OBFS: A File System for Object-based Storage Devices

Feng Wang, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long
Storage System Research Center
University of California, Santa Cruz
Santa Cruz, CA 95064
{cyclonew, sbrandt, elm, darrell}@cs.ucsc.edu
tel +1-831-459-4458
fax +1-831-459-4829

Abstract

The object-based storage model, in which files are made up of one or more data objects stored on self-contained Object-Based Storage Devices (OSDs), is emerging as an architecture for distributed storage systems. The workload presented to the OSDs will be quite different from that of general-purpose file systems, yet many distributed file systems employ general-purpose file systems as their underlying file system. We present OBFS, a small and highly efficient file system designed for use in OSDs. Our experiments show that our user-level implementation of OBFS outperforms Linux Ext2 and Ext3 by a factor of two or three, and while OBFS is 1/25 the size of XFS, it provides only slightly lower read performance and 10%–40% higher write performance.

1. Introduction

Object-based storage systems represent files as sets of objects stored on self-contained Object-Based Storage Devices (OSDs). By distributing the objects across many devices, these systems have the potential to provide high capacity, throughput, reliability, availability and scalability. We are developing an object-based storage system with a target capacity of 2 petabytes and throughput of 100 gigabytes per second. In this system as, we expect, in many others, files will be striped across OSDs. The stripe unit size of the system will determine the maximum object size and will be the most common object size in the system. Because files will generally consist of many objects and objects will be distributed across many OSDs, there will be little locality of reference within each OSD. The workload presented to the OSDs in this system will be quite different from that of general-purpose file systems. In object-based systems that do not employ this architecture we can still expect that files will be distributed across multiple objects, objects will be distributed across multiple OSDs, and there will be little locality of reference. Even so, many distributed file systems employ general-purpose file systems as their underlying file system.

We present OBFS, a very small, highly efficient object-based file system developed for use in OSDs in large-scale distributed storage systems. The basic idea of OBFS is to optimize the disk layout

based on our knowledge of the workload. OBFS uses two block sizes: small blocks, equivalent to the blocks in general-purpose file systems, and large blocks, equal to the maximum object size, to greatly improve the object throughput while still maintaining good disk utilization. OBFS uses *regions* to collocate blocks of the same size, resulting in relatively little fragmentation as the file system ages. Compared with Linux Ext2 and Ext3 [3, 28], OBFS has better data layout and more efficiently manages the flat name space exported by OSDs. Although developed for a workload consisting mostly of large objects, OBFS does well on a mixed workload and on a workload consisting of all small objects. Thus, in addition to being highly suitable for use in high-performance computing environments where large files (and hence objects) dominate, we believe that it may also prove effective in general-purpose computing environments where small files dominate.

Our results show that our user-level implementation of OBFS outperforms Linux kernel implementations of Ext2 and Ext3 by a factor of 2 to 3, regardless of the object size. Our user-level implementation of OBFS is a little slower than a kernel implementation of XFS [19, 27] when doing object reads, but has 10% to 40% better performance on object writes. We expect the performance to improve further once we have fully implemented OBFS in the kernel to avoid extra buffer copies.

OBFS is significantly smaller than Linux XFS, using only about 2,000 lines of code compared with over 50,000 lines of code in XFS. This factor of 25 size difference and the corresponding simplicity of OBFS make OBFS easy to verify, maintain, modify, and port to other platforms. OBFS also provides strong reliability guarantees in addition to high throughput and small code size; the disk layout of OBFS allows it to update metadata with very low overhead, so OBFS updates metadata synchronously.

2. Background

A new generation of high-performance distributed file systems are being developed, motivated by the need for ever greater capacity and bandwidth. These file systems are built to support high-performance computing environments which have strong scalability and reliability requirements. To satisfy these requirements, the functionality of traditional file systems has been divided into two separate logical components: a *file manager* and a *storage manager*. The file manager is in charge of hierarchy management, naming and access control, while the storage manager handles the actual storage and retrieval of data. In large-scale distributed storage systems, the storage manager runs on many independent storage servers.

Distributed object-based storage systems, first used in Swift [6] and currently used in systems such as Lustre [4] and Slice [1], are built on this model. However, in object-based systems the storage manager is an object-based storage device (OSD or OSD) [30], which provides an object-level interface to the file data. OSDs abstract away file storage details such as allocation and scheduling, semi-independently managing all of the data storage issues and leaving all of the file metadata management to the file manager.

In a typical instance of this architecture, a metadata server cluster services all metadata requests, managing namespace, authentication, and protection, and providing clients with the file to object mapping. Clients contact the OSDs directly to retrieve the objects corresponding to the files they wish to access. One motivation behind this new architecture is to provide highly-scalable aggregate bandwidth by directly transferring data between storage devices and clients. It eliminates the file server as a bottleneck by offloading storage management to the OSDs [8] and enables load balancing and high performance by striping data from a single file across multiple OSDs. It also enables

high levels of security by using cryptographically secured capabilities and local data security mechanisms.

Much research has gone into hierarchy management, scalability, and availability of distributed file systems in projects such as AFS [18], Coda [11], GPFS [22], GFS[26] and Lustre [4], but relatively little research has been aimed toward improving the performance of the storage manager. Because modern distributed file systems may employ thousands of storage devices, even a small inefficiency in the storage manager can result in a significant loss of performance in the overall storage system. In practice, general-purpose file systems are often used as the storage manager. For example, Lustre uses the Linux Ext3 file system as its storage manager [4]. Since the workload offered to OSDs may be quite different from that of general-purpose file systems, we can build a better storage manager by matching its characteristics to the workload.

File systems such as Ext2 and Ext3 are optimized for general-purpose Unix environments in which small files dominate and the file sizes vary significantly. They have several disadvantages that limit their effectiveness in large object-based storage systems. Ext2 caches metadata updates in memory for better performance. Although it flushes the metadata back to disk periodically, it cannot provide the high reliability we require. Both Ext3 and XFS employ write-ahead logs to update the metadata changes, but the lazy log write policy used by both of them can still lose important metadata (and therefore data) in some situations.

