UNIVERSITY OF CALIFORNIA
SANTA CRUZ

STORAGE MANAGEMENT IN LARGE DISTRIBUTED OBJECT-BASED STORAGE SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Feng Wang

December 2006

The Dissertation of Feng Wang
is approved:

Professor Scott A. Brandt, Chair

Professor Ethan L. Miller

Professor Darrell D. E. Long

Doctor Richard A. Golding

Professor Carlos Maltzahn

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Storage Management in Large Distributed Object-Based Storage Systems

by

Feng Wang

Driven by the requirements for extremely high bandwidth and large capacity, storage subsystem architectures are undergoing fundamental changes. The object-based storage model, which repartitions the file system functionalities and offloads the storage management functions to intelligent storage devices, is a promising new model. In this model, Object-based Storage Devices (OSDs) manage their own storage space, provide persistent storage, and export an object interface to the data. With abundant on-board computing power, OSDs are able to provide complex services to facilitate high level system designs.

This thesis focuses on the design, performance and functionality of an individual OSD in a large distributed object-based storage system, currently being developed in the Storage Systems Research Center at the University of California, Santa Cruz. Based on the file system workload analysis and the expected object workload studies, I extract unique features of object workloads and design an efficient storage manager, named OBFS, for individual OSDs. OBFS employs variable-sized blocks to optimize disk layouts and improve object throughput. Object metadata and attributes are also laid contiguously together with data, which further improves disk bandwidth utilization. The physical storage space is partitioned into fixed-size regions to organize blocks with different sizes together, which effectively reduces file system fragmentation in the long run. A series of experiments was conducted to evaluate OBFS performance compared with two Linux file systems, Ext2 and XFS. The results show that OBFS successfully limits the file system fragmentation even after long term aging. OBFS demonstrates very good synchronous write performance, exceeding those of Ext2 and XFS by up to 80% on both fresh systems and aged systems. Its asynchronous write performance is about 5% to 10% lower than that of Ext2, but 20% to 30% higher than that of XFS on a lightly used disk. On a heavily used disk, OBFS beats both Ext2 and XFS by 20%. For read operations, the performance of OBFS almost doubles that of Ext2 and is only slightly slower than that of XFS. Overall, OBFS achieves 30% to 40% performance improvements over Ext2 and XFS under expected object workloads, and forms a fundamental building block for larger distributed storage systems.

Acknowledgements

I thank my advisor, Scott A. Brandt for his support, guidance, friendship, and patience. I have been fortunate to work with him in the last five years. He poured uncountable hours into my research as well as my development in general.

I gratefully acknowledge Darrell D.E. Long, Ethan L. Miller, and Richard Golding for their collaborations and suggestions in my research. I also thank Carlos Maltzahn for his generous service in my thesis committee. I specially thank Tyce T. McLarty for providing file system traces and valuable suggestions on this work. The students in the Storage Systems Research Center have been very helpful and supportive over the years. In particular, I thank Bo Hong, Qin Xin, Scott Banachowski, and Caixue Lin, for helping me in ways too numerous to mention.

This research was supported by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714.

To my parents, Yanlin Wang and Xianrong Li,

and my wife, Jing Liu

Chapter 1

Introduction

Simulations on high performance computer systems produce very large data sets. Rapid storage and retrieval of these data sets presents major challenges for high-performance computing and visualization systems. Although computing power and disk capacity have both increased at exponential rates over the past decade, disk bandwidth has lagged far behind. To satisfy the increasing demands for high data throughput, many distributed file systems employ storage clusters to achieve high aggregate bandwidth. However, the traditional file system model may not work well in such environments. The excessive data forwarding as well as the complex file management tasks may make the file servers the bottleneck of the whole system, with disk bandwidth wasted waiting for the file servers to handle requests and forward data from/to clients. High speed *storage area networks* [75, 1] and network attached storage (NAS) enable incremental scaling of bandwidth and capacity by incorporating more storage devices into the network. However, those devices still provide a block-level interface, leaving most of the low-level storage management tasks to file servers, which limits the scalability of such systems.

1.1 Object-based Storage

Object-based storage systems [18, 42] address these limitations through a simple networked data storage unit, the Object Storage Device (OSD). Each OSD consists of a CPU, network interface, local cache, and storage device (disk or small RAID configuration), and exports a high-level data object abstraction on top of the disk (or RAID) block read/write interface. By managing low-level storage details such as allocation and disk request scheduling locally, OSDs provide a building-block for scalability.

OSDs store data in objects. The object is a logical abstraction between the file and the disk block: files are comprised of data objects stored on (possibly many) OSDs and the OSDs handle the translation from objects to disk blocks. Objects hide the physical details related to the disk, while providing better granularity.

The object-based storage model is different from the traditional server-based storage model in that it separates the storage management from the file hierarchy management. The storage management functionalities, such as data allocation, block mapping, and request scheduling, are offloaded to the storage nodes. File servers handle the high-level issues, such as naming, hierarchy management and access control. The storage devices are in charge of the low-level storage management, such as data allocation, block mapping and request scheduling. This reduces the load on the file servers, which now only have to service metadata requests, and enables direct, highly parallel data transfer between clients and storage nodes. Together with the distribution of low-level storage management to the OSDs, this greatly improves the scalability of file systems.

An object-based storage system may be composed of a metadata server cluster (replacing the traditional file servers), multiple OSDs, and a client interface for individual clients, all connected through a high performance storage area network. The metadata cluster handles all namespace related requests, while the OSD clusters are in charge of storing and retrieving the actual file data. The client component talks to both metadata servers and OSDs using a special object protocol. In such a system, metadata operations are decoupled from read and write operations. A client wishing to open a file will first contact the metadata servers to obtain a capability and information allowing it to locate the objects containing file data in the OSD cluster. Subsequent read and write traffic then involves only the client and relevant OSDs without significant metadata server interaction [1]. Such an arrangement allows maximum scalability for system throughput since the primary bottleneck limiting I/O transfer rates becomes the network and the raw number of OSDs, instead of any single component within the system.

1.2 Ceph Storage System

The Ceph Storage System [37], currently being developed at the UC Santa Cruz Storage System Research Center, is a large distributed object-based storage system based on the object-based storage model. It aims to provide petabytes of storage and I/O bandwidth up to one terabyte per second. In Ceph, the contents of each file are striped over a sequence of

[1]Some interaction is required to update metadata as the file is accessed/modified. Lazy update potentially minimizes this interaction.

objects stored on OSDs using a fixed stripe unit size. Files smaller than the stripe unit size and the tail ends of large files are placed in individual objects. Objects are placed on different OSDs according to the *RUSH* [25] policy, which attempts to balance load uniformly among OSDs and minimize the occurrences of OSD hotspots by pseudo-randomly assigning objects to OSDs. Ceph is discussed further in Chapter 2.

The Object-based storage model, as implemented in Ceph, introduces significant changes to the storage management in distributed file systems. By striping file data into objects and distributing them across many OSDs, the workloads seen by Ceph storage devices are dramatically different from those seen in traditional storage servers. The object identifier chosen by the metadata servers no longer reflects an object's physical location. As a result, OSDs have to manage a flat namespace of object identifiers internally.

Some object-based file systems employ general-purpose local file systems, such as Ext2 [7, 77], to manage their OSDs [8]. However, general purpose file systems have very different design assumptions. They optimize for a hierarchical namespace, which makes them quite inefficient in mapping/retrieving objects from a flat object namespace. Their allocation policies make incorrect assumptions about the incoming workloads, which cost them opportunities to further optimize the data layout and reduce potential fragmentation. As a result, we show that general purpose file systems are suboptimal to work as the storage management component on OSDs, and that specially designed object file system can provide greater throughput under expected object workloads.

1.3 Workloads

Understanding the workloads that a storage system is expected to handle is critical to developing a system capable of meeting its performance requirements. While there is a significant body of research on general purpose workloads, relatively little current research discusses the specific workloads that will be encountered in modern high performance computing platforms. In addition, due to the lack of large object-based storage systems, nothing had previously been published on the object workload that results from even well-understood file- and block-level workloads.

This dissertation includes a series of studies to analyze the file-level workloads gathered from several typical scientific applications, as demonstrated in Chapter 4. The results show that, on average, each application has only one or two typical request sizes. Requests tend to be deeply buffered at the client side. Large requests from several hundred kilobytes to several megabytes are very common. Although in some applications small requests account for the majority of all requests, almost all of the I/O data are transferred by large requests. All of these applications show bursty access patterns. Our research in this area has resulted in a better understanding of the expected scientific workloads and enabled both more accurate object workload generation and more informed system development.

1.3.1 Object Workloads in Ceph System

Based on the file system workload analysis, the major characteristics of expected object workloads under the Ceph framework are derived. Since the majority of storage space in scientific environments is consumed by large files, most on-disk objects are the same size as the system stripe unit and the small variable-sized objects will only account for a small fraction of objects in the system. Given a 512 KB stripe unit, more than 83% of all objects will be stripe-unit-sized objects. Furthermore, in scientific environments most of the object requests will read/write whole objects. Partial object accesses are not significant due to the deep client-side buffering and the long sequential file access patterns. Object requests to individual Ceph OSDs will exhibit little or no locality of reference between objects because of the random object placement policy, making useless most general purpose file systems' attempts at clustering.

In Ceph, multiple objects from a small or medium file will seldom be stored on the same OSD.From the OSD point of view, objects will tend to be accessed randomly: it will be rare to observe sequential object accesses on individual OSDs. As a result, the Ceph system requires that the underlying OSD file systems provide fast object name map/retrieval from a single flat and homogeneous object namespace. The object workloads are dominated by write requests. Although the foreground requests from clients have to be served synchronously, there are still a large amount of asynchronous write requests generated from background activity. OSDs must therefore handle both synchronous and asynchronous write requests.

1.4 OBFS

1.4.1 Design

Our Object-Based File System (OBFS) is a very light-weight, highly efficient file system designed specifically for use in OSDs in large-scale distributed object-based storage systems. OBFS aims to optimize storage management on individual OSDs under the Ceph framework. However, it can be employed in other object-based storage systems if they share the same design assumptions: fixed stripe unit and random object placement. As described in Chapter 5, OBFS optimizes disk layout and enhances flat name space management based on knowledge regarding expected object workloads. It uses two block sizes: small blocks, equivalent to the blocks in general-purpose file systems, and large blocks, equal to the Ceph system stripe unit size, to greatly improve the object throughput while still maintaining efficient disk utilization. By collecting blocks of the same size into *regions*, OBFS effectively alleviates file system fragmentation as the system ages, reduces seek overhead, and minimizes recovery time after a failure. Compared to Linux Ext2 file systems [7, 77], OBFS results in better data layout by guaranteeing that any maximum-size object is laid sequentially and any variable-size object is allocated to a single disk region. Object metadata are also laid contiguously with the associated data, further improving disk bandwidth utilization. In addition, hash-based algorithms are employed to map/retrieve objects to/from the persistent storage, efficiently managing the flat name space exported by the OSDs.

1.4.2 Implementation

Based on our design, we implemented OBFS in Linux. Implementation details about the object allocation, read, write, delete, and failure recovery are discussed in Chapter 6. Object allocation involves two steps: region allocation and block allocation. The region allocation policy aims to minimize region internal fragmentation and reduce seek distance by taking into account three factors: the distance between the last-accessed region and the target region, the region fragmentation, and the burstiness of workload. The block allocation uses an extent-based policy to minimize object fragmentation. A small extent list and an extent summary array per region are built on the fly to alleviate allocation overheads. The object read/write/delete implementations aim to reduce extra metadata updates associated with those object operations, while still providing strong reliability guarantees. A set of flags and specific update procedures are introduced to achieve this goal, which leads to an efficient but relatively complex failure recovery procedure. During file system checking, the recovery routine uses flags and region-unique generation numbers to identify active metadata and rebuild the region structures.

1.5 Experiments and Results

A series of experiments, discussed in Chapter 7, were conducted to evaluate the performance of OBFS on both freshly installed and aged file systems. Aged file systems simulate the state of a file system after long term usage, which often lowers file system performance due to fragmentation of file/object data. The results show that OBFS successfully limits the file system fragmentation after long term aging. The aging loads are evenly distributed to all allocated regions. The low overall number of extents for all objects indicates that most objects are laid on disk contiguously.

OBFS demonstrates very good synchronous write performance, exceeding those of Ext2 and XFS by up to 80% on both the fresh systems and the aged systems. Its asynchronous write performance is comparable to those of Ext2 and XFS on lightly used disks and about

20% better on heavily used disks. OBFS read performance nearly triples that of Ext2 and is only slightly lower than that of XFS. Since OBFS is optimized for stripe-sized objects, it shows extremely good performance for streaming workloads that purely contain such objects. The sustained bandwidth on a fresh disk is about 40 MB/s for both synchronous writes and asynchronous writes, which is apparently limited by the maximal bandwidth of the disk used in the test system. Compared with Ext2 and XFS on a fresh system, it almost doubles the throughput of synchronous writes and improves by 50% the throughput of the asynchronous writes. On an aged system, OBFS still leads by 15% to 50%. Putting all those factors into account, OBFS achieves 18% to 200% performance improvements over XFS and Ext2 respectively under expected object workloads.

1.6 Conclusion

OSDs are the basic storage units for large distributed object-based storage systems. Any bandwidth change of individual OSDs will be proportionally reflected in the overall system bandwidth. Thus, a small improvement of the OSD throughput can lead to a large enhancement of the overall throughput. However, the characteristics of the object workloads are dramatically shifted from what the traditional file systems experience and optimize for due to the novel object-based architecture.

OBFS is designed specifically for OSDs under the Ceph framework. It employs two different block sizes and uses regions to organize blocks of the same size together. Combined with its optimized metadata layout and allocation policy, it greatly reduces object fragmentation in the long run. It handles a flat object name space through a hash-based structure, which makes it very efficient in storing and retrieving a large amount of objects. Compared with Ext2 and XFS, it demonstrates very good performance in both read and write operations.

The overall contributions of this work are summarized as below:

- File system workloads in modern high-performance scientific environments have been re-examined and evaluated, which faciliates the design of large scale storage systems.
- Based on the file system workload analysis, an object workload has been studied (under a set of assumptions corresponding to the Ceph architecture). This work provides the characteristics of object workload in Ceph framework. It gives a good example of what the object workload may look like and is useful for systems sharing similar assumptions with Ceph.
- A novel design for file systems on object-based storage devices is presented.
- A completed version of OBFS has been implemented into Linux.
- The performance of our OBFS is studied and evaluated in details using various workloads and benchmarks with different characteristics.

The rest of the disseration is arranged as follows: Chapter 2 provides the background of this work. Chapter 3 discusses the related work on file system design and workload analysis. File system workload studies in scientific environments and object workload analysis of Ceph system are presented in Chapter 4. Chapter 5 describes the overall design of our OBFS and Chapter 6 provides the implementation details. OBFS evaluations and performance comparisons with Ext2 and XFS are presented in Chapter 7 and Chapter 8 respectively. Chapter 9 summaries the design of OBFS and concludes the contributions of this work.

Chapter 2

Background

This chapter provides some background and context for the research described in this dissertation. Although the primary focus of the dissertation is on file systems for object-based storage devices, it does not operate in isolation and its design is informed by the successes and failures of previous systems. Accordingly, we begin with a brief high-level discussion of traditional storage models, including directly-attached storage, server-based storage, and SAN-based storage. We then discuss the new object-based storage model, which addresses some of the limitations of server-based storage. Finally, we discuss the Ceph object-based storage system, in which the work described in this dissertation was done.

2.1 Directly-attached Storage

Directly-attached storage, as its name suggests, is directly connected to clients using SCSI, ATA, or FibreChannel protocols. It exports its storage space through a block interface.

A directly-attached storage device cannot be shared among multiple hosts. Some high-end arrays provide multi-channel interfaces that can be connected to two or more different hosts, but they are primarily used for high availability. Only one of connected servers actually uses the storage during normal operation; all other servers that connect to the same storage device are configured as fail-over servers and only access the device when the primary server is unavailable.

Directly-attached storage is simple and easy to use. It requires minimal configuration and is very robust for normal applications. The close coupling of the directly-attached storage and its host allows for good performance and simplifies the storage management tasks on individual hosts.

However, the strong affinity between storage devices and hosts imposes severe limits on global storage manageability, utilization, and scalability. The directly-attached storage model does not support resource sharing among multiple hosts. In an environment that requires multiple hosts to access the same data set, the hosts have to explicitly move data between different domains. Important storage management tasks, such as backup, have to be perfermed on a host-by-host basis, which significantly increases global management complexity.

Such a non-sharing architecture also greatly reduces global storage utilization. Free space in one host cannot be utilized by others. Balancing storage space between hosts involves physical movements of the storage devices and relatively complicated reconfiguration processes.

Finally, the storage resources managed by individual hosts are hard to scale. A directly-attached storage device consumes a certain amount of host resources, such as main

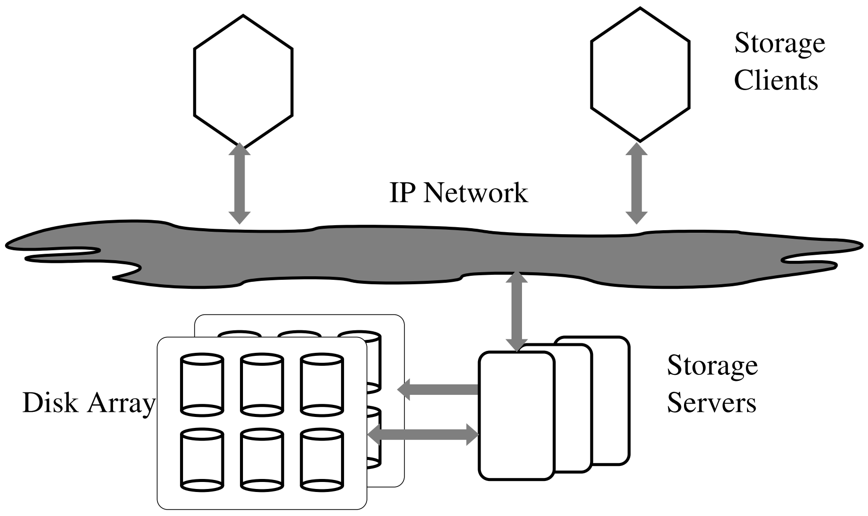


Figure 2.1: The server-based storage models

memory for in-core data structures and bitmaps and CPU cycles for doing I/Os, which are roughly proportional to the size of the device. As more and more devices added to a host, the aggregrate management overhead is significant enough that it may compete with the user applications on the host.

2.2 Server-based Storage

Server-based storage, as shown in Figure 2.1, introduces a dedicated file/storage server that is separated from application servers (storage clients). The storage server manages a set of storage devices and shares them with multiple hosts through the network. All of the storage clients on the same network with the server can access the storage using file-level protocols, such as NFS and CIFS.

The server-based storage model provides a sharing architecture that improves storage manageability and utilization. The server provides a central point for storage management instead of multiple isolated domains. All storage devices exported by the server are shared by the hosts, which balances the storage balance among hosts and effectively improves the overall utilization. The dedicated server can devote all of its resources to storage management, and is thereby able to handle more storage devices. As a result, the storage clients on the network can access much larger storage with relatively low overheads than the hosts using directly-attached storage.

Although the server-based storage model provides better scalability, the centralized server or server cluster is still a performance bottleneck as the requirements for large capacity and high bandwidth scale up. Since the storage devices are attached to the server either directly or through a storage area network, they are typically invisible to the clients. As the only authority in the system, the server or server cluster needs to handle both the high-level hierarchy requests, such as naming, access control, and authentication, as well as the low-level storage management requests, such as data allocation and request scheduling. The server cluster also takes the responsibility for forwarding data between clients and storages devices. A typical I/O process in a system using this model is initiated by a client sending a file request, such as a file open request, to a server in the server cluster. The server looks up the corresponding file metadata and responses the client with a file handle. A series of read/write requests are then sent to the server by the client. The server, on behalf of the client, allocates storage space on disk, updates the metadata, and performs real disk reads/writes.

As the system scales up, an individual server may not be able to satisfy the requirements for ever larger capacity and higher bandwidth. In that situation, a server cluster is used to manage a shared storage pool or a set of directly-attached storage devices and export a single file system image or multiple independent file system images to all clients. The maximal throughput of a system using this model is limited by the aggregrate link bandwidth and processing power of the server cluster and the aggregated bandwidth of the underlying storage devices. If the links and computing power of the server cluster are saturated, increasing the underlying storage bandwidth will not improve overall system throughput. As a result, scaling up the server cluster becomes the key issue that determinates the overall system throughput. However, the coherent requirement of the clustering protocol makes it extremely difficult to scale up the server cluster. Lock contention for both the namespace and the physical resource management severely constrains the size of the server cluster. Moreover, data forwarding imposes significant overhead on the server cluster, further reducing its scalability. As a result, storage systems using the server-based model can hardly satisfy the requirements for tens of petabyte storage and several terabytes per second aggregrate bandwidth, which is required in the next-generation storage systems, especially high-end systems in scientific computing environments.

2.3 SAN-based Storage

The SAN-based storage model further improves bandwidth scalability by enabling direct data transfer between clients and storage devices. In a SAN-based stoarge system, all storage devices, clients, and servers are connected through a storage area network, as shown in Figure 2.2. Since the storage devices are now visible to the clients, the server cluster is no

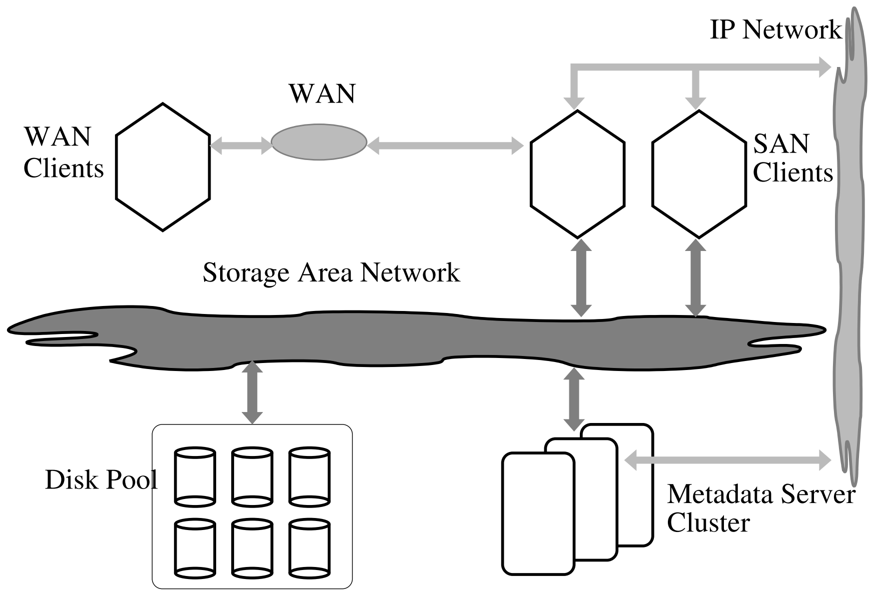


Figure 2.2: The SAN-based storage models

longer responsible for data forwarding between them. A client can initiate direct data transfer using special protocol, such as iSCSI, to any device in the SAN after acquiring appropriate permissions from the server cluster. To read/write a file, a client first issues a file open request to a server through either the SAN or a separate IP network. The server responds the client with a list of file block mappings. The client then uses this information to directly contact disks through the SAN and retrieve data as needed.

By removing the data forwarding step, the SAN-based storage system eliminates the bandwidth constraint imposed by the aggregrate link bandwidth of the server cluster. Overall system throughput is able to scale proportional to the aggregrate storage bandwidth up to the maximal capacity provided by the SAN. Server load is also greatly reduced, which makes it possible to handle more file requests from clients. In addition to the performance benefits, the

SAN-based storage further consolidates storage management in the distributed environment. More storage can be incorporated into a SAN-based system and managed centrally.

Although the SAN-based storage model provides better scalability and manageability, it has its own limitations. Since the clients can communicate directly with the disks, which have limited intelligence on board, security becomes a big issue in the SAN environment. The system built on the SAN-based storage model trusts every host in the same SAN. It relies on strict authentication for SAN clients to guarantee data safety. However, once a host joins the SAN, there is no effective way to prevent it from performing malicious operations. Any compromised host in the SAN may access and damage the whole system.

The processing power of the server cluster can still become a performance bottleneck in a large system. Despite eliminating the data forwarding functionality, the server cluster is still in charge of low-level storage management tasks, such as block allocation and layout optimization. Significant amounts of memory resource, computing power, and link bandwidth are spent in servicing those requests and managing related data structures. For example, a 10-PB file system contains 160-GB block bitmaps, assuming that the block size remains 8 KB. Even with a 256-KB block size, the block bitmaps still consume 5 GB. Fitting such big bitmaps in core imposes great challenges to the server cluster. A common practice is to delegate a portion of the bitmap to every server in the cluster. However, since the namespace partitions among the server cluster do not closely track the physical resource partitions, the mismatch of those two may result in extra data movement and resource locking. For instance, to free a file from one server, it may need to access a portion of the block bitmap managed by another server. This may involve a complicated distributed locking procedure or multiple RPC calls between

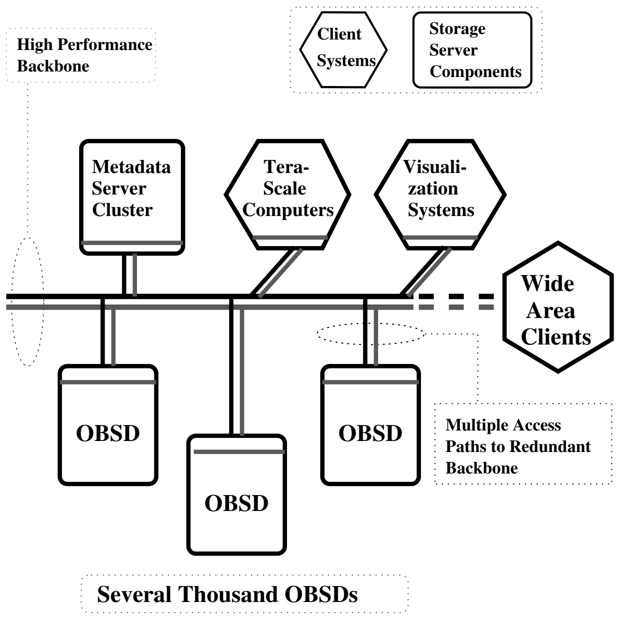


Figure 2.3: OSD distributed file system architecture

servers. As a result, it is hard to scale up the server cluster while still exporting a single file system image, which limits the ultimate scalability of the SAN-based storage systems.

2.4 Object-based Storage

Object-based storage, first proposed by Gibson *et al.* [18], extends the SAN-based storage architecture by offloading the physical storage management functionality to intelligent storage devices and enforcing security policies on them. The intelligent storage device, referred to as an object-based storage device, presents an object interface that encapsulates the physical storage details. The object name, unlike the block number, does not associate directly with any physical resources. It is the OSD's responsibility to map an object into the disk blocks.

2.4.1 Object Abstraction

The *object* is a storage abstraction that lies between the file and the logical disk block. It hides the physical details related to the low-level storage, while providing a better granularity for storage management. The object name, unlike the disk block number, has no direct relation to the object's physical location or its relationship with other objects. Compared with the file abstraction, objects provide a user-visible entity that can be used to facilitate data placement in a distributed environment and enforce security policy at a finer granularity.

The object concept also provides an ideal data layout and security unit that aligns to the storage device boundary in distributed environments. Objects can be used to pack multiple small files together or break down large files to optimize the disk bandwidth and network bandwidth utilization for the OSD cluster. The object access control is handled by individual OSDs, which helps the metadata server cluster to enforce the system security policy.

2.4.2 Architecture

Object-based storage provides an extremely scalable architecture and strong security guarantees. It eliminates the file server as a bottleneck by offloading storage management to the OSDs and enables load balancing and high performance by striping data of a single file across multiple OSDs. It also enables a high level of security by using cryptographically secured capabilities [18, 59] and local data security mechanisms [45].

A file system employing the object-based storage model is depicted in Figure 2.3. In this system, a metadata server cluster services all metadata requests, manages the hierarchical file namespace, handles authentication and protection, and provides clients with the file to object mapping. Clients may then contact the OSDs to retrieve the objects corresponding to the files they want to access. In such a system, metadata operations are completely decoupled from read and write operations. A client wishing to open a file will first contact the metadata servers to obtain a capability and information allowing it to locate the objects containing the file data in the OSD cluster. Subsequent read and write traffic then involves mostly the client and relevant OSDs, only rarely involving the metadata server further. Upon receiving a client request, an OSD will validate this request by comparing it with the companioned capability. Only if the request matches the access control information, encoded in the capability, will the OSD service it.

Object-based storage reduces the server load by offloading the physical storage management to OSDs and thus improves the system scalability. As shown in Figure 2.4, the server cluster is only responsible for meteadata related requests. This greatly reduces the locking contention and data movement caused by the physical resource management, which effectively increases the server cluster scalability. All block allocation and layout optimization functionalities are delegated to OSDs. The overall system processing power scales proportionally to the number of OSDs in the system. Thus, object-based storage allows maximum scalability for system throughput since the primary bottleneck limiting I/O transfer rates becomes the network, instead of any single component within the system.

Compared with SAN-based storage, object-based storage provides much better access controls and much stronger security guarantees. As we mentioned in Section 2.3, a file

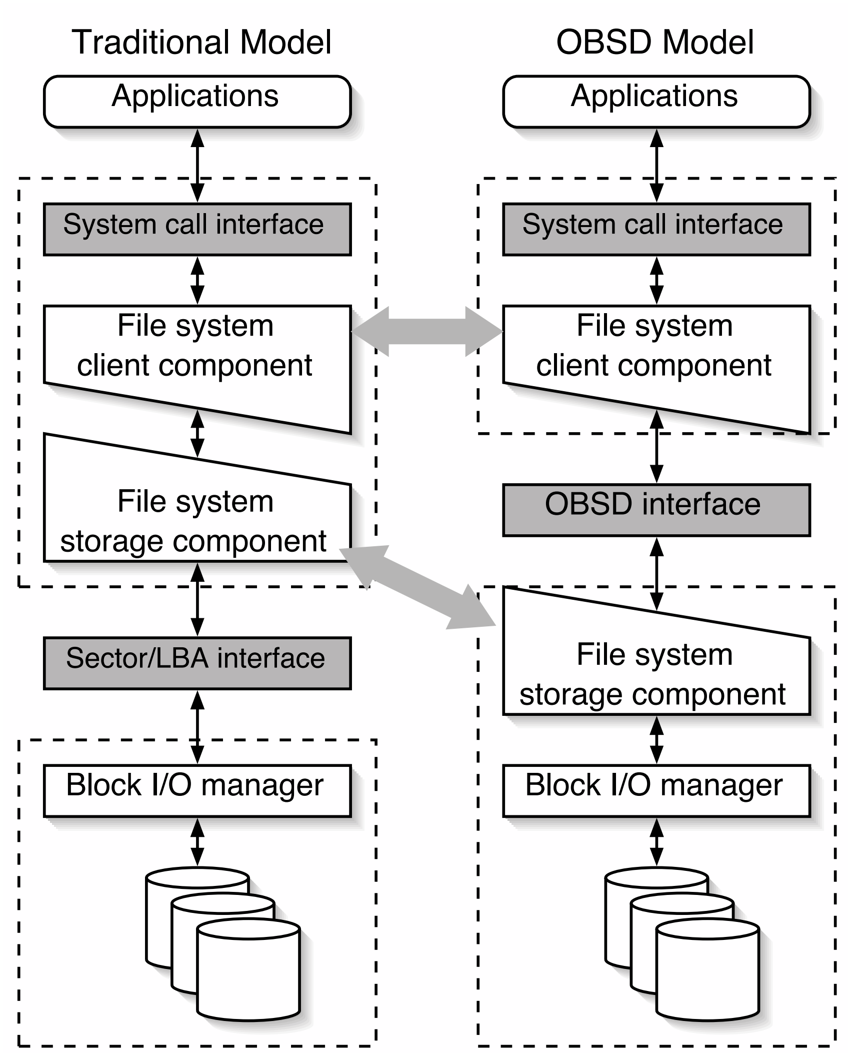


Figure 2.4: Comparison of traditional and OSD storage models (This figure was orginally appeared in the OSD proposal from the technical committee T10 [80].).

system using the SAN-based storage model cannot provide a finer granularity for the access control and the security check because all the hosts in the SAN can perform I/Os directly to the disks. The system either rejects a client access to a disk or allows it to access the entire disk. Object-based storage, on the other hand, benefits from the intelligent OSDs, which can help to enforce complex access-control and security policies. Unlike a dumb disk that allows a client to access all its blocks equally, an OSD requires the client to present a capability for every request it issued. Such a capability contains access-control and authentication information, which is used by the OSD to validate the client request. This capability is typically signed by the metadata server to prevent a malicious client from faking one. Such a security framework only trusts the server cluster and the OSDs without imposing any requirement on the clients. It enables object-based storage to be adopted in an untrustworthy environment.

