Providing Quality of Service Support in Object-Based File System

Joel C. Wu Scott A. Brandt

Department of Computer Science
University of California, Santa Cruz
{jwu,sbrandt}@cs.ucsc.edu

Abstract

Bourbon is a quality of service framework designed to work with the Ceph object-based storage system. Ceph is a highly scalable distributed file system that can scale up to tens of thousands of object-based storage devices (OSDs). The Bourbon framework enables Ceph to become QoS-aware by providing the capability to isolate performance between different classes of workloads. The Bourbon framework is enabled by Q-EBOFS, a QoS-aware enhancement of the EBOFS object-based file system. Q-EBOFS allows individual OSDs to become QoS-aware, and by leveraging on the random element of the CRUSH data distribution algorithm employed by Ceph, it is possible for a collection of independent QoS-aware OSDs to provide class-based performance isolation at the global level. This preserves the highly scalable nature of Ceph by avoiding the introduction of any centralized components or the need to collect and propagate global state information. This paper presents the Bourbon framework by first describing Q-EBOFS, and then examines how a collection of OSDs running Q-EBOFS can work together to provide global-level QoS.

1. Introduction

Storage systems are becoming larger with the ever increasing demand for storage capacity. Storage systems must also handle different types of data objects, many of which have timing constraints [4]. In addition, different accesses to a storage system can have different relative importance. Large storage systems are likely to serve different groups of users and different workloads that have different characteristics and priorities. Without performance management, they compete with each other for bandwidth resource on a first-come first-serve basis, where less important bulk traffics may starve more important traffics. The ability to manage bandwidth as a resource can be beneficial in many situations as it enhances the usability of the storage system.

Ceph [24] is a peta-scale object-based storage system designed to provide high scalability and reliability. This paper examines how quality of service (QoS) support can be added to Ceph, giving it the capability to partition performance amongst different classes of users or workloads (class-based performance partitioning/isolation). The unique architecture of Ceph presents challenges to QoS provisioning unlike that of traditional distributed storage systems. Namely, a file in Ceph is broken up into objects, hashed into placement groups (PG), and then pseudo-randomly distributed to object-based storage devices (OSD) using the CRUSH algorithm [25]. The QoS mechanism in Ceph must accommodate this striping and randomized distribution of data.

Most existing QoS mechanisms for storage system can throttle data traffic for a single logical storage node independently [11, 17, 28]. Some QoS mechanisms for distributed storage can provide performance management at the global level [3], but require centralized components for the gathering and processing of global state information. One of the main design goals of Ceph is extreme scalability. The performance of Ceph scales nearly linearly with the number of OSDs [24]. The main objective underlying the design of Bourbon is that a QoS mechanism should not hamper the scalability of Ceph, and the additional complexities introduced should be minimized. We take a two step approach toward the issue of providing QoS support for Ceph. First, we want to enable individual OSD to become QoS-aware by giving them the ability to shape disk traffic. Second, we investigate how a collection of QoS-aware OSDs working independently can provide global level QoS.

Bourbon is enabled by Q-EBOFS, a QoS-aware object-based file system intended to run locally at the OSD. It is developed from the EBOFS file system originally created by Weil [24]. Q-EBOFS can shape disk traffic to provide class-based bandwidth partitioning at each of the OSDs. By leveraging on the properties of CRUSH, we then use simulations to show how a collection of OSDs running Q-EBOFS can work together to provide global-level QoS without requiring the use of global state information.

The rest of this paper is organized as follows. Section 2 gives an overview of the Ceph object-based file system and its implications on QoS. Section 3 presents the design and implementation of Q-EBOFS. We then discuss achieving global-level QoS in section 4. Related works are presented in section 5, and conclusions in section 6.

2. The Bourbon QoS Framework

This section presents an overview of the Bourbon QoS framework. We first give a brief background description of the Ceph object-based file system that forms the context of of this work, and then outline the objective and design of the Bourbon framework.

2.1. Ceph Object-Based File System

Object-based storage [12] is an emerging paradigm in distributed storage architecture with the potential to achieve high capacity, throughput, reliability, availability, and scalability. The main difference between the object-based storage model and the traditional distributed storage models is that object-based storage offloads the handling of low level storage details to the storage devices themselves—functions such as disk space management and request scheduling are handled by object-based storage devices autonomously. In addition, metadata management is decoupled from data management and clients are able to access the storage devices directly at the object level. A typical object-based storage system are composed of three main components: clients, metadata servers (MDS), and object-based storage devices (OSD). The client locates data by first contacting the metadata server, it can then transfer data directly to and from the OSD.

Ceph is an object-based storage system developed by Weil et al. [24] and is the system under study for this work. In addition to reaping the benefits of object-based architecture, Ceph is designed with extreme scalability in mind, both in its metadata server cluster [26] and its object-based storage devices. The OSDs in Ceph have peer-to-peer capability and manages replication and failure handling autonomously. The collection of OSDs and a number of highly scalable monitor nodes appear as a single large reliable store, referred to as the Reliable and Autonomous Distributed Object Store (RADOS) [22].

One of the key features of Ceph is its data distribution scheme. Ceph breaks a file into objects. The objects are hashed into placement groups, and placement groups are mapped to OSDs through CRUSH [25]. CRUSH is based on a hash function that pseudo-randomly distributes data across OSDs in a deterministic way. Any components in the Ceph system can determine the location of data from running CRUSH and having a cluster map without consulting any centralized entity. The Bourbon QoS mechanism accommodates and leverages on the unique features of Ceph.

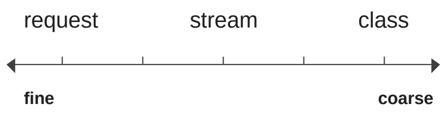


Figure 1. Granularity of assurance

2.2. Bourbon QoS objective

Mechanisms for storage QoS can make assurances at different levels of granularity as shown in Figure 1. On the side of finer granularity, real-time disk schedulers can make assurances for an individual disk request. A request arriving at the disk scheduler can be associated with a deadline, and the scheduler will ensure that the request is serviced before this deadline. Increasing the granularity, a stream (sequence of related requests) can be admitted and assured as a whole, usually with statistical guarantees. In general, finer grained assurances can be made if the requirement for a storage task is known *a priori*. For example, to stream a movie file at a rate corresponding to the frame rate, reservations can be made to the storage ahead of time. Class-based assurances are coarser grained assurances that attempt to divide the total bandwidth between different classes of requests.