These general-purpose file systems trade off the reliability for better performance. If we force them to synchronously update object data and metadata for better reliability, their performance degrades significantly. Our experimental results shows that in synchronous mode, their write throughput is only several MB/second. Many general-purpose file systems such as Ext2 and Ext3 use flat directories in a tree-like hierarchy, which results in relatively poor searching performance for directories of more than a thousand objects. XFS uses B+-Trees to address this problem. OBFS uses hash tables to obtain very high performance directory operations on the flat object namespace.

In our object-based storage system as, we expect, in many others, RAID-style striping with parity and/or replication is used to achieve high performance, reliability, availability, and scalability. Unlike RAID, the devices are semi-autonomous, internally managing all allocation and scheduling details for the storage they contain. The devices themselves may use RAID internally to achieve high performance. In this architecture, each stripe unit is stored in a single object. Thus, the maximum size of the objects is the stripe unit size of the distributed file system, and most of the objects will be this size. At the OSD level, objects typically have no logical relationship, presenting a flat name space. As a result, general-purpose file systems, which are usually optimized for workloads exhibiting relatively small variable-sized files, relatively small hierarchical directories, and some degree of locality, do not perform particularly well under this workload.

3. Assumptions and Design Principles

Our OBFS is designed to be the storage manager on each OSD as part of a large-scale distributed object-based storage system [13], which is currently being developed at the University of California, Santa Cruz, Storage System Research Center (SSRC). Our object-based storage system has three major components, the Metadata Server Cluster (MDSC), the Client Interface (CI), and the Storage Managers (SMs). File system functionality is partitioned among these components. The MDSC is in charge of file and directory management, authentication and protection, distributing workload among OSDs, and providing redundancy and failure recovery. The CI, running on the client

machines, provides the file system API to the application software running on the client nodes, communicates with the MDSC and SMs, and manages a local file system cache. The SMs, running on the OSDs, provide object storage and manage local request scheduling and allocation.

The operation of the storage system is as follows: Application software running on client machines make file system requests to the CIs on those machines. The CIs preprocess the requests and query the MDSC to open the files and get information used to determine which objects comprise the files. The CIs then contact the appropriate SMs to access the objects that contain the requested data, and provide that data to the applications.

In our system, objects are limited by the stripe unit size of the system. Thus, in contrast to a file, whose size may vary from bytes to terabytes, the size variance of an object is much smaller. Moreover, the delayed writes in the file cache at the client side will absorb most small writes and result in relatively large object reads and writes. We provide a more detailed analysis of the object workload characteristics in Section 4.

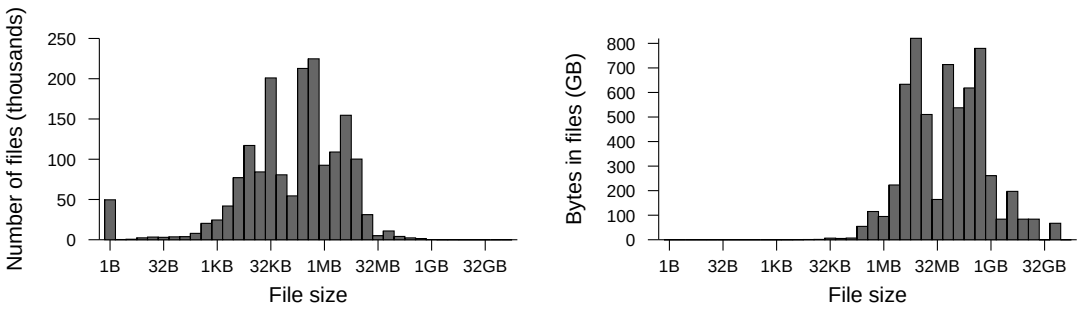
To enable parallel I/O, files are striped into fixed size objects and spread across different OSDs. The specific OSDs are selected based on the overall workload distribution intended to avoid ”hot spots” and increase potential parallelism [13]. From the viewpoint of a single OSD, incoming object accesses will be relatively random. Thus inter-object locality will be insignificant.

Most file systems cache writes for fast response, to coalesce many small writes into fewer larger ones, and to allow the file system to exploit locality of reference within the request stream. In object-based storage systems, most asynchronous writes will therefore be cached by the client. As a result, almost all of the writes to the OSDs will be synchronous. Thus, the SMs should probably not cache incoming writes in the OSDs. Furthermore, because logically contiguous data is distributed across many objects in many different OSDs, there is no locality of reference to be leveraged by caching writes of different objects.

Another caching-related concern arises due to the black-box nature of the OSDs. Because the OSDs provide a very high-level interface to the data, caching can cause the storage system as a whole to believe that the data has been saved, while data has actually been lost due to power failure or other hardware failures. While this may be addressable, we have not addressed it in this version of OBFS.

On each OSD there is a complete lack of information about relationships between objects. Thus a flat name space is used to manage the objects on each OSD. Because hundreds of thousands of objects might coexist on a single OSD, efficient searching in this flat name space is a primary requirement for the SMs.

As mentioned above, most of the incoming write requests will be synchronous. A client expects the data to be on the permanent storage when it commits its writes. This requires the OSDs to flush the objects to permanent storage before committing them. This also means the metadata of those objects should also be kept safely. In effect, OSDs in object-based storage systems are like disks in traditional storage systems, and file systems expect disks to actually store committed write requests rather than caching them.



(a) Number of files of different sizes

(b) Total bytes in files of different sizes

Figure 1. Data distribution in a large high-performance distributed storage system (data courtesy of LLNL)

4. Workload Characteristics

Very large-scale distributed file systems may exhibit very different performance characteristics than general-purpose file systems. The total volume of a large-scale distributed file system may range from several terabytes to several petabytes, orders of magnitude larger than typical general-purpose file systems. The average file size in such a distributed file system may also be much larger than that of current general-purpose file systems. Although our intent is to develop a flexible and general file system applicable in many different situations, one of our performance goals is to handle workloads encountered in high-performance computing environments with tens or hundreds of thousands of processors simultaneously accessing many files in many directories, many files in a single directory, or even a single file. These environments place extremely high demands on the storage system.

