specification is relatively new, there are no readily

SC07 November 10-16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

available hardware platforms.

OSDs can potentially simplify the design of parallel file systems by obviating the need for dedicated I/O and metadata servers. Moving the data and metadata storage logic directly onto the OSDs can effectively decouple the cluster nodes from the storage subsystem. However, this may not improve the performance of parallel file systems in terms of data throughput and latency. OSDs are after all, logical devices based on regular disks and as such the disk throughput and latency of OSDs is equivalent to that of regular disks. However, OSDs offer parallel file systems significant opportunities for scalability and manageability. By removing the most common bottlenecks found in the design of current parallel file systems (dedicated metadata and I/O servers), an OSD-based file system can be easily scaled up. Offloading security, data layout and management to the disks also simplifies global storage management.

The main contribution of our work is to examine the feasibility of OSDs for use in parallel file systems. A new infrastructure for experimenting with OSDs, from initiator to target, is presented. We also introduce a modified version of the Parallel Virtual File System (PVFS) that uses software-emulated OSDs to store data, along with performance results from microbenchmarks and applications. Even with the overhead of emulation, the performance of our OSD-enabled parallel file system is comparable to a traditional PVFS implementation, demonstrating that serverless parallel file systems using OSDs are an achievable goal.

2. BACKGROUND

In the past, distributed file systems, such as NFS [21] and AFS [19], were designed with servers in front of storage that handled requests from multiple clients. The servers were responsible for both data and metadata operations. Though this design is simple, it suffers from problems such as scalability, load imbalance, and network hot-spots. In contrast to this, parallel file systems, such as PVFS [2] and Lustre [4], segregate metadata from data operations to provide improved data throughput. This motivated the deployment of specialized servers for metadata and data operations. Figure 1 shows the architecture used by such file systems. Essential components of this design are the interconnect, the file system clients, the metadata servers hosting metadata and the storage or I/O servers hosting the data.

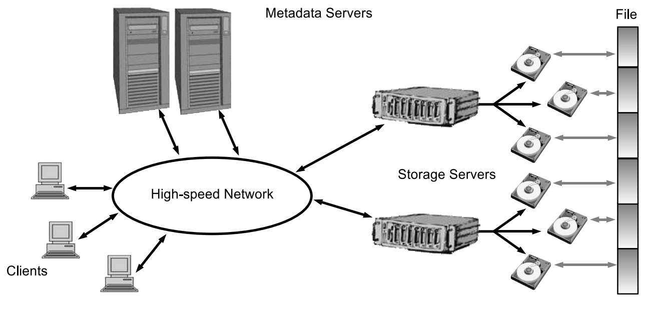


Figure 1: Architecture of Modern Storage Systems.

In such parallel file systems, storage servers are generally fully provisioned computers, which allows for powerful file system semantics [3], at the expense of introducing another layer in the data path. However, systems such as NASD [10] discard this intermediate layer to remove the additional store-and-forward overhead, but end up sacrificing functionality. With OSDs it may be possible to eliminate this intermediate layer to achieve both good performance and powerful functionality. Furthermore, many aspects of the metadata management can be offloaded to OSDs, reducing the number of metadata nodes required for a given performance target.

2.1 Object-Based Storage Devices

An object-based storage device (OSD) organizes and accesses data as objects rather than as a simple stream of bytes. Figure 2 shows how OSDs transform the storage architecture. In the traditional model, the host operating system (OS) is responsible for the data layout and all application requests have to be translated by the OS into logical block addresses. In the OSD model, the job of organizing the data on the storage medium is moved to the OSD.

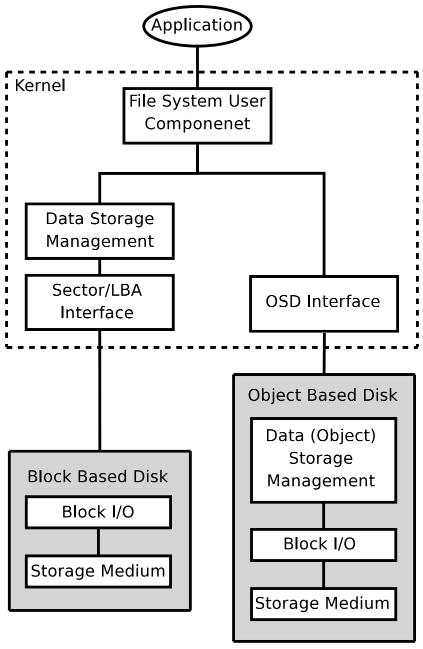


Figure 2: Comparison of Block-based and Object-based Storage Models. The Data Storage Management unit has moved from the kernel to the OSD.

The OSD specification [34] defines an object as an ordered set of bytes associated with a unique identifier. Additionally, each object has a set of mandatory and user-defined attributes, or metadata, about the object. Some of the predefined attributes are familiar from most file system designs, such as size and modification time.

An OSD has four types of objects: the root object, partitions, collections, and user objects. Every OSD has exactly one root object whose attributes control settings for the entire storage device. A partition defines a namespace for collections and user objects. User objects are the entities that contain data and metadata. Collections are used to facilitate fast indexing of user objects based on their attributes.

With block-based storage, the security policy controls each device on a per-drive basis, usually through switch-mediated per-node access controls. With OSDs, the security policy is much more fine grained, in that access to each object can be controlled individually. In typical storage area networks, there is a server or controller that sits in front of devices to enforce security policies. This creates a bottleneck and limits scalability. In contrast, with an OSD, every command is accompanied by a security capability that identifies the rights of the user issuing the command.

Clients interact with OSDs through SCSI commands [33]. Some of the basic and easily recognizable commands are CREATE, READ, WRITE and APPEND. In addition to manipulating an object, each command is optionally able to get and set particular object attributes. The results of the data operation as well as any retrieved attributes are returned to the initiator through normal SCSI mechanisms.