2.4.3 Systems Using Object-based Storage Model

Distributed object-based storage systems, first used in Swift [10] and subsequently used in systems such as NASD [19], Storage Tank [27], Lustre [8], Slice [3], and Panasas [49] are built on this model.

Much research has gone into hierarchy management, scalability, and availability of distributed file systems such as AFS [46], Coda [29], GPFS [65], GFS[73], and Lustre [8], but relatively little research has been published that aims 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 [8]. 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.

2.5 The Ceph Object-based Storage System

Ceph is a large scale object-based storage system currently being developed in Storage System Research Center at UC, Santa Cruz. The Ceph architecture contains four key components: a small cluster of metadata servers (MDSs) that manages the overall file system name space, a large collection of OSDs that store data and metadata, a client interface, and a high-speed communications network. OBFS is designed as a sub-component to run on the individual OSDs in the *Ceph* system.

Ceph's metadata server cluster is based on a dynamic metadata management design that allows it to dynamically and adaptively distribute cached metadata hierarchically across a set of MDS nodes. Arbitrary and variably-sized subtrees of the directory hierarchy can be reassigned and migrated between MDS nodes to keep the workload evenly distributed across the cluster. This distribution is entirely adaptive and based on the current workload characteristics. A load balancer monitors the popularity of metadata within the directory hierarchy and periodically shifts subtrees between nodes as needed. The resulting subtree-based partition is kept coarse to minimize prefix replication overhead and preserve locality. Ceph's OSD cluster is used for storing both data and metadata. Intelligent OSDs allow data replication, failure detection, and recovery activities take place semi-autonomously under the supervision of the MDS cluster. This intelligence in the storage layer allows the OSD cluster to collectively provide a reliable, scalable, and high-performance object storage service to client and MDS nodes.

In Ceph, the contents of each file are striped over a sequence of objects stored on OSDs. To ensure that tens of thousands of clients can access pieces of a single file spread across thousands of object-based disks, Ceph must use a distribution mechanism free of central bottlenecks. This mechanism must accommodate replication, allow for the storage system to be easily expanded, and preserve load balance in the face of added capacity or drive failures. We chose the RUSH_R variant of the RUSH [25] algorithm to distribute data because it meets all of these goals.

The Ceph client combines a local metadata cache, a buffer cache, and a POSIX call interface. The client communicates with the MDS cluster to open and close files and manipulate the file name space, while maintaining a local metadata cache both for efficiency and to "learn" the current partition of metadata across the MDS cluster. Metadata consistency is currently similar to that of NFS: inode information in the cache remains valid for a fixed period before subsequent `stat` operations require contacting the MDS cluster. The client reads from and writes to files by communicating with OSDs directly: once a file has been opened, the inode number and byte offset specify (via RUSH) an object identifier and OSD to interact with. A buffer cache serves to filter out redundant file I/O while maximizing the size of file requests submitted to OSDs.

Chapter 3

Related Work

There has been a tremendous amount of research in file and storage systems. This chapter surveys some of the more closely related research and places the research described in this dissertation in context by comparing and contrasting it to the work of others.

3.1 File Systems Allocation and Layout Policies

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. In this section, we discuss research in allocating disk space and handling metadata.

The Berkeley Fast File System (FFS) [40] and related file systems such as Ext2 and Ext3 [77] are widely used today. They all try to store file data contiguously in *cylinder groups*—regions with thousands of contiguous disk blocks. This strategy can lead to fragmentation, preventing data for larger files from being stored contiguously, so techniques such as extents and clustering [41, 71] are used to group blocks together to decrease seek time. Analysis has shown that clustering can improve performance by a factor of two or three [68, 71], but clustered file allocation becomes less successful in aged file systems because it is difficult to find contiguous blocks for clustered allocation.

To improve the clustering technique, McKusick *et al.* [38] provide a new allocation algorithm, which adds a reallocation step to the original FFS disk allocation algorithm. The reallocation step tries to re-examine and optimize the block allocation before those blocks are written to disk. This delayed allocation scheme can better exploit the existing clusters of free space.

Similar to the approach of the FFS, McVoy *et al.* discuss the file system clustering technique to obtain the extent-like performance for the SUN UFS [41]. They attempt to place logically sequential file data on physically contiguous disk blocks and transfer data in multiple-block chunks. The performance of such file systems is good when the file systems are fresh and nearly empty. However, as file system ages, data fragmentation will compromise the benefits achieved by data clustering.

WAFL [24], a file system layout specially designed for file systems to work in an NFS appliance, is optimized for writes. Rather than pre-allocating fixed disk space for the metadata, WAFL uses a file to organize all of the metadata, allowing it write both data and metadata blocks at any free location. This approach introduces an indirect metadata mapping through the file interface, which enables the flexible metadata writes. However, WAFL aims at the NFS environment, where writes dominates. Like LFS discussed below, its read performance is poor when files are randomly written in and sequentially read out.

Colocating FFS (C-FFS) [16] explicitly groups small files into contiguous disk location and, like WAFL, uses directory files to organize the metadata. In the file allocation process, C-FFS tries to incorporate the new file blocks into an existing group associated with the directory that names the file. Compared with the FFS allocation policy, C-FFS improves performance by 40 to 60%, when the data access pattern matches the namespace locality.

Log-structured file systems [61, 67] group data by optimizing the file system for writes rather than reads, writing data and metadata to segments of the disk as they arrive. 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 [6].

XFS [47, 74] is a highly optimized file system that uses extents and B-trees to provide high performance. Similar to the reallocation technique presented in the FFS of the 4.4 BSD Operating System, XFS employs a delayed allocation technique. Buffered data blocks are not assigned disk space until those blocks need to be flushed to disk. The delayed allocation technique can effectively reduce the small extents which can result from small incremental data writes.

3.2 File System Consistency and Metadata Management

Existing file systems must do more than allocate data; they must also manage large amounts of metadata and directory information. Metadata management introduces significant overhead to file system performance due to the extra disk accesses for metadata and maintaining file system consistency. In this section, we discuss related work in managing metadata and maintaining file system consistency.

FFS provides strong consistency by synchronously writing each block of metadata [40]. The metadata are flushed back to disk in fixed order as they are updated, which guarantees that the file system can be recovered to a consistent state after a system crash. However, the synchronous metadata writes can significantly impair file system performance.

Soft Updates [39] attacks this problem by removing metadata updates from the critical path while still guaranteeing the metadata are written to disk in predefined order. Soft Updates tracks the dependencies of metadata updates and enforces the dependencies as those metadata are flushed back to disk.

Write-ahead logging [21] is a technique that first emerged in database research before becoming widely used in file systems [28, 78, 77, 20]. Metadata operations are written to an auxiliary log and committed to disk before any blocks modified by corresponding metadata operations reach disk. After a system crash, file systems can be recovered to a consistent state by replaying the valid operations in the write-ahead log. If the log is updated synchronously, the file systems provide consistency as strong as that of FFS. If the log is maintained asynchronously, the file systems are more like those using Soft Updates. Another approach in configuring write-ahead log is called metadata group commit. It batches multiple log writes and commits them to disk together synchronously [21], which is quite useful in highly concurrent systems.

Log-structured file systems [61, 67] provide a different approach for metadata management. Rather than updating metadata in a fixed position, LFS writes the modified data into a segment log. The order of the metadata operations in the segment log reflects the actual metadata update order, which guarantees file system recoverability.

Besides those software techniques, the metadata update problem can be addressed by hardware techniques, such as non-volatile RAM (NVRAM). Systems designed with the aid of NVRAM, such as Rio [12] and WAFL [24], exhibit better performance than those that use Soft Updates and write-ahead logging.

Most systems do not store data contiguously with metadata, decreasing performance because of the need for multiple writes. Log-structured file systems [61, 67] and embedded inodes [16] 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 [78] and Soft Updates [39] can reduce the penalty associated with metadata writes, but cannot eliminate it.

3.3 Distributed File and Storage Systems

The Network File System (NFS) [63] was first introduced by SUN Microsystems for use in UNIX operating system environments. NFS is a network protocol. It takes advantage of the Virtual File System (VFS) and supports various local file systems through the VFS interface. The storage management is done by local file systems.

The Sprite File System [53], developed by University of California at Berkeley, provides a single-system image, in which files are equally accessible by any workstation in the network. Sprite provides stronger consistency semantics than NFS. A log-based file system

(Sprite LFS) [61] serves as its local file system.

The Andrew File System (AFS) [46] and its successor the DEcorum File System (DFS) [28] provide a highly scalable architecture. Andrew provides a homogeneous, location-transparent file name space to all the client workstations by using a set of trusted servers. The operating system running on each workstation intercepts file system calls and forwards them to a user-level process called *Venus*, which manages the local data cache and communicates with the file servers.

The Coda File System [64] is a descendant of AFS which was designed to be more resilient to failures by using replication and disconnected operation. It relaxes the consistency requirements to allow efficient disconnected operation. It optimistically assumes that clients work on their own separate home directories, in which very little sharing will happen.

Many scalable storage systems such as GPFS [65], GFS [57], Petal [36], Swift [10], RAMA [44], Slice [3] and Zebra [22] 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 [3]. 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 could take advantage of our file system running on an OSD. In GPFS, allocation is done in large blocks, resulting in fewer disk seeks overall but very low storage utilization for small files.

High-performance, scalable file systems have long been a goal of the high-performance computing (HPC) community. HPC systems place a heavy load on the file system [52, 69, 79], placing a high demand on the file system to prevent it from becoming a bottleneck. As a result, there have been many scalable file systems that attempt to meet this need; however, these file systems do not support the same level of scalability that Ceph does. Some large-scale file systems, such as OceanStore [32] and Farsite [2] are designed to provide petabytes of highly reliable storage, and may be able to provide simultaneous access to thousands of separate files to thousands of clients. However, these file systems are not optimized to provide high-performance access to a small set of files by tens of thousands of cooperating clients. Bottlenecks in subsystems such as name lookup prevent these systems from meeting the needs of an HPC system. Similarly, grid-based file systems such as LegionFS [82] are designed to coordinate wide-area access and are not optimized for high performance in the local file system.

Parallel file and storage systems such as Vesta [13], Galley [51], RAMA [44], PVFS and PVFS2 [11, 33], the Global File System [73] and Swift [10] have extensive support for striping data across multiple disks to achieve very high data transfer rates, but do not have strong support for scalable metadata access. For example, Vesta permits applications to lay their data out on disk, and allows independent access to file data on each disk without reference to shared metadata. However, Vesta, like many other parallel file systems, does not provide scalable support for metadata lookup. As a result, these file systems typically provide poor performance on workloads that access many small files as well as workloads that require many metadata operations. They also typically suffer from block allocation issues: blocks are either allocated centrally or, in the Global File System, via a lock-based mechanism. As a result, these file systems do not scale well to write requests from thousands of clients to thousands of disks. Similarly, the Google File System [17] is optimized for very large files and a workload consisting largely of reads and file appends, and is not well-suited for a more general HPC workload because it does not support high-concurrency general purpose access to the file system.

Recently, many file systems and platforms, including Federated Array of Bricks (FAB) [62], IceCube [26], Lustre [8, 66], GPFS [65], the Panasas file system [81], pNFS [23], Sorrento [76], and zFS [60] have been designed around network-attached storage [18, 19] or the closely related object-based storage paradigm [4]. All of these file systems can stripe data across network-attached devices to achieve very high performance, but they do not have the combination of scalable metadata performance, expandable storage, fault tolerance, and POSIX compatibility that Ceph provides. pNFS [23] and the Panasas object-based file system [81] stripe data across network-attached disks to deliver very high data transfer rates, but they both suffer from a bottleneck in metadata lookups. Lustre [8, 66] has similar functionality: it supports nearly arbitrary striping of data across object storage targets, but it hashes path names to metadata servers. This approach distributes the metadata load, but destroys locality and makes POSIX compatibility difficult, despite approaches such as LH3 [9]. GPFS [65] also suffers from metadata scaling difficulties; while block allocation is largely lock-free, as it is in most object-based storage systems, metadata is not evenly distributed, causing congestion in metadata lookups. Moreover, none of these systems permits a client to locate a particular block of a file without consulting a centralized table. Sorrento [76] alleviates this problem somewhat and evenly distributes data and metadata among all of the servers, but only performs well in environments with low levels of write sharing in which processors work on disjoint data sets.

FAB [62] focuses on continuously providing highly reliable storage; while its performance is acceptable, FAB provides very high reliability at the cost of somewhat reduced performance. Ceph takes the opposite approach: provide very high performance and reasonable reliability.

3.4 Distributed Cache and Cooperative Caching

Caching is well accepted as a viable method for alleviating the performance mismatch between main memory and disk. However, single point caching has limited scalability. Cooperative caching, where cache servers support each other in serving requests for cached objects, has emerged as an approach to overcome this limitation.

Muntz *et al.* studied the multilevel cache architecture on distributed file systems in the LAN environment [48]. They found “disappointingly low” hit rates in the intermediary servers. The reason for the low hit rates is that caches in different servers are managed independently. The same cache management policy is employed in those caches, which results in “inclusive” cache behavior. Those cached objects in server caches may also exist at the client caches. If any request cannot be satisfied at client caches, it cannot hit in server caches.

Dalin *et al.* presents several cooperative caching schemes to improve distributed file system performance [14]. In this study, they try to coordinate all the file caches in clients and servers to form a large efficient file cache. Through trace-driven simulation, they found as much as half of raw disk accesses can be reduced compared with the non-cooperative schemes and the read response time is improved by 73%.

3.5 Scientific File System Workload Study

The I/O subsystem has been a system performance bottleneck for a long time. In parallel scientific computing environments, the high I/O demands make the I/O bottleneck problem even more severe. Kotz and Jain [30] surveyed impacts of I/O bottlenecks in major areas of parallel and distributed systems and pointed out that I/O subsystem performance should be considered at all levels of system design.

Previous research showed that the I/O behavior of scientific applications is regular and predictable [43, 54]. Users have also made attempts to adjust access patterns to improve performance of parallel file systems [69].

There are several studies on file system workload characterizations in scientific environments [5, 31, 43, 50, 58]. They have shown that file access patterns share common properties such as large file sizes, sequential accesses, bursty program accesses, and strong file sharing among processes within a job. A more recent study showed that applications use a combination of both sequential and interleaved access patterns and all I/O requests are channeled through a single node when applications require concurrent accesses [70]; we observe similar phenomena in one of the applications under our examinations.

Pasquale and Polyzos found that the data transfer rates ranges from 4.66 to 131 megabytes/sec in fifty long-running large-scale scientific applications [54]. They also demonstrated that the the I/O request burstiness is periodic and regular [55].

Baylor and Wu showed that the I/O request rate is on the order of hundreds of requests per second [5]; this is similar to our results. They also found that a large majority of requests are on the order of kilobytes and a few requests are on the order of megabytes; our results differ in this regard.

Previous research has mainly investigated scientific workloads in the 1990's, although technology has evolved very quickly since then. In our study, we observe changes in large-scale scientific workloads, and provide guidelines for future file system designs based on a thorough understanding of current requirements of large-scale scientific computing.

Chapter 4

Workload Characterization

Parallel scientific applications require high-performance I/O support from underlying file systems. A comprehensive understanding of the expected workload is therefore essential for the design of high-performance parallel file systems. We re-examine the workload characteristics in parallel computing environments in the light of recent technology advances and new applications. We analyze application traces from a cluster with hundreds of nodes. On average, each application has only one or two typical request sizes. Large requests from several hundred kilobytes to several megabytes are very common. Although in some applications small requests account for more than 90% of all requests, almost all of the I/O data are transferred by large requests. All of these applications show bursty access patterns. More than 65% of write requests have inter-arrival times within one millisecond in most applications. By running the same benchmark on different file models, we also find that the write throughput of using an individual output file for each node exceeds that of using a shared file for all nodes by a factor of 5. This indicates that current file systems are not well optimized for file sharing.

4.1 File System Workload Study

Parallel scientific applications impose great challenges on not only the computational speeds but also the data-transfer bandwidths and capacities of I/O subsystems. The U.S. Department of Energy Accelerated Strategic Computing Initiative (ASCI) projected computers with 100 TeraFLOPS, I/O rates of 50–200 gigabytes/second, and storage system capacities of 0.5–20 PB in 2008. The projected computing and storage requirements are estimated to 400 TeraFLOPS, 80–500 gigabytes/second, and 3–20 PB in 2008 [15]. The observed widening disparity in the performance of I/O devices, processors, and communication links results in a growing imbalance between computational performance and the I/O subsystem performance. To reduce or even eliminate this growing I/O performance bottleneck, the design of high-performance parallel file systems needs to be improved to meet the I/O requirements of parallel scientific applications.

The success of file system designs comes from a comprehensive understanding of I/O workloads generated by targeted applications. In the early and middle 1990s, significant research effort was focused on characterizing parallel I/O workload patterns and providing insights into parallel system designs [5, 31, 43, 70]. The following decade has witnessed significant improvements in computer hardware, including processors, memory, communication links, and I/O devices. At the same time, systems are scaling up to match the increasing demands of computing capability and storage capacity. This advance in technologies also enables new scientific applications. Together these changes motivate us to re-examine the characteristics of parallel I/O workloads a decade later.

We traced the system I/O activities under three typical parallel scientific applications: the benchmark *ior2* [35], a physics simulation, *f1*, running on 343 nodes, and another physics simulation, *m1*, running on 1620 nodes. The detailed descriptions of those applications can be found in Section 4.1.1.2. We studied both static file system and dynamic I/O workload characteristics. We used the results to address the following questions:

- What were the file sizes? How old were they?
- How many files were opened, read, and written? What were their sizes?
- How frequent were typical file system operations?
- How often did nodes send I/O requests? What were the request sizes?
- What forms of locality were there? How might caching be useful?
- Did nodes share data often? What were the file sharing patterns?
- How well did nodes utilize the I/O bandwidth?

4.1.1 Tracing Methodology

All the trace data in this study was collected from a large Linux cluster with more than 800 dual-processor nodes at the Lawrence Livermore National Laboratory (LLNL) [1]. A development version of Lustre Lite [66] is employed as the parallel file system and the Linux kernel in use is a variant of 2.4.18.

[1]Tyce T. McLarty helped us collecting the traces.

4.1.1.1 Data Collection

Tracing I/O activity in large scale distributed file systems is challenging. One of the most critical issues is minimizing the disturbance of tracing on the measured system behaviors. A commonly-used method is to develop a trace module that intercepts specific I/O system calls—a dedicated node in the cluster collects all trace data and stores them to local disks. However, due to time limits, we chose a simpler approach: we employed the *strace* utility with parameters tuned for tracing file-related system calls. The trace data were written to local files. We relied on the local host file systems to buffer trace data.

Although the strace utility greatly simplifies the tedious data collection process, this approach has two shortcomings: first, strace intercepts all I/O-related activities, including parallel file system, local file system, and standard input/output activities. This results in relatively large data footprint. Second, the strace utility relies on the local file system to buffer traced data. This buffer scheme works poorly when the host file system is under heavy I/O workloads. In such a scenario, the host system performance might be affected by the frequent I/Os of the traced data.

The shortcomings were not significant in our trace collection because of the large I/O requests and the relatively short tracing periods. As discussed in Section 4.1.2, I/O requests in such a large system are usually around several hundred kilobytes to several megabytes. Even in the most bursty I/O period, the total number of I/Os per node is still around tens of requests per second. Up to one hundred trace records will be generated on each node per second on average. Buffering and storing these data has only a slight impact on system performance. Moreover, instead of tracing the whole cluster, we only study several typical scientific applications. These applications are usually composed of two stages: a computation phase and an I/O phase. The typical I/O stage ranges from several minutes to several hours. During this period, each node usually generates several hundred kilobytes of trace data, which can be easily buffered in memory.

4.1.1.2 Applications and Traces

All of the trace data were collected from the ASCI Linux Cluster in Lawrence Livermore National Laboratory (LLNL). This machine is currently in limited-access mode for science runs and file system testing. It has 960 dual-processor nodes connected through a Quadrics Switch. Two of the nodes are dedicated metadata servers and another 32 nodes are used as gateways for accessing a global parallel file system. The detailed configuration of this machine is provided in Table 4.1 [34]. We traced three typical parallel scientific applications during July, 2003. The total size of the traces is more than 800 megabytes.

The first application is a parallel file system benchmark, *ior2* [35], developed by LLNL. It is used for benchmarking parallel file systems using POSIX, MPIIO, or HDF5 interfaces. *Ior2* writes a large amount of data to one or more files and then reads them back to verify the correctness of the data. The data set is large enough to minimize the operating system caching effect. Based on different common file usages, we collected three different benchmark traces, named *ior2-fileproc*, *ior2-shared*, and *ior2-stride*, respectively. All of them ran on a 512-node cluster. *ior2-fileproc* assigns an individual output file for each node, while *ior2-shared* and *ior2-stride* use a shared file for all the nodes. The difference between the last

Table 4.1: The ASCI Linux Cluster Parameters

Total Nodes (IBM x355)	960
Compute Nodes	924
Login Nodes	2
Gateway Nodes	32
Metadata Server Nodes	2
Processor per Nodes (Pentium 4 Prestonia)	2
Total Number of Processors	1920
Processor Speed (GHz)	2.4
Theoretical Peak System Performance (TFlops)	9.2
Memory per Node (GB)	4
Total Memory (TB)	3.8
Total Local Disk Space (TB)	115
Nodes Interconnection	**Quadrics Switch**

two traces is that *ior2-shared* allocates a contiguous region in the shared file for each node, while *ior2-stride* strides the blocks from different nodes into the shared file.

The details about the other two applications are limited since they are both secret applications inside LLNL. Although we are assured that they are representative of the types of applications typically run at LLNL, we only have the minimal information about those applications. The second application, *f1*, is a physics simulation run on 343 processes, in which a single node gathers a large amount of data in small pieces from the others nodes. A small set of nodes then write these data to a shared file. Reads are executed from a single file independently by each node. This application has two I/O-intensive phases: the restart phase, in which read is dominant; and the result-dump phase, in which write is dominant. The corresponding traces are named *f1-restart* and *f1-write*, respectively.

The last application, *m1*, is another physics simulation which runs on 1620 nodes. This application uses an individual output file for each node. Like the previous application, it also has a restart phase and a result-dump phase. The corresponding traces are referred as *m1-restart* and *m1-write*, respectively.

4.1.1.3 Analysis

The raw trace files required some processing before they could be easily analyzed. Unrelated system calls and signals were filtered out. Since each node maintained its own trace records, the raw trace for each application is composed of hundreds of individual files. We merged those individual files in chronological order. Thanks to the Quadrics switch, which has a common clock, the traced time in those individual trace files was globally synchronized. Our analysis work, such as request inter-arrival time, was greatly simplified by sorting all requests into a chronologically sorted trace file.

A good understanding of file metadata operation characteristics is important. However, our traces are not large enough to capture general metadata access patterns. Therefore, the following section focuses more on file data I/O characterization.

4.1.2 File System Workload Characteristics

We present the characteristics of the workloads, including file distributions and I/O request properties. We study the distributions of file size and lifetimes and show the uniqueness of large-scale scientific workloads. We focus on three typical applications as described in Section 4.1.1.2 and examine the characteristics of I/O requests, such as the size and number of read and write requests and the burst and the distribution of I/O requests on various nodes.

Table 4.2: File Numbers and Capacity of the 32 File Servers

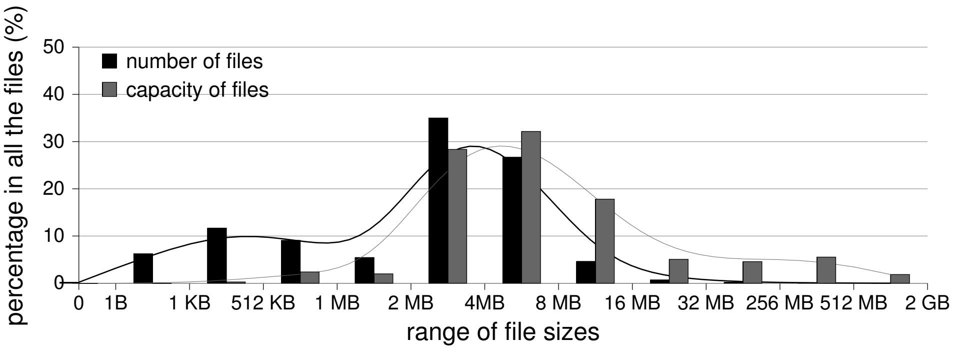
	Num. of files	**Total file size**
mean	305,200	1044.33 GB
standard deviation	75,760	139.66 GB
median	305,680	1072.88 GB
minimum	67,276	557.39 GB
maximum	605,230	1207.37 GB

4.1.2.1 File Distributions

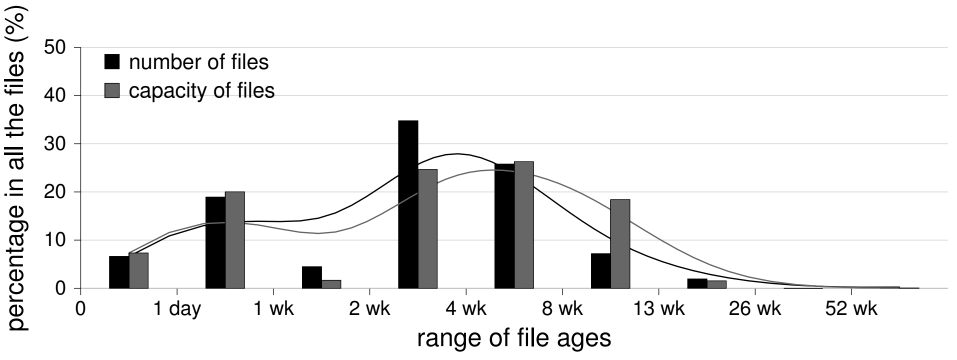
We collected file distributions from thirty-two file servers that were in use for the ASCI Linux cluster during the science runs phase. Each file server has storage capacity of 1.4 terabytes. The file servers were dedicated to a small number of large-scale scientific applications, which provides a good model of data storage patterns. On average, the number of files on each file server was 350,250, and each server stored 1.04 terabytes of data, more than 70% of their capacity. On most of the file servers, the number and capacity of files are similar except for five file servers. Table 4.2 displays statistic values of the number and capacity of files on these servers, including mean, standard deviation, median, minimum and maximum.

Figure 4.1(a) presents file size distributions by number and file capacity. The ranges of file sizes are sampled from 0–1 Byte to 1–2 Gigabytes. Some of the partitions were merged due to space limitations. We observed that over 80% of the files are between 512 kilobytes and 16 megabytes in size and these files accounted for over 80% of the total capacity. Among various file size ranges, the most noticeable one is from 2 megabytes to 8 megabytes: about 61.7% of all files and 60.5% of all bytes are in this range.

We also measure the file ages on those servers using their creation time. We divided



(a) By File Sizes



(b) By File Ages

Figure 4.1: Distribution of Files

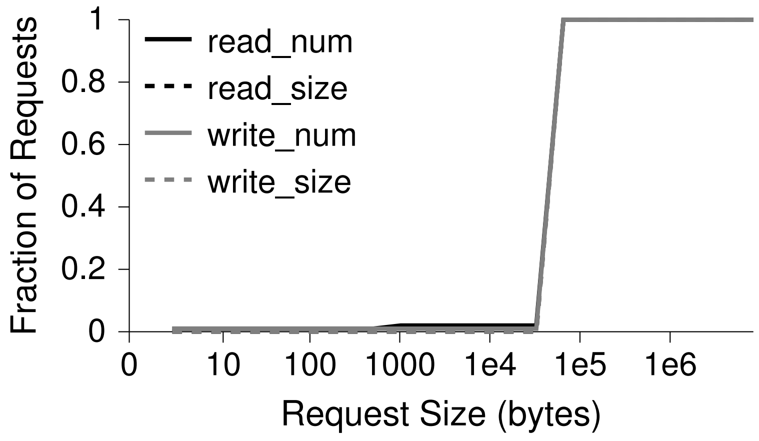


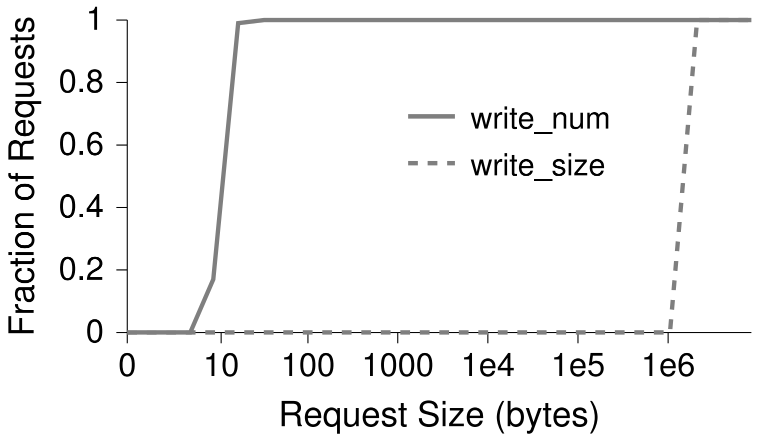
Figure 4.2: Cumulative Distribution Functions (CDF) of the Size and the Number of I/O Requests (X axis-logscale) in *ior2-fileproc*. The *read_num* and *write_num* curves indicate the fraction of all requests that is smaller than the size given in X axis. The *read_size* and *write_size* curves indicate the fraction of all transferred data that live in requests with size smaller than the value given in the X axis.

file ages into 9 categories: from 0–1 day to 52 weeks and older. As illustrated in Figure 4.1(b), 60% of the files and 50% of the bytes lived from 2 weeks to 8 weeks, while 6.6% of the files and 7.3% of the bytes lived less than one day. The lifetime of the traced system is about 1 year, so no files lived longer than 52 weeks.

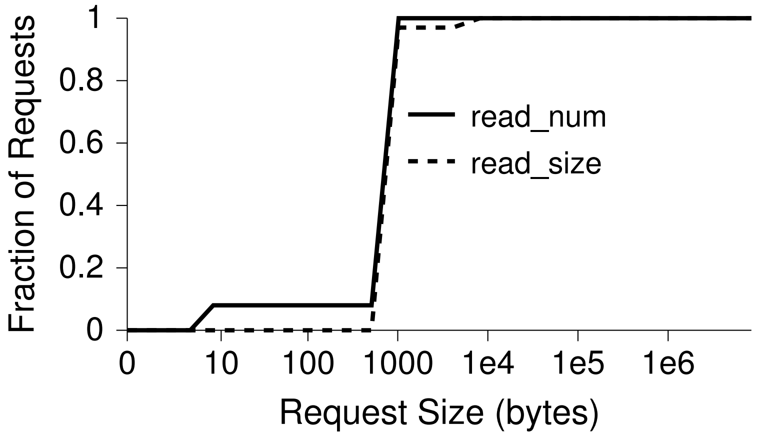
4.1.2.2 I/O Request Sizes

Figures 4.2– 4.4 show the cumulative distribution functions of request sizes and request numbers. Since all three *ior2* benchmarks have identical request size distributions, we only show one of them. As shown in Figure 4.2, *ior2* has only an unique request size of around 64 kilobytes.