Figure 1 shows the data distribution across files in a high-performance distributed file system from Lawrence Livermore National Laboratory (LLNL). Figure 1(a) shows the file size distribution for the more than 1.5 million files in this system. Most of the files are larger than 4 KB and the majority of all files are distributed between 32 KB and 8 MB. Those files that are smaller than 4 KB (a typical block size for a general-purpose file system) only account for a very small portion of the total files. However, almost all of the disk space is occupied by files larger than 4 MB and the majority of all bytes are in files between 4 MB and 1 GB, as shown in Figure 1(b). The total number of bytes in files smaller than 256 KB is insignificant. Though the files larger than 1 GB are only a small percentage of the files, the total number of bytes in such files account for more than 15% of the bytes in the system.

The file access pattern of such systems is also different from that of a typical general-purpose file system. In the LLNL workload, most data transferred between the processors and the file system are in several megabyte chunks. Most files are accessed simultaneously by hundreds of processors. and instead of flushing dirty data directly back to the storage device, each processor caches the data in its local memory and only writes the data once the buffer is full.

Object-based storage may be used for smaller file systems as well. Systems like those traced by Roselli, *et al.* [20] have many small files; in the systems they studied, 60–70% of the bytes transferred were from files smaller than 512 KB. Clearly, an OSD file system must also be able to efficiently handle workloads composed primarily of small objects.

For the OSDs to achieve the high throughput required of the system and to fully take advantage of the object-based storage model, our system stripes file data across the OSDs. This is a very compelling choice, analogous to that of earlier systems such as Swift [6] and Zebra [9], and we believe that this will be an architecture of choice in large-scale object-based storage systems. In such systems, each object stored on an OSD will be a stripe unit (or partial stripe unit) of data from a file.

The system stripe unit size depends on the design requirements of the individual system. Stripe units that are too small will decrease the throughput of each OSD while stripe units that are too large will decrease the potential parallelism of each file. Assuming a stripe unit size of 512 KB, large files will be divided into several 512 KB objects and files smaller than 512 KB will be stored in a single object. Consequently, no object in the system will ever exceed the system stripe unit size. In the LLNL workload we estimate that about 85% of all objects will be 512 KB and 15% of all objects will be smaller than 512 KB. We will refer to objects that are the same size as the system stripe unit size as *large objects* and the rest as *small objects*. Workstation workloads [20] will likely have more small objects and fewer large objects.

Because the object-based storage system is expected to spread the objects evenly across all of the OSD devices, the object size distribution in the workload of a single OSD device will be the same as that of the larger storage system. Thus, a single OSD device under the LLNL workload should expect that 85% of incoming objects are large objects and the rest are small objects. Since files are distributed across many OSDs and directory hierarchies are managed above the OSD level, there is no inter-object locality that can be exploited in the OSDs. The workload of OSDs in this type of system will be dominated by large fixed-size objects exhibiting no inter-object locality. Under workstation workloads, in contrast, the object size distribution will be closer to 25% large objects and 75% small objects. An OSD file system should be able to handle either type of workload.

5. Design and Implementation

As described in Section 4, the expected workload of our OSDs is composed of many objects whose sizes range from a few bytes up to the file system stripe unit size. Therefore, OBFS needs to optimize large object performance to provide substantially higher overall throughput, but without overcommitting resources to small objects. Simply increasing the file system block size can provide the throughput needed for large objects, but at the cost of wasted storage space due to internal fragmentation for small objects. For the LLNL workload, more than 10% of the available storage space would be wasted if 512 KB blocks are used, while less than 1% of the space would be lost if 4 KB blocks are used. In a 2 PB storage system, this 9% difference represents about 18 TB. The situation is even worse for a workstation file system, where 512 KB blocks would waste more than 50% of the space in such a system.

To use large blocks without wasting space, small objects must be stored in a more efficient way. OBFS therefore employs multiple block sizes and uses *regions*, analogous to cylinder groups in FFS [15], to keep blocks of the same size together. Thus, the read/write performance of large objects can be greatly improved by using very large blocks, while small objects can be efficiently stored using small blocks.

Another important feature of OBFS is the use of a flat name space. As the low-level storage manager in an object-based distributed file system, OBFS has no information about the logical relationship among objects. No directory information is available and no useful locality information is likely to

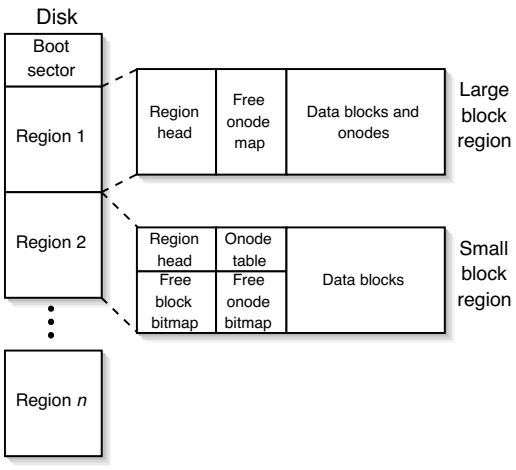


Figure 2. OBFS structure

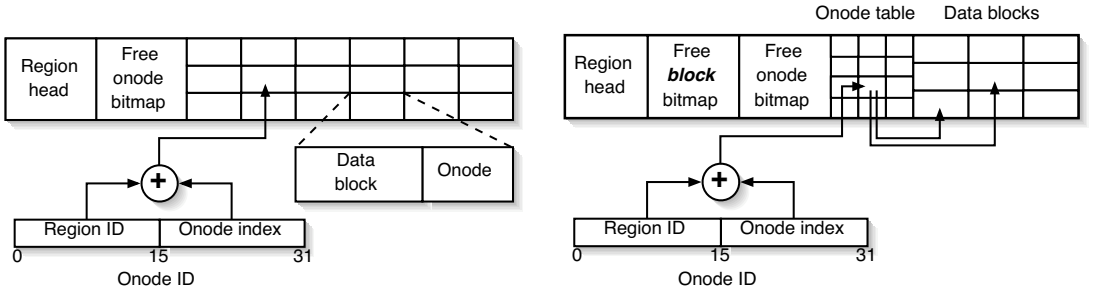
be available. Note that in a small system where an OSD may hold several objects of a file, some locality information may available, but this does not extend to multiple files in the same directory or other tidbits that are useful to general-purpose file systems. Many general-purpose file systems such as Linux Ext2 are extremely inefficient in managing very large directories due to the fact that they do linear search, resulting in $O(n)$ performance on simple directory operations. To avoid this, OBFS uses hash tables (like Ext3 [28]) to organize the objects and achieve much higher performance on directory operations.