In the context of Ceph, the type of assurance that we seek to provide is class-based performance isolation – the ability to assign bandwidth quota akin to the way storage space quota is assigned. The motivation for this stems from the scenario that large storage systems often need to support multiple unrelated workloads or multiple groups of users. For example, a high priority scientific application requiring high storage throughput may not run at the same time as applications generating bulk traffics such as backup or indexing, since the lower priority traffics will likely degrade the performance of the scientific application. But if we can assign fractions of the storage bandwidth to the two classes of workloads, they can both run at the same time without interfering with each other. Class-based performance isolation isolates different workload classes, but do not provide protection against intra-class interferences. Two different workloads within the same class may still interfere with each other. However, if a particular workload is of sufficient significant, it can always be defined as a class of its own. Also, hierarchical sharing [30, 6] can be applied to recursively allocate disk bandwidth if intra-class performance isolation is desired.

In Bourbon, the specification of the target share is through weight values. For example, in a two class scenario with class A and class B, if class A is to receive 80% of the total bandwidth and class B is to receive 20% of the total bandwidth, the weight of class A would be `.8` and the weight of class B would be `.2`. Since Ceph's performance scales nearly linearly with the number of OSDs, knowing the performance of each OSD allows the translation between weight value and the actual bandwidth.

The performance of disk drives are stateful. The service time for a request not only depends on the location of the data, but also depends on the current location of the disk head. Disk performance varies according to the pattern of requests and even the zone on the disk [20]. Exacerbating this issue, disk drives have increased in intelligence and the onboard buffer also increased in size. In addition, large distributed storage have multitudes of resources that can have potential effect on performance. The data path from a client to the disk goes through a multitude of buffers and different caching schemes, and the network link and switches also play a role. True end-to-end QoS requires multi-resource scheduling. This work represents our initial attempt at addressing QoS issues for large storage systems. We make the assumption that the disk drives are the performance bottleneck of the system. Other resources such as the interconnect network are outside the scope of this paper but are parts of the future works1112 plan.

2.3. Bourbon Design

A fundamental requirement for QoS-aware storage system is the ability to differentiate and throttle disk traffic. Bourbon achieves traffic differentiation through client authentication and request tagging. In Ceph, a client must first contact the MDS for authentication and obtain credentials with capabilities that allow it to access the OSDs.

One way to use the QoS mechanism is to associate one or more classes with the client capability. For example, a client running high-priority scientific application can receive the capability to issue requests belonging to a high priority class (more bandwidth share). The same client may also receive capability that allows it to issue requests belonging to other lower priority classes, as it is conceivable that the same client or application may generate both critical and non-critical requests. A client is given the capability to generate requests belonging to one or more classes. Determining which class that a particular request belongs to (and be tagged with) is the responsibility of the clients and its applications, we do not consider this further in this paper. The QoS class is embedded in a data structure we refer to as a *pass*. When the client accesses the OSDs, it presents the pass along with the request, and the request will be treated accordingly. The number of valid classes are defined by administrator and can be changed dynamically. The information on classes are distributed to OSDs by piggybacking on the cluster map. The cluster map is a data structure used to describe the OSD cluster [25].

The ability to throttle disk traffic is the responsibility of each individual OSD. However, with the way that data is striped across OSDs, we must also consider how we can achieve sharing at the global level. An intuitive way to achieve global level QoS is through monitor and react. The global performance statistics experienced by each class can be collected and analyzed, and based on the statistics, the throttling mechanism can be adjusted to steer the actual global sharing toward the target sharing. However, global state information are needed in order to control the throttling. As mentioned in earlier section, Ceph distributes data pseudo-randomly over all OSDs in the system to achieve a statistical balance of workload over all OSDs. Our hypothesis is that if the CRUSH algorithm does a sufficiently good job of distributing the workload, each OSD working independently enforcing the target share will result in the target share being enforced at the global level. The rest of this paper will describe Q-EBOFS and global level sharing in detail.

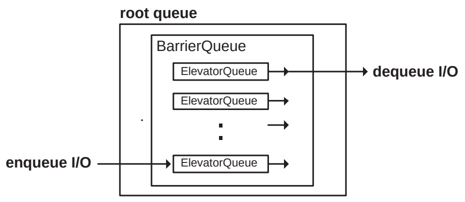
3. Q-EBOFS

Bourbon relies on the underlying QoS support provided by OSDs. This is done by making the local file system on the OSD QoS-aware. The modular architecture of Ceph allows for different object-based file systems to be used. In the context of Ceph, the current file system of choice to run on OSDs is EBOFS [24, 23]. EBOFS is an acronym for Extent and B-tree based Object File System, it encapsulates the block management details and exports an object interface to the upper layers. We enable the OSD to have the ability to throttle disk traffic by enhancing EBOFS. This section presents Q-EBOFS, a QoS enhanced version of EBOFS.

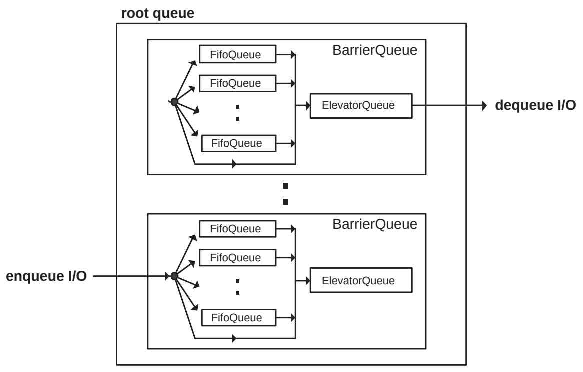
3.1. Queueing Structure

The EBOFS `read` and `write` calls are modified to accept an additional argument of type `pass`. The pass represents QoS credential and indicates which class the request belongs to. All requests coming from clients are pigeonholed into a class. Other top level calls such as `stat` and `rm` are also augmented with the `pass` parameter as they may generate read/write operations to the disk. For requests that arrive with invalid pass credential, they will be put into a default class (class 0).