Figure 4.3(a) shows the write request size distribution of the result-dump stage in

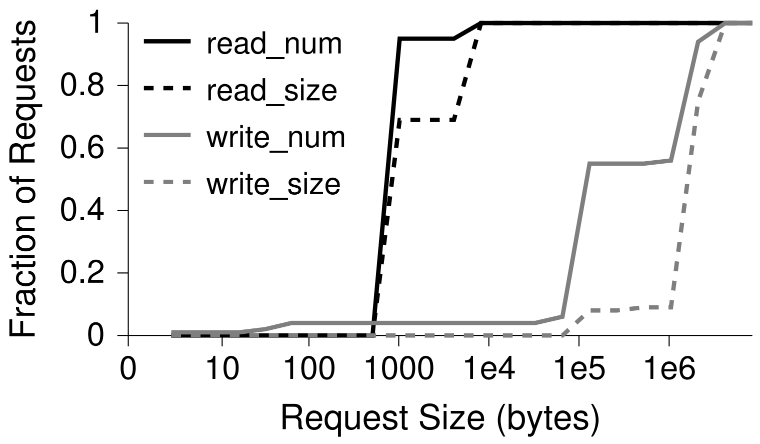


(a) f1-write

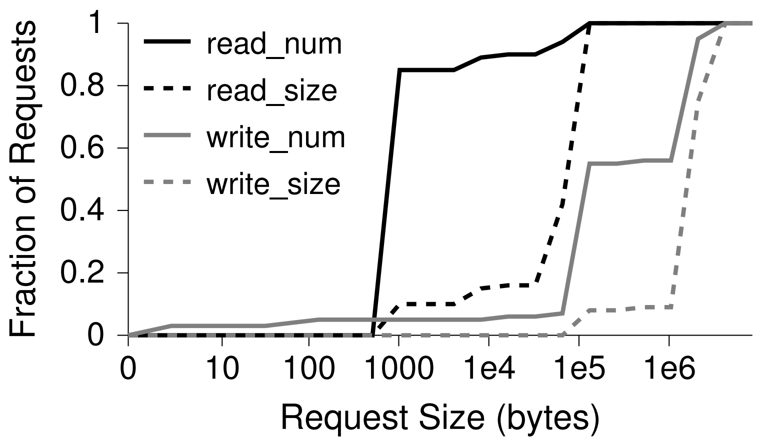


(b) f1-restart

Figure 4.3: Cumulative Distribution Functions (CDF) of the Size and the Number of I/O Requests (X axis-logscale) in *f1*.



(a) m1-write



(b) m1-restart

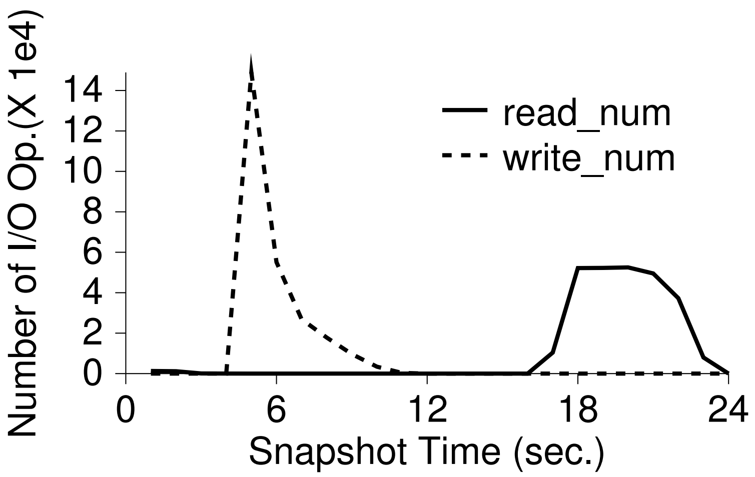
Figure 4.4: Cumulative Distribution Functions (CDF) of the Size and the Number of I/O Requests (X axis-logscale) in *m1*.

the first physics simulation, *f1*. Almost all the write requests are smaller than 16 bytes, while almost all the I/O data are transferred in the requests with sizes larger than one megabyte. This turns out to be a common I/O pattern of scientific applications: a master node collects small pieces of data from all computing nodes and writes them to data files, which results in a huge number of small writes. Other nodes then read and write these data files in very large chunks. There are less than five percent read requests in the result-dump stage so that we ignore the read request curves in figure 4.3(a). Similarly, we ignore the write request curves in figure 4.3(b) because very few write requests can be observed in the restart stage.

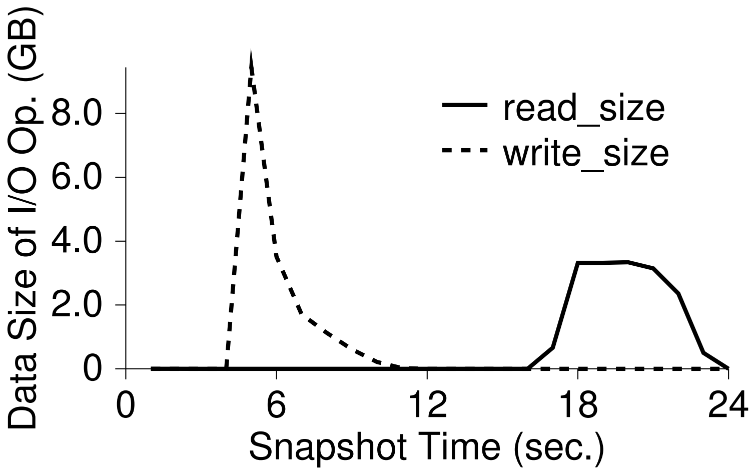
Figure 4.4(a) and Figure 4.4(b) show the same write request distribution in the restart and result-dump stages of the second physics simulation, *m1*. The two spikes in the *write_num* curves indicate two major write sizes: 64 kilobytes and 1.75 megabytes, respectively. Each of them accounts for 50% of all write requests. More than 95% of the data are transfered by large requests, which is also shown in Figures 4.4(a) and 4.4(b). Reads in *m1* are dominated by small requests less than 1 kilobytes. However, a small fraction (less than 3%) of 8 kilobyte requests accounts for 30% of all read data transfer. This is similar to the read distribution in Figure 4.4(b): 5% of the read requests contribute to 90% of all data read.

4.1.2.3 I/O Accesses Characteristics

Figures 4.5–4.10 show I/O accesses characteristics over time. The resolution for these figures is 1 second except Figure 4.8(a), which uses a resolution of 50 seconds. Figures 4.5–4.7 show that the request number distribution and the request size distribution are almost identical in *ior2* due to the fixed size requests used in those benchmarks. The *ior2*-

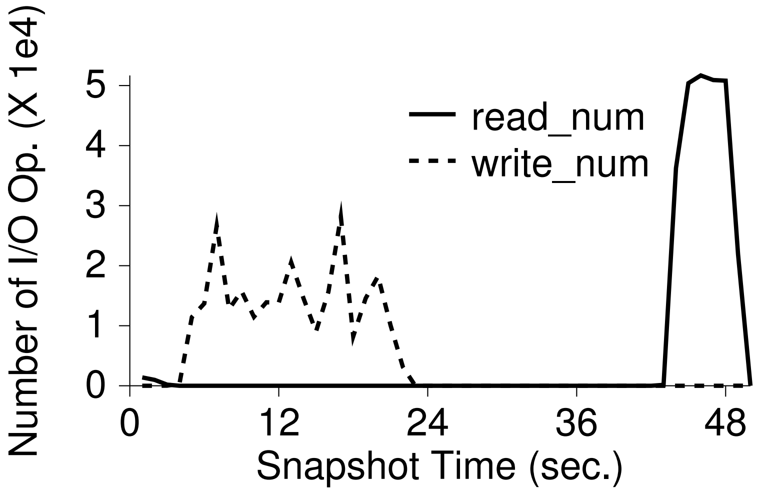


(a) ior2-fileproc number

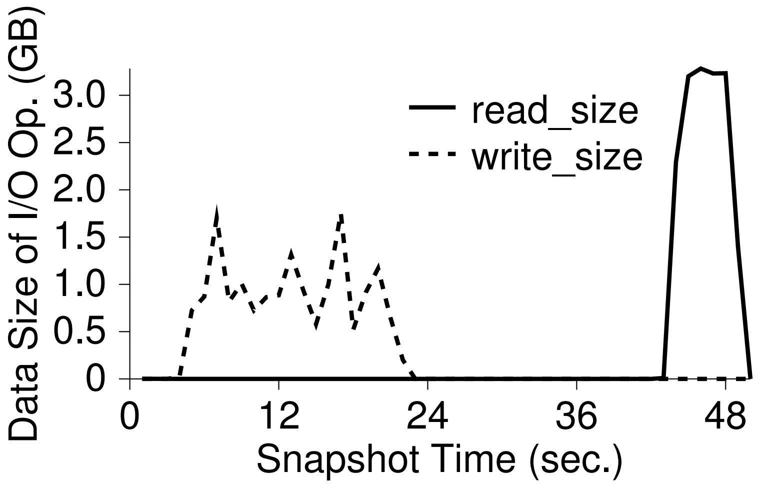


(b) ior2-fileproc size

Figure 4.5: I/O Requests over Time for *ior2-fileproc* Benchmarks

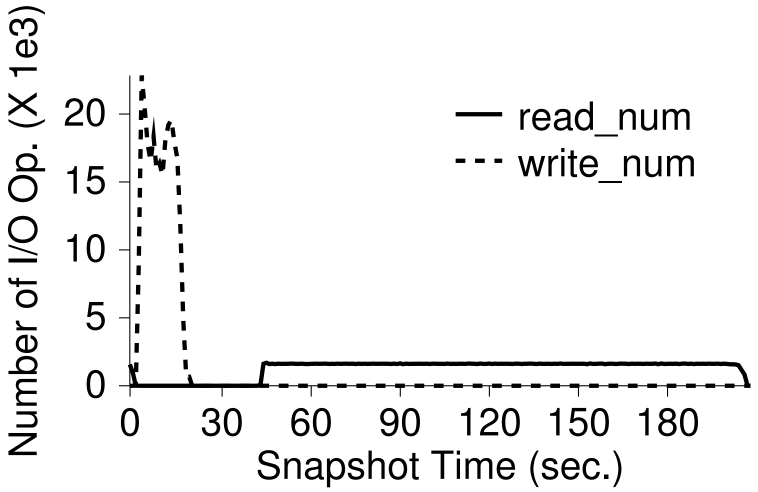


(a) ior2-shared number

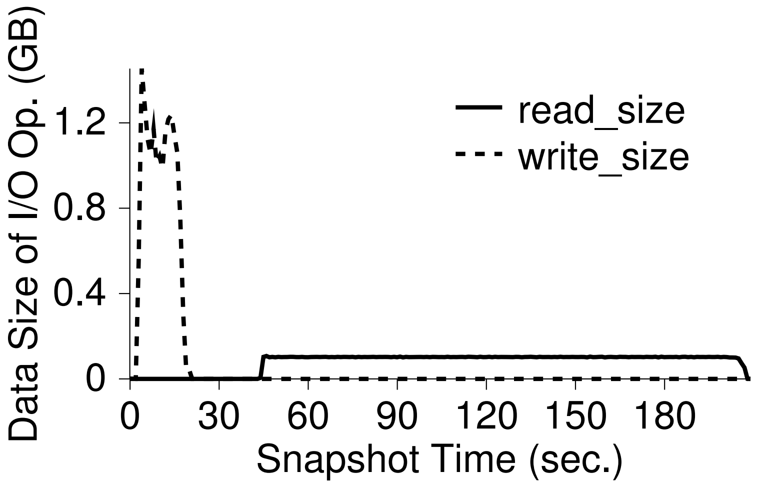


(b) ior2-shared size

Figure 4.6: I/O Requests over Time for *ior2-shared* Benchmarks



(a) ior2-stride number



(b) ior2-stride size

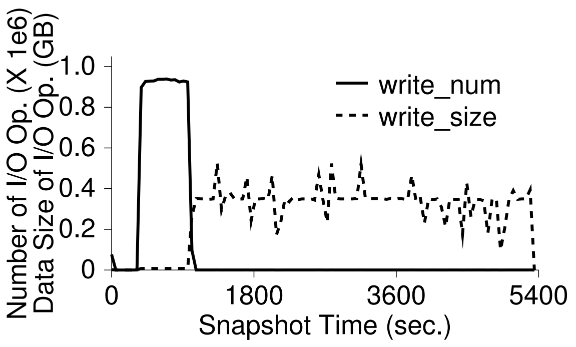
Figure 4.7: I/O Requests over Time for *ior2-stride* Benchmarks

fileproc benchmark, using the one-file-per-node model, presents the best write performance. Up to 150,000 write requests per second, totaling 9 gigabytes per second, are generated by the 512 nodes. However, the *ior2-shared* and *ior2-stride* benchmarks can only achieve 25,000 write requests per second, totaling 2 gigabytes per second. These two benchmarks use the shared-region and the shared-stride file model, respectively. We believe that the performance degradation is caused by the underlying file consistency protocol. This result is somewhat counterintuitive. The shared-region file model appears to be similar to the one-file-per-node model because the contiguous regions in the former are analogous to the separate files in the latter. Therefore, their performance should be comparable as well. The severe performance degradation implies that the shared-file model is not optimized for this scenario.

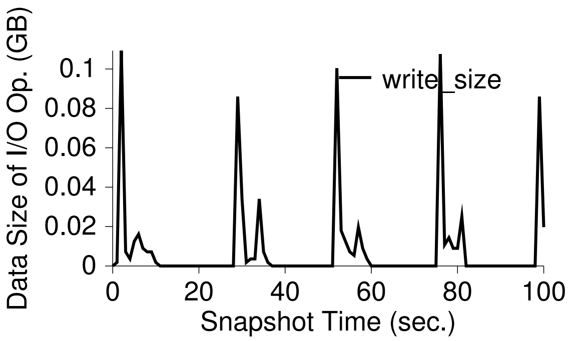
After a write, each node reads back another node's data as soon as it is available. The gaps between the write and read curves in each sub-figure reflect the actual I/O times. Obviously, the *ior2-fileproc* benchmark demonstrates much better performance: only 10 seconds are used in this model, while more than 20 seconds are needed to dump the same amount of data when using the shared file model. Since reads must be synchronous, we can easily figure out the file system read bandwidth from the *read_size* curve. The *ior2-fileproc* and *ior2-shared* benchmarks have comparable read performance. However, the *ior2-stride* has the worst read performance, only 100 megabytes per second for 512 nodes. This result is not surprising: the stride data layout in shared files limits the chances of large sequential reads.

Figure 4.8 shows the I/O access pattern of the application *f1*. As we mentioned before, *f1-write* has very few reads and *f1-restart* has very few writes. Therefore, we can ignore those requests in the corresponding figures. In Figure 4.8(a), we chose a resolution of 50 seconds because it becomes unreadable if we use finer time resolutions. The spike of the *write-num* curve is caused by the activities of the master node to collect small pieces of data from other computing nodes. At its peak time, nearly 1 million file system requests are issued per second. However, due to the very small request size (8 to 16 bytes), this intensive write phase contributes negligable amounts of data to the overall data size. In the rest of the application, large write requests from 48 nodes dominate the I/O activities. Requests are issued in a very bursty manner. Figure 4.8(b) zooms in on a small region of Figure 4.8(a) with 1 second resolution. It shows that sharp activity spikes are separated by long idle periods. At the peak time, up to 120 megabytes per second of data are generated by 48 nodes. In the restart phase of *f1*, read requests become dominant. However, both the number and the data size of read requests are small compared to those in the write phase.

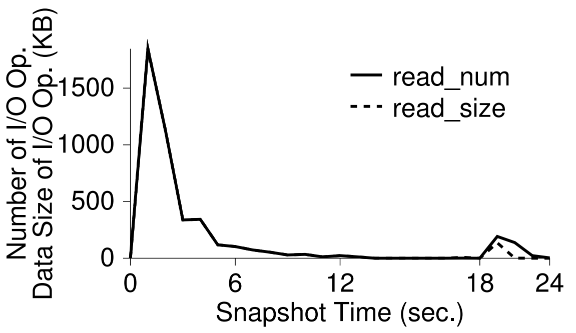
Figures 4.9 and 4.10 presents the I/O access pattern of the physics application *m1*. It demonstrates very good read performance: nearly 28 gigabytes per second bandwidth are achieved by 1620 nodes, thanks to the large read size (1.6 megabytes – 16 megabytes). Like *f1*, its write activities are also bursty. We observed that the write curves have similar shapes in Figure 4.10: They all begin with a sharp spike followed by several less intensive spikes. One possible explanation is that the file system buffer cache absorbs the coming write requests at the begin of the writes. However, as soon as the buffer is filled up, the I/O rate drops to what can be served by the persistent storage.



(a) time-f1-write

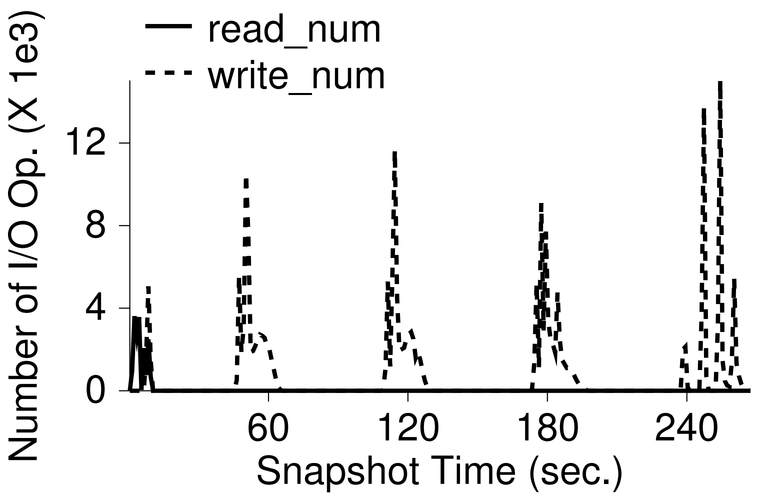


(b) time-f1-write-short

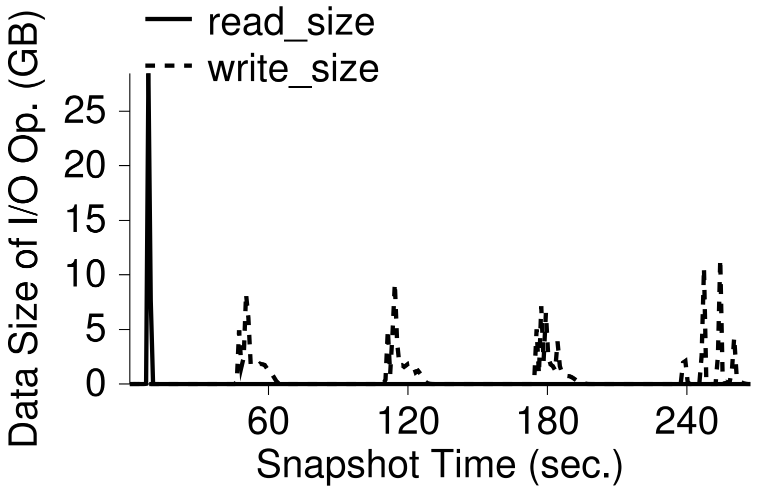


(c) time-f1-restart

Figure 4.8: I/O Requests over Time for *f1* Application. The number of I/O operations plotted in figure (a) is the average amount per fifty-second bucket.

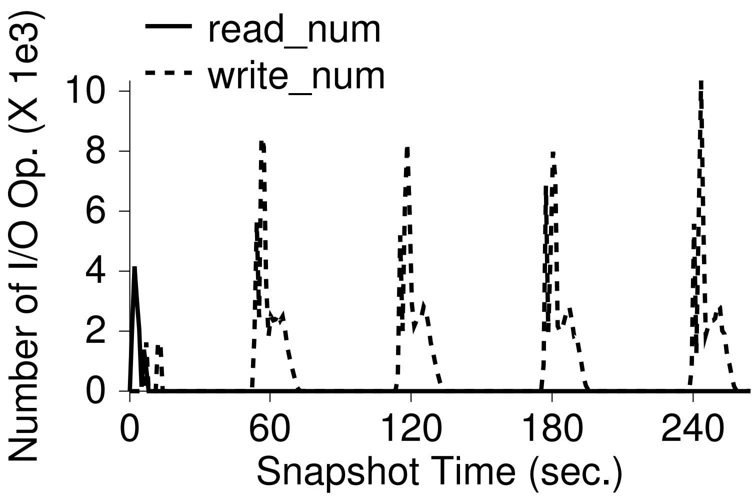


(a) m1-restart-num

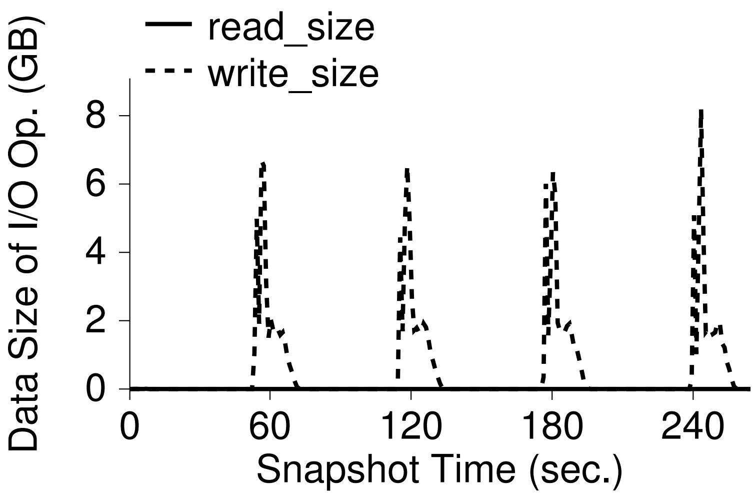


(b) m1-restart-size

Figure 4.9: I/O Requests over Time during *m1* Restart Phase.

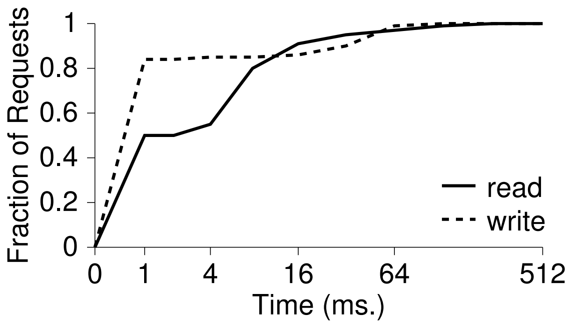


(a) m1-write-num

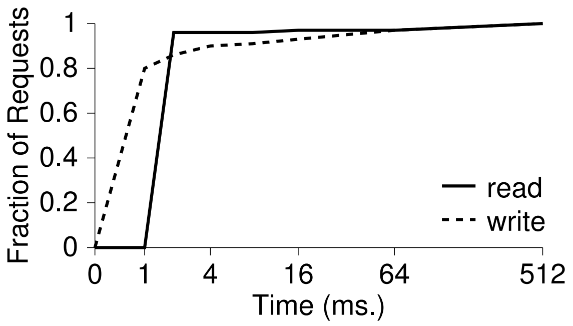


(b) m1-write-size

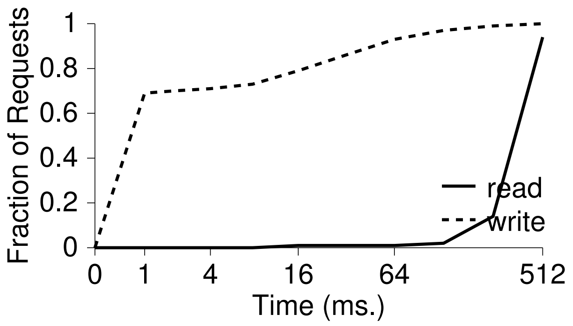
Figure 4.10: I/O Requests over Time during *m1* Write Phase.



(a) inter-ior2-fileperproc

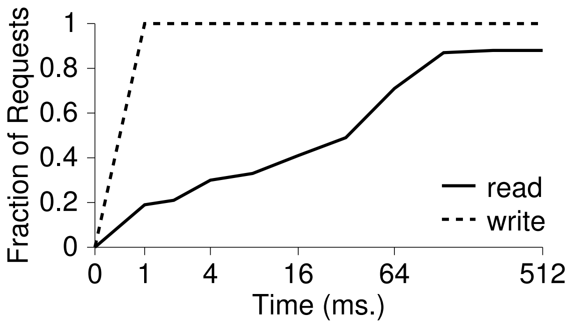


(b) inter-ior2-shared

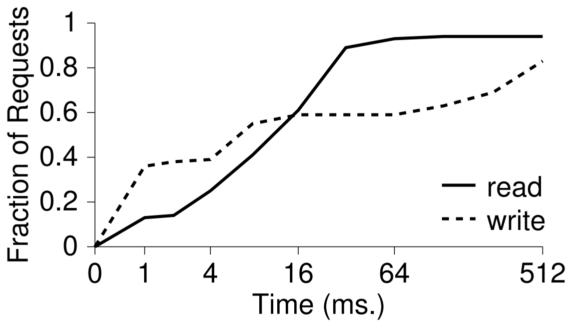


(c) inter-ior2-strided

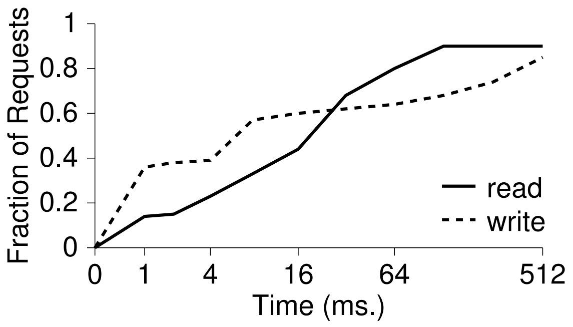
Figure 4.11: Cumulative Distribution Functions (CDF) of Inter-arrival Time of I/O Requests (X axis-logscale) for *ior2*.



(a) inter-f1-write



(b) inter-m1-write



(c) inter-m1-restart

Figure 4.12: Cumulative Distribution Functions (CDF) of Inter-arrival Time of I/O Requests (X axis-logscale) for application *f1* and *m1*.

4.1.2.4 I/O Burstiness

To study I/O burstiness, we measured I/O request inter-arrival times. Figures 4.11 and 4.12 show the cumulative distribution functions (CDF) of I/O request inter-arrival times. Note that the x-axis is in the logarithmic scale. Write activity is very bursty in the *ior2* benchmarks and the *f1* application: about 65–100% of write requests have inter-arrival times within 1 millisecond. In *ior2* and *f1*, most of the write activities are due to memory dump and I/O nodes can issue write requests quickly. However, write activity on *m1* is less intensive than on *ior2* and *f1* because *ml* distributes I/O activities evenly on a larger (3X–5X) cluster, which has 3 to 5 times more nodes than those for *ior2* and *f1*.

On the other hand, read requests are generally less intensive than write requests because reads are synchronous. In particular, Figure 4.11(c) indicates that *ior2* under shared-strided files suffers low read performance, as described in Section 4.1.2.3. In this scenario, data are interleaved in the shared file and read accesses are not sequential.

4.1.2.5 I/O Nodes

In this section, we study the distributions of I/O request sizes and numbers across the different nodes, as shown in Figure 4.13– 4.17. For the *ior2* benchmarks, read and writes are distributed evenly among nodes, as shown in Figures 4.13(a) and 4.13(b), because each node executes the same sequence of operations in these benchmarks.

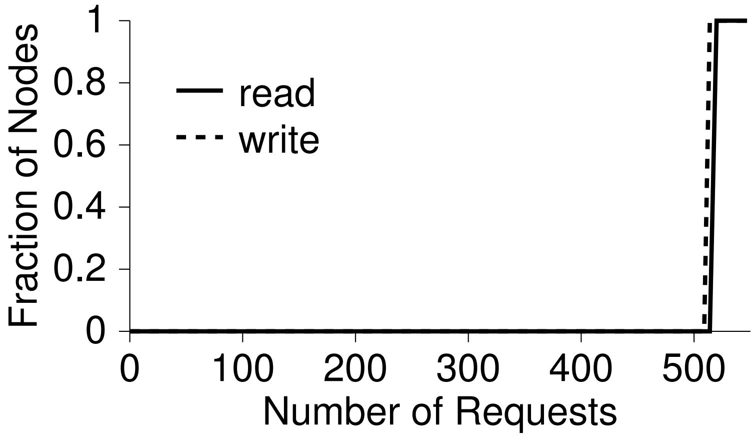
In the physics application *f1*, a small set of nodes write gathered simulated data to a shared file. Therefore, only a few nodes have significant I/O activity in their write phase and most of the transfered data are from large write requests (14% of the write requests), as shown in Figures 4.14(a) and 4.14(b). There is little read activity in the write phase. However, read requests are evenly distributed among nodes in the restart phase and their sizes are around 1 megabyte, as shown in Figures 4.15(a) and 4.15(b). There is little write activity in the restart phase.

In the restart and write phases of the physics application *m1*, I/O activity is well balanced among nodes, as shown in Figures 4.16(a)–4.17(b). We also observe significant write activity in the restart phase.

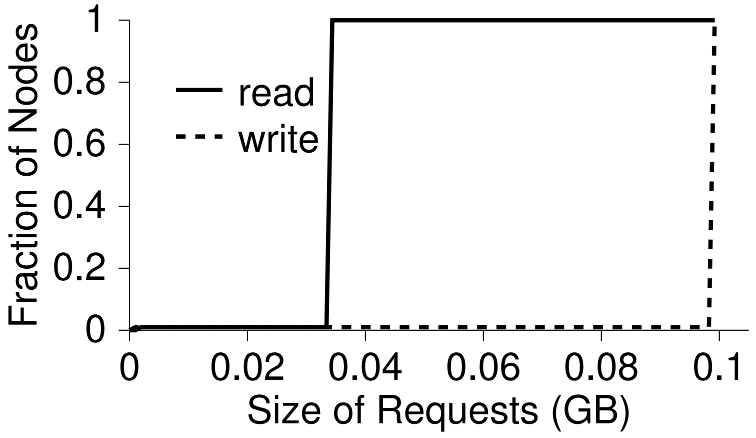
4.1.2.6 File Opens

We also studied the file open patterns of the applications. All file open data are retrieved directly from the application traces. Since our traces are at the file system level, every trace record, corresponding to a file operation, contains a file handle number, either as an operation parameter or as a return value. For example, a file open record saves the opened file handle number as the return value and a file read record saves the file handle number as the parameter. Using the file handle number in every trace record, we compiled the file activities between pairs of open/close operations and retrieved their characteristics, shown in Table 4.3 and Table 4.4.

In both tables, we differentiate *data file* operations from the other operations. The *data file* is defined as the file that actually stores the application data. In large distributed storage systems, especially in scientific computing environments, I/Os to the application data dominate. Their characteristics are more interesting and have more impact on the file system design. Therefore, we list them separately in both tables.