5.1. Regions and Variable-Size Blocks

The user-level implementation of OBFS separates the raw disk into regions. As shown in Figure 2, regions are located in fixed positions on the disk and have uniform sizes. All of the blocks in a region have the same size, but the block sizes in different regions may be different. The block size in a free region is undefined until that region is initialized. Regions are initialized when there are insufficient free blocks in any initialized region to satisfy a write request. In this case, OBFS allocates a free region and initializes all of its blocks to the desired block size. When all of the blocks in a used region are freed, OBFS returns the region to the free region list.

Although our region policy supports as many different block sizes as there are regions, too many different block sizes will make space allocation and data management excessively complicated. In our current implementation, OBFS uses two block sizes: small and large. Small blocks are 4 KB, the logical block size in Linux, and large blocks are 512 KB, the system stripe unit size and twice the block size of GPFS (256 KB). Those regions that contain large blocks are called *large block regions* and those regions that contain small blocks are called *small block regions*. With this strategy, large objects can be laid out contiguously on disk in a single large block. The throughput of large objects is greatly improved by the reduction in seek time and reduced metadata operations that are inherent in such a design. Only one disk seek is incurred during the transfer of a large object. OBFS eliminates additional operations on metadata by removing the need for indirect blocks for large objects. Dividing the file system into regions also reduces the size of other FS data structures such as free block lists or maps and thus make the operations on those data structures more efficient.

This scheme reduces file system fragmentation, avoids unnecessary wasted space and more effectively uses the available disk bandwidth. By separating the large blocks in different regions from the



(a) Large Block Region (b) Small Block Region

Figure 3. Region structure and data layout.

small blocks, OBFS can reserve contiguous space for large objects and prevent small objects from using too much space. Region fragmentation will only become a problem in the rare case that the ratio of large to small objects changes significantly during the lifetime of the system, as described in Section 5.6.

Higher throughput in OBFS does not come at the cost of wasted disk space. Internal fragmentation in OBFS is no worse than in a general-purpose Unix file system because the small block size in OBFS is the same as the block size in most Unix file systems. Large blocks do not waste much space because they are only used for objects that will fill or nearly fill the blocks. The only wasted space will be due to objects stored in large blocks that are nearly, but not quite, as large as a stripe unit. This can be limited with a suitable size threshold for selecting the block size to use for an object. One minor complication can occur if an object starts small and then grows past this threshold. Our current implementation recopies the object into a large block when this occurs. Although this sounds expensive, it will happen rarely enough (due to aggressive write coalescing in the client caches) that it does not have a significant impact on system performance, and the inter-region locality of the small blocks makes this a very efficient operation.

5.2. Object Metadata

Object metadata, referred as an *onode*, is used to track the status of each object. Onodes are preallocated in fixed positions at the head of small block regions, similar to the way inodes are placed in cylinder groups in FFS [15]. In large block regions, shown in Figure 3, onodes are packed together with the data block on the disk, similar to embedded inodes [7]. This allows for very low overhead metadata updates as the metadata can be written with the corresponding data block.

Figure 3 shows that each onode has a unique 32-bit identifier consisting of two parts: a 16 bit region identifier and a 16 bit in-region object identifier. If a region occupies 256 MB on disk, this scheme will support OSDs of up to 16 TB, and larger OSDs are possible with larger regions. To locate a desired object, OBFS first finds the region using the region identifier and then uses the in-region object identifier to index the onode. This is particularly effective for large objects because the object index points directly to the onode and the object data, which are stored contiguously.

In the current implementation, onodes for both large and small objects are 512 bytes, allowing OBFS to avoid using indirect blocks entirely. The maximum size of a small object will always be less than the stripe unit size, which is 512 KB in our design. Because the OBFS layout policy assigns objects to a single region, we can use the relative address to track the blocks. Assuming the

region size is 256 MB and the small block size is 4 KB, there will be fewer than 2^{16} small blocks in a region, allowing a two-byte addresses to index all of the blocks in the region. In the worse case, a small object will be a little smaller than 512 KB, requiring 128 data blocks. Thus, the maximum amount of space that may be needed to index the small blocks in an object is 256 bytes, which can easily fit into a 512 byte onode.

5.3. Object Lookup

Given an object identifier, we need to retrieve the object from the disk. In a hierarchical name space, data lookup is implemented by following the path associated with the object to the destination directory and searching (often linearly) for the object in that directory. In a flat name space, linear search is prohibitively expensive, so OBFS uses a hash table, the *Object Lookup Table* (OLT), to manage the mapping between the object identifier and the onode identifier. Each valid object has an entry in the OLT that records the object identifier and the corresponding onode identifier. The size of the OLT is proportional to the number of objects in the OSD: with 20,000 objects residing in an OSD, the OLT requires 233 KB. For efficiency, the OLT is loaded into main memory and updated asynchronously.

Each region has a region head which stores information about the region, including pointers to the free block bitmap and the free onode bitmap. All of the region heads are linked into the Region Head List (RHL). On an 80 GB disk, the RHL occupies 8 MB of disk space. Like the OLT, the RHL is loaded into memory and updated asynchronously. After obtaining an onode identifier, OBFS searches the RHL using the upper 16 bits of the onode identifier to obtain the corresponding region type. If the onode belongs to a large block region, the object data address can be directly calculated. Otherwise, OBFS searches the in-memory onode cache to find that onode. A disk copy of the onode will be loaded into the onode cache if the search fails.

5.4. Disk Layout Policy

The disk layout policy of OBFS is quite simple. For each incoming request, OBFS first decides what type of block(s) the object should use. If the object size is above the utilization threshold of the large blocks, a large block is assigned to the object; otherwise, it uses small blocks.