Q-EBOFS consist of a buffer cache layer on top of a block device driver layer. The enforcement of sharing is



(a) Original queues in EBOFS



(b) Enhanced queueing structure in Q-EBOFS

Figure 2. Enhancing the queues in EBOFS for QoS

done at the block device driver's request queues, before requests are dispatched to the disk. In Q-EBOFS, there is a FIFO queue for each QoS class and a single dispatch queue (elevator queue). As requests arrive, they are sorted into their respective FIFO queue first. Requests from the FIFO queues are then selectively moved to the dispatch queue to enforce the sharing. Requests in the dispatch queue are arranged in elevator order to increase disk throughput. The requests are then dispatched from the elevator queue to the disk.

The sharing is enforced by how the requests are moved from the FIFO queues to the dispatch queue. The modular architecture of Q-EBOFS allows different schemes to be used here. Current implementation of Q-EBOFS uses weighted round robin (WRR) for proportional sharing. For example, a 20% - 80% sharing between two classes would entail the dequeue of one request from the first FIFO queue for every four requests from the second FIFO queue. The dequeueing is work conserving, if the next FIFO queue to dequeue from is empty, the algorithm will simply move on to the subsequent FIFO queue without waiting.

While most metadata requests can be associated with a class and sorted into the appropriate FIFO queue, there are some file system originated requests that can not be associated with any particular class. These requests are inserted directly into the dispatch queue. Although we do not expect these requests to impact the effectiveness of the QoS mechanism under typical usage, a separate FIFO queue can be set up for them to provide more control and reduce interference if it becomes an issue.

Modern disk schedulers that rearrange requests must also support the barrier function. A barrier request ensures that all requests sent before the barrier will be completed before requests after the barrier. They are useful to support upper layer functionalities such as journaling file system that requires requests to be committed to disk in certain order to ensure integrity. EBOFS supports barrier functionality to prevent starvation from occurring in the elevator queue, and Q-EBOFS must also preserves the barrier functionality in the same way as originally implemented by Weil [24, 23]. The queueing structure (multiple FIFO queues and an elevator queue) described in the previous paragraph are actually encapsulated within an abstract barrier queue. A Q-EBOFS file system has a single queue as viewed from above (buffer cache) and below (disk), referred to as a root queue. A root queue is composed of a list of barrier queues. When requests are added to the root queue, they are added to the barrier queue at the tail of the list. When requests are dequeued from the root queue, they are dequeued from the head of the list. When a barrier request arrives, a new barrier queue is created at the end of the list. New requests are always added to the barrier queue at the end of the list, while requests going to disk are always drained from the barrier queue at the head of the list. Figure 2(b) shows the queueing structures of original EBOFS and Q-EBOFS.

3.2. Buffer Cache Management

The queueing structure where sharing is enforced is in the block device layer below the buffer cache. This works well for read requests. However, EBOFS handles all write calls asynchronously. As shown in Figure 3, a write call to EBOFS will return as soon as the data is in the buffer cache (in memory of the OSD) and the request is added to the block device driver's queue. A write to EBOFS will never block except when the buffer cache is approaching full. If the size of dirty buffer in the buffer cache exceeds a threshold value, write requests will be blocked while flushing takes place (trimming of the buffer cache). Once the size of dirty buffer has been reduced, the write will then be unblocked.

A consequence of this design is that throttling write requests at the queues in the block device driver will enforce the proper sharing going to the disk, but will not enforce the sharing as seen by entities above the buffer cache. Since a write will only be blocked when the buffer cache is full and the blocking occurs indiscriminately regardless of the class of the request, throttling below the buffer cache will change the composition of the buffer cache but not the way the sharing is viewed by the clients above the buffer cache.

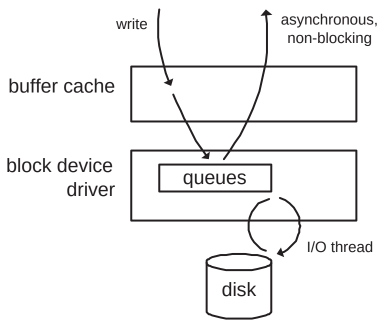


Figure 3. EBOFS Writes are asynchronous - once the data is in the buffer cache and the request placed in the queue, the write call returns immediately.

To enable the desired sharing to work for EBOFS writes, the buffer cache must be managed as well. Since the only blocking point in the write path is when the buffer cache is full, by selecting which request to block when the buffer cache is full, we can control and enforce the desired sharing. A simple way to achieve this is by dividing up the buffer cache space between classes according to the desired share. For example, if there are two classes with a sharing of 20% and 80%, we can split the buffer cache into a 20% and a 80% portion. Whenever the portion of a buffer cache is full, write request to that class will be blocked while flushing of the cache takes place to create space for the blocked request. This correlates the blocking (throttling) of requests for a class to the target share. While simple, this method has the drawback that a request may be blocked even if the cache as a whole still has sufficient space.

A more efficient approach than the strict partitioning of buffer cache is to allow a class to use more than its share of the space if sufficient space are available. This can be beneficial in situations where one class does not fully utilize its share. The spare capacity belonging to this class can be reallocated to other classes that is demanding more than their share. However not all spare capacity should be reallocated. Some spare capacity for a class not fully utilizing its share must be reserved to avoid penalizing them if the class started to have requests arriving and it must be blocked because the buffer cache is being used by another class.

Q-EBOFS allows a class to use any free buffer space until the amount of buffered data exceeds some high watermark. After that point, each class is limited to its reserved space and blocks if it is using more than that. This allows for greater utilization of the buffer cache. Q-EBOFS uses a simple algorithm, shown in Figure 4, to decide whether to block a write request. This simple algorithm decides

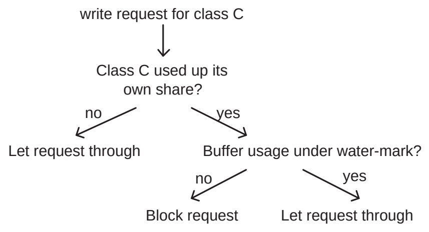

write request for class C
Class C used up its own share?
no
yes
Let request through
Buffer usage under water-mark?
no
yes
Block request
Let request through

Figure 4. Throttling writes at the buffer cache

whether to block a write request or not. It is the only potential blocking point in the write path. The selective blocking of requests achieves bandwidth partitioning for asynchronous writes through the management of buffer space usage.