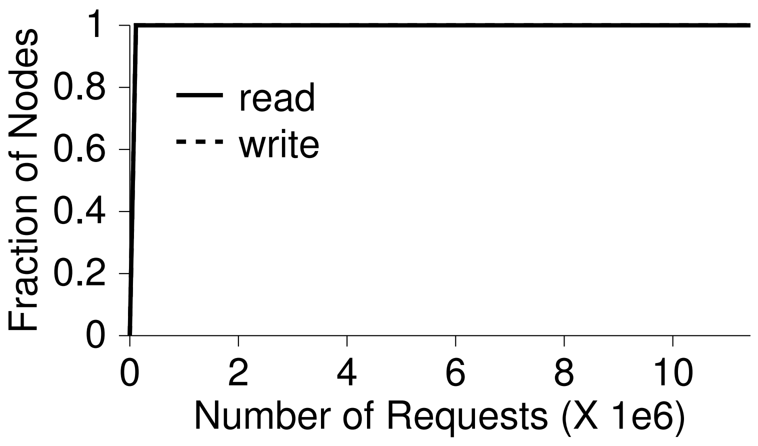


(a) node-ior2

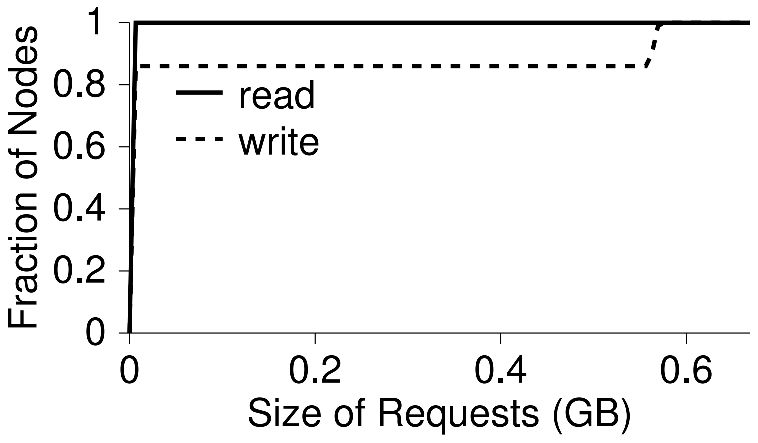


(b) node-ior2

Figure 4.13: Cumulative Distribution Functions (CDF) of the Size of I/O Requests over Nodes for *ior2*.

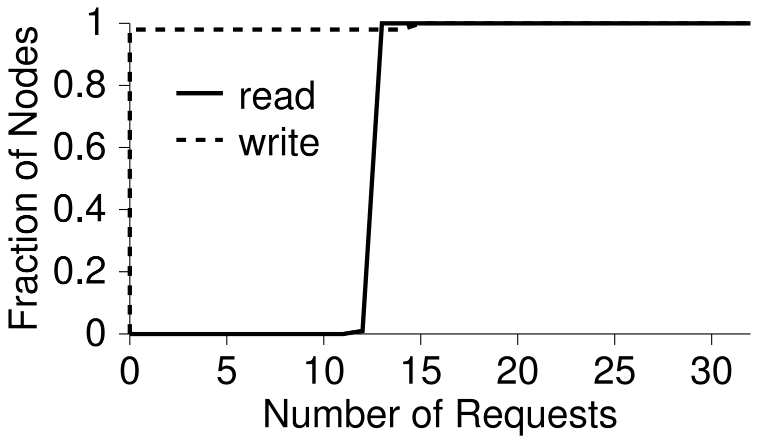


(a) node-f1-write-num

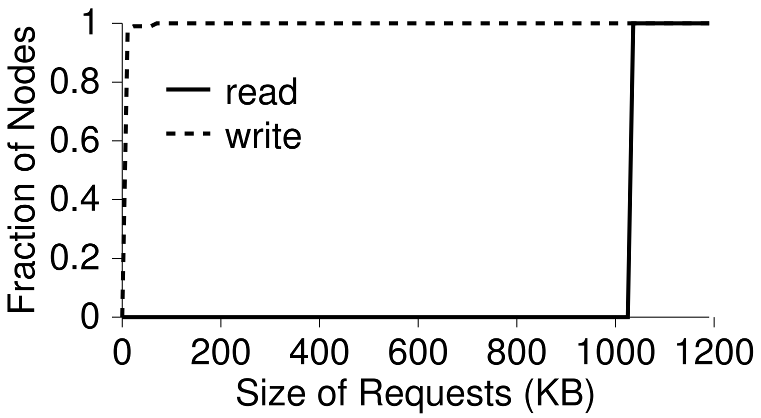


(b) node-f1-write-size

Figure 4.14: Cumulative Distribution Functions (CDF) of the Size of I/O Requests over Nodes during *f1* write phase.

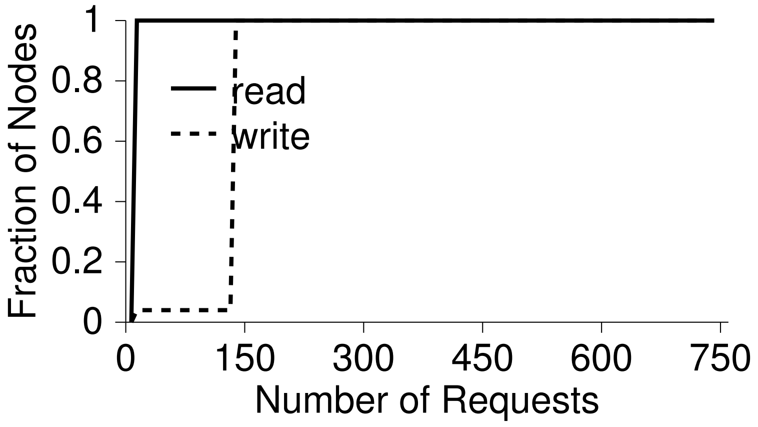


(a) node-f1-restart-num

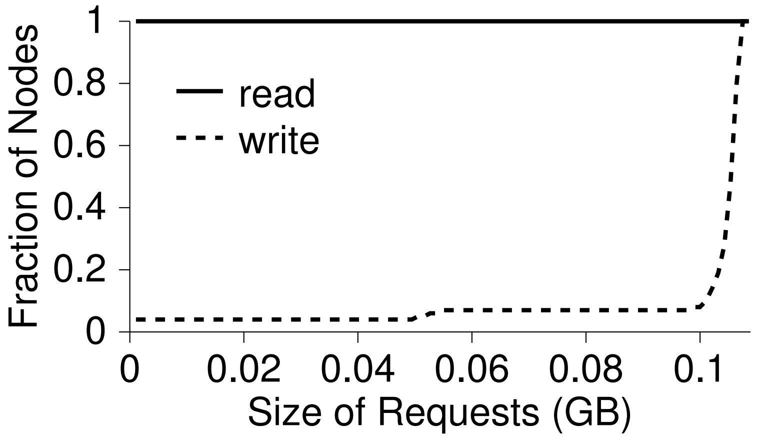


(b) node-f1-restart-size

Figure 4.15: Cumulative Distribution Functions (CDF) of the Size of I/O Requests over Nodes during *f1* restart phase.

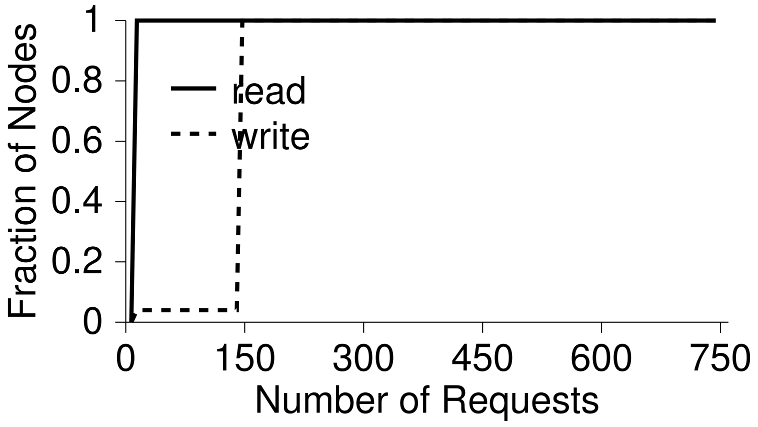


(a) node-m1-write-num

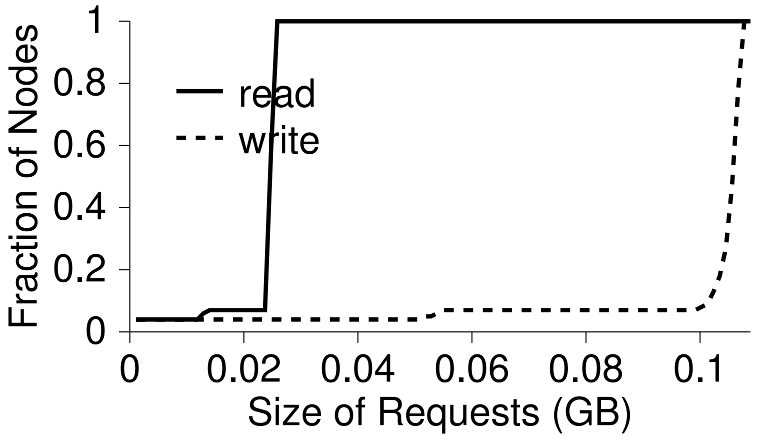


(b) node-m1-write-size

Figure 4.16: Cumulative Distribution Functions (CDF) of the Size of I/O Requests over Nodes during *m1* write phase.



(a) node-m1-restart-num



(b) node-m1-restart-size

Figure 4.17: Cumulative Distribution Functions (CDF) of the Size of I/O Requests over Nodes during *m1* restart phase.

Table 4.3: File Open Statistics

Applicatons	Overall Number of File Opens			Number of Data File Opens		
	Read/Write	**Read**	**Write**	**Read/Write**	**Read**	**Write**
ior2	6,656	5,121	0	1,024	0	0
f1-write	3,871	6,870	718	98	10	34
f1-restart	3,773	6,179	0	0	343	0
m1-restart	17,824	22,681	12,940	0	1,620	12,960
m1-write	17,824	21,061	12,960	0	0	12,960

In all applications, files tend to be opened either for both reading and writing or only for reading. We observe significant write-only files only in the physics application *m1*, as shown in Table 4.3. However, data files are opened either read-only or write-only except for the benchmark *ior2*. The open operations on the data files only account for a small portion of the overall files opened. Given the fact that the data file operations dominate the overall I/O, the small number of data file opens implies longer open time and more I/O operations during each open. As listed in Table 4.4, the open duration of data files ranges from several seconds to several hundred seconds, which is typically 2 to 20 times longer than overall file open durations. The average number of operations and the size of data files on each open operation are also much larger than those on the overall files. For example, up to 400 MB data are transferred during each data file open in physical application *f1-write*.

4.1.3 File System Workload Summary

In this study, we analyzed application traces from a cluster with hundreds of processing nodes. On average, each application has only one or two typical request sizes. Large requests from several hundred kilobytes to several megabytes are very common. Although

Table 4.4: Operations During File Open

Applications	Avg. open time		Avg. IOs per Open		Avg. IO Size per Open	
	Overall	Data File	Overall	Data File	Overall	Data File
ior2-fileproc	0.4 sec	4.5 sec	44.4	512.0	2.8 MB	32.8 MB
ior2-shared	0.7 sec	5.2 sec	44.4	512.0	2.8 MB	32.8 MB
ior2-stride	7.6 sec	26.57 sec	44.4	512.0	2.8 MB	32.8 MB
f1-write	20.2 sec	504.9 sec	14.8	142161	2.4 MB	3993.5 MB
f1-restart	0.02 sec	0.1 sec	0.5	1	$<<$ 1 MB	$<<$ 1 MB
m1-restart	1.2 sec	3.9 sec	4.2	15.3	3.7 MB	8.5 MB
m1-write	1.2 sec	2.4 sec	4.3	17	3.1 MB	6.5 MB

in some applications, small requests account for more than 90% of all requests, almost all of the I/O data are transferred by large requests. All of these applications show bursty access patterns. More than 65% of write requests have inter-arrival times within one millisecond in most applications. By running the same benchmark on different file models, we also find that the write throughput of using an individual output file for each node exceeds that of using a shared file for all nodes by a factor of 5. This indicates that current file systems are not well optimized for file sharing. In the applications examined, almost all I/Os are performed on a small set of files containing the intermediate or final computation results. Such files tend to be opened for a relatively long time, from several seconds to several hundred seconds. And a large amount of data are transferred during each open.

4.2 Object Workload Study

Our research on the scientific application workloads has led to a better understanding of the expected environments for the large scale distributed file systems and enabled more informed system development. However, such information is not adequate in designing a file system for object storage devices. As we described in Section 2.5, *Ceph* services file system requests through the client component in each client. The file system requests will be buffered at the client side and eventually translated into object requests. The object workload has unique characteristics that directly impact our OBFS design. Different architectures of high-level file systems and various data management policies may result in different object workloads. In this section, we study the object workload characteristics in *Ceph* and discuss their impact on the design of our OBFS.

4.2.1 Static Object Workload Characterization

Ceph breaks files into multiple objects using a fixed stripe unit size. Because the majority of disk space in scientific environments is occupied by large files–much larger than the system stripe unit size–most objects will be uniformly sized. However, small files and tails of large files will be packed into individual object, which will generate non-trivial number of small objects. In the following discussion, we refer to stripe-unit-size objects as large objects and smaller non-uniformly-sized objects as small objects.

Objects are distributed across the OSD cluster according to object placement policies. Object size is an important parameter that balances the aggregrated bandwidth of sequential file accesses and the overheads of object managements. A small object size increases the total number of objects in the system for a fixed mount of file data. Extra space and computational overheads associated with each object, such as object metadata and attributes, impose more cost for file I/Os as the number of objects scales up. Disk and network bandwidth uti-

Table 4.5: Object Workload Analysis Using LLNL Workloads

Stripe size (KB)	**Total Obj. Num. (million)**	**Avg. obj. per file**	**small obj. ratio**
32	399.3	130	3.3%
64	200.4	65	5.2%
128	100.9	33	7.8%
256	51.2	16	12.3%
512	26.3	8	16.8%
1024	13.8	4	28.4%

lization are also reduced due to the smaller object sizes. On the other hand, a large object size may reduce the average number of objects per file. In scientific environments, file I/Os are typically bounded by the aggregated bandwidth of underlying OSDs containing those data. Thus, performance benefits gained from parallel I/Os to multiple OSDs decrease proportional to the size of the objects.

Table 4.5 presents the characteristics of the expected object workloads based on the LLNL file system snapshots. As we can see from the table, small stripe sizes, such as 32 KB and 64 KB, introduce huge amount of objects into the system: about 10 to 30 times more than that using the largest stripe size. The majority of all objects are uniformly-sized. The small objects account for only 3.3% to 28.4% of the number of objects when the stripe size varies from 32 KB to 1024 KB. Consequently, the OBFS design optimizes for those fix-sized objects.

4.2.2 Object Access Pattern

In scientific environments, applications tend to access files sequentially. Although huge numbers of small random requests are observed in some situations, the total amount of data transfered by them can be ignored because client caches can effectively absorb those small requests and hide them from the underlying storage devices. Therefore, file data are typically dumped to the underlying storage devices in large chunks. As a result of the deep client buffering, most object requests tend to access objects as a whole. Using the LLNL application *m1* as an example, more than 85% of all object requests access entire objects if the system has a stripe-unit size of 512 KB.

In the *Ceph* architecture, objects are distributed across the OSD cluster randomly. After mapping file requests into object requests, the sequential access pattern can hardly be observed on an individual OSD. The object sequential access is defined as the object access order strictly following the order they were initially written to the OSD. The objects comprising small and medium files will typically be spread across different OSDs. Sequentially accessing those files will lead to random object accesses in otherwise unrelated OSDs. Very large files may have multiple objects on each OSD. Long sequential accesses to those large files may result in sequential object accesses on individual OSDs. However, the run lengths of such sequential object accesses are greatly reduced due to the large OSD cluster and evenly distributed schemes. For example, if the OSD cluster has one hundred OSDs, a long sequential access to a file with one thousand objects may result in a ten-object sequential access on each OSD. Such short sequential object accesses can be further destroyed by interleaved object accesses from other clients. In short, sequential objects accesses will be rare and those that do occur will generally have short run-lenghs. From an individual OSD point of view, accesses to objects will be random, and little benefit will be realized from attempting to optimize the location of related objects.

An individual OSD is likely to see more write requests than read requests. Both because many read requests can be served by large client caches and because the replication schemes used in the large distributed object system double or triple the total number of write requests. For example, *Ceph* can ensure data safety by double or triple mirroring objects to different OSDs. A write request to an object can therefore result in two or three write requests to different OSDs, including both synchronous and asynchronous requests. In order to guarantee data safty, OSDs must synchronously serve foreground write requests. However, some background requests such as those due to object migration or data recovery may be served asynchronously to maximize disk bandwidth utilization.

4.2.3 Object Workload Characteristics Summary

Most objects are the same size as the system stripe unit size. Based on the LLNL file system snapshots and assuming a system stripe size of 512 KB, around 83% of all objects will be large objects. In scientific environments, the majority of all object requests access whole objects. Partial object accesses are not significant. Objects on an individual OSD show very weak relationship with each other due to the need for objects of an individual file to be placed on different OSDs to allow for high-performance parallel access to a single file. Objects from a small or medium file are rarely placed on the same OSD and few objects from a large file will ever be on the same OSD. From the OSD point of view, objects tend to be accessed randomly and seldomly show a sequential pattern. Object workloads are dominated by write requests. Although the foreground requests from clients have to be served synchronously, there are still a lot asynchronous write requests generated from background activities. OSDs have to handle both synchronous and asynchronous write requests efficiently.

Chapter 5

OBFS Design

The expected object workload of an individual OSD shows unique characteristics. A specially optimized file system for such object workloads is able to fully utilize the underlying storage bandwidth and provide better performance than general-purpose file systems. OBFS has been designed to be the storage manager on an individual OSD in the Ceph system. It optimizes disk layout and enhances flat-namespace management based on our understanding of the expected object workload. In this chapter, we present the OBFS design as follows: a design overview is provided in Section 5.1; We then discuss the variable-sized blocks in Section 5.2 and the self-contained region structure in Section 5.2.1; The metadata and file system structures and their on-disk layouts are presented in Section 5.3, followed by a brief discussion about their benefits and overheads in Section 5.4; We conclude this chapter in Section 5.5.

5.1 Design Overview

As described in Section 4.2, the expected workload of our OSDs is composed of many objects, with sizes ranging from a few bytes up to the file system-stripe-unit size. Therefore, OBFS needs to optimize large object performance to provide a high overall throughput without over-committing resources to small objects. 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 20 PB storage system, this 9% difference represents about 180 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 manner. To address this need, OBFS employs multiple block sizes and uses *regions*, analogous to cylinder groups in FFS [40], to keep blocks of the same size together. An object cannot span multiple regions. Thus, the read/write performance of large objects can be greatly improved by using very large blocks, while small objects can be efficiently stored by using small blocks. Regions can also greatly reduce the system fragmentation as the file system ages. Large objects are always laid contiguously on disk using large blocks. While small objects can be fragmented inside regions, such fragmentation cannot span a region boundary. In the scientific workloads that we are interested in there are far fewer small block regions than large block regions. Thus, the fragmentation is constrained to a limited number of regions no matter how long the file system is exercised. In the worst case–say the workload is composed of huge numbers of small objects–OBFS degenerates to a general purpose local file system containing all small block regions. However, in the normal case in which large objects and small objects are mixed together, OBFS can significantly reduce the overall fragmentation, especially after long-term running.

In addition to reducing fragmentation, regions also provide failure boundaries for a file system. OBFS partitions all important metadata into regions and manipulates them separately. Combined with a boundary check technique that validates I/O addresses of normal operations, OBFS is able to isolate any random error inside a region boudary.

Another important feature of OBFS is the use of a flat namespace. 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 or useful locality information is likely to be available. Note that in a small system in which an OSD may hold several objects of a file, some locality information may be available. However, 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 [77], to organize the objects and achieve much higher performance on directory operations.

5.2 Multiple Block Sizes

The multiple-block-size strategy used by OBFS aims to lay out variable-sized objects sequentially without overcommitting disk space for small objects due to the block internal fragmentation. 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 scheme, 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 high-level system-stripe-unit size and twice the block size of GPFS (256 KB). The block sizes are system-tunable parameters that have direct impacts on the overall performance and the disk utilization. The detailed performance comparisons for different block sizes are available in Section 7.3.

Those regions that contain large blocks are called *large block regions* and those that contain small blocks are called *small block regions*. Objects with system-stripe-unit size are always allocated into large block regions, where each object only occupies one large block. Regions that have not been initialized are referred as *free regions*. All other objects are allocated into small block regions, which, like files in a traditional file system, may span multiple small blocks inside a region. With this strategy, large objects can be laid out contiguously on disk in a single large block. OBFS further simplifies the large object layout by collocating the object metadata, the onodes, together with large blocks. Only one disk seek is incurred during the transfer of a large object. The throughput of large objects is greatly improved, thanks to the reduced metadata operations and the shortened seek time inherent in this design. For small objects, 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 sizes of other file system data structures such as free block lists or maps and thus makes the operations on those data structures more efficient.

The idea of large/small blocks is similar to that of the block/fragment in FFS. Like FFS, which uses bigger blocks to optimize disk bandwidth utilization and small fragments to improve space utilization, OBFS achieves the same goal by employing large/small blocks. However, their optimization scopes have some difference due to the different assumptions of expected workloads. A file in FFS is composed of multiple blocks and fragments, while an object in OBFS can only employ either a large block or multiple small blocks. Since most objects are laid contiguously on disk through large blocks, OBFS needs to optimize the layout of the small block objects, which is done by organizing them into regions. While in FFS, fragments are generated on demand by breaking down a block and spreading it across the whole file system. It may easily fragment the free space as the system ages, depending upon workload characteristics. Moreover, if many blocks are partially used due to fragments, it may incur extra overheads by packing and relocating those fragments.

5.2.1 Self-contained Region Structure

OBFS design partitions a raw disk into self-contained regions. As shown in Figure 5.1, regions are located in fixed positions on the disk with uniform sizes, except the last one, which may be a partial region. All blocks in a region have the same size, but the block sizes of different regions may be different. Any given object can only live inside one region,

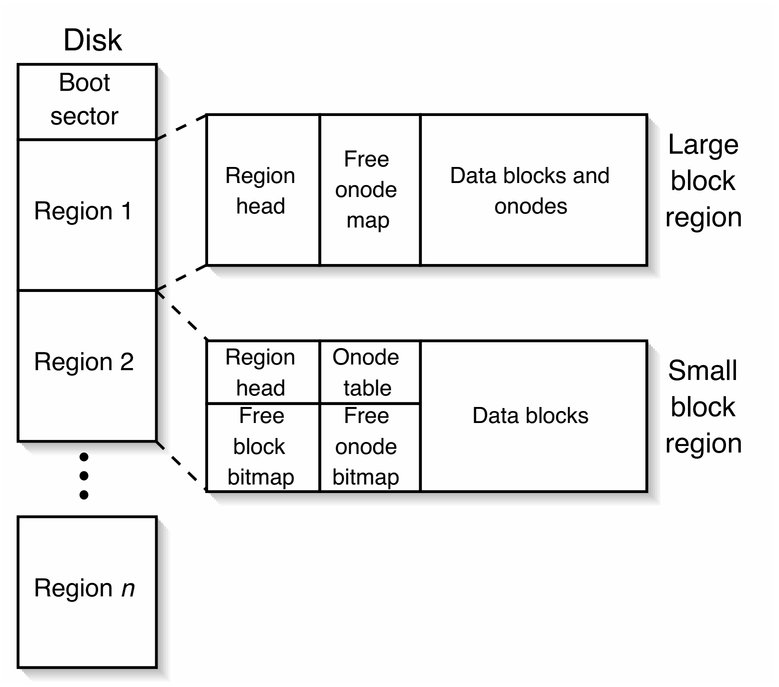


Figure 5.1: OBFS structure

with all necessary metadata encapsulated in that region. The block size in a free region is undefined until that region is initialized on demand. When there are not enough blocks to satisfy a write request or the free space is too fragmented, OBFS will allocate a free region and initialize it to contain the desired block size. Used regions are inserted back into the free region list when all of their blocks are freed.

A region is self-contained in that it is able to store and retrieve all its objects independent of other regions and global data structures. All the metadata for its objects are packed inside the region. The global data structures in OBFS are designed to facilitate region-based management of objects. However, in the scenarios in which those global structures are damaged, OBFS is still functional and can re-assemble them using the redundant information stored in the regions. This eliminates global data structures as a single point of failure in the system. In the case of non-recoverable errors, such as media failures or software bugs that destroy important metadata, it is still possible to retrieve most objects in an OSD.

The region structure also provides a failure boundary in a finer granularity inside a file system. Regions manage their own resources independently, which prevents random errors in one region from corrupting data and metadata in others. OBFS maintains such failure boundaries by enforcing address checks for every I/O: the disk address of an I/O, initiated to one region, must fall inside the region's address space. Those checks typically happen during mapping of the objects or region structures to disk blocks. As a result, any random error is always confined inside a region and will not introduce catastrophic events that may potentially destroy the whole file system. For example, a random bit flip in a block bitmap may cause a used block to be allocated again. If such a duplicated block happens to be used by an inode or an indirect block, it can potentially redirect a regular file write into any position on disk, which may destroy the whole file system. In OBFS, duplicated blocks across region boundaries can be detected before actual data is written back to disk. Therefore, data corruption caused by various random failures can only happen inside a region. If any error is found in some region, it is adequate to only scan the region to bring the whole system back to a consistent state. We will discuss the failure scenarios and recovery details in Section 6.2.

The region concept is similar to the allocation group or allocation unit in general-purpose Unix file systems. They all partition the disk space into smaller units and optimize data layout by co-allocating file/object metadata together with data blocks in the same unit. However, the allocation group does not provide a clear logical boundary: a file/object can still be placed across different allocation groups. Although an allocation group contains sufficient metadata, such as the block bitmap, to manage all its physical resources, it lacks some logical structures that enable it to independently manage its file/object namespace. It has to rely on other allocation groups and global structures to store and retrieve files/objects. For instance, to read/write a file/object in one allocation group, it needs to first load the corresponding directory entry, which may be located in another allocation group. The region structure in OBFS extends the allocation group concept and provides an isolated domain for both physical resources and logical namespace. Such a design is enabled by the object workload characteristics under the Ceph framework: the object size is bounded by the system-stripe-unit size and the logical relationship among objects is opaque at the OSD level.

5.2.2 Fragmentation

The design of the region structure and the variable-sized blocks can potentially reduce file system fragmentation, avoids unnecessary wasted space, and more effectively uses the available disk bandwidth. In this section, we discuss several different types of fragmentation: object fragmentation, free space fragmentation, region fragmentation, and block internal fragmentation. The object fragmentation refers to the situation that an object is not laid contiguously on the disk. The free space fragmentation means that the disk free space is partitioned into small pieces, which are not able to provide contiguous space for new objects. The region fragmentation refers to the unusable space due to mismatch of the object type and the region type. And the internal fragmentation refers to the wasted space inside a block due to the allocation alignment.

The large block in OBFS guarantees that all stripe-unit-sized objects are laid contiguously on disk no matter how long the system ages, since those objects occupy exactly one large block each. In traditional block-based file systems, a file is composed of multiple blocks. During a file allocation, the block allocator tends to select as the candidate the nearest free block from the last allocated block of the same file. In a system with plenty of free space, such a scheme is able to place a file contiguously on disk. However, in a heavily used system or an already fragmented system, the block allocation policy tends to spread a file into multiple disk locations and further fragment the free space because it only does one block allocation at a time. Thus, the traditional block-based file systems usually exhibit decreased performance as the system ages and disk usage increases. The extent-based file systems, on the other hand, manage the free space in variable-sized extents. Their allocation policies have a global view of the free space and are able to achieve nearly optimal allocation results. Under the expected object workloads, the optimal layout of large objects in OBFS guarantees that the fragmentation of those large objects is much less than that in the block-based file systems and is comparable to or a little better than that in the extent-based file systems.

The small object allocation, as described in Section 6.1, employs an extent-like allocation policy. It builds up a very compact free extent list in core for each small block region, which tracks large free extents of that region. The allocation for large extents can be done very quickly by looking up the in-core list, while small extents are allocated through scanning the block bitmap. Such a scheme is able to achieve much better layout of small objects than those in the traditional block-based file systems, and a little worse or comparable layout than those in the extent-based file systems.

OBFS improves the global free space fragmentation by separating small objects into different regions from large objects. As a result, the free space fragmentation is bounded inside the small block regions. In a scenario in which objects come and go frequently, a traditional file system can be severely fragmented due to the mix of small and large objects. For example, an extent-based file system may receive a small object allocation request after removing a large object. It will break down the freed large object space to serve the outstanding small object request if no matched small extent is found. If such a pattern keeps repeating, the free space of the file system will be fragmented. OBFS adapts to the mixed workloads much better than the traditional file systems. No matter how long a file system ages, the large block regions are still able to provide contiguous space for large objects.

By introducing the region concept, OBFS can significantly alleviate the file system aging problem. However, it can potentially incur a region fragmentation problem. Specifically, an OBFS may run into a situation that regions of one type are sparsely occupied by live objects, while regions of the other type are short of free space. Although the overall disk utilization may be far from full, an object write to the region of the latter type will find no free space. The file system has to stop and clean those regions. However, such a situation can only happen in the very rare case that the ratio of large to small objects changes significantly during the lifetime of the system. By choosing the right region size and using the region-packing technique described in Section 6.3 we can dramatically decrease the region clean events.

Higher throughput in OBFS does not come at the cost of wasted disk space. Internal fragmentation in OBFS is comparable to that 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 the blocks used 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 happens rarely enough (due to aggressive write coalescing in the client caches) that it does not have a significant impact on system performance. Also, and the inter-region locality of the small blocks makes this a very efficient operation.

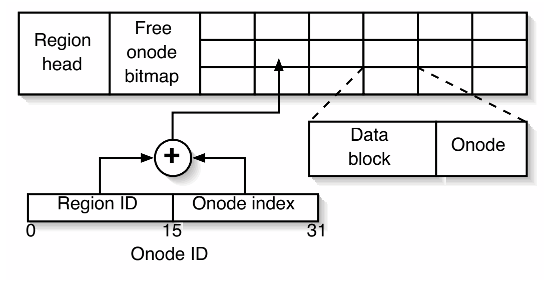
5.3 Metadata and File System Structures

One goal of the OBFS design is to minimize file system metadata to make them as efficient as possible. As mentioned in Section 4.2, OBFS is designed specifically for managing a flat namespace without worrying about the complicated directory hierarchy. This gives us an opportunity to simplify the metadata structures to reduce metadata-related overhead and maximize data throughput. To achieve this goal, we design different metadata schemes for large and small objects, respectively. The important metadata include the onode, *bitmaps* for both onode and data blocks, *region head* for region-related information, and *object lookup table*.

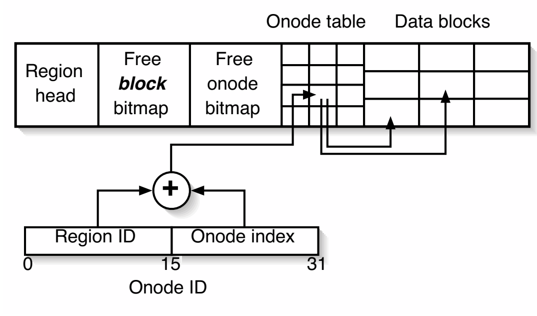
5.3.1 Object Metadata

Object metadata, referred as an onode, is used to track the status of an object. Onodes are pre-allocated in a fixed position at the head of a small block region, similar to the way inodes are placed in cylinder groups in FFS [40]. In a large block regions, shown in Figure 5.2, onodes are packed together with data blocks on disk, similar to embedded inodes [16]. This allows for very low overhead of metadata updates since the metadata can be written together with the corresponding data block in a large, sequential disk write.

Figure 5.2 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



(a) Large Block Region



(b) Small Block Region

Figure 5.2: Region structure and data layout.

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 large onode size also allows OBFS to pack more small files (less than 384 bytes) directly inside onodes, which can greatly improve the small file performance and reduce wasted space. Since the identifier of a large object can be used to directly address the block, an onode of a large block is exempted from storing address information. Instead, it is dedicated for object attributes and file system book-keeping information.