For those objects that use large blocks, OBFS only needs to find the nearest large-block region that contains a free block, mark it as used, and write the object to that block. For objects that use small blocks, an FFS-like allocation policy is employed. OBFS searches the active region list to find the nearest region that has enough free small blocks for the incoming object. After identifying a region with sufficient space, OBFS tries to find a contiguous chunk of free blocks that is large enough for the incoming object. If such a chunk of blocks is not available, the largest contiguous chunk of blocks in that region is assigned to the object. The amount of space allocated in this step is subtracted from the object size, and the process is repeated until the entire object is stored within the region. If no region has the desired number and type of free blocks, the nearest free region will be initialized and put into the active region list. The incoming object will then be allocated to this new region.

The OBFS data allocation policy guarantees that each of the large objects is allocated contiguously and each of the small objects is allocated in a single region. No extra seeks are needed during a large object transfer and only short seeks are needed to read or write the small objects, no matter how long the file system has been running. Compared with a general-purpose file system, OBFS

is much less fragmented after running for a long time, minimizing performance degradation as the system ages.

5.5. Reliability and Integrity

As mentioned in section 5.3, OBFS asynchronously updates important data structures such as the OLT and the RHL to achieve better performance. In order to guarantee system reliability, OBFS updates some important information in the onodes synchronously. If the system crashes, OBFS will scan all of the onodes on the disk to rebuild the OLT and the RHL. For each object, the object identifier and the region identifier are used to assemble a new entry in the OLT. The block addresses for each object are then used to rebuild each region free block bitmap. Because the onodes are synchronously updated, we can eventually rebuild the OLT and RHL and restore the system. As mentioned above, OBFS updates metadata either without an extra disk seek or with one short disk seek. In so doing, it keeps the file system reliable and maintain system integrity with very little overhead.

5.6. Region Cleaning

Since OBFS uses regions to organize different types of blocks, one potential problem is that there will be no free regions and no free space in regions of the desired type. Unlike LFS [21], which must clean segments on a regular basis, OBFS will *never* need cleaning unless the ratio between large and small objects changes significantly over time on an OSD which has been nearly full. This can only happen when the object size characteristic of the workload changes significantly when the file system is near its capacity. We do not expect this to happen very often in practice. However, if it happens, it can result in many full regions of one type, many underutilized regions of the other type, and no free regions. In this situation, the cleaner can coalesce the data in the underutilized regions and create free regions which can be used for regions of desired type.

If all of the regions are highly utilized, cleaning will not help much: the disk is simply full. Low utilization regions can only be produced when many objects are written to disk and then deleted, leaving “holes” in regions. However, unlike in LFS, these holes are reused for new objects without the need for cleaning. The only time cleaning is needed is when all of the holes are in the wrong kind of region e.g., the holes are in small block regions, and OBFS is trying to write a large block. This situation only occurs when the ratio between large objects and small objects changes. In our experiments, we only observed the need for the cleaner when we artificially changed the workload ratios on a nearly full disk.

Because cleaning is rarely, if ever, necessary, it will have a negligible impact on OBFS performance. However, cleaning can be used to improve file system performance by defragmenting small-block regions to keep blocks of individual objects together. This process would copy all used blocks of a region to a free region on the disk, sorting the blocks as it goes. Because this would occur on a region-by-region basis and because a new region will always have enough free space for all of the blocks in an old region, it would be trivial to implement. The system need never do this, however.

6. OBFS Performance

We compared the performance of OBFS to that of Linux Ext2, Ext3 and XFS. Ext2 is a widely-used general-purpose file system. Ext3 is used by Lustre for object storage and has the same disk layout as Ext2 but adds a journal for reliability. XFS is a modern high-performance general-purpose file

Capacity	80 GB
Controller	Ultra ATA/133
Track-to-track seek	0.8 ms
Average seek	8.5 ms
Rotation speed	7200 RPM
Sustained transfer rate	24.2–44.4 MB/s

Table 1. Specifications of the Maxtor D740X-6L disk used in the experiments

system that uses B-trees and extent-based allocation. While Ext2, Ext3, and XFS run as in-kernel file systems, the version of OBFS used in these experiments is a user-level file system. An in-kernel implementation of OBFS would take advantage of the very effective caching provided by the Linux kernel, but our user-level implementation cannot. Thus, in order to allow for a fair comparison, we executed the following experiments with the system buffer cache bypassed: all of the file systems were mounted using the "-o sync" parameter, which forced the system buffer cache to use a write-through policy. The results generated evaluates disk layout policies of different file systems. With caching enabled, all three file systems will achieve higher performance. We expect the performance change of OBFS to be comparable to those of XFS, Ext2, and Ext3.

6.1. Experimental Setup

All of the experiments were executed on a PC with a 1 GHZ Pentium III CPU and 512 MB of RAM, running Red Hat Linux, kernel version 2.4.0. To examine the performance of the file systems with minimal impact from other operating system activities, we dedicated an 80 GB Maxtor D740X-6L disk (see Table 1) to the experiments. This disk was divided into multiple 8 GB partitions. The first partition was used to install file systems and run experiments. The rest were used to backup aged file system images. We used aged file systems to more accurately measure the long-term performance of the file systems. For each experiment, we copied an aged file system to the first partition of the disk, unmounted the disk and rebooted Linux to clean the buffer cache, then mounted the aged partition to run the benchmarks. We repeated these steps three times and took the average of the performance numbers obtained.

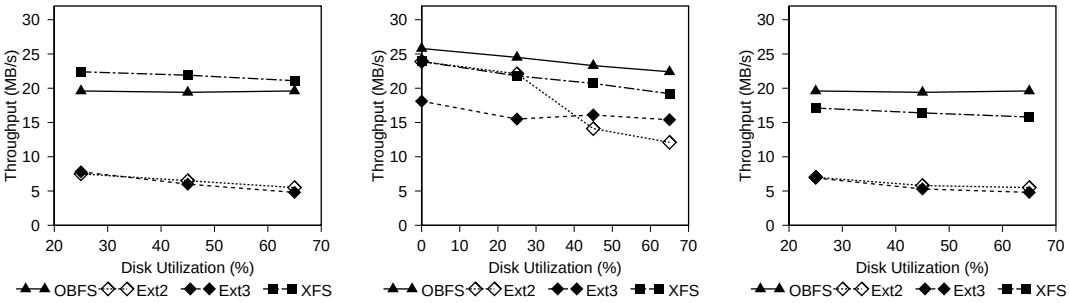
Smith, *et al.* [25] used file system snapshots and traces to approximate the possible activities in file systems. No object-based storage system snapshots are currently available so we used the simplest approach: generate 200,000 to 300,000 randomly distributed create and delete requests and feed these requests to a new file system. The create/delete ratio was dynamically adjusted based on the disk usage, which guaranteed that it neither filled nor emptied the available disk space.