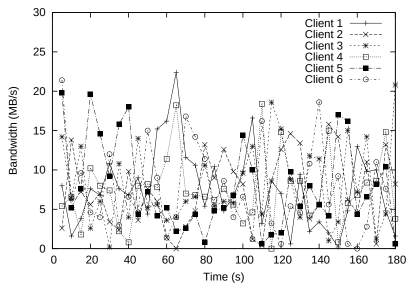
3.3. Using Q-EBOFS with other QoS mechanisms

In addition to being used as an integral part of Ceph, Q-EBOFS can also be used independently of Ceph, either by linking directly with the user program or access through FUSE [19]. The fact that Q-EBOFS (and EBOFS) are implemented in user space using C++ makes it modular and portable. A secondary goal of this work is to make Q-EBOFS a modular QoS-aware file system that can be used to test other QoS approaches for disk based storage. Developers can plug in their QoS mechanism without having to modify other parts. For example, the current Q-EBOFS implementation provides sharing through WRR and uses number of requests as unit of throttling. Methods other than WRR as well as other cost function (such as data size) can be used. The different algorithms can simply be plugged into EBOFS without extensive modifications to other parts of the file system.

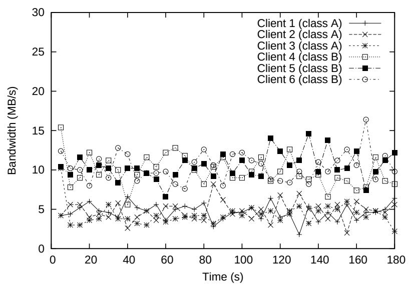
3.4. Experiments

This section presents the experimental results of Q-EBOFS. The purpose of these experiments is to validate the ability of Q-EBOFS's request throttling mechanism to differentiate and isolate traffics. The following experiments were performed on a PC with 1.8 GHz Pentium 4 processor and 512 MB of RAM. The disk is a 80 GB Maxtor DiamondMax STM3200820 formatted with Q-EBOFS.

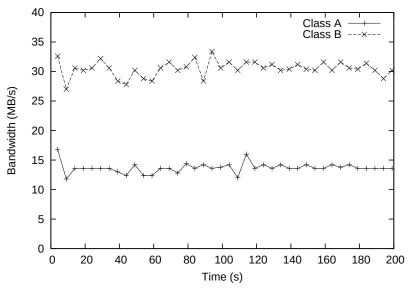
First, we examine the performance of original EBOFS without QoS enhancement. Figure 5 shows an experiment with 6 clients writing simultaneously to EBOFS. All the clients are greedy, they issue writes as fast as they can. Without any performance isolation mechanism, the resulting bandwidth received by each client through time is plotted in Figure 5(a). Each client receives approximately the same performance – an average of 7.69 MB/s per client,



(a) Original EBOFS with no bandwidth isolation



(b) Q-EBOFS with two classes, class A with 30% and class B with 70%



(c) Aggregate throughput of the two classes

Figure 5. Q-EBOFS allocating bandwidth between different classes

with the minimum of 7.22 MB/s and the maximum of 8.1 MB/s. The aggregate throughput of all clients is 46.13 MB/s.

To show the result when using the throttling mechanism, we set up two different classes with a sharing of 30% and 70%. Class A receives 30% of the bandwidth and class B receives 70% of the bandwidth. Clients 1, 2, and 3 are allocated to class A, and clients 4, 5, 6 are allocated to class B. The result is shown in Figure 5(b). It can be seen clearly that the three clients of class A receive lower bandwidth than the three clients of class B. This shaping of the writes are the effect of the buffer cache management scheme. Figure 5(c) plots the aggregate amount of data transferred for the entire class. The amount of data transferred over the experiment's run time for the two classes are 31% and 69%. Sufficiently approximating our target share.

Next we show the throttling mechanism on reads. In this experiment we have 20 clients reading from EBOFS. In the original EBOFS without QoS mechanism, the result is shown in Figure 6(a). Because of the large demand, each client receives only around 1 MB/s, the data points are overlapped and appear as a single line in the figure. Next we run the same experiment on Q-EBOFS with a two class setup of 20% and 80%. The result for the aggregate throughput of the classes is shown in Figure 6(b), and it matches the 20%-80% target share.

Figure 7 shows an experiment mixing reads and writes. We define two classes with sharing of 25% for class A and 75% for class B. There are four clients in each class. Each client issues mixed read and write requests. The read/write distribution has a read percentage set to 68%. In this case both the queues and the buffer management mechanisms contributes to the shaping of the bandwidth. The figure plots the aggregate throughput each class receives. Class A receives 25.8% of the bandwidth and class B receives 74.2% of the bandwidth, closely matching the target share.

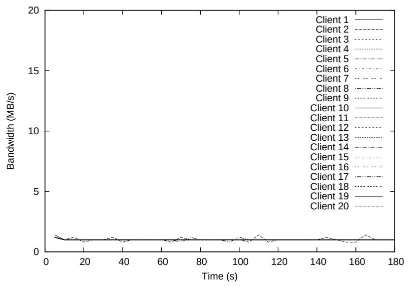
4. Global-level QoS

We have shown how Q-EBOFS can allocate fractions of disk bandwidth to different classes of workloads at each OSD. However, in Ceph, a file is broken up and striped across OSDs pseudo-randomly through CRUSH. Any piece of data can potentially end up on any OSD. There is no correlation between a particular client or class to any particular OSD due to the random element in CRUSH. We now return to examine the question of how independent OSDs can work together to provide global-level QoS.

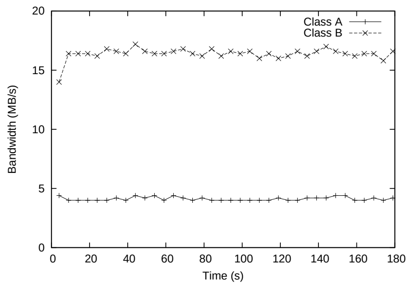
4.1. Global level class-based sharing

In the context of this work, global level class-based sharing implies that the storage system, although made up of large numbers of distinct underlying storage devices, can allocate bandwidth to different classes of data traffic as a whole. The global level sharing are not be visible at an individual OSD or client, but are visible when we consider the entire storage system as a whole. Recall the concept of RADOS, where the collection of OSDs function and behave as a large single logical store. Our goal of class-based performance isolation is to allow the bandwidth of this single large logical store to be allocated to different classes of workloads (Figure 8).