Small objects, on the other hand, are composed of multiple small blocks and require onodes to keep track of such address information. In the OBFS design, the maximum size of a small object is always less than the stripe unit size, which is 512 KB. Because the OBFS layout policy assigns objects to a single region, we can use the relative address to index the blocks. The number of bytes for a block address depends upon the region size and is initialized during the OBFS format process. For example, assuming that 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 address to index all of the blocks in the region. If an OBFS employs a region size larger than 256 MB, it can either change the small block size proportionally to maintain the two-byte small block address or simply assign more bytes for small block addresses.

Blocks allocated to an object are organized into extents, which are represented by tuples: $<$block address, object offset, length$>$ inside an onode. In the normal configuration, the size of a tuple is four bytes. The first two bytes are used for the *block address*. The rest are assigned to *object offset* and *length*, one byte each. By using the *object offset* parameter, OBFS allows holes inside an object. However, the boundary of a hole has to align to the underlying small block size. Given an onode size of 512 bytes, it might be possible that a heavily-fragmented object will not fit all its address tuples into the onode since OBFS reserves the first 128 bytes of an onode for object attributes and other book-keeping information. For example, an object with 96 or more noncontiguous extents requires 384 or more bytes insides an onode, which cannot be satisfied by the current 512-byte onode. To address this problem, the OBFS allocation policy limits the maximal number of extents for an object. If the free space of a region is too fragmented, OBFS will either skip that region or start a clean process to clean the region and produce more contiguous space. The clean action rarely happens because it implies that all small regions are heavily used and that no free region is available, which is only possible when the disk utilization is extremely high. OBFS keeps monitoring the fragmentation of the small block regions. If it reaches a predefined threshold, the clean process will be triggered in the background. Such background cleaning can further reduce the occurrence of a foreground clean process and improve performance even when the system utilization is very high.

Besides the address information, three important fields are also stored in both the large and small object onodes, including an object identifier, a generation number, and a valid flag. The object identifier provides a reverse mapping from an onode to the object it stores. The generation number and the valid flag are used to identify whether an onode is active. Those fields contains redundent structural information. OBFS uses them to rebuild important data structures if the file system experiences serious damages, as described in Section 6.2.

5.3.2 Region Head and Bitmaps

The region head data structures summarize the region status and maintain the bitmaps for both onodes and blocks. Region heads for large block regions are quite simple in that they contain only region summaries together with block bitmaps. Since the onodes for large blocks are packed with large blocks, the onode bitmaps are merged with the block bitmaps. A small region head is composed of a region summary, an onode bitmap, a block bitmap, and a small delete log.

Region heads are laid at the beginning of regions on disk and organized into the Region Head List (RHL) in core. The RHL serves the same purpose as the block bitmaps and inode bitmaps in the traditional file systems. However, unlike the bitmaps in traditional file systems, those in the RHL need not be updated synchronously to guarantee data reliability. Instead, OBFS stores a valid bit in each onode, from which the block and onode bitmaps can be rebuilt.

Each region maintains a monotonically-increasing generation counter. Any update to the region structures as well as the onodes will obtain a generation number from the counter, and then increase the generation counter by one. The generation number will be stored together with those bitmaps and onodes. To obtain more finely grained updates, bitmaps are broken into to 4 KB chunks. The first four bytes of each chunk are reserved for the generation number. In the face of various failures, the one with a larger generation number is more trustable. For example, after an OBFS crash, the *fsck* process will check each region by scanning the onodes and comparing their generation numbers with those of the onode bitmaps. If the generation number of an onode is larger than that of the onode bitmap, it will use the onode valid bit to update the onode bitmap and the extent tuples to update the block bitmap. This approach gives OBFS more flexibility to choose which metadata to update. It potentially reduces the mandatory metadata updates to guarantee data durability. Synchronous writes can benefit significangly from this scheme. If we assume that a region head has been loaded into memory, a large object create/write to that region requires only one disk access, which writes a large block and the contiguous onode. A small object create/write requires at most one more metadata update to write the onode in addition to the data block updates. The onode bitmap and block bitmap are only flushed to disk periodically. OBFS chooses to update the onode bitmap during delete operations. Multiple delete operations can be packed and updated to the disk in one single bitmap write. A more detailed discussion of data reliability and recoverability is presented in Section 6.2.

Two additional flags, a dirty flag and a corruption flag, are stored in the region head. The former flag is set on when an on-disk onode in that region is updated and cleared as the onode and block bitmaps are flushed back to disk. The latter flag is set on whenever OBFS detects a corrupted metadata in that region. It can only be reset after the region is scanned and fixed.

The size of a region head varies depending upon its type. The large-block-region head consumes less than 4 KB, while a small-block-region head occupies 32 KB disk space, assuming that the region size is 256 MB and the small block size is 4 KB. For a 500 GB disk, the maximal disk space used by region heads is about 64 MB if there are all small block regions. If we assume that 80% of all regions are large block regions, the total region head size is about 20 MB. Although we expect all region heads to be loaded into memory given their relatively small sizes under normal object workloads, OBFS does load them on demand and page them out if more memory space is desired. An in-core region head list works as a region head cache to manage all active region heads in the LRU fashion. After a region head has been loaded into memory, it will be inserted into the list. Those inactive region heads at the list tail can be purged out of memory if memory usage is under pressure.

5.3.3 Object Lookup Table

Given an object identifier, we need to retrieve the object from the disk. In a hierarchical namespace, data lookup is implemented by following the path associated with the object to the destination directory and searching (often linearly) for the object in the directory. In a flat namespace, linear search is prohibitively expensive. OBFS uses a hash table, the *Object Lookup Table* (OLT), to manage the mapping between object identifiers and onode identifiers. Each valid object has an entry in OLT that records its object identifier and onode identifier.

The OLT is a very compact data structure compared with the namespace management structures of other local file systems. The size of OLT is proportional to the number of objects in the OSD: with 200,000 objects residing in an OSD, OLT requires around 2 MB. Each hash entry contains 8 bytes, 4 bytes for the object identifier and 4 bytes for the onode identifier. Sixteen hash entries are packed into a hash line and every four contiguous hash lines are allocated to a block, which is 512 bytes. Hash entries are loaded into main memory in the unit of a block. The last entry in each hash line is reserved as a link pointer to connect more hash lines if more entries are needed for the same hash buckets. The number of hash buckets initiated is based on the disk size and the estimation of the average object size during the *mkfs* process. Given a disk with 500 GB capacity and the average object size of 250 KB, OBFS uses 2^{15} hash buckets and pre-allocates 2^{19} hash entries. This accounts for one hash line for each hash bucket. The total space pre-allocated to OLT is a little more than 4 MB. The pre-allocated part of OLT is called the *base part* of OLT. It has a fixed location on disk, contiguously after the super block. If the *base part* of an OLT cannot hold all of the entries, OBFS will dynamically allocate a meta file in a small block region to collect the extra entries. If OBFS needs to append a new hash line to a hash bucket, it will allocate the new hash line in a meta file and put the onode identifier of that meta file and in-file offset of the new hash line into the last entry of the target hash bucket. Those extra hash lines are referred as the *extensible part* of OLT.

During a normal file system mount, the *base part* of OLT is loaded into main memory and stays there until the file system is unmounted. This in-memory structure would appear to waste a certain amount of main memory if the file system is seldom touched. However, in normal situation, the overall throughput is significantly improved because the in-core OLT greatly reduces lookup overhead. Compared with the directory entry cache used by traditional file systems to retrieve the structural information, OBFS's OLT has a much smaller footprint in memory. Given a system with 400,000 objects, an OBFS can fully load all the object-mapping information in core using less than 5 MB memory resources. While for a traditional local file system, more than 20 MB memory resources are needed to store all the directory entries for the objects. The small memory footprint of OLT effectively increases the data cache size, reduces disk accesses, and improves the lookup speed.

5.4 Metadata Overhead

The metadata overheads of a file system include the extra space used for storing its metadata, additional I/Os to read and write them, and computation time for retrieving desired information from them. The design of OBFS focuses on reducing the overheads and achieving both space efficiency and time efficiency. Compared with a general-purpose file system, the functionalities of the small file system on OSDs are much simpler. Traditional file system structures, including the popular VFS layer, are too heavily weighted in this scenario and introduce too much overhead because they have to satisfy a wide range of requirements and implement various features. OBFS metadata, specializing for the flat namespace and space management, remove unnecessary data structures and make them as compact as possible. This results in better space utilization and fewer I/Os for metadata.

5.4.1 Space Overhead

OBFS reserves one onode for each large block and one for every eight small blocks. In the default configuration, every 512-KB large block consumes one 512-byte onode. In small regions, every eight small blocks, totaling 32 KB, have one pre-allocated onode. The size ratio of onode to data ranges from 0.1% to 1.7%. If the workload is composed of 80% of large objects and 20% of small objects, only 0.4% of all used space will be occupied by onodes. Besides the onodes, OLT and region heads also have persistent copies on disk. The size of the OLT is proportional to the total number of objects on disk. Because each object has one 8-byte entry in the OLT, the space used by the OLT is negligible. Region heads have different structures for large and small regions. A large region head contains only the book-keeping information and a block bitmap. For a 256 MB region, 32 KB disk space is enough for its region head. However, OBFS reserves the first 512 KB to make the region head align with the large block boundary. In the default configuration, a 256 MB small region has a 8 KB onode bitmap, a 1 KB onode map, and a 64 KB log. OBFS reserves 64 KB for a small region head and 64 KB for its log. Therefore, around 0.05% to 0.2% of disk space is reserved for region heads. In summary, the disk space occupied by OBFS metadata typically ranges from 0.6% to 2% of all the space in normal situations.

The in-core file system structures are also very compact, as we described in Section 5.3.3 and Section 5.3.2. By using large blocks, the bitmaps for those large blocks require much less memory resources. For example, a traditional file system with a 4-KB block size on a 256 GB disk needs at least 8 MB block bitmaps and several hundred KB inode bitmaps. A random workload with frequent write/delete operations may touch most regions of those bitmaps, which forces those bitmaps to stay in core and consumes large amount of memory resources. In OBFS, assuming 80% of disk space is occupied by large blocks, the total space consumed by bitmaps is a little more than 1.6 MB. Such a small footprint makes them much easier to fit into memory and incurs much less disk traffic. Considering that disk requests for bitmaps are typical in the critical paths of object operations, reducing those requests can significantly improve the overall throughput. Another improvement of OBFS structures is the OLT, which completely replaces the role of dentry cache in traditional Unix file systems. Each entry in the OLT is only 8 bytes, sixteen times smaller than the typical size of a dentry. Several megabytes are enough to hold the entire OLT into main memory, which results in fast name lookups. In general, the compact in-core data structures of OBFS save a lot of main memory for data cache, eliminate a large number of disk accesses, and shorten the critical paths of many object operations.

5.4.2 Run Time Overhead

Object operations need to first read their metadata to retrieve physical layout information and related attributes. Some operations, such as object creates/writes, have to modify their metadata as well. Those metadata references and modifications consume a significant amount of computational time and incur a lot of small disk I/Os. The OBFS design tries to minimize both costs.

Many general-purpose file systems, such as Linux Ext2, assume small to medium directory sizes. To find a file in a directory, they simply go through the directory linearly and lookup file entries one by one. Such a linear search scheme becomes excessively inefficient as directories become very large. In a directory with thousands of objects, an object lookup operation has to examine, on average, half of the directory entries to find out the matched one. Create operations have even worse performance because they have to compare all the existing entries to make sure the name is unique. Although it is possible to add an additional hash layer to manage large directories, it is quite inefficient to implement such a layer on top of current directory structures. Some of the latest file systems, such as XFS, use $B+$ tree or hash-based structures to manage the directory structure. Searching dentries in those file systems would be much easier. However, the size of directory entries in those file systems are still the same as that of Ext2. As a result, they can buffer fewer dentries than OBFS, given the same amount of memory, which imposes more pressure on main memory and causes more disk accesses for file name translations.

Besides its compacted size, the OLT data structure is also designed for efficient hash insertion and retrieval. Each hash line in the OLT has 128 bytes and contains 16 entries. Most lookups for a hash bucket can be done in one or two memory loads. On the other hand, lookups in the normal dentry cache have to load the entire dentry into CPU for comparison, which can easily trash a whole CPU cache line. Searching a hash bucket with 16 entries may require 16 or more memory loads. Although currently we are more interested in the disk bandwidth optimization, simple file system structures will find their place as OSDs become larger and more complicated. In the scenario of multiple clients accessing the same OSD, the system might be memory bound, making simple and efficient data structure highly desirable.

Metadata updates typically generate small non-contiguous disk requests. In a system dominated by synchronous file writes, those metadata updates have to be synchronous to guarantee file data persistence. As a result, normal data accesses are often disrupted by them and a significant amount of disk bandwidth is wasted in serving them. For example, a file create operation needs at least five writes, four of which are to metadata: one modifying the inode bitmap, one modifying the block bitmap, one updating the allocated inode, and the last one inserting a dentry into its parent directory. To guarantee the atomicity of this operation, there might be an additional log write. Such extra metadata writes would hurt small file performance badly. Journaling file systems batch those small I/Os into logs to delay their impacts to the foreground activities. However, those updates have to be written to their original positions eventually. If the system is heavily loaded, the extra writes to the log may further hurt the overall performance. That is why journaling file systems may become slower under heavy loads.

OBFS solves this problem by re-designing the metadata structures so that at most one synchronous metadata update is required for each object operation. Such a design relaxes the dependency of OBFS to the transaction log. For a synchronous object create/write, OBFS only needs to write its onode together with the data to guarantee its persistence. All other updates to OLT and region headers can be delayed and batched with other requests together to disk. This approach is further augmented by collocating onodes for large objects. As a result, a synchronous large object create/write requires only one disk access to sequentially write both its data and onode, assuming OLT and the corresponding region head are already loaded in memory. The large object read benefits from the addressable onode identifier, which can directly calculate its disk address without loading its onode. This greatly improves the large object read throughput. A synchronous small object create/write generates one additional write to its onode besides the data block writes. The read operation of a small object is similar to that in the traditional file system. It has to first load the onode to retrieve the object to block mapping and then read back requested data blocks.

In summary, the OBFS design minimizes the metadata operations associated with object data operations. Combined with the always-in-core name lookup service, OBFS manages to maintain the overall metadata overhead at a very low level.

5.5 Summary

As a storage manager on individual OSDs, OBFS is expected to see object-level workloads, which have quite different characteristics than those of the file-level and block-level workloads. Based on our object workload analysis under the Ceph system, we present a simple, highly efficient design for OBFS and discuss the rationale behind the design by comparing it with what the general-purpose file systems normally do.

OBFS employs variable-sized blocks to achieve better sequential layout for differently-sized objects. It uses the self-contained region structure to organize blocks of the same kind together, which is able to reduce free space fragmentation as the system ages. The file system metadata are also partitioned into regions and operated independently. This provides better failure isolation and reduces recovery overheads. To better manage a flat namespace, OBFS employs a compact hash-based structure that enables fast mapping between the global object name and its internal identifier. Finally, OBFS collocates large blocks and their metadata together to minimize the metadata I/Os associated with the object data operations. In summary, the OBFS design adapts to the special object workload under the Ceph system. Such a design is able to provide potentially high throughput, maintain reasonable disk utilization, and guarantee strong data reliability.

Chapter 6

OBFS Policy and Implementation Details

The OBFS design employs variable-sized blocks and partitions disk space into regions. It optimizes for a flat and homogeneous object namespace through a hash-based indexing structure and a small set of specially designed object metadata. The disk layout in OBFS uses the hints from the expected object workloads to reduce disk seeks between object operations and minimize the file system fragmentation in the long run. These design choices provide a good framework for supporting the expected object workloads. At the same time, they are quite different from the design of the general-purpose file systems, which means that they change various policies and implementation details including allocation, reliability, and the possible need for region cleaning. In this chapter, we first describe the allocation policy in Section 6.1. The file system reliability, failure scenarios, and recovery schemes are discussed in Section 6.2. We further study the region cleaning problem and present several techniques to mitigate it in Section 6.3. Finally, we conclude this chapter in Section 6.4.

6.1 Allocation Policy

The allocation policy of OBFS involves three allocation decisions: first, which type of blocks the object should use; second, which region to put the object in; and finally, which blocks in that region will store the object. For every new object, OBFS first decides what type of block(s) it should use. In the Ceph system, the decision as to the object type is assisted by the client component, which informs OSDs of the expected size of an object. Thus, OBFS can identify the object type even if it is only partially written. If the expected object size is equivalent to the system stripe unit, a large block is assigned to the object; otherwise, it uses small blocks. OBFS then selects an appropriate region from which the block(s) are to be allocated. If no qualified region is found, a free region will be initialized to the desired type. The final step is to allocate blocks and an onode inside that region.

6.1.1 Delayed Allocation

OBFS buffers new objects in the object cache and delays allocating space for the objects until they are flushed to the disk. The actual allocation happened when an object is issued to the disk-request queue. This scheme can potentially reduce object fragmentation by aggregating multiple small pieces of a partially written object together and allocating a large contiguous extent for it. It also increases the chance for clustering multiple unallocated objects during flush time. Since the object allocation happens right before the object being served by the disk, the allocator can approximate the disk head position and allocate objects to the nearby regions to reduce the seek overhead.

6.1.2 Region Allocation/Selection Policy

During the region allocation/selection step, OBFS searches the Region Head List (RHL) to find a candidate region that contains enough free blocks of a given type. Three factors are taken into consideration during the region allocation/selection: the distance between the last-accessed region and the target region, the region fragmentation, and the workload burstiness.

All region heads in RHL are organized into three sub-lists: a large block region list, a small block region list, and a free region list. All of the sub-lists are sorted by the region addresses. There are two in-core pointers associated with the first two sub-lists respectively: an outstanding large region pointer that indicates the region where the previous large object I/O happens, and an outstanding small region pointer that points out the last-accessed small block region. The outstanding pointers are updated whenever an I/O request hits a region that differs from the current outstanding regions.

The allocator uses the outstanding region pointers to approximate the disk head position at the allocation time. It tends to allocate objects to the nearby regions to minimize the seek overhead. Using a small object allocation as an example, the region allocation/selection process starts from the outstanding small block region. The allocator first evaluates the potential object fragmentation in that region through a pre-allocation process. The detailed discussion of the pre-allocation process can be found in Section 6.1.3. If the pre-allocation process shows that the region excessively fragments the new object, OBFS will scan the small region list sequentially, starting from the current outstanding region, to retrieve adjancent regions and repeat the evaluation process. If no qualified region is found, OBFS chooses the nearest free region and formats it into the requested type.

The region allocation policy used in the current implementation is conservative in allocating a free region: it prefers to use existing regions rather than allocating a new one. For example, between a free region with a distance of one and an existing region with a distance of five, OBFS tends to consume available blocks in the existing region, instead of allocating that free region. This conservative approach may lead to some performance degradation as the file system ages due to long seeks between regions. However, such degradation is typically amortized by consequent write requests into the same region and thus has less impact on the overall performance. More importantly, it is compensated for by preserving more free regions that are valuable in handling different types of workloads.

The last heuristics OBFS uses involve allocating/selecting regions based on the intensiveness of the write workload. The allocation routine keeps monitoring the request arrival rate to the disk-request queue. As the arrival rate becomes high and the disk queue builds up with many write requests pending, it is better to allocate a fresh new region, where the large number of write requests can be laid sequentially, rather than using an existing region with only a few free blocks scattered around.

6.1.3 Block Allocation Policy

Block allocation for a large object is straightforward because it only needs to find one free block in the target region, mark it as used, and store the object in it.

For objects that use small blocks, an extent-like allocation policy is employed. OBFS goes through the region block bitmaps trying to find a free extent that is large enough to hold the incoming object. In the scenario that multiple large extents are found, OBFS uses the worst-fit policy, breaking down the largest free extent to store the object and returning the unused portion to the free list. If such a free extent is not available, the largest extent 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 allocated.

6.1.3.1 Extent-summary Array and Large-extent List

Two in-core data structures, an extent-summary array and a large-extent list, are maintained for each region on the fly to facilitate the allocation process. The extent-summary array is designed to guide the extent search. It tracks the total number of extents in different-sized categories in a region. In our current implementation, an array contains 128 integers that correspond to extent size from 4 KB to 512 KB. The value of the Nth integer in the array represents the total number of N-block extents in the region. For example, a region that contains M 16-KB extents will set the values of the fourth bucket in its summary array to be M. The large-extent list records up to 128 large extents, sorted by their disk addresses. It helps the allocator to directly grab a large extents without first scanning the block bitmap.

6.1.3.2 Small Block Allocation

In searching for a given-sized extent, the allocator first looks up the extent summary array to see if there is a perfect match. This step is done by simply examining the value of the corresponding bucket in the array. A positive value of that bucket means the perfect match exists. It then scans the block bitmap to retrieve the address of the extent. If no perfect match is found, the allocator picks up the largest extent from the head of the large-extent list, breaks it down to desired size if it is larger than the requested size, and allocates it to the object. This process can be repeated until the requested space is fully allocated.

6.1.3.3 Pre-allocation Process

Besides the normal allocation process, OBFS also introduces the pre-allocation process, which is used to evaluate the potential fragmentation of an object if it is allocated into a region. The pre-allocation process has the similar procedure as the normal allocation except that it will not scan the bitmap to retrieve the extent details. Instead, it only looks up the extent-summary array to calculate the total number of fragments an object will have. Such information can be used to assist the region allocation to decide the final destination, as described in Section 6.1.2.

6.2 Reliability and Integrity

As a storage manager at the OSD level, OBFS provides strong guarantees of data reliability and system integrity. Although the OBFS design focuses on object throughput by minimizing metadata read/write per object operation, no compromise of data safety has happened. In the face of various failures, OBFS only trusts the onodes, the onode bitmaps, and the delete log, as described in Section 6.2.3. All other data structures can be rebuilt directly or indirectly from them. In addition, OBFS tags every onode and bitmap write with a generation number, which gives OBFS a hint about the write generation when those two structures do not agree with each other.

The simple metadata update scheme gives good performance without sacrificing data reliability and file system structure consistency. However, good performance does not come without cost. Compared with the journaling approach, such as the one used in Ext3, OBFS takes a relatively long time to recover from a system crash because it needs to scan all active onodes and onode bitmaps to rebuild the OLT and the block bitmaps. However, this process is not excessively expensive since the important data structures are partitioned into regions and located in the fixed locations among regions. Those metadata only account for a small fraction of the overall disk space, which can be easily loaded from disk and parsed in memory. On the other hand, a journaling file system may have to perform a full system scan due to some internal structure corruptions, such as bad recovery logs or corrupted directory structures. In such a case, OBFS shows much better recovery performance thanks to its simple metadata structures.

In the rest of this section, we first discuss the failure scenarios that are expected to be handled, followed by detailed descriptions of the object create/write and delete operations, and how OBFS guarantees data reliability during those operations. Then we will explain how the region structure provides a failure boundary to isolate random errors. Finally, we present the recovery scheme in the current design.

6.2.1 Failure Scenarios

OBFS considers two types of failures. One is the loss of volatile memory, such as power failures and unclean shutdowns. The other is the random error, such as software bugs and bit rot, both in memory and on disk. The first type of failure is easy to detect because the whole system is halted and the normal operation is interrupted. The latter one, on the other hand, may not be detected at the time that it happens. It will stay in the system and propagate its effects to other parts of the system until it causes serious damage to file system structures or corrupts user data. For example, a flipped bit in a block bitmap may cause an already-used block to be allocated again. If this duplicated block lives in an important metadata structure, such as an inode or a directory block, it may cause serious damages to file system structures. Such errors are random and usually have much worse impacts on the system. Although some techniques, such as mirroring and checksum, may protect the system from being compromised by a random error, they are not sufficient to defend against internal software bugs, which are inevitable in large systems.

The design of OBFS minimize the impact of random errors through a self-contained region structure. It introduces a boundary check for every I/O operation, which helps to quickly detect random errors and confine them inside the region in which they happen. In other words, the damage caused by a random error will not propagate beyond the region boundary. As a result, OBFS can preserve most of its objects in the face of various hardware and software failures.

6.2.2 Object Create/Write

As a low level storage device, an OSD should provide strong data reliability guarantees. Before committing a write, it is important for an OSD to make sure that all of the data as well as the necessary metadata are on persistent storage. Therefore, unlike other general-purpose file systems that assume asynchronous writes, OBFS treats every object write from clients as synchronous unless the asynchronous flag is explicitly set. The strong guarantee for the user data also implies a strong guarantee for file system structure consistency. If the structures used for retrieving data are damaged, there is no way to ensure data reliability even if the data is stored stably somewhere on the persistent storage. In this sense, the synchronous file write in some general-purpose file systems–*e.g.*, Ext2–is not good enough since the structural changes are delayed. Although a file system scan can retrieve the data of the lost objects, their names may be permanently lost, which makes them essentially useless in a distributed environment. OBFS solves this problem by storing the global object identifier in its onode. Together with a valid bit in the onode, OBFS can always identify if an onode is valid by scanning the fixed location onode table and retrieving the object data, no matter how badly the system structures have been damaged.

Besides the synchronous create/write workload, OSDs may also see a great amount of asynchronous create/write traffic among themselves. Those operations are mostly for replication or management purposes. For example, objects are replicated to protect against a single point of failure. The primary copy should be written synchronously to an OSD, while the secondary and the tertiary copies may be asynchronously written before committing. Other cluster management work, such as the workload balance between clusters, uses asynchronous writes to improve the overall throughput and minimize the disruption of foreground activities. For such workloads, OBFS forces the update ordering between the object data and their onodes. An object onode is always pinned in memory until its data has reached disk. Only after all of its data has been flushed back to disk is it unpinned and ready for I/Os. This simple ordering can avoid the situation in which the object onode has reached the disk while its data is lost. Since the onode size in OBFS is exactly the disk sector size, onode I/Os are independent of each other. No Soft Update style techniques are needed, which further simplifies the OBFS design.

In additional to the onode update, an object create/write operation may also need to update the block/onode bitmaps and the OLT. OBFS chooses to lazily flush them back to disk. There are several advantages by doing so: first, the synchronous create/write performance can be significantly improved since the bitmap or OLT updates are decoupled from the critical paths. Second, multiple updates to the bitmaps and the OLT can be packed together, which can greatly reduce small metadata writes. However, asynchronously updating the bitmaps and the OLT requires a relatively complicated check scheme to restore the system consistency after a crash. We will discuss it in detail in Section 6.2.6.

6.2.3 Object Delete

In OBFS, object deletion is implemented as an asynchronous operation. Every delete operation updates the in-core data structures by adding an entry into the region delete log, marking its onode invalid, and clearing the used bit in the block/onode bitmaps. Dirty logs are flushed to the disk every second or when there are more than 16 pending entries. The modifications to the onode will be piggybacked with the adjacent onode operations to reduce disk I/Os. As soon as the onode reaches the disk, the delete operation is considered complete and a completion entry is added to the log.

Because the log updates are asynchronously written to the disk, it may happen that the deleted onode is reused while the completion log has not reached the disk. If the system crashes right at this moment, some objects could be identified as deleted objects and cleared from the system during the recovery process. To avoid this, every log write and onode write are tagged with a generation number, unique in a region. An object is marked as deleted only when the generation number of the delete log entry is larger than the one with its onode. The asynchronous delete operations can achieve batched delete performance and minimize their impacts on the foreground workload. However, there is a short window in which those deletes are committed while the on-disk data structures are not changed. System crashes during that window may void the deletes and leave unreferenced objects in OSDs. If users are more concerned about the data security or the wasted disk space used by those orphan objects, they can force synchronous object delete, which guarantees that the deleted objects are removed from the namespace before return. Alternatively, these orphaned objects can be found by scanning the metadata in a high-level filesystem consistency check.

6.2.4 Boundary Check

As demonstrated in section 4.2, the object workload has two important aspects: object sizes are always smaller than the system stripe unit size, and the objects in an OSD are largely independent of one another. The OBFS design makes use of these characteristics to partition the global storage space into relatively small independent regions. Each region has its own self-contained data structures and a clear physical boundary. Since an object can only live in a single region, all I/Os to an object must fall inside that region. OBFS enforces the boundary check for every disk I/O during the logical address to physical address mapping. For instance, OBFS will check if an object I/O will cross the region boundary and whether it will touch the metadata areas. The same checks apply to the metadata operations. Those checks provide a failure boundary for each region. An operation can only read/modify its target region. Any I/O outside of that region will be treated as an error and denied immediately.

In addition to the corrupted region metadata, a damaged OLT may also redirect an object request into a wrong region. For example, a bad OLT may mistakenly map an object, say A, into object B's location. If a user updates object A, it could trash object B's data. However, this wrong mapping is not able to corrupt the region metadata or further damage the whole system structures because the metadata of object B is still consistent. Updates to object A have to follow the extent information inside the onode of object B, which prevents them from writing to an arbitrary place on the disk.

In a traditional file system, the physical space is a global resource. Every object obtains the space from the same resource pool. Given an error in a file system, there is no limit to how far it can propagate. A good example is that a corrupted block address in a file inode may direct the disk I/O to the wrong place, which may happen to be owned by the global block bitmaps. Any further operations may get incorrect allocation information from the bitmaps and further spread the errors out. Therefore, a full system check is mandatory if some structural errors are detected, such as duplicated blocks and corrupted inodes, since the file system has no idea how long the error has existed and how badly it has affected the system. The journaling technique does not help here because it is a method to protect the system from interrupted transactions rather than from random failures. With a relatively larger OSD, such a full check for a traditional file system may last for hours or even longer.

OBFS shows its unparalleled advantage in this area. Any random error that occurs in a region is confined in that region. System check only applies to the region where the error has been detected. Only when the OLT or the superblock are damaged does OBFS need to scan the full system. Since the OLT and the superblock are much smaller than the rest of the file system, the chance of corrupted OLT due to random errors is fairly small. We can further reduce the corruption of important metadata by protecting them with erasure coding or replication schemes. This provides bounded recovery time no matter how large the OSD is. Moreover, such a check can be done online without taking the whole OSD offline. The errors can even be masked from the user. In the worst case, users only notice certain delays for some object I/Os.

6.2.5 File System Consistency

OBFS consistency comprises of two different components: region internal consistency and OLT consistency. Region internal consistency means that the region metadata are in agreement with each other. OLT consistency means that the OLT agrees with all the metadata in every region. The system recovery process involves both the individual region recovery and the global OLT re-construction.

6.2.5.1 Region Metadata Update Process and Internal Consistency

As we mentioned in Section 6.2.2 and Section 6.2.3, OBFS requires only one metadata update before committing a create/write or a delete opertion. Whenever a new onode is written to a region, the on-disk structure of that region becomes temporarily inconsistent. Such inconsistency can be eliminated only after all dirty bitmaps of that region are flushed back to disk. Since OBFS flushes the dirty bitmaps periodically, there is a short window of inconsistency. If the system crashes during this window, OBFS has to scan all the onodes of that region to rebuild the region bitmaps. To faciliate the recovery process, OBFS maintains a dirty flag, described in Section 5.3.2, for each region. It is set on before writting a new onode to the disk and cleared after all dirty bitmaps are flushed back to the disk. During a recovery process, this flag indicates whether a region is in the consistent state.

6.2.5.2 OLT Update Process and Consistency

The OLT consistency is tracked by the OLT generation array, as described in Section 5.3.3, which contains the outstanding generation numbers from all the regions. It is always updated together with the OLT. OBFS can check whether the on-disk OLT is consistent with an on-disk region by comparing the generation number in both the region head and the generation-number array.

The OLT update process starts by freezing the in-core OLT and the generation-number array. All the new updates to them are temporarily buffered in a new cache using the copy-on-write technique. OBFS then sets the update flag of the on-disk OLT to indicate that the OLT update is in progress. Dirty blocks in the OLT are then issued to disk, followed by the generation-number array updates. Finally, the update flag is cleared. After the completion of all data flushing, the newly-updated OLT entries in the temporary cache are merged back with the OLT and the generation-number array.