3. SYSTEM DESIGN

This section explains the design and implementation details of our OSD infrastructure. The software has two main components, the OSD initiator and the OSD target. Figure 3 shows the components of the architecture and their relationships. The initiator and the target communicate with each other using iSCSI [26]. The iSCSI protocol enables the transport of SCSI commands over a network, usually TCP/IP. The following sub-sections explore the design issues and the choices made by us during our implementation of an OSD initiator and target. Issues related to integration of a parallel file system with the OSD infrastructure are discussed in Section 4.

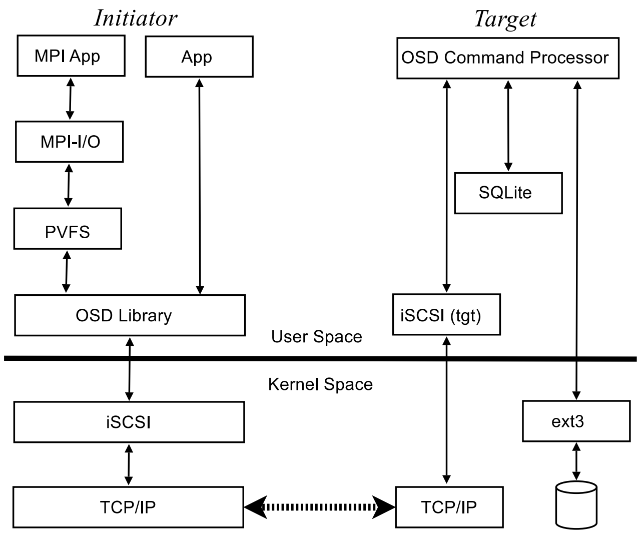


Figure 3: Software architecture of the OSD infrastructure.

3.1 OSD Initiator

The OSD initiator library exports the interface used by client applications to communicate with the OSD target. The library can be used directly, or as a building block for file systems or other middleware. It provides calls to create OSD commands, submit them to devices and retrieve results. All operations are asynchronous and independent. The initiator also manages device enumeration and mapping of device names to system interfaces.

In order to send a command to a target, an application first uses our initiator library to create a Command Descriptor Block (CDB). This is a 200-byte chunk of data defined by the specification [34] that conveys most of the information required to designate a command: operation code, partition, collection and object identifiers as necessary, offset and length, capabilities for access control and optional get and set attribute parameters. Along with the CDB, the application also provides pointers to output data, input data and a "sense" buffer for returned error information.

To transport the command to the target device, we use iSCSI for its convenience and ubiquity. In particular, we use the existing Linux in-kernel iSCSI client implementation. The kernel version offers good performance and is a mature code base that is present on all modern Linux systems. It currently supports two transport layers: TCP/IP and InfiniBand [16]. Our initiator directly generates SCSI commands and submits them to the kernel through its "SCSI mid-layer," making it possible to use any SCSI transport such as locally attached SATA or fibre channel drives.

The initiator library submits the fully formed command descriptor block to the kernel SCSI mid-layer through an interface called "bsg." This is a recent development in the Linux kernel meant to serve as a generalization of the existing "sg" interface to all block devices, not just SCSI. The key feature for us in choosing bsg is its ability to handle CDBs that are longer than 16 bytes and to support bidirectional commands. As any OSD command can also retrieve attributes, bidirectionality is frequently required. Commands are passed to the kernel via a write to a character device. Completed commands are retrieved by reading the character device, which can be polled for status too. The split-phase asynchronous nature allows us to manage multiple outstanding commands to one or many target devices.

Beyond the inclusion of patches to add the bsg interface, other changes to the mainline Linux kernel are also required, the bulk of which are needed to support bidirectional commands in the SCSI mid-layer and iSCSI transport. One handy change to bsg that we added is the support for I/O vectors to minimize data copying in the initiator library. The effort to maintain these forward-looking patches is worth the ability to leverage large amounts of working infrastructure related to SCSI and iSCSI processing. Over time, the changes will certainly be absorbed into mainline Linux.

One of the more difficult features to handle cleanly in the initiator is the ability to set and retrieve arbitrary attributes along with any command. Each command can retrieve one or more attributes of an object or it can retrieve a well-defined "page format" of attributes. It can also set one or more attributes. There are two forms to specify the attribute requirements in the CDB, with different limitations. The list format requires sending output data with a list of the attributes requested. When retrieved, attributes are delivered at a specified offset in the input data that is returned. There are alignment constraints on the offsets, and output and input buffers must be manipulated so that the data and attributes land at the correct locations.

While our initiator design could have placed on the user the onus of constructing the proper CDB and input and output buffers, this would result in clumsy application code and exposure of unnecessary details to the applications. Instead, we crafted the interface to build the CDB given a concise list of the attributes to set and get. After the command completes, another helper routine resolves the returned attributes and arranges for their retrieval by concise pointer dereferences. Furthermore, there is no copying of input or output data due to the bsg I/O vector patch.

3.2 OSD Target

The OSD target is responsible for iSCSI session management, processing requests from initiators, accessing or creating objects, as well as potentially setting and retrieving attributes, and finally sending back an appropriate response. This response includes an indication of the final result of the command, and in the event of an error, the appropriate sense code [33].

There is currently no iSCSI target support in mainline Linux, nor does that appear to be likely in the future. But there are a number of iSCSI target implementations available, ranging from pure software to hardware-specific drivers [29]. We chose *tgt* [9], a user-space iSCSI target implementation for block-based devices. It requires no kernel modifications and matches the performance of other in-kernel iSCSI target implementations. The iSCSI target code maintains connections with initiator devices and demarshals their requests. It sends the command descriptor block and data pointers to the OSD command processing layer. Finally the response and output data is marshaled and sent to the initiator.