Existing QoS mechanisms providing global-level QoS support require the use of global state information. Typically some centralized components are involved, or the global state information needs to be propagated to each storage node for coordination. One of the design goals for Ceph is extreme scalability. The introduction of centralized component(s) or the need to propagate global state information creates potential bottlenecks that may hamper scalability. Even well designed and highly scalable centralized management entities can undesirably increases the complexity of the system. We consider simplicity and scalability as highly desirable to preserve the elegance of Ceph as it is.



(a) Original EBOFS with no bandwidth isolation (lines overlapped)



(b) Q-EBOFS with two classes, class A with 20% and class B with 80%. Aggregate throughput of each class is plotted.

Figure 6. 20 clients reading from EBOFS

In Ceph, because of the random element in how CRUSH distributes data, device load is on average proportional to the amount of data stored [25]. Randomized data distribution has the property of creating a probabilistically balanced distribution, where on average, all devices will be similarly loaded. This remains true even across different types of workloads [25], as different workloads would end up generates the same random distribution of requests [15]. Because of this property, an interesting question arises: if the random data distribution sufficiently disperses the data (and workload) onto all OSDs in the system as it is supposed to, if each OSD enforces the desired sharing independently, will the global sharing approximates the desired sharing? Figure 9 depicts this scenario. For example, if we set up two classes, A and B, where class A is to receive 25% of the global share and class B is to receive 75% of the global share. If each OSD enforces the 25%-75% share independently, will the resulting share observed at the global level also matches 25%-75%.

4.2. When sharing matters

Bourbon is intend to be work conserving. The OSD should not sit idle while there are requests in the queue. This is in contrast with other scheme where artificial delays are inserted to ensure global proportional sharing [21]. As noted in that work, sometimes it is not possible to be work conserving while maintaining global proportional

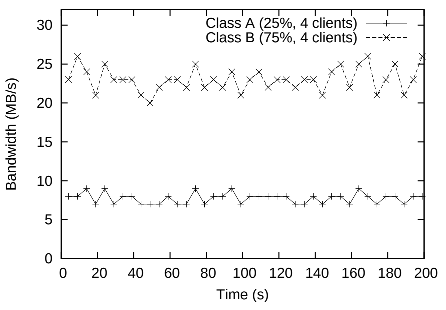


Figure 7. Mixed read/write

sharing. In Bourbon, the allocation of bandwidth between classes should approximate the ideal Max-Min fair share with weights [8]. A consequence of being work conserving is that the actual sharing will match the target sharing only when all classes demand more than their share.

In the following description, we consider the system to be overloaded when the supply of bandwidth exceeds aggregate demand from all classes; and a class is satisfied when its demand is being met.

When system is not overloaded. Then all classes are satisfied, and one of the following is true.
1. All classes are demanding less than their share.
2. Some classes are under-demanding and some classes are over-demanding. The aggregate slack from under-demanding classes are greater than the aggregate over-demands (demand beyond share) from over-demanding classes.

When system is overloaded. One of the following is true.
1. All classes are over-demanding.
2. Some classes are over-demanding and some are under-demanding. The aggregate slack from under-demanding classes are less than the aggregate over-demands from over-demanding classes.

As we can see, being work conserving, there is only one scenario in which the observed sharing will match target sharing – when all classes are demanding more than their share. In other scenarios, the actual sharing observed will not match the target sharing (except for coincidental cases). But not matching the target share is acceptable since all classes are satisfied (and desirable for the reason of work conserving). Therefore, the objective of Bourbon is not to maintain the target sharing at all time, but to allocate bandwidth according to the weights in such a way to (1) ensure a class is satisfied when its demand is not greater than its allocated share, and (2) allow a class to receive more than its share if slacks are available, and (3) ensure a class will receive at least its share if the class over-demands.

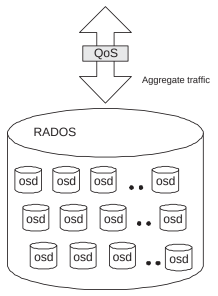


Figure 8. RADOS: A collection of OSDs functioning as a single logical store

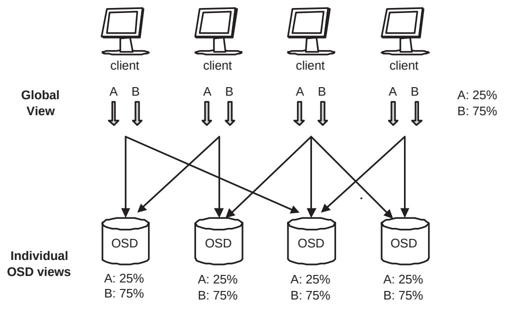


Figure 9. Global sharing and local sharing

4.3. Experiments

We implemented a simulator to study the use of independent QoS-aware OSDs to achieve global-level sharing. The simulator uses the source code for data distribution from Ceph (hash + CRUSH). Simulation is beneficial in this case because it enables us to focus on the question at hand and bypassing other irrelevant complexities and details. It also allows the exploration on the behavior of much larger systems that would otherwise not be possible. The simulator has client and OSD objects and simulates the performance of a large scale Ceph system. The stateful property of the disk is incorporated into the simulation by using a Gaussian distribution for service time [15]. We are examine the behavior from a higher level, the goal is not to mimic the behavior of a particular model of hard disk, but just to introduce variable service time into the experiments.

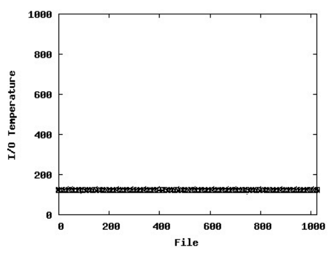
Recall that Ceph leverages on the random property of CRUSH for data distribution, and on average all OSDs should be similarly loaded. In a distributed storage system, more balanced load will lead to better performance, and in the case of Bourbon, more chances that individual OSDs will be able to enforce the target sharing. Since Q-EBOFS is work conserving, it needs a pool of requests from different classes to pick and choose from. If the workload is sufficiently dispersed across all the OSDs, an OSD will receive requests from all classes, and with sufficient backlog, should be able to attain the target sharing.