The generation-number array helps OBFS to reduce the number of regions that needs to be scanned during the OLT recovery process. Since the generation counter in a consistent region always reflects the generation number of the last update to the region metadata, OBFS can compare it with the corresponding generation number in the OLT generation array to see if the OLT is up to date. A mismatch of the values indicates that there are new updates to the region metadata after the current OLT was written. Thus, the onodes in that region need to be scanned to rebuild the OLT. Otherwise, the OLT is consistent with that region and the recovery process can simply skip it.

6.2.6 Recovery Schemes

In Section 6.2.1, we classified the file system failures into two categories: interruption of normal operations and random errors. We introduce two recovery processes, the crash recovery process and corruption recovery process to handle the two types of failures.

6.2.6.1 Crash Recovery

The crash recovery aims to bring the system back to a consistent state after a system crash. It assumes that the inconsistent state is caused by losing volatile memory rather than random corruptions. After the recovery, no committed requests should be lost.

The crash recovery process scans and recovers individual regions independently. It starts by scanning all the region heads sequentially. If the corruption flag is set, it will switch to the corruption recovery, as specified in Section 6.2.6.2. Assuming that the corruption flag is off, it continues to check the dirty flag. An off dirty flag means that the region is internally consistent and there is no need to perform the crash recovery for that region. Otherwise, it will load the entire region metadata to rebuild the region structures. The final step is to compare the region generation counter with the corresponding generation number in the OLT generation array. If they do not match, the recovery process will initiate the OLT rebuild process. In the rest of the sub-section, we will focus on the individual region recovery and the OLT rebuild process.

The individual region recovery is the most important component in the crash recovery process. It restores the internal consistency of a region by scanning all the onodes and the delete log to rebuild the onode and block bitmaps. During the onode scanning, it checks the valid flag of every onode to see if the onode is active. For a valid onode, it then compares the generation number of the onode with the one in the onode bitmap. If the onode's generation number is larger, it will set the corresponding bit in the onode bitmap and updates the block bitmap using the extent information inside the onode. After scanning all the onodes, the recovery process will parse the delete log and apply changes to the bitmaps. As we mentioned in Section 6.2.3, the delete log replay process compares the generation number of the log with the one in the related onode to decide whether an onode is valid. It will use this information to clear corresponding bits in the onode and block bitmaps. The maximum generation number found during the recovery process will be used to update the region generation counter. The last step is to flush all those bitmaps and the region heads back to disk.

The OLT rebuild process restores the consistency between a valid region and the OLT. It also needs to scan all the onodes in that region. For each valid onode, it retrieves the corresponding OLT entry using the object identifier stored in the onode. If the entry is out-dated or invalid, it will remove the old entry and add a new entry to reflect the changes. This process relies on the internal consistency of the region. It should alwasy check the region internal consistency before running this process.

6.2.6.2 Corruption Recovery

Corruption recovery handles various failures caused by random errors. Unlike crash recovery, it does not trust any piece of metadata and has to perform more comprehensive checks for them. After recovery, the system should be brought back to the consistent state. However, it may lose some of the objects if the error is serious enough.

The corruption recovery is performed independly on individual regions. Similar to the crash recovery, it first scans all the onodes. For each valid onode, it performs a sanity check to make sure the flags and values in the onode are in reasonable/predefined ranges. It then updates the onode bitmaps and the block bitmaps using the extent information in that onode. The recovery routine applies boundary checks during this step to find invalid addresses. An additional scan is needed if any duplicated block is detected. A duplicated block refers to the block that is referenced by two or more objects in a region. To resolve the conflicts caused by the duplicated block, OBFS simply replicates the block and assigns them to both objects that reference the duplicated block. A corruption flag is set in both objects to inform the user that both objects' data are not trustable.

After scanning the onodes, the recovery process checks the sanity of the delete log. If the delete log is valid, it will be replayed to update the onode and block bitmaps. At this stage, all the onodes and the bitmaps should be consistent. The following step is to rebuild the region head, which mainly involves in recalculating the statistical information such as the free block count. All metadata are then flushed to the disk and the corruption flag is cleared. Finally, the recovery process initiates the OLT rebuild process, which is identical to the one described in the crash recovery process.

6.2.6.3 Recovery Overheads

Both the crash recovery and the corruption recovery require the scanning of a number of regions. Compared with the recovery processes in a journaling file system, the crash recovery is expected to be slower while the corruption recovery should be much faster.

The crash recovery in OBFS needs to check those inconsistent regions with the dirty flag on. It scans onodes in those regions and rebuilds the bitmaps and the OLT. However, this process is not excessively expensive. First, OBFS uses the dirty flag and the generation number to identify inconsistent regions and minimize the number of regions to be scanned. Second, the simple region data structures are guaranteed consistent after dirty bitmaps are flushed back to disk. Therefore, the total number of inconsistent regions is bounded by the active regions between two consecutive metadata flushings. Since OBFS tends to create objects in a few regions during a short period, the number of such active regions should be small under normal workloads. Finally, the region metadata is very compact and laid out contiguously on disk. One large sequential I/O is enough to load all region metadata, including bitmaps and onodes, into the disk. The recovery routine can parse the onodes and rebuild the bitmaps fully in core. For example, the total space occupied by onodes and bitmaps is around 4 MB in a small block region. A large sequential disk I/O can load it into memory within 100 ms.

The corruption recovery has similiar overheads as the crash recovery. It shows much better performance when compared to general purpose file systems. If any metadata corruption is identified inside an general-purpose file system, the entire file system has to be taken offline and all the metadata have to be scanned. OBFS, on the other hand, provides a failure boundary through the self-contained region structure. A corruption in a region requires only that region to be fixed. Since we can decide the region size, the recovery time for a region is bounded. Therefore, for a single random corruption, the corruption recovery time is bounded no matter how large the file system might be. While in a general-purpose file system, this time is proportional to the size of the entire file system.

The failure boundary enables OBFS to perform online region repairs, which might be important for a continuously running system. The design of breaking down a large system into small regions favors a background-checking process. OBFS can periodically check those long-time-untouched regions one by one in the background without severely affecting the foreground activities. Overall, the design of OBFS provides a simple but robust system for the OSDs. It has some key features, such as the failure isolation and fast online scan, that can be extremely useful for the high-level distributed storage system design.

6.3 Region Clean

Because OBFS uses regions to organize different types of blocks, one potential problem is that it may run out of free regions and free space in regions of the desired type. Unlike LFS [61], which must clean segments on a regular basis, OBFS never needs cleaning unless the ratio between large and small objects changes significantly over time on an OSD that is nearly full. We expect that such a scenario will rarely happen in practice. The cleaning process coalesces data in the underutilized regions and creates free regions that can be used for regions of desired types. One side effect of region cleaning is that some small block regions may be de-fragmented during the cleaning process. Fragmented objects can be laid contiguously into a new region during the data relocation.

The cleaning process for large block regions is very simple since the large object is of fixed size and its metadata are collocated together with its data block. The cleaning process starts by selecting a least-utilized region and reads out all of its objects into memory. It then searches the rest of the large block regions to dump all those objects. After one object has been relocated, it needs to update the OLT to reflect the change.

The small region clean may result in more overhead than that of the large region clean process because small objects may be heavily fragmented. OBFS adopts a different approach to clean the small block region. Instead of following the onodes to read in individual objects, OBFS scans the onode and block bitmaps to identify all valid onodes and blocks. It then merges those onodes and blocks into large chunks and issues raw disk I/Os to read them all together into the memory. Finally, it parses those data blocks in memory to build temporary objects that associates the in-core inodes with the in-core extents. Those in-core objects are flushed to their new destination using the same procedure as the one in the large block region clean process. This method eliminates the reading dependence of object data blocks to the onodes and it greatly improves the disk bandwidth utilization by merging small extents into large ones.

6.3.1 Non-fixed Object Location and Region Packing

In traditional file systems, file creation is expensive because it needs at least four writes to update all modified metadata. As a result, most file updates happen in-place rather than through relocating the whole file into a new place. OBFS, on the other hand, makes it easy to relocate an object because of its simple metadata design: no extra writes for a large object creation and only one extra write for a small object creation. Such relocation on update is referred to as *floating write*. An object can be freely relocated into another region if it is fully loaded into memory. In theory, this approach can achieve a WAFL-like performance because it greatly improves the locality of object I/Os. However, it has several limitations. The object to be relocated must be completely loaded into memory with its onode. If a part of an object is missing in memory, the cost of reading the object and migrating it to another region may be more expensive than simply updating in place. Second, this approach leaves multiple valid copies of an object on disk. If the system crashes, it cannot tell which one is the latest copy. OBFS solves this problem by storing the generation number in the OLT. To relocate an object, OBFS will compare the old copy's generation number with the new one obtained from the new location. The largest number will be incremented by one and used as the new object generation number. The region generation number will be advanced to the new generation number. By doing so, OBFS can recognize the latest copy of an object from multiple out-dated copies, even after a system crash. This results in a larger OLT and a slightly more complicated recovery process. The last limitation is that *floating write* does not help with asynchronous writes. A large object cache can effectively exploit the locality of existing objects and coalesce them into large sequential I/Os. In such a scenario, relocating a small object may even increase the cost because it introduces one more write to the new onode location.

In our current implementation, OBFS employs the *floating write* technique for two purposes. The first is performance improvement. OBFS applies this technique only when the following conditions are satisfied. First, the operation must be a synchronous write. Second, the request must write the whole object and all necessary data to memory. Finally, the object onode must be in memory if it is a small object. Given all three conditions, OBFS will treat the object as a new object and select the optimal location for it. In case if the new location has no advantage over the old one, OBFS will use the old one. The purpose of using the *floating write* is to consolidate under-utilized regions. It is preferable to relocate objects from a lightly-utilized region to a highly-utilized region when the ratio of free regions becomes low. By doing so, it is possible to reduce the overhead of the region cleaning. OBFS will trigger this migration process once the number of free regions drops below a certain threshold.

6.4 Summary

This chapter provides the essential policy and implementation details of OBFS. We focus on the allocation policy, the reliability and recovery issues, and the region cleaning policy. They are demonstrated through a series of detailed presentations on the related data structures, discussions on execution procedures, and comparisons with other approaches in existing general-purpose file systems.

The allocation policy in OBFS is composed of both the region selection/allocation policy and the block allocation policy. During an object allocation, OBFS first decides which type of blocks the object should use, assisted by the hints from the high-level file systems. It then selects the region for the object allocation. The region allocation policy used in this step takes into account the region distance, free space fragmentation, and workload burstiness. It tries to balance between the optimal layout and the seek distance to provide a better region selection during run-time. After the region selection/allocation, OBFS finally allocates blocks in that region for the object. With the aids from two in-core data structures, OBFS achieves extent-like allocations with relatively low overhead.

As a low-level storage manager on individual OSDs, OBFS is designed to provide strong reliability and recoverability. OBFS guarantees the data safety after it commits the object requests. A piece of redundant information is stored inside each onode, which can be used to restore the system back to consistent state even after the system is seriously damaged. The self-contained region structure helps OBFS to enforce the failure boundary. Combined with the boundary check scheme, OBFS can always limit a random data/metadata corruption inside a region. Based on those data structures, we further discuss the failure scenarios that OBFS is expected to handle and present two recovery schemes: the crash recovery that restores a system back to consistent state after a crash, and the corruption recovery that fixes metadata corruptions. Compared with those recovery processes in a journaling file system, the crash recovery in OBFS is expected to be slower, while the corruption recovery should be much faster.

Finally, we discuss the region clean precedures and the scenarios that may trigger those precedures. A region packing technique is also provided, which will potentially reduce the chance of region cleaning.

Chapter 7

OBFS Design Evaluation

OBFS introduces several unique features such as variable-sized blocks, self-contained region structures, and optimized region/block allocation policies. The design goal is to minimize the disk fragmentation and make best use of disk bandwidth in the long run. Several key parameters in the OBFS design have significant impacts on overall system performance. In this chapter, a series of experiments demonstrate the effectiveness of the design decisions and evaluate the best choices of some key parameters. In the rest of the chapter, Section 7.1 describes the experiment setup; Section 7.2 introduces several synthetic object workloads and benchmarks; Section 7.2.1 explains the aging technique and describes the synthetic aging workloads used in our experiments; Section 7.3 presents how OBFS's throughput varies as the region size and the block size change; The long term aging effects and the file system fragmentation are studied in Section 7.4

Table 7.1: Specifications of the Maxtor D740X-6L disk used in the experiments

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

7.1 Experiment Setup

All of the experiments are executed on a PC with a 1 GHz Pentium III CPU and 512 MB of RAM, running Red Hat Linux, kernel version 2.6.9. To examine the performance of the file systems with minimal impact from other operating system activities, we dedicate an 80 GB Maxtor D740X-6L disk (see Table 7.1) to the experiments. This disk is divided into multiple 8 GB partitions. The first partition is used to install file systems and run experiments. The rest of the partitions are used to backup aged file system images. We use aged file systems to more accurately measure the long-term performance of the file systems. For each experiment running on the aged system, we copy the aged file system image to the first partition of the disk, unmount the disk to clean the buffer cache, then mount the aged partition to run the benchmarks. We repeat these steps three times and take the average of the performance numbers obtained.

7.2 Synthetic Workloads Used in the Experiments

Several synthetic object workloads are used to measure the OBFS performance. If the workloads have write requests, it can be switched between synchronous mode and asyn-

Table 7.2: Synthetic workloads

	TwoValue	SynWrite	SynRead	ObjectBench
r/w	Write	Write	Read	Read & Write
properties	†Two request sizes. †*ratio* parameter adjust their ratio.	†80% large object requests. †20% small object requests. †Small object requests uniformly distributed between 4 KB and 512 KB.	†60% random reads. †40% sequential reads.	†40% read requests. †36% write requests. †24% rewrite requests. †80% writes to large objects. †20% writes to small objects. †Small object writes uniformly distributed between 4 KB and 512 KB.

chronous mode. If not specified, all the requests generated by synthetic workloads reference whole objects, e.g., no partial object I/Os are used in the synthetic workloads. In this performance evaluation, we adopt a closed loop scheme. A new request will be generated only after completion of the previous one. Such workloads evaluate the maximum throughput for an individual thread.

The simplest object workload, referred to as the *TwoValue* workload, employs only two request sizes. One is the same as the large object size, which is 512 KB in the default setting. The other is 32 KB. The ratio of these two requests is adjusted by the *ratio* parameter, which indicates the fraction of all requests that use large object size. For example, a *TwoValue* workload with a 0.8 *ratio* means 80% of all requests have a fixed size of 512 KB. The workload generator randomly draws a number between zero and one, and compares it with the *ratio* specified in the workload. If smaller, a request to a large object will be generated. Otherwise, a small object request will be issued. With a *ratio* value of one or zero, the workload is reduced to the fixed-size micro benchmark. The *TwoValue* workload is used for detailed study on the performance impact of the block and region sizes chosen in OBFS.

The *SynWrite* workload simulates more realistic environments. It mixes large object requests and small object requests with a fixed ratio of 4 to 1. The size of small object requests are uniformly distributed between 4 KB and 512 KB.

The *SynRead* workload generates both sequential reads and random reads. Sequential reads, in this context, mean that the read requests issued to a set of objects follow exactly the same order as they are written. For example, assuming object B is written immediately after object A, a read request to object B is called a sequential read if the previous read request is to object A. The term sequential in the context does not necessarily mean that those objects are laid sequentially on disk. For example, on a fragmented disk, two objects written together may be allocated far apart due to the lack of sufficient contiguous space. A random read, on the other hand, arbitrarily selects an on-disk object without considering the write.

The last workload, called *ObjectBench*, simulates real scientific workloads. The key parameters of this workload are derived from the LLNL file system workloads, as described in Chapter 4. It is composed of 40% read requests, 36% write requests, and 24% rewrite requests. Among them, 80% of all requests reference large objects and 20% reference small objects. Like the *SynWrite* workload, the size of the small objects is uniformly distributed between 4 KB and 512 KB. 60% of all write and rewrite requests are synchronous and the rest are asynchronous.

7.2.1 Aging Workload

Most file systems show very good performances on fresh disks. However, aged file system performance is more important since it is more representative of real working environments. A file system has to be evaluated under both fresh and aged circumstances to comprehensively demonstrate its advantages.

File systems age slowly under normal conditions. The long aging process can hardly satisfy the time-constrained evaluation requirements. Special aging techniques are introduced to achieve fast aging progress while still maintaining good similarity between the real image and the artificially generated one.

7.2.1.1 Existing Aging Workload Generation Techniques

Smith, *et al.* [72] used file system snapshots and traces to approximate the possible activities in file systems. They examined the difference between two consecutive snapshots and generated a series of operations to transform an old snapshot into a new one. To simulate the temporal I/Os between snapshots, they introduced a lot of random writes/deletes, which did not affect the states of each snapshot but still extensively exercised allocation policies of the underlying file systems. Combining both the transform operations and the random I/Os, they eventually created a file system aging workload. Applying the aging workload to a fresh file system, they could easily generate an aged testing platform in a relatively short time.

7.2.1.2 Multiple-stream Object Aging Workload

We followed Smith's approach and developed our own aging workload, called the multi-stream aging workload, to facilitate the evaluation of our OBFS. There are two or more independent request streams in the workload. One of them simulates the transformation requests. The rest create a number of temporal requests. For each stream, there is a workload generator thread associated with it. It controls the request generation based on a set of workload parameters, which are specified before the aging process starts. Only the create and delete requests are generated during the aging process so that we can fully exercise the file system allocation policy in relatively short periods. All those streams use the closed-loop model: a new request is generated only after the previous one is completed.

7.2.1.3 Aging Parameters

Several key parameters control the the workload stream characteristics, including the large/small object ratio, create/delete ratio, the mean inter-arrival time, and the mean object size. All the ratio numbers are between zero and one. In current implementation, we employ the uniform distribution to generate the object size and the Poisson distribution to generate the request inter-arrival time. Since our workload generator use the closed-loop model, the inter-arrival time means the time between the completion of outstanding request and the issue of the new request.

The object identifier space is partitioned into streams. Each stream owns a set of object identifiers exclusively so that operations from one stream will not affect the activities of other streams. There is a hash-based data structure for each stream that tracks the objects it creates. The workload generator uses this structure to quickly identify an existing object so that it can generate a delete request without querying the underlying file system. It also helps to avoid issuing multiple create requests for the same object.

7.2.1.4 Aging Workload Generation Process

To generate a new request, the workload generator first decides the request type. It generates a random number and compares it with the create/delete ratio. If the random number is smaller than the ratio value, an object create request will be issued. Otherwise an object delete request will be issued. For a create request, it further decides the object type, large or small, by generating another random number and comparing it with the large/small object ratio. After obtaining the object type, the generator allocates an object identifier from its own id space and creates the object size, as specified in Section 7.2.1.3. Finally, it generates an inter-arrival time and issue the request after the time expires. For a delete request, it generates a random number and uses it to index the object table to find an existing object. A delete request to that object will be issued after the inter-arrival time expires.

A challenging task is to age the disk to a desired utilization with arbitrarily large number of requests. It requires the workload generator to keep track of the target disk usage. When the usage is far below the desired one, it tends to issue more write requests. As the usage exceeds the target one, it is more likely to issue delete requests. Such controls are implemented by the variable create/delete ratio for different streams. The create/delete ratio for the temporal request stream is fixed at 0.5 so that total create and delete request should be equal in the long run. While the transformation stream maintains a ratio table that contains variable create/delete ratios for different disk utilizations. The generator uses the actual disk usage to index the table in order to obtain the current create/delete ratio. The ratio table is pre-computed. When the actual usage is lower than the desired usage, the ratio is larger than 0.5, which means the possibility of generating a create request becomes larger. The bigger the usage difference, the larger the ratio. Similiarly, when the actual usage is higher than the desired usage, the ratio is smaller than 0.5. The bigger the usage difference, the smaller the ratio.

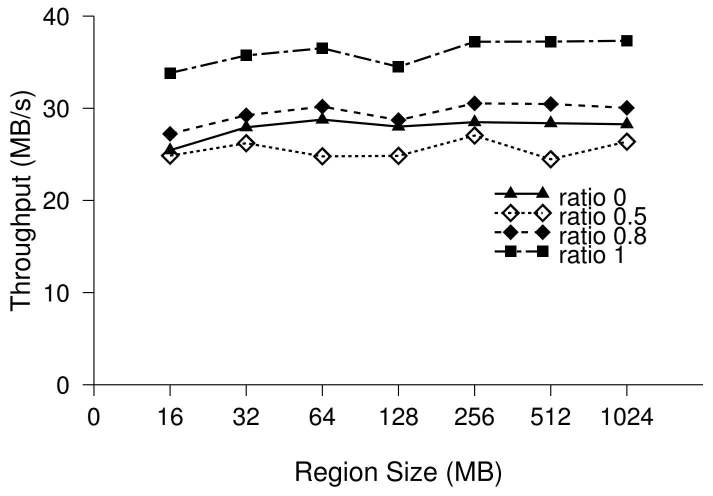
Such an aging scheme simulates more realistic environments: workload characteristics, e.g. create/delete ratio, fluctuates significantly from time to time due to random number generation, which causes the disk usage to vary a lot in a short period of time. However, the overall disk usage will eventually meet with our expection since the pre-defined table dynamically adjusts the create/delete ratio. Using this approach, we can easily age a disk to a specific utilization. The create/delete ratio for the temporal request stream is fixed at 50%, which means that none of the temporal requests will be persistent on disk in the long run. The temporal I/O streams are tuned to be more intense than the transformation stream in order to speed up the aging process.

In our current setup, we employ two streams, one for transformation requests and the other for temporal requests. A total of 100,000 requests are issued for the transformation purpose and more than 200,000 requests are issued by the temporal I/O stream. The key parameters of the transformation stream are exactly the same as the SynWrite workload. The characteristics of the temporal I/O stream are chosen to be fully random with only one restriction on the maximum object size. Although it is hard to evaluate the effectiveness of our aging method due to the lack of a existing real system, we do find that our approach successfully exercises the underlying file systems, as demonstrated in section 7.4.

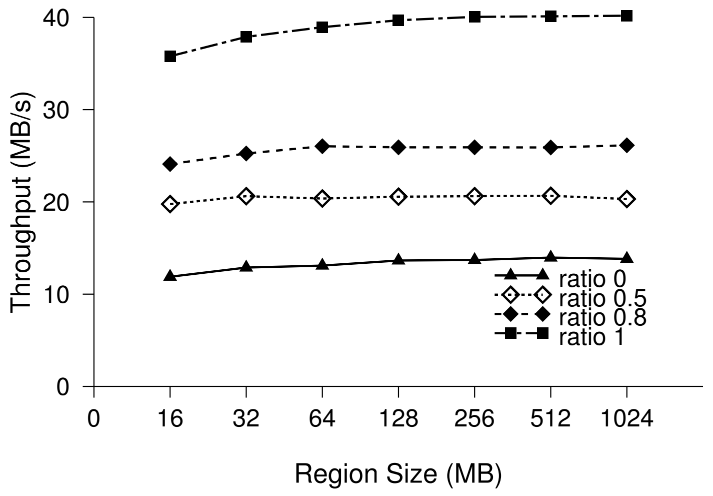
7.3 Region Size and Block Size

One important parameter in traditional file systems is the block size. A file system block is the smallest allocation unit. File data smaller than the block size consumes a full block on disk. The space wasted due to the block allocation is referred to as the internal fragmentation. Choosing a smaller block size will reduce internal fragmentation. However, a smaller block size increases file system metadata sizes and associated management overheads. As a file system ages, constant file allocation and deallocation will cause free space to be fragmented, which often lead to non-sequentiality of file allocations and worse performance in the long run. This effect is exaggerated by the small block size since it reduces the minimal contiguous disk space. Therefore, choosing a balanced block size is crucial in designing a file system. OBFS further complicates this process because it introduces the region concept. The disk space is partitioned into fixed size regions. Regions may have different block sizes, but all the blocks inside a region are of the same size. In this section, we focus on studying the performance impacts of the region size and the block size through a series of micro benchmarks. All experiments were run on an empty disk partition using the *TwoValue* workloads. Each experiment was repeated three times and the mean value of the three results is reported.

Figure 7.1 shows the overall system throughput under different region sizes. We ran our experiments using the *TwoValue* workloads. The *ratio* parameter indicates the fraction of



(a) Asynchronous Write



(b) Synchronous Write

Figure 7.1: OBFS performance as the region size varies. *TwoValue* micro benchmarks are used in the experiments. The *ratio* parameter indicates the fraction of the large object requests.

the large object requests among all requests. We vary the region size from 16 MB to 1024 MB and evaluate them using *TwoValue* workloads with *ratio* of 0, 0.5, 0.8, and 1, respectively. The block sizes in the experiments are set to 512 KB for large blocks and 4 KB for small blocks. Experiments in Figure 7.1(a) employ asynchronous writes, while those in Figure 7.1(b) use synchronous writes. From the figures, we can see the region size has a slight impact on the overall performance. On average, write throughput improves as the region size increases. However, the improvements can barely be observed: there are only two to four MB/s difference between the largest region size and the smallest region size, which account for 5% to 10% of the overall performance. We can see from both figures that overall throughput is almost unchanged when the region size grows beyond 256 MB. This indicates that any region size larger than 256 MB would be a good choice for OBFS.

OBFS shows the best performance under the large object workloads, which maintains 37 to 40 MB/s under all region sizes. It demonstrates consistent throughput whether synchronous writes or asynchronous writes are used. This result is of no surprise since we collocate large blocks with their metadata. A pure large object workload will result in a sequential disk write stream. The memory buffering during asynchronous writes has no chance to further improve the already optimal request stream. In fact, we observe a small performance degradation using asynchronous writes. This is due to the computational overhead of examining the disk queue and managing the cache.

The performance of small object requests, on the other hand, suffers under synchronous writes. In Figure 7.1(b), a small object stream with request size of 32 KB shows average throughput from 12 to 14 MB/s, as shown by the line with *ratio* of 0. With only 20%

of small object requests, the average throughput drops significantly from 38 MB/s to 25 MB/s. The main reason for the performance degradation is that the small objects employ the traditional layout scheme, which separates the object data from its metadata. OBFS writes both the data and metadata before committing the request. This causes disk seeks and decreases the disk bandwidth utilization. Asynchronous write helps a lot in such situations, as depicted in Figure 7.1(a). All of the small request workload and mixed workloads present good performance around 25 to 31 MB/s.

It is interesting to note that the mixed workload with a *ratio* value of 0.5 shows the worst performance under asynchronous writes. After close examination of the traces, we find that two factors are contributed to this result: the ratio between the object metadata and data, and the frequency of region switching. With the aid of write buffering, most object requests can be packed into large sequential disk requests. Thus, the actual disk bandwidth utilization is dominated by the metadata/data ratio. The larger the ratio, the lower the utilization. Another factor, the frequent region switching, is observed only in mixed workloads. In those workloads, OBFS has to switch between large block regions and small block regions to enforce the block allocation rule. This causes frequent long seeks. The small object workload has the worst metadata/data ratio. However, it has no region switch overhead, which accounts for better throughput compared to the mixed workload with the *ratio* of 0.5. For the workload with the *ratio* of 0.8, the better disk bandwidth utilization dominates over the region switch overhead. Therefore, OBFS shows better throughput in such workload than in the small object workload.

Another important parameter, the large block size, is evaluated through a series of experiments, as demonstrated in Figure 7.2. We vary the large block size from 32 KB to

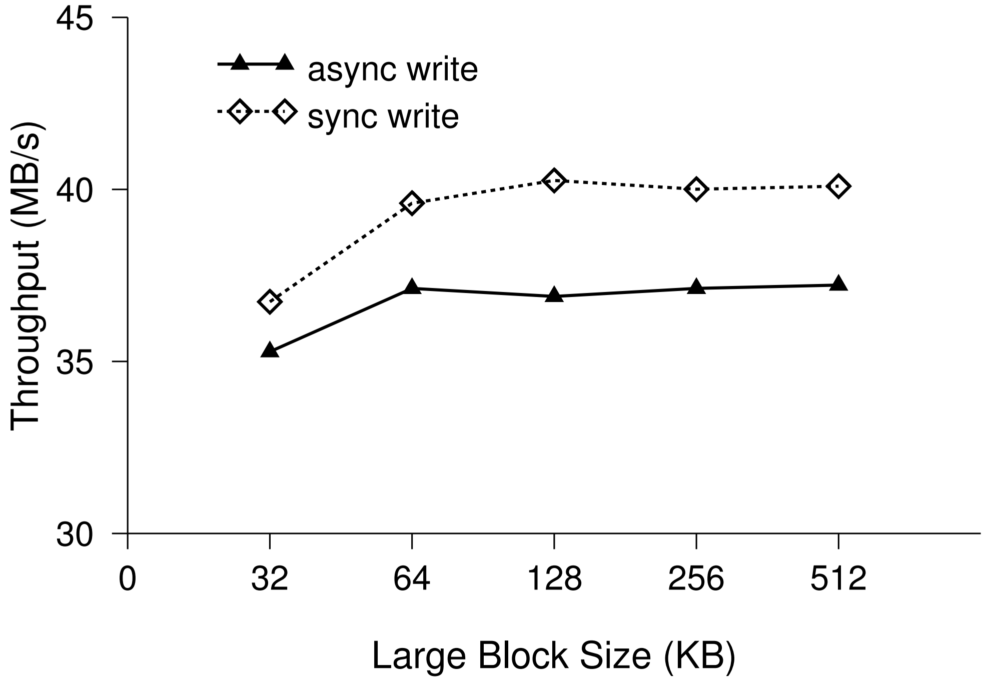


Figure 7.2: OBFS performance as the large block size varies.

512 KB and fix the region size 256 MB. All experiments write 2 GB of data, in the form of large objects. As we can see from the figure, a small block size decreases the overall throughput by 2 to 5 MB/s. However, as the large block size exceeds 64 KB, it is hard to observe any performance difference. Although any block size beyond 64 KB will fit OBFS's needs, we still prefer the larger block size based on the the analysis of the scientific file system workloads in section 4.1. Considering more than 90% of all disk space is occupied by 10% of extremely large files, the bigger the large block is, the less blocks are consumed by those files. Therefore, the less management overhead, both static and dynamic, is associated with them.

The small block size is not evaluated since many traditional file systems suggest the use of 4 KB. Given that the small block allocation and usage patterns in OBFS are quite similar to those in other traditional file systems, it is normal to expect the 4 KB block size to be a good choice. Although some researcheres indicated that efficient handling of extremely small files with smaller block sizes has significant impact on file system performance, it is important to note that such conclusions are drawn from typical office and engineering environments, where small files dominate. Our distributed file system, *Ceph*, is targeting scientific computing environments. As an OSD storage manager in *Ceph*, OBFS has to efficiently handle object workloads derived from such environments. The file system workload studies in Chapter 4 suggest that the number of small files are less significant in scientific environments. The object workloads derived from them are dominated by large uniformly-sized objects. This makes small block size less important. As a result, OBFS simply adopts the 4-KB block size without further evaluating other alternative sizes. In the following evaluation, we set the large block size 512 KB and the small block size 4 KB.

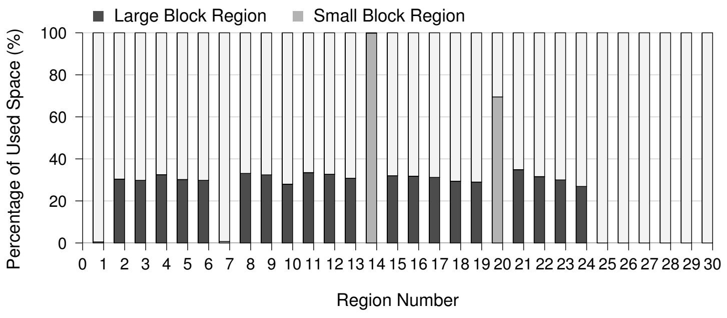
7.4 Allocation Policy

OBFS's allocation policy consists of two parts: the region allocation policy and the block allocation policy, as described in Section 6.1. The region allocation policy has slow performance impacts in the long run, while the block allocation policy has immediate effects.