Because our user-level implementation of OBFS bypasses the buffer cache, all three file systems were forced to use synchronous file I/O to allow for a fair comparison of the performance. Ext2 uses asynchronous metadata I/O to achieve high throughput even if synchronous file I/O is used, so we mounted the partitions in synchronous mode to force them to always flush the data in the buffer cache back to disk.

The benchmarks we used consisted of semi-random sequences of object requests whose characteristics were derived from the LLNL workload described in Section 4. On average, 80% of all objects were large objects (512 KB). The rest were small objects whose size was uniformly distributed between 1 KB and 512 KB. To examine the performance of the various file system, we generated two kinds of benchmarks: microbenchmarks and macrobenchmarks. Our microbenchmarks each

	Benchmark I	Benchmark II
	# of ops(total size)	# of ops(total size)
Reads	16854 (7.4GB)	4049 (1.8GB)
Writes	4577 (2.0GB)	8969 (4.0GB)
Rewrites	4214 (1.8GB)	8531 (3.8GB)
Deletes	4356 (1.9GB)	8147 (3.9GB)
Sum	30001 (13.1GB)	29696 (12.5GB)

Table 2. Benchmark parameters



(a) Reads (b) Writes (c) Rewrites

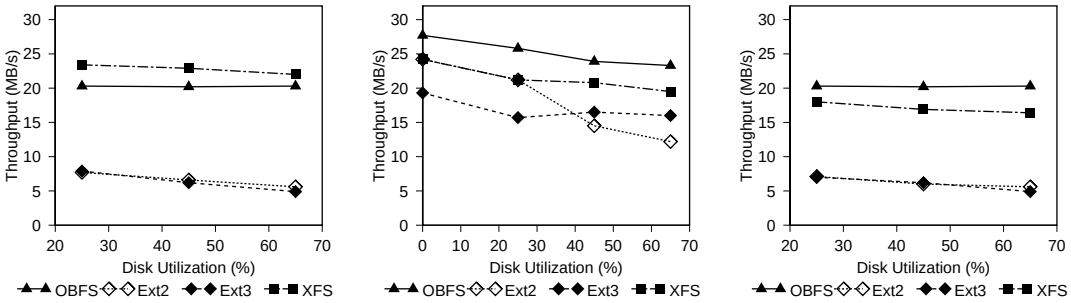
Figure 4. Performance on a workload of mixed-size objects.

consisted of 10,000 requests of a single request type—read, write, or rewrite—and allowed us to examine the performance of the file systems on that request type. Our macrobenchmarks consisted of synthetic workloads composed of create, read, rewrite, and delete operations in ratios determined by the workload mentioned above. These allowed us to examine the performance of the file systems on the expected workload. We used two different macrobenchmarks, Benchmark I and Benchmark II, whose parameters are listed in table 2. Benchmark I is a read-intensive workload in which reads account for 56% of all requests and the total size of the read requests is around 7.4 GB. The writes, rewrites, and deletes account for 15.3%, 14.0%, and 14.5% of the requests. In Benchmark II, reads account for 13.6% of the requests and writes, rewrites, and deletes account for 29.8%, 28.4%, and 27.1%.

6.2. Results

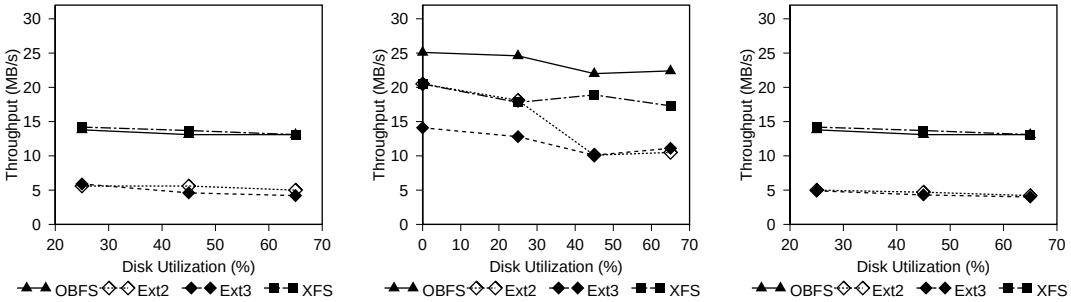
Figure 4 shows the performance of Ext2, Ext3, XFS, and OBFS on a mixed workload consisting of 80% large objects and 20% small objects[1]. As seen in Figure 4(b), OBFS exhibits very good write performance, almost twice that of Ext2 and Ext3 and 10% to 20% better than XFS. The large block scheme of OBFS contributes a lot to the strong write performance. With large blocks, contiguous space has been reserved for the large objects, allowing large objects to be written with only one disk seek. Because OBFS uses regions to organize large and small blocks, limiting the amount of external fragmentation, the performance of OBFS remains good as disk usage increases. At the same time, the performance of Ext2 and Ext3 drops significantly due to the insufficient availability of large contiguous regions, as seen in Figures 4(b), 5(b), and 6(b).

[1]Note that in all of the microbenchmark graphs write performance is displayed starting at 0% disk utilization but because reads and rewrites cannot be done on an empty disk we chose to start those experiments at 25% utilization.



(a) Reads (b) Writes (c) Rewrites

Figure 5. Performance on a workload of large objects.



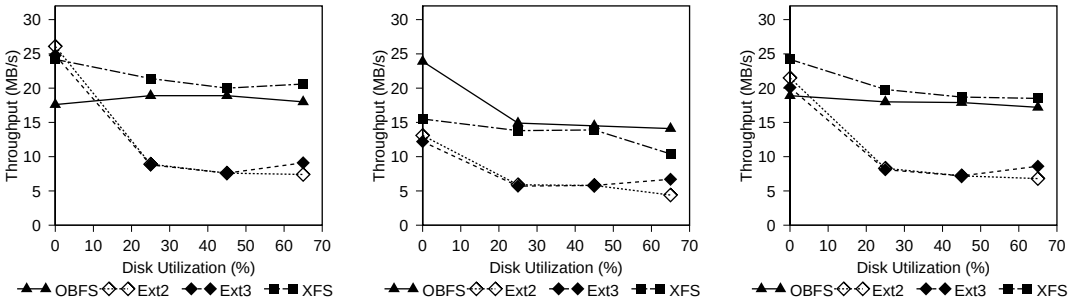
(a) Reads (b) Writes (c) Rewrites

Figure 6. Performance on a workload of small objects.