To test how well CRUSH can distribute workloads, consider a scenario with 1024 files of 1 GB each, where each file has a different loading. We use I/O temperature [9], a concept that quantifies (unit-less) how ”hot” a file is, to represent how heavily loaded a file is. We consider three different scenarios: (1) when all files have the same I/O temperature, (2) when half of the files have the same I/O temperature and the other half are inactive (I/O temperature of 0), and (3) a highly skewed workload modeled by Zipf distribution. Zipf distribution and its derivatives are often used to model file popularity. The three distributions of I/O temperature over the set of files are shown in Figure 10. For each of the three sets of file loading distribution, the files are broken up into objects and its associate loads are distributed to 1024 OSDs with an object size of 1 MB. The resulting load distributed to the OSDs are shown in Figures 11. We can see that no matter how the load is distributed across the files, it is effectively dispersed across all the OSDs, resulting in OSDs with similar loadings (temperatures).

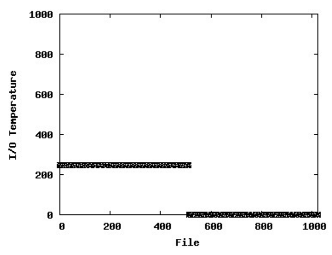
Next we want to test the same scenario with smaller files. The previous set of experiments uses a file size of 1 GB and an object size of 1 MB, so each file is broken up into 1024 objects. In the perfect scenario, each object would be placed on a distinct OSD, since there are a total of 1024 OSDs. When the data distribution is approximating normal distribution, the loading should approximate normal distribution as well. In the next experiment, we reduced the file size to 32 MB. Therefore each file will only have 32 objects to stripe across 1024 OSDs. The resulting loads on the OSDs should be more skewed. The result is shown in Figure 12. As expected, the result for the Zipf distribution of file loading has a larger standard deviation. However the overall loads across the OSDs are still relatively even.

To study the behavior in large systems with heavy loading, consider a scenario with 0.5 million files and 5000 OSDs. Each file is 1 GB and the object size is 1 MB. We use the three different file loading distributions shown in Figure 13(a), 13(b), and 13(c). The resulting distribution across OSDs are shown in Figure 13(d), 13(e), and 13(f). As shown in the figures, when the loading is heavy, the resulting loads on the OSDs are virtually identical for the three different file loadings.

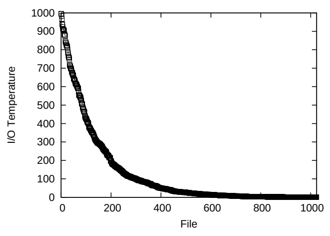
Now, we want to see how local sharing would add up to global sharing. Figure 14 shows the result compiled from multiple sets of experiment runs. Each data point in the figure is the result of a simulation run. In this figure, the X-axis is the total load (demand) placed on the system as a percentage of the total capacity. The Y-axis is the load of each class, also as a percentage of the total capacity. There are two classes, A and B, set up to receive 25% and 75%



(a) File: Constant

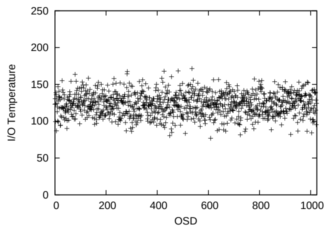


(b) File: 50%-50%

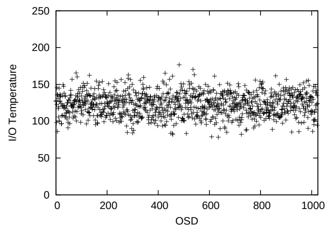


(c) File: Zipf

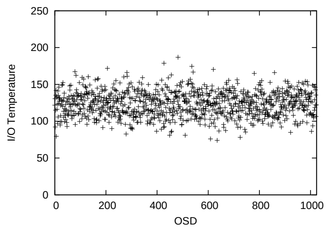
Figure 10. Distribution of workloads across files



(a) OSD: Constant (stddev: 15.51)

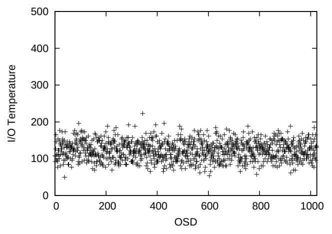


(b) OSD: 50%-50% (stddev: 15.86)

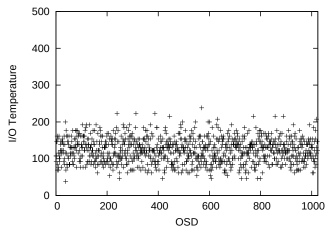


(c) OSD: Zipf (stddev: 16.91)

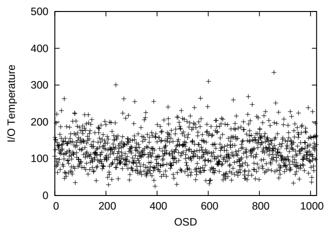
Figure 11. Resulting distribution of workloads across OSDs. 1024 files, 1 GB file size, 1 MB object size, 1024 OSDs



(a) OSD: Constant (stddev: 16.35)



(b) OSD: 50%-50% (stddev: 33.30)



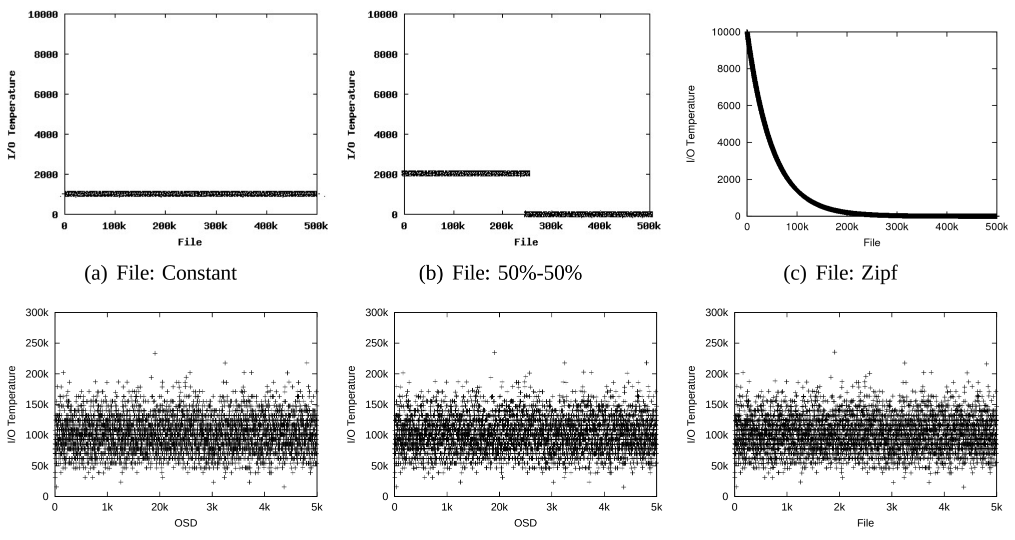
(c) OSD: Zipf (stddev: 46)

Figure 12. Resulting distribution of workloads across OSDs. 1024 files, 32 MB file size, 1 MB object size, 1024 OSDs

(the dotted horizontal lines) of the total capacity. There are 1000 clients and 1000 OSDs, with about 100 placement groups for each OSD [24]. The experiments were ran at each loading level with increments of 5.