Beyond iSCSI processing, we also need code to process OSD commands, not just the block commands supported by most targets. There are a few OSD target implementations available, but none was suitable either because of reliance on an old and possibly incompatible iSCSI stack [28, 17], or because of unavailability of source code [6, 15]. The existing available implementations are also incomplete and based on an older version of the OSD specification. They would have also required a major overhaul of the code to work with *tgt*. This led us to develop an OSD command processing engine in the context of the *tgt* framework. Our goal for the target is to achieve conformity with the OSD specification [34] and to attain a comparable level of performance with existing servers. We are not attempting to address broader industry objectives such as device integration of an OSD target. This fits with the purpose of our current work which is to examine the feasibility of OSDs in the context of parallel file systems, and not on development of the OSD target itself.

OSD commands can be classified into the following categories: object manipulation, input/output, attribute manipulation, security, and device management. Our OSD target currently implements all object manipulation, input/output, and attribute manipulation commands. Work on the other commands, including optional commands for collections, is in progress. Almost all OSD commands can be accompanied by attribute manipulation requests, which our OSD target fully supports.

The primary task of the target is the manipulation, creation and destruction of objects and their attributes. The actual layout and management of objects on disk can be implemented in a couple of ways. One is to rely on an underlying file system like ext3 [30] for low level storage operations, while still controlling the high level organization of objects. Another option is to manage block allocations ourselves and access a raw block-based partition directly. This approach may provide better performance, but has considerable management overhead and invariably would lead to much code duplication. Out of simplicity, we have chosen the first option and rely on an underlying local file system. For higher level object organization and management, our implementation currently uses a one-to-one mapping between user objects and files.

One of the interesting features of OSDs is per-object user-defined attributes which allows increased semantic control over objects. The desire for fast look-ups and flexible manipulation of attributes, compounded with the potentially large number of attributes per objects, suggested the use of a database for their management. We use SQLite [14], a light-weight embedded SQL database, to manage object attributes. However there are some trade-offs involved with SQLite. For instance, while SQLite provides rich SQL semantics, it lacks a framework for multi-threaded applications, like fine-grained table and row locks. That said, the real benefits of an SQL database will arise when dealing with collections, which requires fast attribute operations on multiple objects.

4. MAPPING PARALLEL FILE SYSTEMS TO OSDS

In this section we explore issues related to mapping parallel file system requirements to OSD capabilities. The exercise involves choice of the object size used for mapping the file chunks, data layout, metadata mapping, security integration among other things. We will explain the issues we encountered while mapping PVFS onto OSDs and follow it up with mismatches between parallel file system requirements and current OSD capabilities.

4.1 Mapping PVFS to OSDs

Parallel Virtual File System (PVFS) is an open-source distributed parallel file system. The file data is striped across multiple I/O servers to increase overall throughput, and co-operating clients can use parallel I/O calls to access data concurrently. It also supports parallelism in the metadata space: multiple metadata servers can share the load for operations such as file look-ups, creates, and directory traversals. The metadata aspect of a file system workload can be quite significant at times [23].

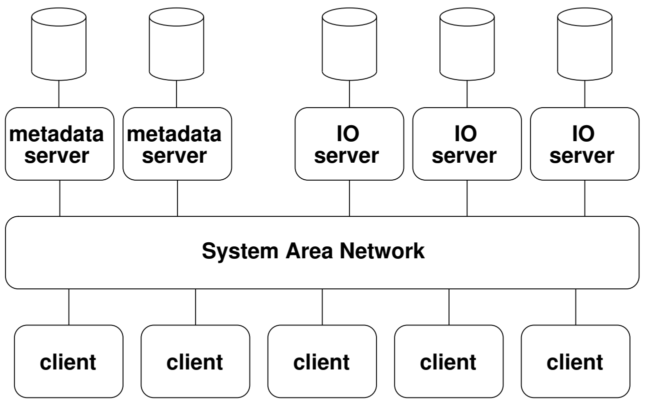


Figure 4: Architecture of Typical Server-Fronted Storage System.

Our long-term vision is to discover just how much parallel file system functionality can be supported purely by using standards-compliant object storage devices. The first step on that path is to move the I/O workload to OSDs. In PVFS, each server process can act as either an I/O server or a metadata server or both, depending on the configuration; this architecture is shown in Figure 4. To construct a PVFS environment using OSDs as the data store, we disable the I/O server functionality but continue to use the metadata server functions unmodified, as shown in Figure 5. The I/O servers are replaced with emulated OSD targets.

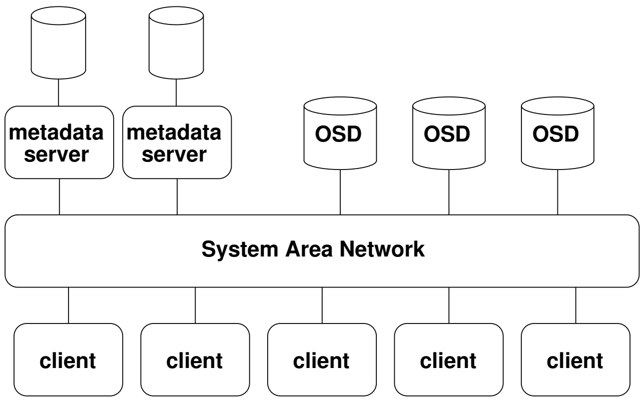


Figure 5: Architecture of Parallel File System using OSDs as I/O Targets.

The bulk of the modifications to PVFS are in the client libraries. Instead of sending requests to the PVFS I/O servers, clients must generate SCSI commands and send them to an OSD server. This transformation of the software is fairly straightforward due to the abstractions already present in PVFS. State machines are used as the mechanism to manage multiple phases of a single logical operation such as a create that may need to communicate with multiple servers (or OSDs). Most client operations in PVFS use a "message pair" state machine that essentially implements a parallel remote procedure call to the servers. For OSDs, we replaced this building block with one that submits SCSI commands and retrieves the results.