OBFS outperforms Ext2 and Ext3 by nearly 3 times, but is about 10% slower than XFS, as Figure 4(a) shows. We suspect that a more optimized implementation of XFS contributes to its slightly better read performance. As seen in Figure 4(c), the rewrite performance of OBFS beats that of Ext2 and Ext3 by about 3 times, and beats XFS by about 20–30%. The poor performance of Ext2 and Ext3 in both read and rewrite can be explained by their allocation policies and small blocks. XFS uses extents rather than blocks to organize files, so most files can get contiguous space. This results in excellent performance in both read and write. However, OBFS still shows slightly better performance on object rewrite.

Figure 5 shows the performance of the four file systems on large objects and Figure 6 shows the performance on small objects. Figure 5 is almost the same as Figure 4 because large objects dominate the mixed workload of Figure 4. In Figure 6, we see that OBFS meets the performance of XFS, almost triples the performance of Ext2 and Ext3 when doing reads and rewrites, and exceeds the performance of all three when doing creates.

The benchmark results are shown in Figures 7 and 8. As described above, Benchmark I is a read-intensive workload and Benchmark II is a write-intensive workload. Notice that in our benchmarks, XFS beats both Ext2 and Ext3 by a large margin in all cases; this differs from other benchmark studies [5] that found that Ext2 and XFS have comparable performance. There are three factors in our experiments that favor XFS over Ext2 and Ext3. First, our benchmarks include many large objects, which benefit from XFS extent-based allocation, especially when disk utilization is high. Second, while other benchmarks used fresh disks, our benchmarks use disks subjected to long-term aging [25] to reflect more realistic scenarios. After aging, the performance of Ext2 and Ext3 drops



(a) Reads (b) Writes (c) Overall

Figure 7. Performance under Benchmark I.

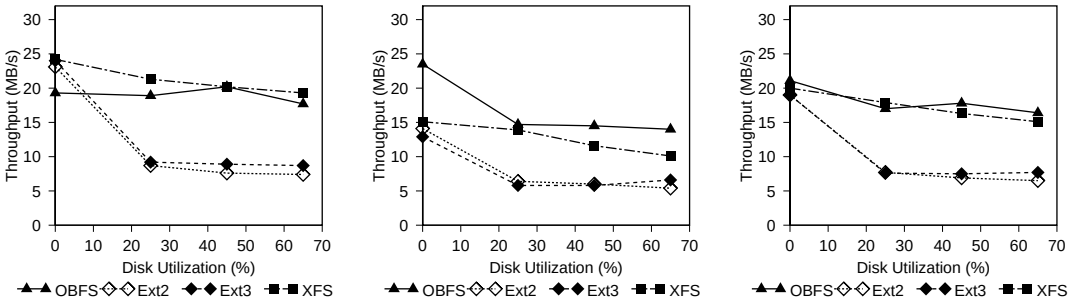
dramatically after aging due to disk fragmentation, while XFS maintains good performance because of its extent-based allocation policy. Third, in our object-based benchmarks, we assume a flat name space in which all objects are allocated in the root directory in all file systems. The linear search of directories used by Ext2 and Ext3 performs poorly when the number of objects scales to tens of thousands. XFS uses B+trees to store its directories, ensuring fast name lookup even in very large directories.

Ext2 and Ext3 outperform OBFS when the disk is nearly empty, as shown in Figures 7(a) and 8(a). This is due in part to the significant likelihood that an object will be in the buffer cache because of the low number of objects that exist in a nearly empty disk. For example, an 8 GB partition at 10% utilization has only 800 MB of data. With 512 MB of main memory, most objects will be in memory and Linux Ext2, Ext3 and XFS reads will proceed at memory speeds while our user-level implementation of OBFS gains no advantage from the buffer cache and must therefore always access the disk. However, as disk usage increases, this effect is minimized and Ext2 and Ext3 read performance decreases rapidly while OBFS performance remains essentially constant. The net result is that OBFS read performance is two or three times that of Ext2 and Ext3. OBFS is still about 10% slower than XFS on reads, similar to the results from earlier read microbenchmarks. OBFS outperforms all three other file systems on writes, however, as Figures 7(b) and 8(b) show. For writes, OBFS is 30–40% faster than XFS and 2–3 times faster than Ext3. Overall, OBFS and XFS are within 10% of each other on the two macrobenchmarks, with one file system winning each benchmark. OBFS clearly beats both Ext2 and Ext3, however, running three times faster on both benchmarks.

Although our macrobenchmarks focused on large-object performance, Figure 6 shows that OBFS meets or exceeds the performance of the other file systems on a workload consisting entirely of small objects, those less than 512 KB. OBFS doubles or triples the performance of Ext2 and Ext3 and matches that of XFS on reads and rewrites and exceeds it by about 25% on writes. As OBFS also does well on large objects, we conclude that it is as well suited to general-purpose object-based storage system workloads as it is to terascale high-performance object-based storage system workloads.

7. Related Work

Many other file systems have been proposed for storing data on disk; however, nearly all of them have been optimized for storing files rather than objects. The Berkeley Fast File System (FFS) [15] and related file systems such as Ext2 and Ext3 [28] are widely used today. They all try to store



(a) Reads (b) Writes (c) Overall

Figure 8. Performance under Benchmark II.

file data contiguously in *cylinder groups*—regions with thousands of contiguous disk blocks. This strategy can lead to fragmentation so techniques such as extents and clustering [16, 24] are used to group blocks together to decrease seek time. Analysis [23, 24] has shown that clustering can improve performance by a factor of two or three, but it is difficult to find contiguous blocks for clustered allocation in aged file systems.