As Figure 14 shows, when the load on the system is less than 100%, both classes are satisfied. They both receive what they demanded. Although the Q-EBOFS at OSD enforces the 25%-75% sharing, the work conserving nature of Q-EBOFS enables all requests to be served. After the load passes beyond 100%, the system can no longer satisfy all requests. We see the bandwidth received for class A started to drop, while the bandwidth for class B continues to increase as it remains satisfied. This trend continues until the bandwidth received for each class converges at the 25%-75% target share.

The loading placed on the system versus the utilization of the system corresponding with this result is shown in Figure 15. In a hypothetical ideal system, the utilization should closely follows the load, as all devices in the system should be perfectly balanced and equally busy (similar queue lengths, etc.). The degree of mismatch between the load and the utilization indicates unbalanced distribution in that some devices are overloaded while others have slacks. In the figure, the load on the X-axis is the demand placed on the system as a percentage of the system capacity. The Y-axis is the utilization of the entire system, derived from the utilization of all the OSDs. As the figure shows, the Ceph's data distribution component does an excellent job of distributing the workload across OSDs. When the load placed on the system is 100 %, the system is more than 90% utilized.



(a) File: Constant (b) File: 50%-50% (c) File: Zipf

(d) OSD: Constant (e) OSD: 50%-50% (f) OSD: Zipf

Figure 13. Distribution of workloads. 0.5 million files, 1 GB file size, 1 MB object size, 5000 OSDs

In the experiment shown in Figure 14, the two classes generates equal loads. In the next experiment, we show the scenario when the two classes generate unequal loads on the system. Figure 16 depicts this results. In this experiment, class B always demand 100% of the system capacity, while class A increases its demand gradually and linearly. As the figure shows, as the demand for class A increases, the demand for class B decreases, down to its allotted share of 25%, then it does not drop below that. Class A continues to increase at the expense of class B until class A reaches its allotted share of 75%, after which case the actual sharing matches the target sharing of 25%-75%. Figure 17 shows the load vs. utilization plot of this experiment. When the load is 100%, utilization is approximately 96.6%. Utilization approaches 100% when the load is about 160%.

These experiments show that Ceph does a sufficiently good job of distributing loads across all OSDs. When the system is not overloaded, the demands from all classes are satisfied. The effect of Bourbon is not visible. When the system is running at or above 80% load, then the effects of Q-EBOFS's throttling begin to show. Classes with lower target share started to become unsatisfied, while classes with higher target share remains satisfied. The actual bandwidth received by each class fully matches the target sharing when the system is about 170% loaded. At which point there are sufficient requests queued up at each OSD that the work conserving WRR traffic shaper in Q-EBOFS can always find a request from a class that it needs to dequeue from, therefore able to enforce the target share while being work conserving. The experiments confirm the properties outlined in section 4.2.

5. Related Work

QoS mechanisms for a single disk are most often found as disk schedulers. Motivated by the need to support homogeneous workload of storage-bound soft real-time applications [7], these type of schedulers associate a deadline with a request, and orders the requests based on some combination of real-time scheduling technique and disk seek optimization scheme (*e.g.* SCAN-EDF [13]). More elaborate disk schedulers (whether real-time or not), which take into account the seek time and rotational latency, require the use of accurate disk models [5, 29]. The use of such low-level information can provide predictable service guarantees [14], but limits their effectiveness to special-purpose systems due to the difficulties of obtaining and maintaining intricate knowledge of disk drive internals.

Some QoS-aware schedulers are designed specifically to address mixed workloads [4, 18, 27]. These are typically two-level schedulers that classify disk requests into categories such as real-time, best-effort, and interactive. Each class has its own scheduler, and the requests from different classes are merged by a meta-scheduler. YFQ [1] is a scheduler that offers similar capability to Q-EBOFS, using different mechanisms. Seelam [16] proposed a virtual I/O scheduler (VIOS) that can provide fairness and performance isolation. The per-class FIFO queues in Q-EBOFS are similar in concept to the application queues in VIOS. VIOS can optionally have two layers of schedulers.

QoS mechanisms can also be implemented above the disk scheduler and treating the disk as a black box [30, 11]. These mechanisms are interposed between the clients and the disks, they intercept and throttle disk requests for traffic shaping.