PVFS uses BMI [1] for communication between clients and servers. Implementations exist for TCP, InfiniBand and Myrinet. We continue to use the TCP communication layer to interact with the metadata servers, but not with OSDs. For them, a new BMI module is used to minimize code changes. It does not implement the basic send and receive calls, but does provide name resolution and manages a list of outstanding messages. OSD commands are submitted to the upper layer job interface as a result of state machine operations. BMI picks up the results of these jobs and allows the state machine to continue to the next phase.

During initialization, PVFS looks at a configuration file on the metadata server where a line specifies if OSDs are being used as the I/O targets instead of PVFS servers. Other minor changes are required to integrate PVFS with OSDs, such as disabling commands that modify server logging levels. Most operations afford obvious translations, such as create, remove, read, write and flush. Others require more care; for example, the "ping" server command becomes a SCSI "test unit ready" command, and "stat" for the file size becomes a "get attribute" command for the object logical size.

The biggest design implications result due to the differences in the presumed flow control implementations of the underlying networks. In PVFS, a "flow" is used by the client to keep some number of pipelined communications going in the network. OSDs use any of a number of SCSI transports, all of which are expected to do their own flow control. With SCSI and iSCSI in particular, the target is responsible for the flow control, permitting it to manage its limited buffer space when serving multiple clients. Thus, for our implementation we do not use the flows in PVFS but just submit commands directly and let the target control the data motion by issuing the existing iSCSI mechanism.

Another issue is related to the internal identifiers that PVFS uses to track files and directories. These identifiers are 64-bit integers that are partitioned in ranges across all the servers in the configuration. This makes it easy to identify which server holds a particular object by looking up the handle in the static table. Mapping PVFS handles to OSD 64-bit object identifiers seems like an obvious approach. However, when allocating a new handle, a PVFS client instructs each I/O server what range of handles it may use. This API is not present in OSDs—either the client specifies a particular handle or allows the device to choose one. Currently we rely on the fact that our implementation always chooses increasing object identifiers and "seed" the target at formatting time with the lower bound of the handle range. A more flexible handle mapping is necessary, perhaps by encoding both an object identifier and server identifier wherever handles are stored in PVFS.

4.2 Possible OSD Extensions

Our experience with the integration of PVFS and OSDs has exposed us to both the powerful features of OSDs, and the mismatches between the requirements of PVFS and the capabilities of OSDs. On the positive side, OSDs enable aggregate operations such as the creation of multiple objects using a single command. This capability can be used to speed up the file create operation in PVFS [5]. On the other hand, there are some infrastructure drawbacks and deficiencies in OSD capabilities.

The SCSI transport layer permits a target device to return exactly one buffer, either for a read or bidirectional command. For a read command that requests attributes, both the read results and the retrieved attributes appear in the single "data in" buffer that SCSI returns. The offset of the retrieved attributes in this buffer is chosen by the client in advance. If, for example, the client eagerly tries to read up to 10 MB of an object but also wants to retrieve an attribute, the client will set the offset to be just past 10 MB in the returned buffer. If the object turns out to be smaller than this size, the server must send 10 MB of zeroes over the network just to fill in the attribute at the requested offset. Currently we get around this restriction by issuing two commands: one for the data and a second for the attribute. The OSD will truncate the data result at the actual length. A better solution would involve letting the OSD specify the retrieved offset for attributes or adding a scatter capability to the SCSI transport layer.

Related to this issue is the lack of a scatter/gather facility on the SCSI target. MPI-I/O calls that use complex data layouts can generate fragmented views of the file data, so that a particular write operation from the client will be scattered in many chunks to the logical object stored on the OSD. Using the PVFS I/O server protocol, the client can send a description of how to scatter the data. The OSD specification does not support this, forcing the client to use many smaller contiguous operations instead. Having a simple scatter/gather facility at the target would both simplify operations from the client point of view and improve performance.

OSDs lack atomic operations such as Fetch-and-Add and Compare-And-Swap that enable serialization on a device. There are other ways to gain exclusive control of a device, but they are too coarse grained to use as building blocks for locking and coherence. It is possible to use a distributed lock manager for serving locks, but atomics-capable OSDs would further improve the scalability of the system. Moreover, implementing atomics should not be too costly since the existing attribute infrastructure could be leveraged. These remote atomic operations can further offload file system operations such as directory entry creation and removal. They could also help to realize disk-managed distributed databases [22].

5. EXPERIMENTS

The goal of the experiments described in this section is to evaluate our OSD infrastructure and to show that using object-based storage with a real parallel file system is feasible. In spite of the fact that the OSD is emulated in software, the performance is comparable to the usual server-fronted storage approach. When hardware OSDs become available, the performance will most likely be better than what we show here.

Our experimental platform is a Linux cluster consisting of 30 nodes. Each node has dual AMD Opteron 250 processors, 2 GB of RAM and an 80 GB SATA disk. The cluster runs the Linux operating system, version 2.6.20. The onboard Tigon 3 Gigabit Ethernet NIC is used for communication, with a single SMC 8648T 48-port switch.

5.1 Overheads in OSD Target Emulator

First we attempt to quantify some of the overheads introduced by our emulator by running a simple command to get attributes. Table 1 shows the time spent in each of the four phases of command processing. "SQLite" represents the cost of object and attribute look-ups. "CDB" covers the request parsing and response creation costs. "iSCSI" includes SCSI transport layer processing. Finally, "Initiator" represents the command processing time in the test application, client-side iSCSI overheads and network communication latencies.