Log-structured file systems [21] group data by optimizing the file system for writes rather than reads, writing data and metadata to segments of the disk as it arrives. This works well if files are written in their entirety, but can suffer on an active file system because files can be interleaved, scattering a file's data among many segments. In addition, log-structured file systems require cleaning, which can reduce overall performance [2].

XFS [19, 27] is a highly optimized file system that uses extents and B-trees to provide high performance. This performance comes at a cost: the file system has grown from 50,000 to nearly 200,000 lines of code, making it potentially less reliable and less attractive for commodity storage devices because such devices cannot afford data corruption due to file system errors. In addition, porting such a file system is a major effort [19].

Gibson, *et al.* have proposed network-attached storage devices [8], but spent little time describing the internal data layout of such devices. WAFL [10], a file system for network-attached storage servers that can write data and metadata to any free location, is optimized for huge numbers of small files distributed over many centrally-controlled disks.

Many scalable storage systems such as GPFS [22], GFS [26], Petal [12], Swift [6], RAMA [17], Slice [1] and Zebra [9] stripe files across individual storage servers. These designs are most similar to the file systems that will use OSDs for data storage; Slice explicitly discusses the use of OSDs to store data [1]. In systems such as GFS, clients manage low-level allocation, making the system less scalable. Systems such as Zebra, Slice, Petal, and RAMA leave allocation to the individual storage servers, reducing the bottlenecks; such file systems can take advantage of our file system running on an OSD. In GPFS, allocation is done in large blocks, allowing the file system to guarantee few disk seeks, but resulting in very low storage utilization for small files.

Existing file systems must do more than allocate data. They must also manage large amounts of metadata and directory information. Most systems do not store data contiguously with metadata, decreasing performance because of the need for multiple writes. Log-structured file systems and embedded inodes [7] store metadata and data contiguously, avoiding this problem, though they still

suffer from the need to update a directory tree correctly. Techniques such as logging [29] and soft updates [14] can reduce the penalty associated with metadata writes, but cannot eliminate it.

8. Conclusions

Object-based storage systems are a promising architecture for large-scale high-performance distributed storage systems. By simplifying and distributing the storage management problem, they provide both performance and scalability. Through standard striping, replication, and parity techniques they can also provide high availability and reliability. However, the workload characteristics observed by OSDs will be quite different from those of general purpose file systems in terms of size distributions, locality of reference, and other characteristics.

To address the needs of such systems, we have developed OBFS, a very small and highly efficient file system targeted specifically for the workloads that will be seen by these object-based storage devices. OBFS currently uses two block sizes: small blocks, roughly equivalent to the blocks in general purpose file systems, and large blocks, equal to the maximum object size. Blocks are laid out in regions that contain both the object data and the onodes for the objects. Free blocks of the appropriate size are allocated sequentially, with no effort made to enforce locality beyond single-region object allocation and the collocation of objects and their onodes.

At present, we have tested OBFS as a user-level file system. Our experiments show that the throughput of OBFS is two to three times that of Linux Ext2 and Ext3, regardless of the object size. OBFS provides slightly lower read performance than Linux XFS, but 10%–40% higher write performance. At a fraction of the size of XFS—2,000 lines of code versus over 50,000 for XFS—OBFS is both smaller and more efficient, making it more suitable for compact embedded implementations. Ultimately, because of its small size and simplicity, we expect that it will also prove to be both more robust and more maintainable than XFS, Ext2, or Ext3.

Finally, we successfully implemented a kernel-level version of the OBFS file system in about three person-weeks. The short implementation time was possible because of OBFS's simplicity and very compact code. At present the performance of our in-kernel implementation does not match that of our user-level implementation because our carefully managed large blocks get broken into small blocks by the Linux buffer management layer, as encountered by the XFS developers. We intend to rewrite the buffer management code, as they did, to avoid this problem. With this change, we expect the in-kernel OBFS performance to exceed that of the user-level implementation, further solidifying OBFS's advantage over general-purpose file systems for use in object-based storage devices.

Acknowledgments

This research was supported by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. The Storage Systems Research Center is supported in part by gifts from Hewlett Packard, IBM, Intel, LSI Logic, Microsoft, Overland Storage, and Veritas.

References

[1] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000.

[2] T. Blackwell, J. Harris, , and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 277–288. USENIX, Jan. 1995.

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, Oct. 2000.

[4] P. J. Braam. The Lustre storage architecture, 2002.

[5] R. Bryant, R. Forester, and J. Hawkes. Filesystem performance and scalability in Linux 2.4.17. In *Proceedings of the Freenix Track: 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

[6] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[7] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, Jan. 1997.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[9] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[10] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[12] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, 1996.

[13] D. Long, S. Brandt, E. Miller, F. Wang, Y. Lin, L. Xue, and Q. Xin. Design and implementation of large scale object-based storage system. Technical Report ucsc-crl-02-35, University of California, Santa Cruz, Nov. 2002.

[14] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the Fast File System. In *Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference*, pages 1–18, June 1999.

[15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[16] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the Winter 1991 USENIX Technical Conference*, pages 33–44. USENIX, Jan. 1991.

[17] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, 1997.

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

[19] J. Mostek, B. Earl, S. Levine, S. Lord, R. Cattelan, K. McDonell, T. Kline, B. Gaffey, and R. Ananthanarayanan. Porting the SGI XFS file system to Linux. In *Proceedings of the Freenix Track: 2000 USENIX Annual Technical Conference*, pages 65–76, San Diego, CA, June 2000. USENIX.

[20] 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.

[21] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[22] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.

[23] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 249–264, 1995.

[24] K. A. Smith and M. I. Seltzer. A comparison of FFS disk allocation policies. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 15–26, 1996.

[25] K. A. Smith and M. I. Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 203–213, 1997.

[26] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 319–342, College Park, MD, 1996.

[27] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 1–14, Jan. 1996.

[28] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux EXT2/EXT3 filesystem. In *Proceedings of the Freenix Track: 2002 USENIX Annual Technical Conference*, pages 235–244, Monterey, CA, June 2002. USENIX.

[29] U. Vahalia, C. G. Gray, and D. Ting. Metadata logging in an NFS server. In *Proceedings of the Winter 1995 USENIX Technical Conference*, New Orleans, LA, Jan. 1995. USENIX.

[30] R. O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug. 2002.