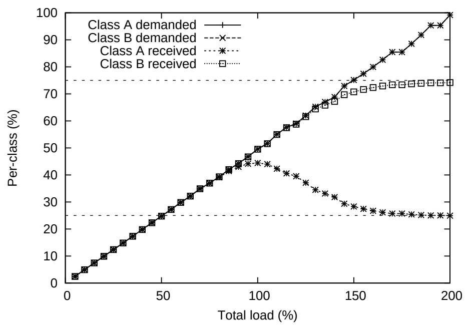


Figure 14. Demand vs. Receive

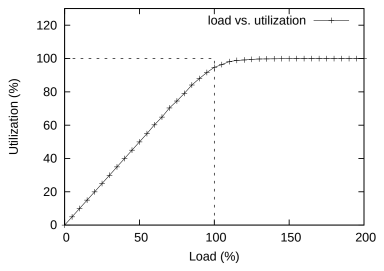


Figure 15. Load vs. Utilization

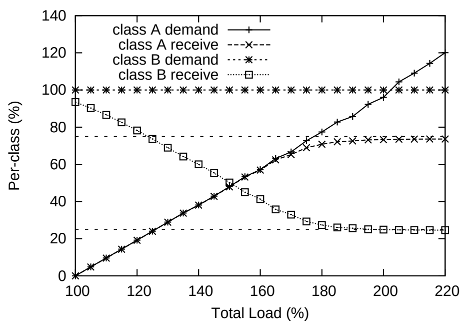


Figure 16. Demand vs. Receive

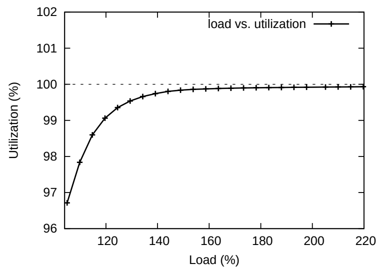


Figure 17. Load vs. Utilization

A number QoS mechanisms for distributed systems are designed to manage a single logical storage node [11, 17]. They do not have the mechanism to support global level QoS management across the entire storage system. Existing works in distributed storage that can achieve global level sharing relies on the use of global state information. SLEDS [3] uses special gateway devices interposed between clients and storage nodes. The gateway device intercepts requests and collect statistics. Based on the actual rates different classes are receiving, the gateway will throttled request streams to attempt to steer the actual sharing to match the desired sharing. Swift [2] is a distributed storage system that also stripes objects across different storage nodes. A QoS mechanism for Swift was proposed [10]. However, this mechanism differs in that it provides reservation through admission control for storage tasks that are known *a priori*, instead of global level sharing between classes where the loads may not be known in advance.

Wang [21] proposed an approach to achieve global level proportional sharing. This approach also relies on the feedback of global state information. The global state information is propagated to the storage nodes by piggybacking on normal requests to reduce the overhead. Based on the global state a storage node will insert extra delay to ensure proportional sharing holds at the global level. Bourbon differs in that it does not require any global state information.

QoS has been studied extensively in the domain of networking. However, techniques for network QoS can not be directly applied to storage, since the available disk bandwidth fluctuates due to variations in seek times. However, techniques in storage QoS management are often adapted from network QoS [21].

6. Conclusions

This paper presents Bourbon, a QoS framework designed for use in the Ceph object-based storage system. Bourbon is enabled by Q-EBOFS, a QoS-aware object-based file system running locally at the OSD. Q-EBOFS is a QoS-enhanced version of EBOFS, the current object file system of choice for Ceph. Q-EBOFS achieves performance isolation through class-based queueing and the management of buffer cache with selective blocking of asynchronous writes. Experiments showed that Q-EBOFS can provide class-based performance isolation between different classes of workloads. The Bourbon framework consists of a collection of QoS-aware OSDs running Q-EBOFS. Our experiments showed that being work-conserving, the actual share observed at the global level will not match the target share unless the system is heavily loaded (well beyond 100 %). However, this is desirable because when the observed share does not match the target share, demands from all classes are either satisfied or at least receiving their full target share. So although the actual sharing does not match the target sharing, the QoS goals are still being met. The target share at the global level will be visible only after the system is overloaded with a sufficiently large number of requests in the queues of the OSDs.

Currently, Bourbon (and Q-EBOFS) can only provide soft assurances and cannot be used to support real-time applications with hard deadlines. Future work will focus on using more rigorous methods to achieve tighter assurances, as well as formal analysis of the workload distribution.

Acknowledgments

We are grateful to Sage Weil, the author of Ceph and EBOFS, for assistance in all aspects of this work. This work would not be possible without his indispensable input and support. We would also like to thank Lawrence Livermore, Los Alamos, and Sandia National Laboratories; and the UCSC/Los Alamos Institute for Scalable Scientific Data Management (ISSDM) for support.

References

[1] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[2] L.-F. Cabrera and D. D. E. Long. Swift: A distributed storage architecture for large objects. In *Digest of Papers, 11th IEEE Symposium on Mass Storage Systems*, pages 123–128, Monterey, Oct. 1991. IEEE.

[3] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 109–118, 2003.

[4] Z. Dimitrijevic and R. Rangaswami. Quality of service support for real-time storage systems. In *Proceedings of the International IPSI-2003 Conference*, October 2003.

[5] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Diskbench: User-level disk feature extraction tool. Technical report, UCSB, November 2001.

[6] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[7] J. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, 1995.

[8] S. Jha and M. Hassan. *Engineering Internet QoS*. Artech House, 2002.

[9] G. karche, M. Mamidi, and P. Massiglia. *Using Dynamic Storage Tiering*. Symantec Corporation, Cupertino, CA, 2006.

[10] D. D. E. Long and M. N. Thakur. Scheduling real-time disk transfers for continuous media applications. In *Proceedings of the 12th IEEE Symposium on Mass Storage Systems (MSST 1993)*, April 1993.

[11] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *USENIX Conference on File and Storage Technology*, 2003.

[12] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–900, August 2003.

[13] A. L. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of ACM Conference on Multimedia*, pages 225–233. ACM Press, 1993.

[14] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*. IEEE, December 2003.

[15] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55, Santa Clara, CA, June 2000. ACM Press.

[16] S. R. Seelam and P. J. Teller. Virtual i/o scheduler: a scheduler of schedulers for performance virtualization. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE '07)*, pages 105–115, 2007.

[17] SGI. Guarantee-rate i/o version 2 guide, 2004.

[18] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS*, 1998.

[19] M. Szeredi. File System in User Space. http://fuse.sourceforge.net, 2006.

[20] R. Van Meter. Observing the effects of multi-zone disks. In *Proceedings of the USENIX Annual Technical Conference*, pages 19–30, January 1997.

[21] Y. Wang and A. Merchant. Proportional share scheduling for distributed storage systems. In *5th USENIX Conference on File and Storage Technologies (FAST '07)*, February 2007.

[22] S. Weil, C. Maltzahn, and S. A. Brandt. Rados: A reliable autonomic distributed object store. Technical Report SSRC-07-01, University of California, Santa Cruz, Jan 2007.

[23] S. A. Weil. Leveraging intra-object locality with ebofs. http://ssrc.cse.ucsc.edu/˜sage, May 2004.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006. USENIX.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *sc06*, Tampa, FL, Nov. 2006.

[26] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004. ACM.

[27] R. Wijayaratne and A. L. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 487–492, June 1999.

[28] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: storage performance as a managed resource. In *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 06)*, April 2006.

[29] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–156, 1995.

[30] J. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk scheduling for multimedia systerms and servers. In *Proceedings fo the ACM International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '05)*, pages 189–194, June 2005.