Processing phase	Overhead
SQLite	81.3 ± 0.3 μs
CDB	2.2 ± 0.5 μs
iSCSI	29.9 ± 1.7 μs
Initiator	125.6 ± 2.9 μs
Total	239.0 ± 1.1 μs

Table 1: Target emulation overheads for the processing phases.

As these experiments use an emulator of an object-based storage device, the results will not be representative of true device behavior. An integrated device would be expected to have dedicated hardware for most critical path functions, such as basic SCSI and iSCSI command processing. It may also have features such as content-addressable memory to make object and attribute look-ups fast.

5.2 Stat Microbenchmark

Figure 6 (left) shows the latency of a "stat" operation on OSD and PVFS I/O servers as a function of the number of I/O elements. This operation is used to find information about files such as the size or modification time, and requires communicating with every storage device. Both curves have similar slope, although there is slightly more overhead with OSDs.

A simpler "ping" microbenchmark produces similar results, but without the processing costs of object look-up, roughly 83 μs as shown in Table 1. The round-trip time with the OSD emulator is 101 μs, compared to 124 μs for the I/O server. The scaling as more server elements are added is slightly better in the OSD case, which requires 1.7 ms compared to 2.0 ms for the PVFS I/O server.

5.3 Create Microbenchmark

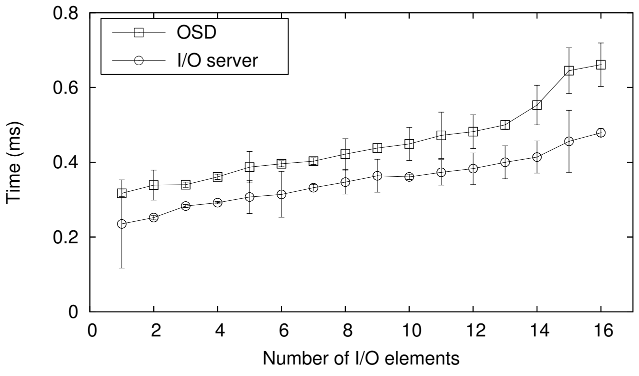
Figure 6 (right) shows the latency of a "create" operation on OSD and PVFS I/O servers. Note that during this test the metadata and I/O servers use a RAM-based file system to remove the effects of physical disk seek times. Again, there is a slight performance overhead in the OSD system due to the impact of the iSCSI and SQLite layers, but performance scales identically with increasing number of I/O elements.

Figure 7 shows the aggregate rate of create operations with one I/O element as clients are added. This is a stress test designed to see what limitations on create are caused by the PVFS I/O server or OSD. On the left are the performance results using a real disk, where both implementations are limited by seek time. In order to analyze protocol processing costs, the results using RAM-based storage are shown on the right. OSD create throughput is about 80% of that of a PVFS I/O server. The reason for this can be seen directly from the processing time for SQLite, as shown in Table 1. The PVFS I/O server uses a simpler DB4 database. Use of fine-grain multi-threading can further boost performance of OSD based PVFS.

5.4 I/O Throughput

To measure throughput, we use `perf`, an MPI-I/O benchmark from ROMIO. In `perf`, each parallel process writes a data array to a shared file, then reads it back, using non-collective I/O operations with individual file pointers. MPI invokes PVFS to perform the file accesses, which in turn sends requests to PVFS I/O servers, or initiates SCSI requests. Figure 8 shows the throughput for one client communicating with one PVFS server (on the left), or one OSD (on the right).

The curves for read and write are similar in both plots and represent the maximum network throughput achievable by either the I/O server or OSD. The lower curve shows the throughput when the data is flushed to disk after the write, and reflects the limit of the single SATA drive used for these tests. The maximum throughput for the OSD emulator of about 80 MB/s is lower than the 95 MB/s measured with the PVFS I/O server. This is due to the transfer limit of 256 kB imposed by the SCSI layer and the lack of pipelining



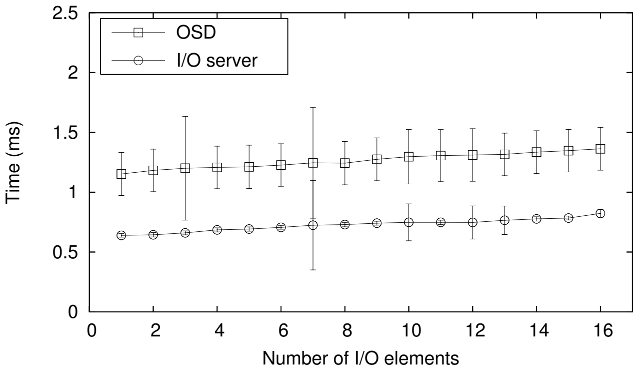
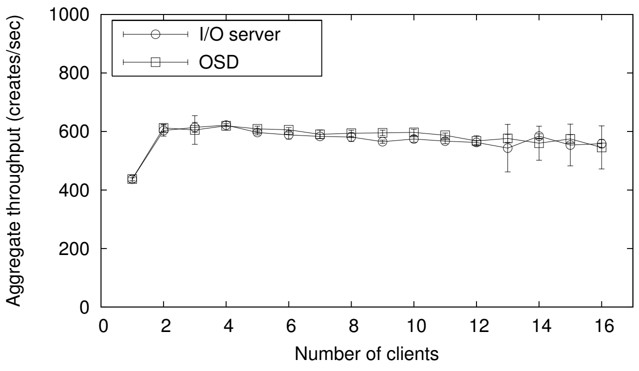


Figure 6: Latency as a function of the number of I/O elements; left: PVFS stat; right: PVFS create.