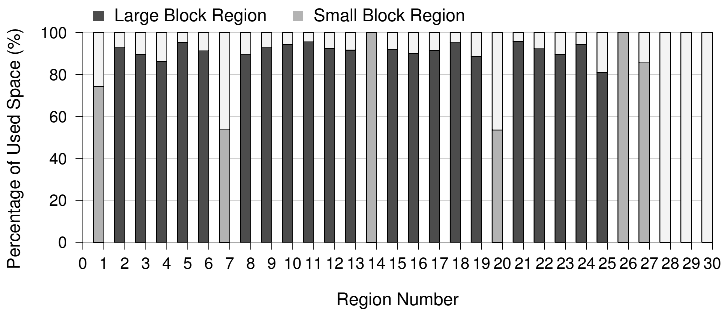
7.4.1 Region Allocation

While serving a new request, OBFS has two different choices for allocating space: either allocating a new region close to the current location or seeking a long distance to use the free space in an existing region. Using the free space in a remote existing region can degrade performance temporarily, but it effectively conserves empty regions. These empty regions are very valuable, especially when disk usage is high, because they provide more buffer space in case the workload experiences temporary changes. For example, if all regions are allocated with a few highly-utilized small block regions and many under-utilized large block regions, OBFS may not be able to find enough space for small objects as the workload suddenly changes to one that is dominated by small objects. This forces OBFS to start the expensive region clean process. In our current design, OBFS adopts the conservative approach. It always tries to consume the free space in existing regions before allocating a new one. Although this scheme comes with some performance penalty, it is advantageous in the long run since the region clean process will only occur in very rare cases. As we will demonstrate in the following experiments, we have not seen the clean process being triggered in any of our aging processes, even when we purposely disturb the aging workload to drastically change the large/small object request ratios.

Figure 7.3 plots the OBFS region distribution after applying the aging workload to the fresh file systems. The dark bars in the figure represent the large block region and the gray bars represent the small block region. The empty regions are depicted using the white bars. The aging workloads, as described in Section 7.2.1, are composed of two streams, one *Syn-Write* workload stream to simulate the long term transformation effects and the other random write/delete stream to mimic the temporal requests. The transformation stream has a fixed large/small object request ratio, where large object requests account for 80%. The random stream generates more small object requests. To further test the region allocation policy, we change the average request inter-arrival time of the random stream several times during the aging process. Smaller inter-arrival time of the random stream makes requests from the random stream more intense, which effectively increases the number of small object requests in a given period of time. This results in different large/small object request ratios. Two aged system images are demonstrated in the figure. After aging, one represents the heavily loaded system where 80% of all disk space is occupied. The other consumes only 20% of the disk space. In the lightly loaded system, there are 20 large block regions, 4 small block regions, and 5 empty regions. The large block regions maintain a fairly balanced utilization, ranging from 20% to 30%. The small regions show very different utilizations. The first two small regions, region 1 and 7, are almost empty, while region 14 and 20 have very high utilization. Although we intentionally adjust the large/small object request ratio throughout the aging process, OBFS still preserves five empty regions. A similar pattern can be observed as the disk utilization increases. There are still three regions left untouched and large block regions have



(a) Disk Utilization 20%



(b) Disk Utilization 80%

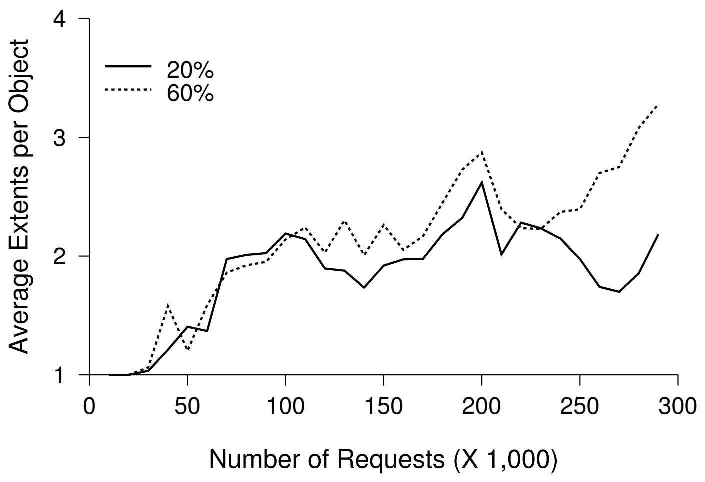
Figure 7.3: Region usage distribution under different disk utilization.

even usages from 80% to 95%. Another nice properity is that the small block regions are evenly distributed between the large block regions. On average, this tends to produce good seek distances.

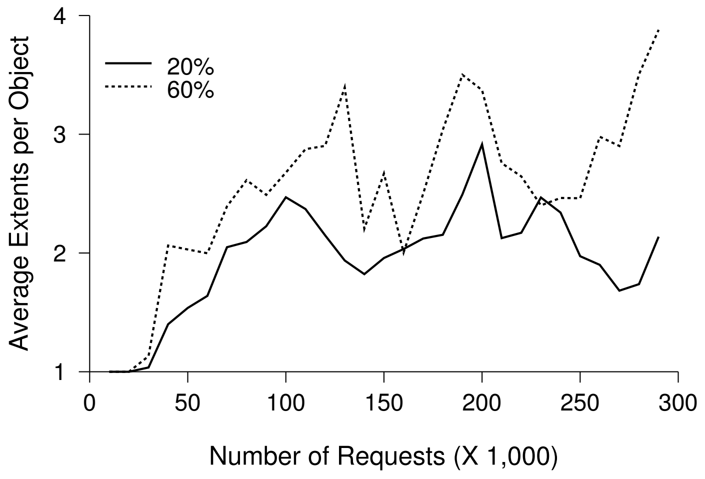
7.4.2 Object Fragmentation

OBFS aims to reduce disk fragmentation in the long run. Large objects in OBFS are always contiguous on disk due to the large block. Therefore, we are more interested in small object fragmentation and only measure its results. All the fragmentation data are collected during the file system aging processes, as described in Section 7.2.1 The average number of extents and average extent size are logged every ten thousand requests.

Figure 7.4(a) shows the average number of extents per object on disk. Two different disk utilizations are plotted. Both lines show the same trend. At the very beginning, OBFS can easily allocate contiguous space to every write request since the disk is empty. After 30,000 requests, the on-disk small objects start to fragment. The average number of extents per object reaches two at 75,000 requests and fluctuates around two thereafter. The sharp rise and drop of the curves are the results of changing the inter-arrival time of the temporal stream. A shorter inter-arrival time implies that more small object requests arrive in the given time. Since OBFS tends to serve requests in adjacent regions, the intensive small object request stream increases the possibility of a single small block region being heavily exercised. Thus, it boosts the average number of extents per small objects. High disk utilization incurs more pressure on the block allocation policy, which can be clearly observed after 120,000 requests. The small objects on the high utilization disk are more fragmented than those on the low utilization disk.



(a) On-disk small object fragmentation



(b) Overall small object fragmentation

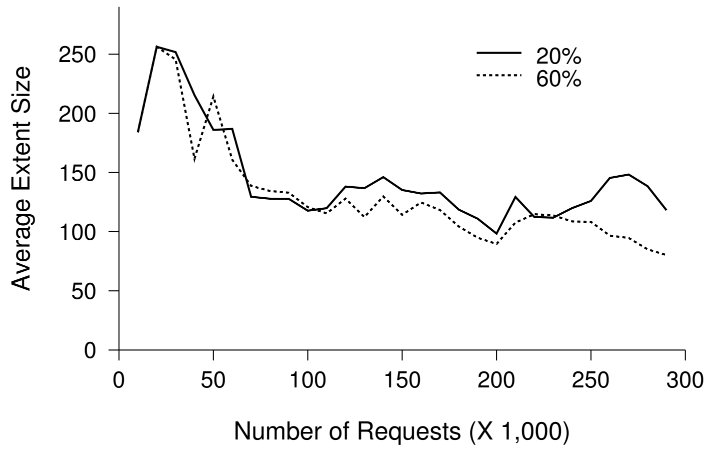
Figure 7.4: Average number of extents per small object as the file system ages. The disk aged to 60% full is plotted in a dashed line. The one with 20% utilization is depicted in a solid line.

At the end of the aging process, the small objects on the 60% full disk have a little more than three extents.

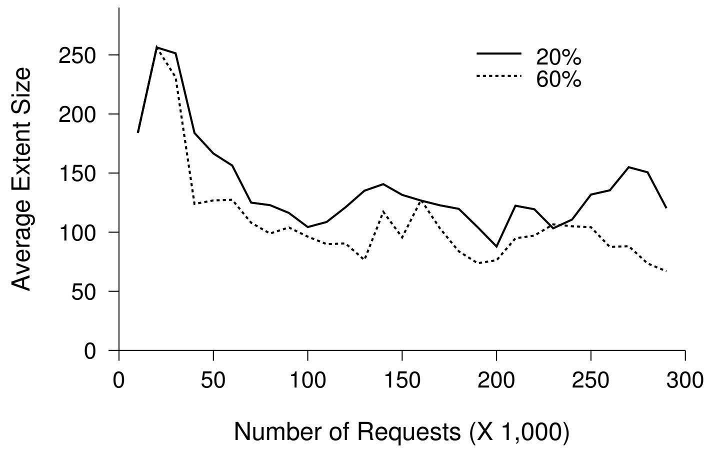
Figure 7.4(b) counts all allocated small objects, either on disk or being deleted. Unlike the on-disk objects that result primarily from the transformation stream, the overall objects reflect both the transformation stream and the random stream. Since the random stream simulates the temporary requests that tend to be removed in a short period of time, it is tuned to be more intense and more bursty than the transformation one. Such bursty patterns tend to increase the space pressure in the outstanding region and result in more fragmentations in object allocations. As we can see from Figure 7.4(b), the 20% curve is similar to the one in Figure 7.4(a), while the 60% line is quite different from the one in Figure 7.4(a) as the deleted objects are counted in. It shows bigger fluctuation and more fragmentation, which demonstates that at high disk utilization, OBFS is more sensitive to workload changes, as expected.

From these experiments, we see that OBFS can manage to keep fragmentation low among small objects. The average number of extents per small object is between 2 and 3. Considering the always-contiguous large objects, which account for the majority of on-disk objects, the average number of extents per object is significantly lower than 2.

The average extent size is depicted in Figure 7.5. The ideal value of the average extent size should be around 250 KB since we randomly distribute the small objects between 4 KB and 512 KB. The figure indicates that in practice OBFS can maintain its value around 100 KB. As expected, the higher the disk utilization, the smaller the average extent size.



(a) On-disk small object extent size



(b) Overall small object extent size

Figure 7.5: Average extent size as the file system ages.

7.5 Summary

OBFS design is specially optimized for the object workloads derived from the scientific computing environments. To faciliate the performance evaluation, we introduced several micro benchmarks and synthetic workloads, which are employed in this chapter and the following chapter. An object file system aging technique as well as a sythetic aging workload are also presented to simulate a more realistic environment.

Using these micro benchmarks, we study several key design parameters of OBFS, such as the large block size and the region size. Our results show that the region size has only a slight impact on the overall throughput. On average, write throughput improves as the region size increases. However, those improvements are small, ranging from 5% to 10%, that they can barely be observed. The overall throughput is almost unchanged when the region size grows beyond 256 MB, which indicates that any region size larger than 256 MB would be a good choice for OBFS. We also evaluated the best large object size for OBFS and concluded that the large block sizes between 128 KB and 1024 KB have similiar performance impacts. Thus, we choose 512 KB as the default large block size to better cope with the object size chosen by the high level file system design. We further study the region and object allocation policies under long-term aging workloads. OBFS region allocation policy successfully balances the object workload across all used regions and preserves a significant amount of empty regions even after long-term aging. Small object regions are evenly distributed among the large object regions, which tend to leverage the seek distances in mixed object workloads. The object fragmentation are well controlled using the object allocation policy. The small object fragmentation slowly grows from 1 to 2 and flucturates between 2 to 3 during aging process. Considering the always-contiguous large objects, which account for the majority of on-disk objects, the average number of extents per object is significantly lower than 2.

Chapter 8

Performance Comparison

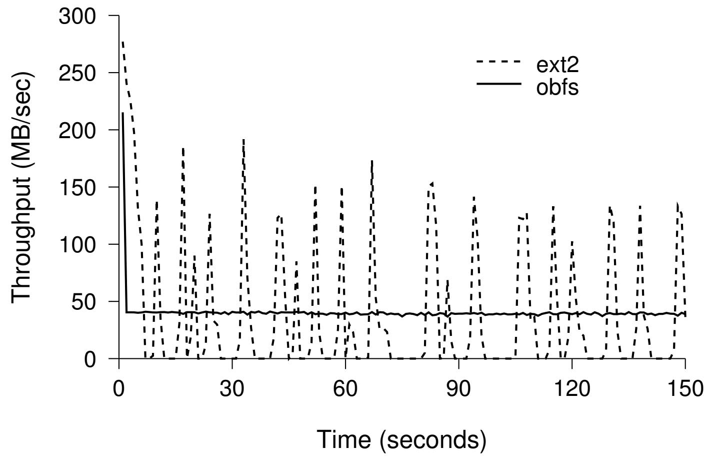
In this chapter, we study in detail the I/O characteristics of OBFS and compare them with those of general purpose file systems, Ext2 and XFS. We are interested in how OBFS would respond to various workload parameters. Since no real object workload exists, we expect to learn OBFS's advantage or weakness under workloads with wide ranges of parameters through those experiments, demonstrating the OBFS effectiveness even as the target object workload shifts from its expected ranges. The workloads and benchmarks used in this chapter are the same as those described in Section 7.2.

Objects are mapped into individual files under the root directory in Ext2 and XFS. As we mentioned in Section 5.4.2, Ext2 suffers from big directories due to the linearly-name-lookup scheme. It will introduce significant performance degradation if all objects are placed in the same directory. To mitigate this problem, we create multiple directories, 256 in the experiment systems, at the root of those file systems. An object is hashed into one of those directories using a simple hash function: the high two bytes of its object identifier are XORed with its lower two bytes, whose result is then used modulo the number of the directories under the root. This approach alleviates the performance degradation caused by the big directory in Ext2 and enables us to fairly compare performance difference between Ext2 and OBFS.

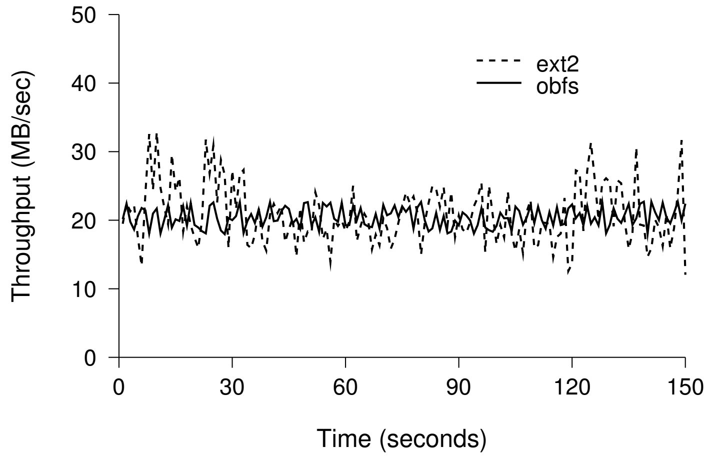
8.1 Read/Write Throughput Variance

OBFS is designed to provide high sustained bandwidth with minimal variation. Some applications, such as databases, are sensitive to throughput variation because their performance is severely bounded by log updates. Their normal activities stop every time log writes block. Big throughput variations typically result in longer response time and lower overall transaction throughput. In this section, we evaluate the throughput variation of OBFS and compare it with that of Ext2. We choose the *TwoValue* workload with the *ratio* parameter of 0.8 to study both the synchronous and asynchronous write characteristics. To extract the read characteristics, we conduct experiments with both the random read and sequential read workloads. All experiments run on fresh file systems. The throughput data are collected every second.

Figure 8.1 shows the first 150 seconds of write experiments. As we can see from Figure 8.1(a), OBFS has pretty constant asynchronous write throughput with a small variation of less than 3 MB/s. Its sustained throughput floats around 40 MB/s after the buffer writes in first several seconds. On the other hand, the asynchronous write throughput of Ext2 presents extreme variation. The curve repeats in similar patterns: a short period of peak throughput followed by several seconds of idleness. Linux buffer cache contributes to this throughput



(a) Asynchronous write



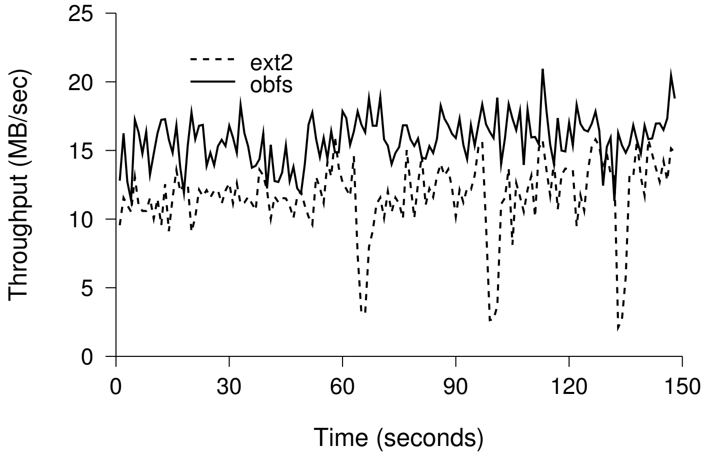
(b) Synchronous write

Figure 8.1: Short term write throughput comparison between OBFS and Ext2. The *TwoValue* workload with the *ratio* parameter of 0.8 is employed in both experiments.

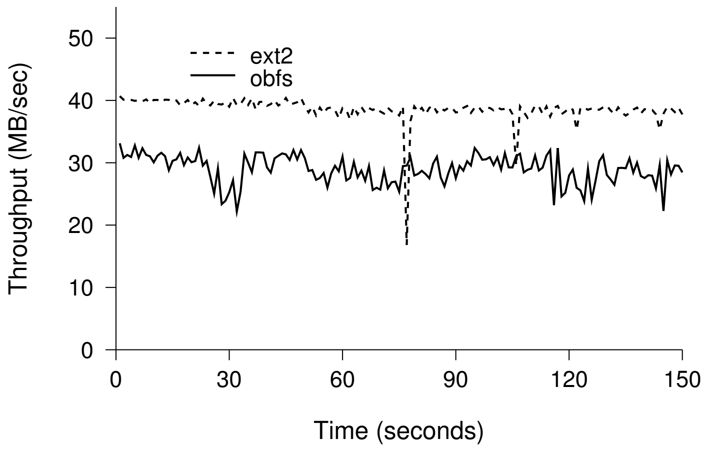
variation. Since Linux initiates its buffer flushing thread (*bdflush*) in blocking mode when the memory pressure is high, Ext2 will choke during the dirty buffer flushing time, as indicated by the zero throughput region in the figure. As soon as the cache space is available, Ext2 starts to consume more buffer to serve external requests until the next flushing time. During buffer writes, the write throughput is very high. However, the underlying disk stays idle, which hurts the sustained throughput. OBFS addresses the variation problem by managing its own object buffer cache in the page cache. Rather than blocking all requests until large amount of page frames are claimed, the object buffer cache returns each page frame to the OS as soon as it is available. Such claimed pages can be immediately used to serve pending requests. Therefore, the cache flushing thread runs in parallel with the memory allocation thread, which reduces the throughput variation and maximizes the disk bandwidth utilization. The average write throughput for Ext2 is around 37 MB/s, 3 MB/s less than that of OBFS.

Figure 8.1(b) plots the synchronous write throughput variation over time. Both Ext2 and OBFS show comparable performance around 20 MB/s. However, the causes of their variation are different: OBFS needs to jump between different regions to service different type of requests; Ext2 has to synchronously write the file data as well as related metadata, such as the inode and directory entry. OBFS shows a little bit less variation in synchronous write mode.

We ran several experiments to examine the read throughput variation, as depicted in Figure 8.2. All experiments ran on file systems with 30,000 objects, 80% large objects and 20% small objects. The random read workload simply selects one of the on-disk objects and reads it back, while the sequential read workload reads back all objects following their write



(a) Random read



(b) Sequential read

Figure 8.2: Short term read throughput comparison between OBFS and Ext2. 30000 objects are written to a fresh file system before conducting the read experiments. The *random read* experiments generate read targets by randomly selecting an object from disk. The *sequential read* experiments read back all objects following the order that they were written.

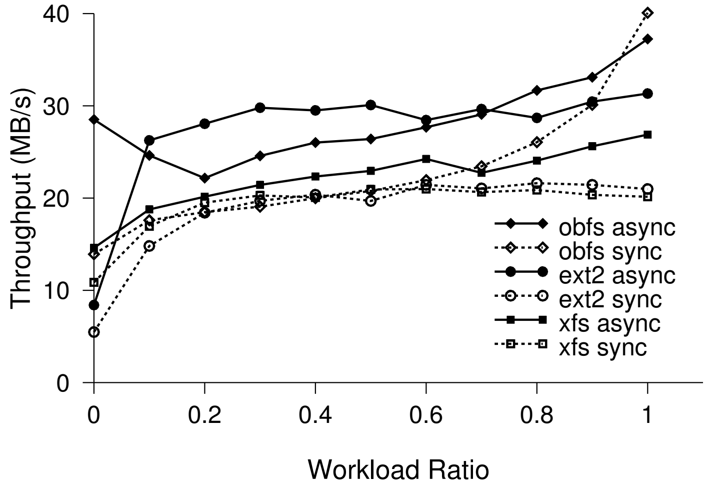
order. All file systems were unmounted before starting experiments to clean up the buffer cache. In the random read experiments, as shown in Figure 8.2(a), both OBFS and Ext2 have low throughput due to the random seeks. OBFS has a 3 to 4 MB/s advantage over Ext2, largely owing to the collocation of the data and metadata of large objects. Although the throughput variations are comparable to each other, we notice that Ext2 has several sharp drops every 30 seconds. One possible explanation is that the OS starts to release the buffer cache due to memory pressures, which holds the entire cache for a short period and decreases the overall throughput.

The sequential read experiments are plotted in Figure 8.2(b). Ext2 demonstrates significantly better performance than OBFS in this scenario, with 10 MB/s advantage compared with 30 MB/s throughput of OBFS. OBFS is not well prepared for such strict sequential read workloads because it treats large objects and small objects differently. Large objects and small objects are allocated in different regions even if they arrive contiguously in the same stream. As the strict read workloads arrive, OBFS still needs to switch between regions to retrieve those objects, while Ext2 translates the workload into nearly perfect sequential disk I/Os. However, as we mentioned in Section 4.2, such strict sequential read workloads are unrealistic since they require a single server to exclusively access a set of OSDs at both write and read time and the workload stream happens to contain both large and small object requests. This scenario is rare in a large parallel/distributed system.

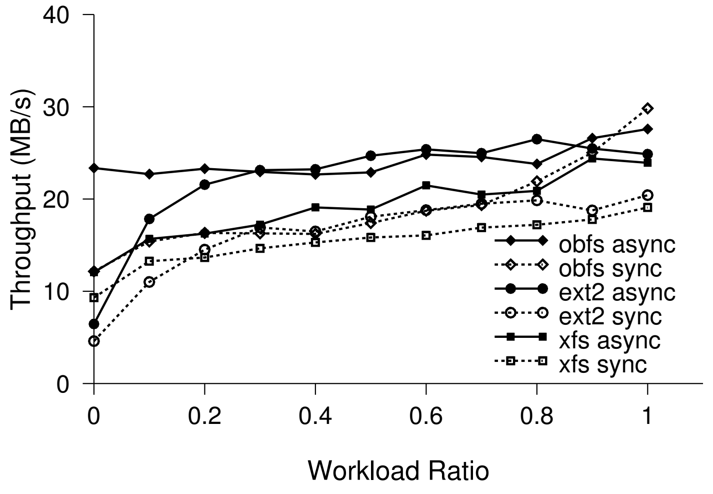
8.2 Mixing Small and Large Objects

OBFS is designed to optimize for large objects while still maintaining good performance for small objects. Although we conclude in Section 4.2 that the object workloads for scientific environments are typically composed of more than 80% large object requests and 20% small object requests, we are still interested in studying how OBFS behaves under a wider range of workloads. Such knowledge will help us in deciding the applicable environments of OBFS. We set up experiments on both fresh and aged file systems and compare OBFS against Ext2 and XFS. The benchmarks used are the *TwoValue* workloads with the *ratio* parameter ranging from 0 to 1. The large block size and small block size are set by default to 512 KB and 4 KB, respectively. The request size in the workload is either 512 KB or 16 KB. One GB of data are written for each run. Each experiment is repeated three times. The average throughput is plotted in the figure.

The *x* axis in Figure 8.3 indicates the fraction of large object requests in all requests. The *ratio* value of zero means all requests reference small objects; the *ratio* value of one means that all requests are large object requests. The curves in Figure 8.3(a) show that almost all file systems improve their write throughput, whether synchronous or asynchronous, as the fraction of large object requests increases. The only exception is that OBFS exhibits decreased throughput as the *ratio* increases from 0 to 0.2 during the asynchronous write experiments. The main reason for this trend is related to the region switching. When the *ratio* is zero, OBFS deals with only one region type. With the help of efficient metadata design, OBFS can achieve extremely good performance compared to Ext2 and XFS. However, with large object requests



(a) Fresh File System



(b) Aged File System (60% full)

Figure 8.3: Write throughput varies as workload small/large ratio changes.

mixed in, OBFS starts to jump between regions, which degrades the performance. As the *ratio* increases beyond 0.2, the benefits of large object requests offset the region switching cost and lead the overall throughput into an up trend. Ext2 demonstrates very good asynchronous write performance with sustained throughput around 30 MB/s. OBFS lags behind in the range between 0.1 to 0.6. As the *ratio* exceeds 0.7, OBFS shows its advantages on the large object optimization by a factor ranging from 10% to 20%.

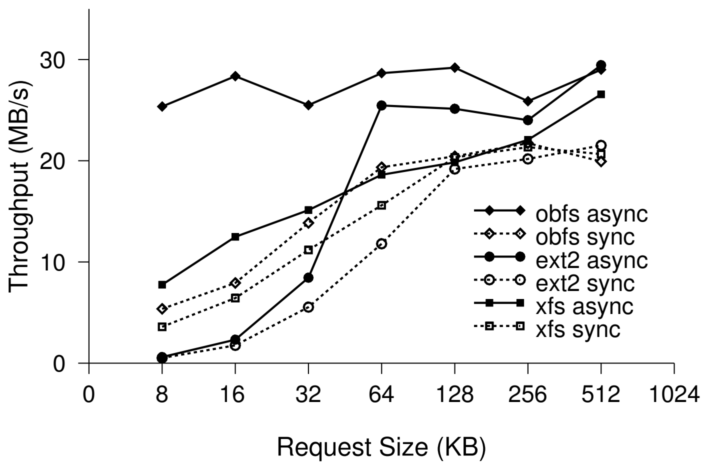
All file systems have similiar synchronous write performances as the small object requests dominate. The sustained throughput is around 20 MB/s. OBFS differs from the other two when the workload contains more large object requests. Its synchronous write performance improves greatly after the *ratio* of 0.6, while Ext2 and XFS stay at 20 MB/s with almost no changes. It is interesting to note that OBFS has higher synchronous write throughput than the asynchronous one when the workload contains purely large object requests. The reason for this, as we explained before, is that OBFS allocates both object data and metadata contiguously on the disk. Workloads with purely large requests will result in perfectly sequential disk I/Os. Buffering such workloads has no benefit. On the contrary, memory management overheads associated with the object cache degrade the overall throughput.

Figure 8.3(b) plots the write throughput on the aged file systems. Compared with those on the fresh file systems, both the synchronous and asynchronous writes have about 15% to 25% of performance degradation. It is obvious that the fragmented space of the aged file systems decreases the possibility of allocating contiguous space for objects and merging buffered requests into big disk I/Os. Again, OBFS has clear edges on both high *ratio* range and low *ratio* range. For the middle range, OBFS is comparable to Ext2 and shows a little advantage over XFS.

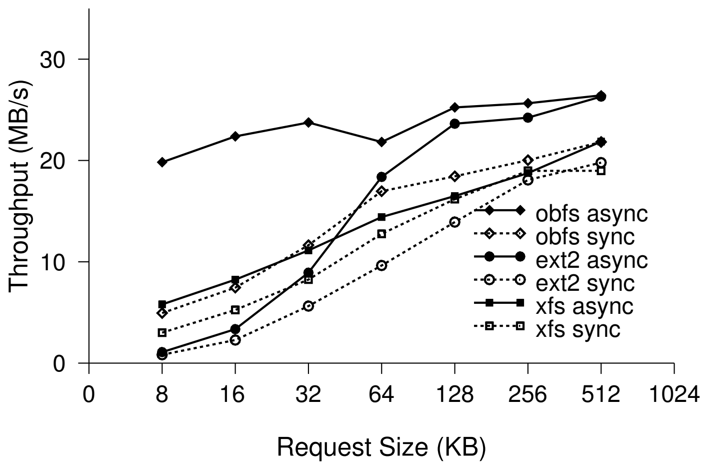
8.3 Request Sizes

In previous experiments, it is of no surprise to see that OBFS has clear advantage on the large object side since its goal is to optimize for the large object workloads. However, we did not expect to see OBFS beats the other two file systems with large margins in the small object end. In this section, we run a series of experiments with fixed request size to illustrate the OBFS performance for small objects. The basic setup is identical to the previous one except that the workloads used here have only one fixed size.

OBFS shows consistent throughput for asynchronous writes. The overall throughput is around 27 MB/s and 22 MB/s for the fresh system and the aged system respectively. Ext2 throughput becomes very low when the request size is small. With merely 1 MB/s throughput at the request size of 8 KB, OBFS exhibits a twenty-fold performance advantage. The main reason for such low throughput is the inability of the Ext2 directory structure in handling large number of small objects. Ext2 employs linear search to locate a file under a directory. Such search has to be executed for every object write/create since it needs to either find the entry or validate that there is no duplicated copy. Although the hash-based *dentry* cache can reduce the overhead in this situation, only a small fraction of *dentries* can be loaded into memory given the large size of a typical *dentry*. Once missing in the *dentry* cache, Ext2 has to perform the linear search on the expanding directory file. OBFS addresses this flat name space problem by maintaining a very compact mapping table. On average, each object only occupies a little



(a) Fresh File System



(b) Aged File System

Figure 8.4: Write throughput varies as request size changes.

more than 8 bytes in this table. The mapping table with millions of objects can fit easily into memory. Thus object lookup is extremely efficient for OBFS. OBFS also benefits from the minimal metadata update associated with each object operation. Unlike Ext2 and XFS, which require up to four different I/Os to update all related metadata, OBFS only needs one additional I/O for object metadata. Fewer seeks are incurred and more disk bandwidth is available for user data.

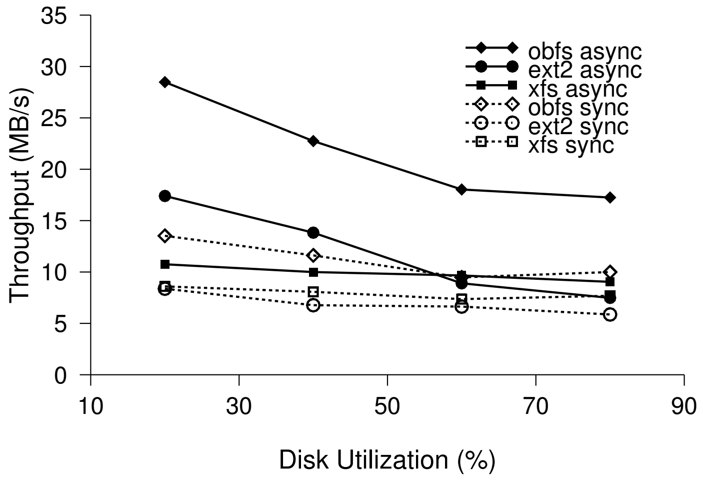
OBFS demonstrates extremely good write performance for workloads containing purely large objects or small objects. For mixed workloads, the frequent switches between different regions offset some of the advantages. However, it is still comparable to the other two file systems for asynchronous writes and even shows some improvements for synchronous writes.

8.4 Aged File System Performance Study

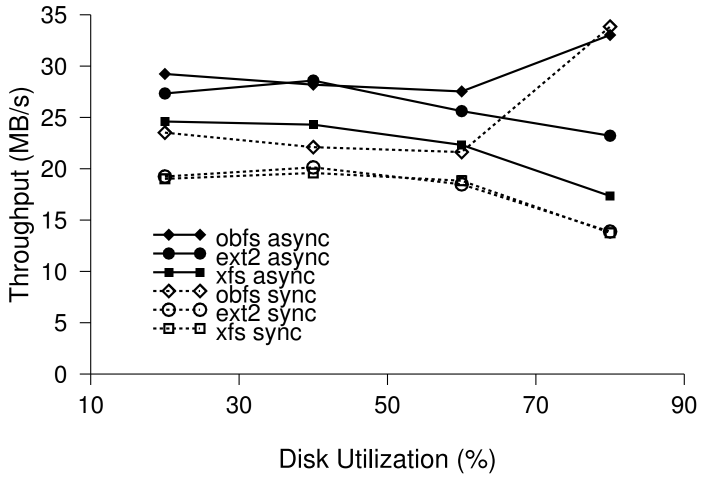
Aged file systems provide more realistic environments for performance evaluations. File system throughput measured on an aged system is more useful to demonstrate its behavior in the real world. In this section, we measure the write performance of all three file systems on aged disks using the aging technique described in section 7.2.1. The aging workloads employ two streams with a total write/delete requests of around 450,000. To set up the experiments, we format a disk using one of the three file systems and apply the aging workload until a specific number of requests have been issued. One parameter of the aging workload determines the target disk utilization. During the aging process, the disk utilization may fluctuate, but it will eventually settle on the target that we specified. Such aged disk images are backed up to another disk so that we could run multiple experiments without repeating the time-consuming aging process. For each run, a total data of 1 GB are dumped into the aged systems using the *TwoValue* workload. The large request size is 512 KB and the small request size is 16 KB.

We can see from Figure 8.5 and Figure 8.6 that file system performance generally decreases as the disk utilization is increased with a few exceptions. On average, Ext2 is more sensitive to disk usage. It shows good asynchronous write performance when the disk usage is low, while its performance drops significantly as 80% of the disk space is used up. As shown in Figure 8.5(a), when facing the small request benchmark, Ext2 throughput at 80% disk usage is almost half of that at 20% disk usage. For mixed workloads with *Ratio* of 0.5 and 0.8, its performance degrades by a factor of 25% and 30% respectively. OBFS and XFS, on the other hand, are less sensitive to the aged disk usage. Their curves are flatter due to their better space management policies. Both file systems try to prevent large extent of contiguous space from being fragmented, which is beneficial when free space is limited.

In general, OBFS performs the best for synchronous writes and shows some advantages on a high utilization disk for asynchronous writes. It is a little surprising to see that Ext2 performs better than XFS in almost all experiments. Several factors contribute to Ext2's success. First, Ext2 deeply buffers asynchronous writes before dumping them all together to the disk. This increases the potential for large sequential disk I/Os. However, it also causes big variance on throughput. We notice that Ext2 tends to choke for several seconds under intensive workloads during almost all of our experiments. Such "choke" behavior is not desirable in parallel environments since it might hold up other OSDs that depend on it. Second, XFS is

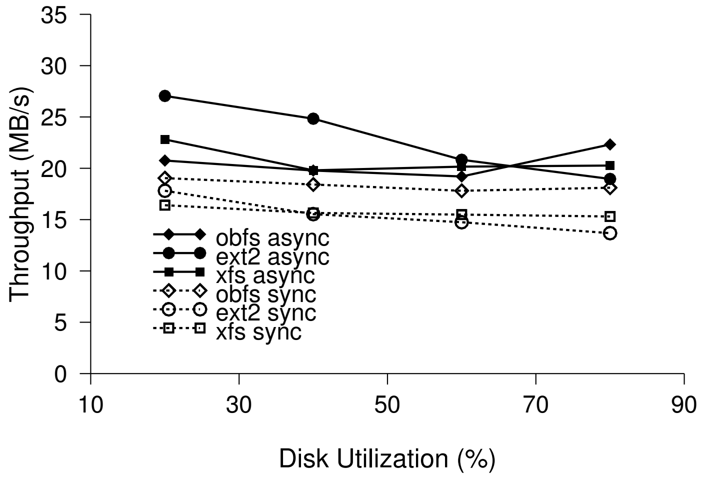


(a) Small Objects (*Ratio* 0)

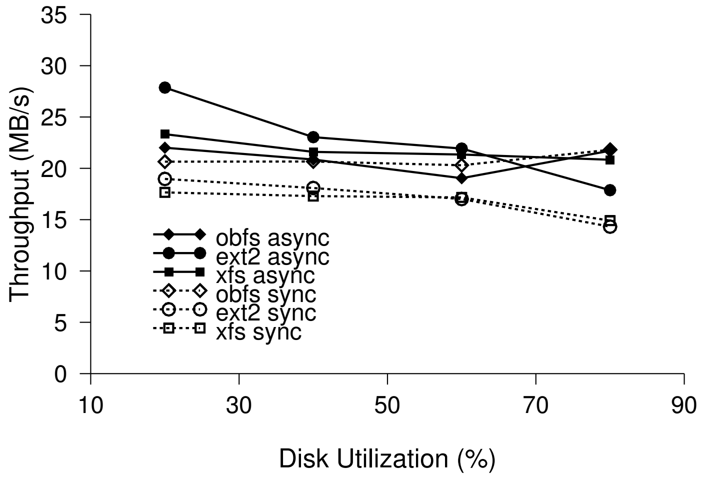


(b) Large Objects (*Ratio* 1)

Figure 8.5: Aged file system performance study using small object and large object workloads.



(a) *Ratio* 0.5



(b) *Ratio* 0.8

Figure 8.6: Aged file system performance study using mixed object workloads

a journaling file system, which spends additional I/Os to log file system activities/transactions and carefully chooses update orders to achieve fast recovery. Ext2 does not care about recovery and freely writes buffered data, which achieves the best performance. Third, a synchronous write to Ext2 only forces it to flush the user data back to the disk. The metadata is still kept in memory. This greatly reduces small random I/Os but increases data loss probability. It is still possible to lose an object even if it has been synchronously written to Ext2. Finally, XFS is designed for large systems. It applies the B-Tree structure extensively internally, which is efficient as the file system grows. Our current setup employs only an 8 GB partition for all experiments. This may not be large enough for XFS to show significant advantage over Ext2. Considering all these facts, we can see why XFS performs worse than Ext2 in this series of experiments.

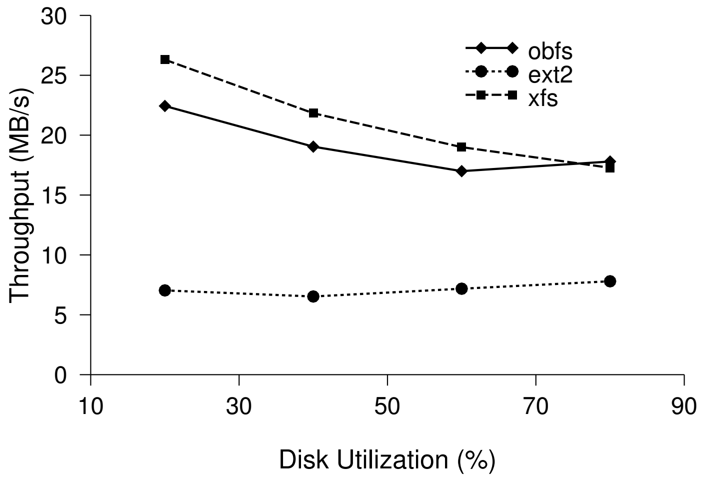
Another interesting point worth mentioning is that OBFS throughput actually increases from 60% to 80% disk utilization. This trend is quite siginificant in Figure 8.5(b). The throughput surges from 21 MB/s to 33 MB/s. After studying the underlying I/O traces, we determine the OBFS allocation policy is the key factor. OBFS tends to allocate objects in already used regions rather than touching empty regions. As showed in figure 7.3(b), OBFS still manage to keep three empty regions on an 80% full disk after long-term aging process. All other regions have very high utilization. Most of them are larger than 90%. When the write workloads arrives, those existing regions are easily filled up. OBFS has to allocate the empty regions to service more requests. Writing to those empty regions is no different from writing to a fresh system. Thus, it is not surprsing to see the performance surge. In the future we may explore alternative policies for region selection.

8.5 Read/Write Throughput

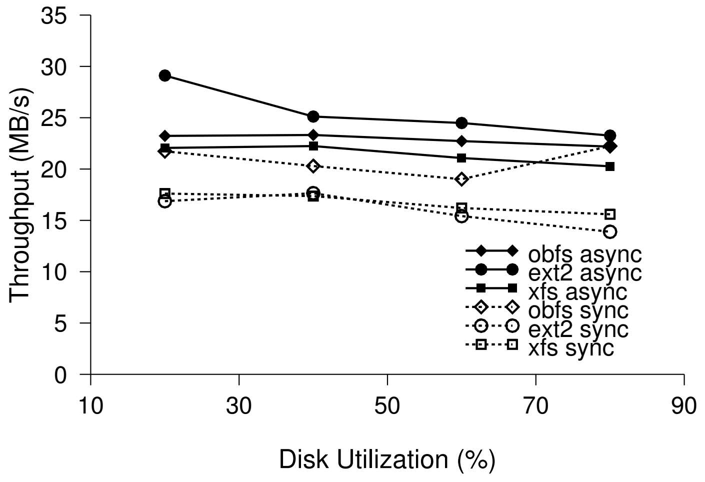
In this section, we further evaluate the read/write performance of all three file systems using the *SynRead* and the *SynWrite* workloads. The *SynRead* workload fetches 20% of on-disk objects, 60% of all requests are random reads and the rest are sequential reads. No object will be read more than once in order to avoid cache effects. Detailed descriptions of these workloads can be found in section 7.2.

Figure 8.7(a) shows the read performance on aged file systems. Before each experiment, the file system cache is cleared through unmount. XFS definitely wins in this area, leading OBFS by 3 to 5 MB/s on average. Ext2 performs extremely poorly with the average read throughput around merely 8 MB/s. The random read pattern in the *SynRead* workload really hurts the Ext2 performance. Compared with hash table in OBFS and the B-Tree in XFS, Ext2 simply organizes all its directory entries flat in a directory file. To find a file inside a directory, Ext2 has to do linear search in those directory entries. In our test systems with thousands of objects on the disk, this process can be very slow. Another disadvantage of Ext2 is the block-based allocation. Since its *inode* can only hold twelve direct block addresses, it requires an additional indirect block for objects large than 48 KB in our test systems. Therefore, an object read may require more than three disk I/Os before it can actually read the object data. Finally, the block allocation policy of Ext2 has much greater potential of fragmenting existing free space since it has no contiguous free space concept. All these facts contribute to the poor read performance of Ext2.

OBFS shows its advantages again on the synchronous write. It beats both Ext2



(a) read performance



(b) write performance

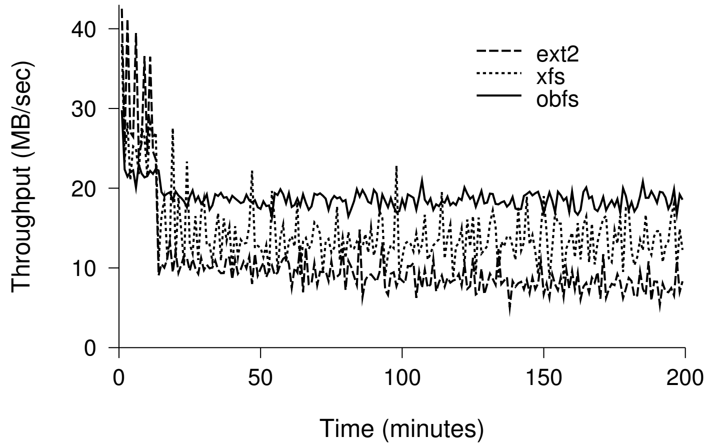
Figure 8.7: Overall read/write performance on aged file systems.

and XFS by 4 to 10 MB/s. Ext2 exhibits good asynchronous write performance. OBFS's asynchronous write throughput, dragged down by the overhead of frequent region switching, is about 2 MB/s slower than that of Ext2. However, it is still faster than XFS by a margin of 1 to 2 MB/s. As we discussed in the previous section, the fast asynchronous write of Ext2 sacrifices the throughput consistency and file system recoverability, which can have negative effects in parallel environments.

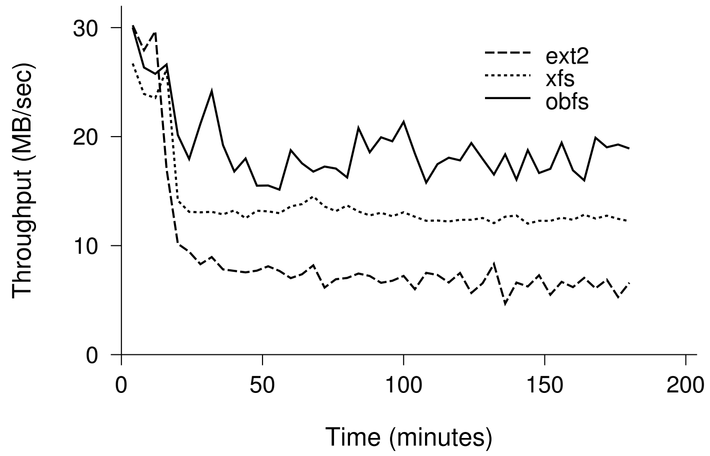
8.6 Sustained Write Throughput

Although Ext2 exhibits good asynchronous write performance in short term benchmarks, we notice consistently that Ext2 is much slower in long term aging processes. It typically requires twice as much time as those of OBFS and XFS. Figure 8.8 shows the sustained write throughput during first two hundred minutes of aging processes. The aging workload used in Figure 8.8(a) has only one aging stream. It collects throughput data every minute. The aging workload used in Figure 8.8(b) contains two aging streams, one of which is to simulate the temporal requests. The throughput data are logged every ten minutes. An aging stream contains only object writes and deletes. All of the requests are asynchronous requests. A more detailed description of the aging workloads can be found in Section 7.2.1.

As plotted in Figure 8.8(a), the average write throughput of Ext2 is slightly less than 10 MB/s when facing one aging stream. This is only one half of what OBFS has and about 50% slower than that of XFS. The aging process starts by filling the disk with certain amount of objects. As soon as the disk usage reaches the target value, the delete requests



(a) Single-stream aging workload



(b) Multi-stream aging workload

Figure 8.8: Sustained write throughput during aging process. Sub figure (a) uses a single aging stream and collects the throughput data every minute. Sub figure (b) uses two aging streams and collects the throughput data every ten minutes.

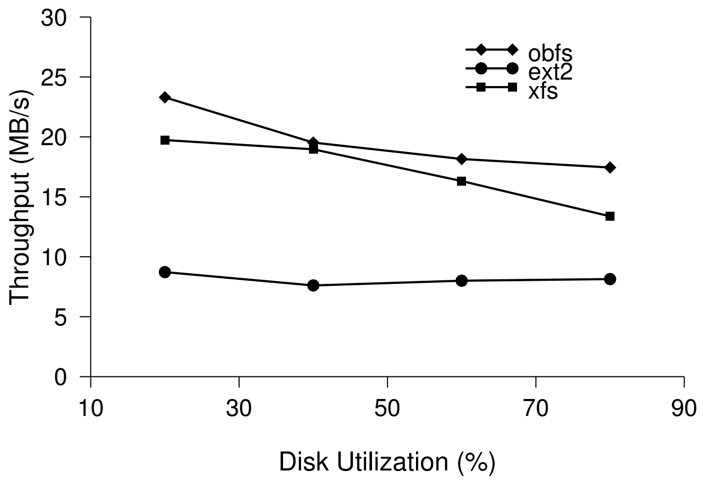
kick in. In our experiments, the object deletes initiate at one minute in the time line. We can see from the figure, in the range between one and ten minutes, OBFS write throughput drops to 22 MB/s after deletes start arriving. Both Ext2 and XFS still maintain high throughput beyond 30 MB/s. This is because OBFS tends to allocate objects in existing regions. When object deletes clean up some slots in existing regions, OBFS turns back to utilize them instead of allocating new regions. On the other hand, Ext2 and XFS tend to consume fresh disk space to maximize throughput. As soon as all large chunks of contiguous space are used up, they are forced to utilize the fragmented space between existing objects. This reflects in the figure as a sharp performance drop around ten minutes in the time line. Ext2 suffers from the mixed workload of writes and deletes. Since the objects to be deleted are selected randomly from disk, they exercise the Ext2's name space management extensively. Like read requests, the delete requests in Ext2 create many small I/Os to update the directory structures and allocation bitmaps, they severely interfere with the write stream and drag down the overall throughput. Both XFS and OBFS use journals to log the delete transaction and minimize the impact of delete operations on the foreground write stream. We observe the same pattern in Figure 8.8(b). The sustained throughputs of Ext2 and XFS hang around 7 MB/s and 14 MB/s respectively. OBFS shows a bit larger variance in multiple-stream aging workloads. This is because one of the aging stream generates large amount of small write/delete requests from time to time to simulate the temporal requests. OBFS is more sensitive to the sudden changes of the large/small request ratio, which leads to fluctuations on the write throughput.

8.7 Synthetic Object Benchmark

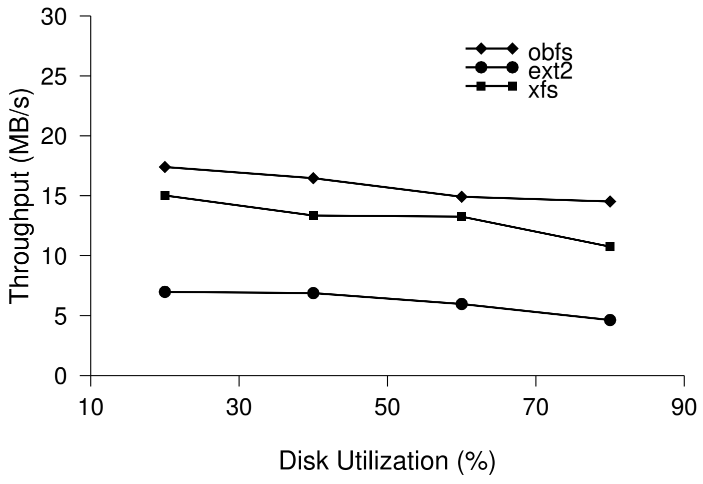
To obtain a comprehensive overview of OBFS performance under more realistic environments, we test OBFS performance using the synthetic object benchmark on the entire disk, 80 GB in total. The key parameters of this benchmark are derived from the LLNL file system workloads, as described in Section 4.2. The benchmark contains 40% read requests, 36% write requests, and 24% rewrite requests. The random read accounts for half of all read requests and the rest are sequential reads. Among the write and rewrite requests, 60% are synchronous and 40% are asynchronous. 80% of all requests reference to the large objects, whose sizes are 512 KB. The small object size distributes uniformly between 4 KB and 512 KB. All experiments were run on aged disks with four different disk utilizations: 20%, 40%, 60%, and 80%. The aging workloads consist of two aging streams with a total of 4 million requests, whose characteristics are described in section 7.2.1.

For comparison, we also show both benchmark results from the 8 GB partition configuration and the whole disk configuration. As we can see from figure 8.9, all file systems show 20% performance degradations under the whole disk setup. This is largely due to the bandwidth difference between different regions on the disk platter. The outer-most region has the highest bandwith, which is around 44 MB/s. The inner-most region can only achieve 24 MB/s bandwith. Since the benchmark workloads cover the entire disk evenly, the overall throughputs in this setup are much lower than those in the 8 GB partition setup.

As we can see from Figures 8.9(a) and 8.9(b), Ext2 shows the worst performance. Its overall throughput ranges from 4 MB/s to 8 MB/s. Several factors contribute to the low



(a) 8 GB disk partition



(b) 80 GB disk

Figure 8.9: Performance comparison using synthetic object benchmark.

throughputs. First, the block-based allocation policy of Ext2 tends to fragment objects on an aged disk. It relies on the buffer cache to coalesce small extents and improve disk bandwidth utilization. However, most requests in the object benchmark are synchronous writes and random reads, which does not benefit significantly from the buffer cache. Thus, the less-optimized allocation policy results in more disk I/Os and effectively reduces the disk bandwidth utilization. Second, Ext2 cannot effectively handle a flat name space. To find an object, it has to linearly search the directory containing the object, which forces the loading of almost all entries of the directory into memory. Both the computation time and disk bandwidth are wasted due to the inefficient linear search algorithm. Ext2 uses the traditional directory entry to represent a file/object. The in-core data structure of a directory entry contains a lot of unnecessary information for a storage manager at the OSD level, which decreases the total number of entries that can be cached in the memory. Thus, the object-lookup process of Ext2 is very slow and expensive. Third, Ext2 metadata layout is not optimized for the object workloads. Several metadata need to be updated before an object is stored reliably on the disk. The frequent synchronous write requests in the benchmark introduce a significant amount of synchronous metadata I/Os. It therefore decreases the overall throughput.

XFS demonstrates much better performance than Ext2. Its extent-based allocation policy minimizes free space fragmentation, which helps to lay out objects sequentially on the disk. XFS delays the extent allocation of a newly-written object until it is flushed to disk. As a result, multiple partial writes of an object can potentially be coalesced together, which further reduces object fragmentation. XFS employs B+Trees to manage its directory structure. Retrieving an object from a directory involves only a few lookups in the B+Tree structures and then loads the corresponding directory entry. This process is very fast and can avoid the loading of too many unrelated directory entries into memory, which saves disk bandwidth and memory space. Compared to Ext2, XFS improves the overall throughput by a factor of 2.

OBFS demonstrates the dominant performance compared to Ext2, as shown in Figure 8.9(b). Its overall throughput is about two to three times higher than that of Ext2. It is also higher than XFS by 18% to 38%. OBFS's advantages on large object handling, efficient metadata management, and optimized allocation policy contribute to most of its success in this benchmark. Similar to XFS, OBFS achieves extent-like allocation through pre-allocated large blocks and extent-based allocation policy. It employs a very compact hash structure that enables fast object mapping and retrieving. Each object occupies an 8-byte mapping entry, which is 10 times smaller than the traditional directory entry used by XFS. Thus, OBFS can buffer much more object-indexing information than XFS, which effectively reduces related disk I/Os and speeds up the object lookup process. Metadata layout in OBFS is designed to minimize disk seek overheads. The onode of a large object is co-located with the large block. A large sequential I/O is sufficient to store or retrieve a large object. In the benchmark with mostly large object requests, OBFS greatly improves the overall performance through efficiently large object handling.

8.8 Summary

OBFS benefits from various special designs optimized for the expected object workloads, as demonstrated in a series of experiments. The results show that OBFS successfully limits the file system fragmentation after long term aging. OBFS preserves reasonable amount of regions untouched even in a highly used disk. The aging loads are evenly distributed to all allocated regions. The average number of extents for small objects is between 2 to 3 after 300,000 aging requests. Since large objects are always laid out contiguously on the disk, the overall number of extents for both large and small objects is far below 2. OBFS demonstrates good synchronous write performance, exceeding those of Ext2 and XFS by up to 80% on both the fresh systems and the aged systems. Its asynchronous write performance is about 5% to 10% slower than that of Ext2 but 20% to 30% faster than that of XFS on a lightly used disk. While on a heavily used disk, OBFS beats both Ext2 and XFS by a factor of 20%. OBFS read performance almost triples that of Ext2 and only slightly slower than that of XFS. Since OBFS is specially optimized for stripe-unit-sized objects, it shows extremely good performance for streaming workloads that purely contain such objects. The sustained bandwidth on a fresh disk is about 40 MB/s for both synchronous writes and asynchronous writes, which is apparently limited by the maximum bandwidth of the disk used in the test system. Compared to Ext2 and XFS on a fresh system, it almost doubles the throughput for the synchronous writes and improves 50% for the asynchronous writes. On an aged system, OBFS still leads by a factor of 15% to 50%. Putting all those factors into account, OBFS achieves 18% to 200% performance improvements over XFS and Ext2 respectively under expected object workloads.

Chapter 9

Conclusions

The fast growing needs for high performance and large capacity storage are best served by distributed storage systems in both high-performance and general-purpose computing environments. Traditional solutions, exemplified by NFS [56], provide a straightforward distributed storage model in which each server exports a file system hierarchy that can be mounted and mapped into the local file system name space. While widely used and highly effective, this model was originally designed for small, low-performance (by modern standards) storage systems and is relatively inflexible, difficult to grow dynamically, and incapable of providing performance that scales with the number of servers in a system.

Object-based storage model [42] promises to address these limitations through a simple networked data storage unit, the Object Storage Device. This new model is different from traditional storage model in that it separates the storage management from the file hierarchy management, which reduces the the load on the file servers, enables direct data transfer between clients and storage nodes, and improves the scalability of the file systems. The new storage model introduces dramatic changes in the design and development of a reliable, scalable, high-performance object-based storage system. Among them, how to design a small but efficient file system for OSDs is of great challenge.

This thesis focuses on the design, performance, and functionality of an individual OSD in a large distributed object-based storage system, currently being developed at the UCSC Storage Systems Research Center. Based on file system workload analysis and object workload studies, I extract the unique features of the object workloads and design an efficient storage manager, named OBFS, for individual OSDs. OBFS employs variable-sized blocks to optimize disk layout and improve object throughput. Large blocks, the same size as the system stripe unit, are used to optimize the layout of large objects. The overall throughput can be greatly improved since the large objects account for the majority of all objects in the expected workloads. Small 4-KB blocks are employed to lay out non-stripe-unit-sized objects to minimize space wasting and improve disk utilization. Large object metadata and attributes are laid contiguously with its data, which further improves disk bandwidth utilization. The physical storage space is partitioned into fixed size regions to organize different sized blocks. Such design can effectively reduce free space fragementation even after long-term aging. The file system metadata are also partitioned into regions and operated independently. This provides better failure isolation and reduces recovery overheads. A compact hash-based structure is introduced to manage the flat object namespace, which preserves the important memory space and enables fast mapping between the global object name and the internal onode identifier.

A series of experiments have been conducted to evaluate OBFS performance compared with two Linux file systems, the Ext2 and XFS. The results show that OBFS successfully limits the file system fragmentation even after long term aging. OBFS demonstrates very good synchronous write performance, exceeding that of Ext2 and XFS by up to 80% on both fresh systems and aged systems. Its asynchronous write performance is about 5% to 10% lower than that of Ext2, but 20% to 30% higher than that of XFS on a lightly used disk. While on a heavily used disk, OBFS beats both Ext2 and XFS by 20%. The read performance of OBFS almost doubles that of Ext2 and is only slightly lower than that of XFS. Putting all these factors into account, OBFS achieves 30% to 40% performance improvements over Ext2 and XFS under expected object workloads, and forms a fundamental building block for the larger distributed storage systems.

Bibliography

[1] Adaptec Corportaion. Fibre channel, storage area networks, and disk array systems – a white paper. Technical report, 1998.

[2] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[3] 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.

[4] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 165–176, Apr. 2003.

[5] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '95)*, pages 16–29, Apr. 1995.

[6] 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.

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

[8] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[9] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290–298, Apr. 2003.

[10] 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.

[11] P. H. Carns, W. B. Ligon, 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, Atlanta, GA, Oct. 2000.

[12] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Oct. 1996.

[13] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[14] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Nov. 1994.

[15] DOE National Nuclear Security Administration and the DOE National Security Agency. Proposed statement of work: SGS file system, Apr. 2001.

[16] 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.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003. ACM.

[18] 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.

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

[20] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3), Mar. 1988.

[21] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Mateo, California, CA, USA, 1993.

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

[23] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. Technical Report CITI-05-1, CITI, University of Michigan, Feb. 2005.

[24] 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.

[25] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[26] IBM Corporation. IceCube – a system architecture for storage and Internet servers. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB_Hardware/.

[27] IBM Corporation. IBM white paper: IBM storage tank – a distributed storage system, Jan. 2002.

[28] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer 1990 USENIX Technical Conference*, pages 151–164, 1990.

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

[30] D. Kotz and R. Jain. I/O in parallel and distributed systems. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 40, pages 141–154. Marcel Dekker, Inc., 1999. Supplement 25.

[31] D. F. Kotz and N. Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Parallel and Distributed Technology*, 3(1):51–60, 1995.

[32] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[33] R. Latham, N. Miller, R. Ross, and P. Carns. A next-generation parallel file system for Linux clusters. *LinuxWorld*, pages 56–59, Jan. 2004.

[34] Lawrence Livermore National Laboratory. ASCI linux cluster. http://www.llnl.gov/linux/alc/, 2003.

[35] Lawrence Livermore National Laboratory. IOR software. http://www.llnl.gov/icc/lc/siop/downloads/download.html, 2003.

[36] 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.

[37] D. Long, S. Brandt, E. Miller, F. Wang, Y. Lin, L. Xue, and Q. Xin. Design and implementa-

tion of large scale object-based storage system. Technical Report ucsc-crl-02-35, University of California, Santa Cruz, Nov. 2002.

[38] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison–Wesley, 1996.

[39] 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.

[40] 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.

[41] 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.

[42] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8), Aug. 2003.

[43] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing '91*, pages 567–576, Nov. 1991.

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

[45] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

[46] 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.

[47] 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.

[48] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. Technical Report 91-3, University of Michigan Center for IT Integration (CITI), Aug. 1991.

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

[50] A. L. Narasimha Reddy and P. Banerjee. A study of I/O behavior of perfect benchmarks on a multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 312–321. IEEE, 1990.

[51] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

[52] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, Oct. 1996.

[53] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[54] B. K. Pasquale and G. C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, Portland, OR, 1993. IEEE.

[55] B. K. Pasquale and G. C. Polyzos. Dynamic I/O characterization of I/O-intensive scientific applications. In *Proceedings of Supercomputing '94*, pages 660–669. IEEE, 1994.

[56] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, Maastricht, Netherlands, May 2000.

[57] K. W. Preslan, A. Barry, J. Brassow, M. Declerck, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe. Scalability and failure recovery in a Linux cluster file system. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Oct. 2000.

[58] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)*, pages 165–172. IEEE Computer Society Press, 1995.

[59] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, Jan. 2000.

[60] O. Rodeh and A. Teperman. zFS—a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Apr. 2003.

[61] 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.

[62] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

[63] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119–130, 1985.

[64] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[65] 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.

[66] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[67] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 307–326, Jan. 1993.

[68] 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.

[69] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 49–59. IEEE, 1996.

[70] E. Smirni and D. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] D. Tang. Storage area networking – the network behind the server. Technical report, Gadzoox Microsystems, 1997.

[76] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004.

[77] 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.

[78] 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.

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

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

[81] B. Welch and G. Gibson. Managing scalability in object storage systems for HPC Linux clusters. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 433–445, Apr. 2004.

[82] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw. LegionFS: A secure and scalable file system supporting cross-domain high-performance applications. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*, Denver, CO, 2001.