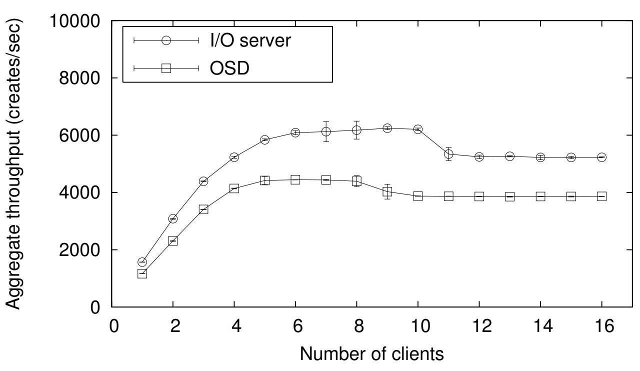


Figure 7: Aggregate create throughput; left: disk-based storage; right: RAM-based storage.

in our OSD emulator. However, for small message sizes, the OSD implementation outperforms the PVFS I/O server due to the use of multiple control messages for each data message in the (non-OSD) PVFS I/O protocol. Enlarging the cut-off for "small" I/O transfers beyond its current value of 16 kB would help for small-message workloads.

The throughput of the iSCSI initiator to the emulated OSD target, without the MPI or PVFS layers, shows that its performance is not the limiting factor. At a 64 kB data transfer size, write throughput is 70 MB/s and read throughput is 91 MB/s. For 256 kB transfers, both write and read throughput increase to 93 MB/s. The write operation is slower than read for smaller message sizes due to the extra "ready to send" iSCSI message used by the target to control data flow. The maximum SCSI transfer size of 256 kB is due to limitations on the size of the scatter/gather table, and hence number of mapped user pages, as imposed by the Linux SCSI mid-layer. PVFS sends multiple SCSI messages to achieve larger transfers. Testing against an in-memory block device target, rather than an OSD target, produces essentially identical results.

5.5 I/O Scaling

The next set of graphs show the results of a stress test, where a single client reads or writes to a file striped across some number of servers. In Figure 9, the file size is 64 kB multiplied by the number of servers, thus the work per server stays constant as more servers are added, but the single client moves an increasing number of bytes. For this small size, the write and write+sync rates increase steadily for both the I/O server and the OSD, and the OSD system saturates the client's network interface with many fewer nodes than the PVFS I/O server system. For both, read performance drops quickly to below 5 MB/s when the number of servers goes much beyond four.

This massive drop-off in throughput can be explained by looking at the TCP congestion control algorithm. For a read, PVFS sends requests to all the servers at the same time. Their responses create a packet storm at the client. The switch must drop some of these packets if the total size of the burst exceeds its internal buffering capacity. As an experiment, we configured the TCP maximum read buffer size on the client to 4 kB, or about two 1500-byte packets plus headers. This effectively disables the TCP congestion control algorithm and limits the size of the burst such that there are no drops. With this setup, the read rate stays high, around 80 MB/s. The problems with the additive increase, multiplicative decrease nature of TCP congestion control algorithms are well known [18]. Our results use Vegas, the best performer of the 11 algorithms available in our version of Linux.

5.6 Parallel Applications

To evaluate application performance, we use two bench-

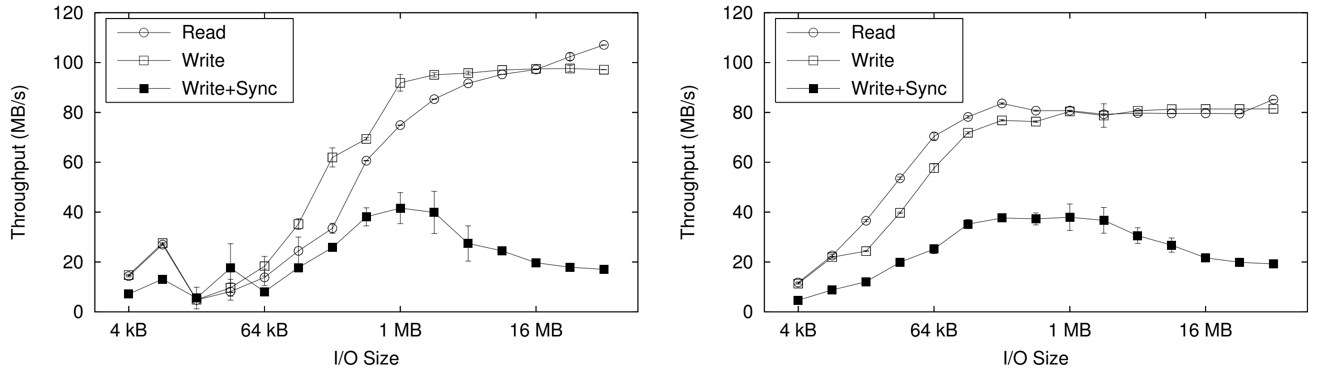


Figure 8: Throughput as a function of message size; left: PVFS with I/O servers, right: PVFS with OSDs.

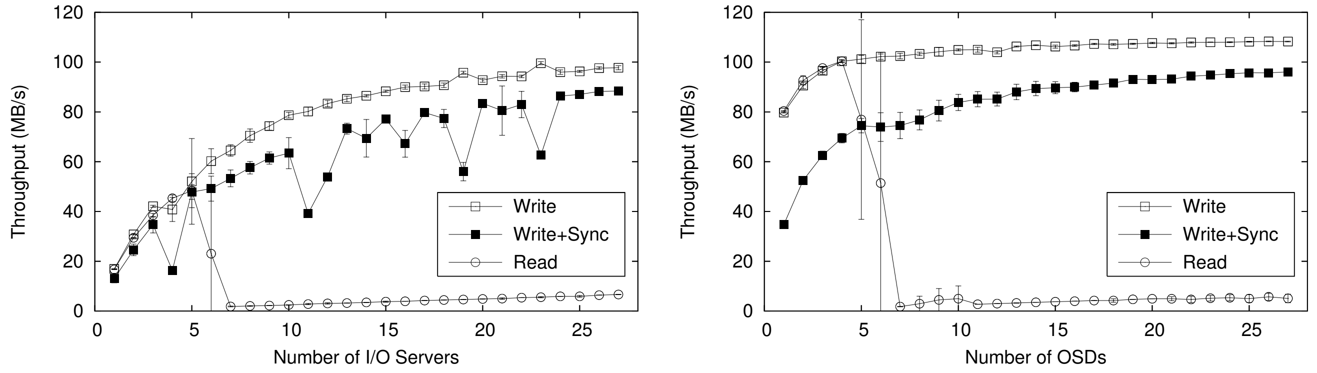


Figure 9: Throughput as a function of the number of I/O elements for a 64 kB message size; left: PVFS with I/O servers, right: PVFS with OSDs.

marks that are designed to mimic full application behavior. The first is the BTIO benchmark, which is a variation of the BT application, of the NAS Parallel Benchmarks [36] suite. BTIO solves systems of block-tridiagonal equations in parallel, and performs periodic solution checkpoints using MPI-I/O calls and non-contiguous data transfers. We use the *full* version of BTIO which uses collective I/O to combine data accesses of multiple processes into large, regular I/O requests. Figure 11 shows the execution time of BTIO as a function of the number of clients. Here we compare a parallel file system of PVFS I/O servers against one that uses OSDs as data servers. We varied the number of clients from 1 to 25 (BTIO requires that the number of processors be a perfect square, i.e. 1, 4, 9, 16 etc.) while keeping the number of PVFS I/O servers and OSDs fixed at 4. The execution times are nearly identical, although perhaps the OSD system achieves a slightly lower overall time to completion due to its comparatively better performance at small message sizes.

The other application we present is the FLASH I/O benchmark [25]. It is designed to approximate the I/O characteristics of the FLASH Code [24], which is used to study astrophysical thermonuclear flashes. Figure 10 compares the performance of the PVFS I/O server against the OSD. The data shown represents only the checkpoint file test. The OSD tests are much faster than the PVFS I/O server tests due to overheads in PVFS for the particular patterns of writes in this benchmark.

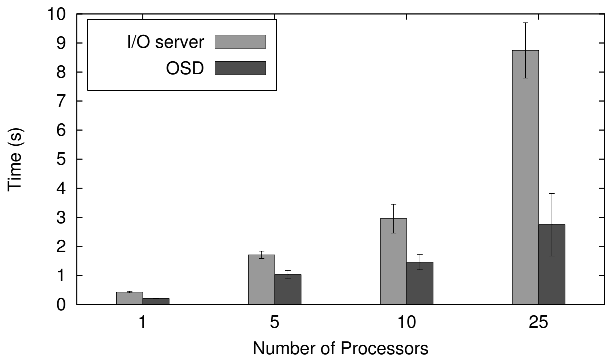
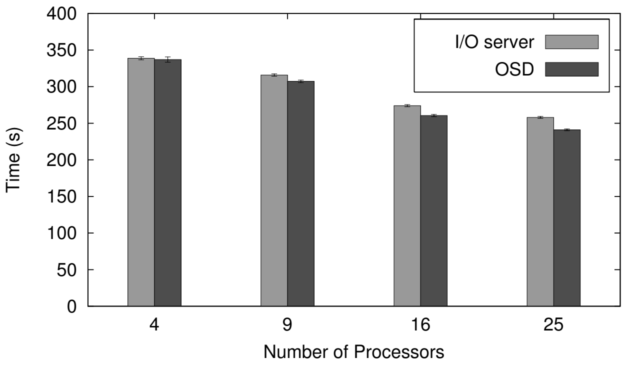


Figure 10: Execution time for FLASH I/O benchmark.

The FLASH code uses HDF5 to store data from all tasks to a single file, with the file contents arranged by variable, causing each task to make multiple writes at various locations in the file. Each task first writes about 200 16-byte chunks to disk, which takes half a second for the 25-client case for both the PVFS I/O server and OSD. The bulk of the data is written in 24 separate 300 kB write calls from



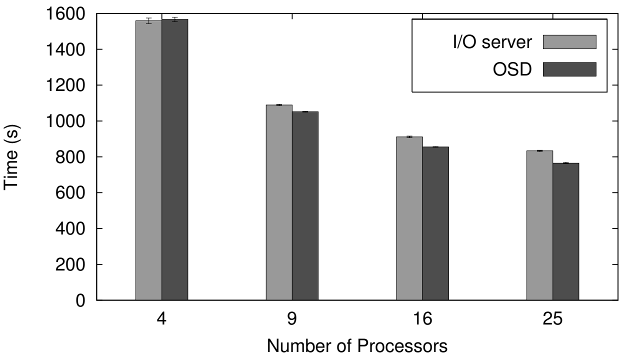


Figure 11: Execution time for BTIO benchmark, left: Class B; right: Class C.

each task. With a PVFS stripe size of 64 kB, this results in writes to each of the four servers of roughly 64 kB to 100 kB, depending on the file offset. This is because PVFS with I/O servers perform poorly relative to PVFS with OSD, when writes are in the range of 64 kB, as shown in Figure 8.

6. RELATED WORK

Using an object-based interface and directly accessing disks is not a new concept. Gibson *et al.* proposed Network-Attached Secure Devices [11, 10], which supported direct data transfer between the drive and clients, asynchronous oversight, cryptographic integrity and an object-based interface.

There exists previous work in OSDs themselves. Factor *et al.* presented the latest developments in OSD technologies [7], focusing on issues related to standardization and deployment as well as the challenges facing the adoption of OSDs. The inherent security capabilities of object-based storage is also an active area of research [8].

Several reference implementations of OSD targets are available. IBM has developed a prototype of an object-based controller called ObjectStone [15]. Its front-end interface implements the OSD protocol. IBM has open sourced their initiator but the OSD target is released only as a binary. Both Intel [17] and Sun [28] have their own iSCSI stacks and partial OSD initiator and target implementations. OSDFS [6] is a virtual file system interface on top of object storage that exports a typical block interface to applications, enabling use of OSDs with legacy applications but not exploiting any new capabilities in the interface. EBOFS [35] is an object storage backend that writes to disk without a local file system, but currently does not implement attributes.

In addition to PVFS, there are are a number of parallel file systems that include object features. Lustre [3, 4], PanFS [20] and Ceph [35] explicitly deal with objects, but store data using various non-standard object interfaces that effectively require server-fronted storage targets. Parallel NFS [12] is an extension to the NFSv4 protocol [27] that extends the delegation mechanism to account for multiple servers. It supports file, block, and OSD data storage types [13].

7. CONCLUSIONS AND FUTURE WORK

In this work, we have demonstrated the successful integration of OSDs in a parallel file system. We have exploited the increased semantic capabilities of OSDs and have shown the feasibility of serverless file systems by replacing PVFS I/O servers with OSDs. We have also analyzed mismatches between parallel file system requirements and OSD capabilities. In spite of the overhead associated with emulating the OSD target, the performance of PVFS integrated with OSDs is comparable to a server-fronted configuration. By publication time we will release all software discussed in this work as open source.

Using OSDs for I/O operations concludes the first step in the process of integration of OSDs in a parallel file system. In the future we plan to investigate their utility in metadata management and issues related to flow control and caching. Work is in progress with respect to implementation of optional features such as collections and development of a multi-threaded OSD target. Currently our target uses TCP/IP for communication and we plan to investigate its performance on RDMA capable transports like InfiniBand and iWARP.

8. REFERENCES

[1] P. H. Carns, W. B. Ligon III, R. Ross, and P. Wyckoff. BMI: a network abstraction layer for parallel I/O. In *Proceedings of IPDPS'05, CAC workshop*, Denver, CO, Apr. 2005.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[3] Cluster File Systems, Inc. Lustre frequently asked questions. http://www.clusterfs.com/faq.html.

[4] Cluster File Systems, Inc. Lustre: a scalable high-performance file system. Technical report, Cluster File Systems, Nov. 2002. http://www.lustre.org/docs/whitepaper.pdf.

[5] A. Devulapalli and P. Wyckoff. File creation strategies in a distributed metadata file system. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, Mar. 2007.

[6] D. Du, D. He, C. Hong, J. Jeong, et al. Experiences in building an object-based storage system based on the OSD T-10 standard. In *Proceedings of MSST'06*, College Park, MD, May 2006.

[7] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In *Global Data Interoperability—Challenges and Technologies*, Sardinia, Italy, June 2005.

[8] M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The OSD security protocol. In *Security in Storage Workshop (SISW'05)*, San Francisco, CA, Dec. 2005.

[9] T. Fujita and M. Christie. tgt: framework for storage target drivers. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[10] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, Nov. 2000.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, et al. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 92–103, 1998.

[12] G. Goodson, B. Welch, B. Halevy, D. Black, and A. Adamson. NFSv4 pNFS extensions. Technical Report draft-ietf-nfsv4-pnfs-00.txt, IETF, Oct. 2005.

[13] B. Halevy, B. Welch, J. Zelenka, and T. Pisek. Object-based pNFS Operations. Technical Report draft-ietf-nfsv4-pnfs-obj-00.txt, IETF, Jan. 2006.

[14] D. R. Hipp et al. SQLite. http://www.sqlite.org/, 2007.

[15] IBM Research. ObjectStone. http://www.haifa.il.ibm.com/projects/storage/objectstore/objectstone.html.

[16] InfiniBand Trade Association. *InfiniBand Architecture Specification*, Oct. 2004.

[17] Intel Inc. et al. Intel open storage toolkit. http://sourceforge.net/projects/intel-iscsi/, 2007.

[18] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3), July 1997.

[19] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[20] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.

[21] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer Technical Conference*, pages 137–152, 1994.

[22] O. Rodeh. Building a distributed database with device-served leases. Technical report, IBM Haifa labs, 2005.

[23] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.

[24] R. Rosner, A. Calder, J. Dursi, B. Fryxell, et al. Flash code: Studying astrophysical thermonuclear flashes. In *Computing in Science and Engineering*, volume 2, pages 33–41, Mar. 2000.

[25] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application I/O on linux clusters. In *Proceedings of SC '01*, Denver, CO, 2001.

[26] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI). Technical report, IETF RFC 3720, Apr. 2004.

[27] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. Technical report, IETF RFC 3530, Apr. 2003.

[28] Sun Inc. et al. Solaris object storage device (OSD). http://www.opensolaris.org/os/project/osd/, 2007.

[29] F. Tomonori and O. Masanori. Analysis of iSCSI target software. In *SNAPI '04: Proceedings of the international workshop on storage network architecture and parallel I/Os*, pages 25–32, 2004.

[30] S. Tweedie. Ext3, journaling filesystem. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2000.

[31] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'04)*, pages 283–300, College Park, MD, Apr. 2004.

[32] 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 Twentieth IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004.

[33] R. O. Weber. Information technology—SCSI Primary commands - 3 (SPC-2), revision 23. Technical report, INCITS Technical Committee T10/1416-D, May 2005.

[34] R. O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 1. Technical report, INCITS Technical Committee T10/1729-D, Jan. 2007.

[35] 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 OSDI'06*, pages 307–320, Seattle, WA, Nov. 2006.

[36] P. Wong and R. der Wijngaart. NAS parallel benchmarks I/O version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, Jan. 2003.