QoS support in Object-Based Storage Devices

Joel C. Wu Scott A. Brandt

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

Abstract

Quality of Service (QoS) is an essential aspect of any large storage system. The object-based storage model is an emerging distributed storage architecture that differs from the NAS and SAN storage models. The difference warrants a reexamination of how to provide QoS support under the new object-based model. In this position paper we examine the way in which QoS is achieved in existing distributed storage systems, outline the issues in providing QoS support in our object-based storage system, and describe the design of OBIS, a mode-changing disk scheduler that can provide the underlying QoS support for object-based storage devices.

1. Introduction

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 (OSD). In addition, metadata management is decoupled from data management and clients are able to access the storage devices directly at the object level.

We are developing *Ceph*, a petabyte scale object-based storage system with a target capacity of 2 PB and throughput of up to 1 TB/s in high-performance computing environments with tens of thousands of clients [17]. There are three major components in Ceph: the metadata server (MDS) cluster, the client interface (CI), and the storage managers (SM). The metadata server cluster is responsible for metadata operations such as file and directory management and authentication [18]. The client interface provides the file system API to applications on the client nodes. The storage manager encapsulates storage allocation, scheduling, and management within each OSD. To access data, the client first contacts the MDS cluster to authenticate itself and obtain the location of the data. The client then transfers data directly to and from the OSDs, resulting in a very high degree of data parallelism.

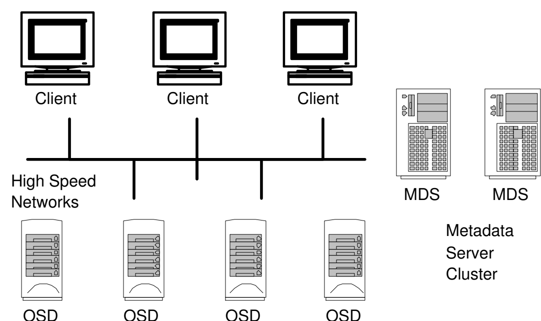

Figure 1. The object-based storage architecture of Ceph

In the context of Ceph, as well as any other large scale storage system, Quality of Service (QoS) is an essential issue. The storage system must be able to deliver satisfactory performance under expected (and unexpected) workloads. In addition, many scientific and visualization applications have timing constraints associated with their storage access [6] that benefit from performance assurances. Very large storage sys-

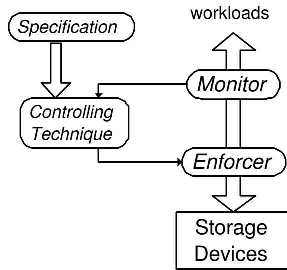

Figure 2. Throttling approach of the storage QoS model

tems are also likely to serve different groups of users and different workloads that have different characteristics and priorities. In such environments, it would be useful to have the ability to allocate performance (bandwidth) dynamically just as storage space is allocated. The virtualization of performance [8] is the next logical step after the virtualization of storage space.

Ceph's object storage architecture has given rise to a multitude of new QoS issues that require new solutions. In this paper, we first examine how existing distributed storage systems provide QoS. We present a generalized model that captures the essential aspects of providing performance assurance through feedback-based mechanisms. We outline the unique challenges that we are facing in the development of a QoS architecture for Ceph and outline the direction of our solutions and future research. We then propose the design of an object-based I/O scheduler called OBIS. It is a QoS-aware disk scheduler designed for use in OSDs. OBIS enables the OSDs to be QoS-aware and forms the basic building block in the overall QoS framework for our object-based storage system.

2. Storage QoS

Quality of service for storage encompasses many aspects of the system. The most common way of defining quality is in terms of the performance that the storage system can deliver. Many storage-bound applications, exemplified by multimedia streaming, must be able to retrieve data from storage in time for processing before the deadline of the task expires. QoS in this context implies the single dimension of bandwidth, or data rate, but QoS can have the additional dimensions of availability and reliability. Although important, these additional metrics are outside the scope of this paper and the rest of this paper will focus on quality of service in terms of performance.

A fundamental approach to assuring QoS for a distributed system is to analyze the expected workloads in order to adequately provision the system. If the initial provisioning becomes inadequate, reprovisioning may be necessary to ensure that the desired performance goals can be attained. The task of provisioning is not trivial; underprovisioning will not guarantee the desired performance while overprovisioning can be prohibitively expensive and wasteful of resources. Research exists in this area to assist the designers in automating the task [1]. While these tools attempt to minimize the amount of overprovisioning, they require accurate knowledge of the expected workloads and are slow to adapt to changes as new resources need to be incorporated.

In addition to provisioning, flexibility can be gained by having the ability to allocate performance by prioritizing workloads during system overload as well as preventing it from occurring in the first place. A number of distributed storage systems have been developed that can manage performance dynamically [11, 5, 10]. These systems work by throttling workload streams. This is essentially a feedback-based technique that allocates bandwidth based on a comparison of the current load condition with the desired load condition. The rate of a stream is ensured by throttling the rate of competing streams. This throttling approach can be simplified and generalized into a generic model with four major components: (1) *Specification* (2) *Monitor* (3) *Enforcer*, and (4) *Controlling Technique*.

Specification—The *Specification* mechanism allows the bandwidth requirements to be specified. It allows the desired quality level that a particular entity (client, group of clients, class of applications, etc.) is to receive and the system is to assure that the entity receives this minimum level of service. This is the agreement on quality that the system will attempt to satisfy. These values are usually specified by the system administrator. The process may or may not involve admission control wherein, for example, a saturated system will reject a new streaming request. SLED [5] uses average response time (IOPS,

bytes/sec) as its specification. Facade [11] uses a pair of curves specifying read and write latency as a function of request rate.

Monitor—The *Monitor* is the component that monitors the rate at which different entities are receiving service. Different parameters of the system can be monitored to extract the status of the system and derive the actions to be taken. Status such as queue length, average completion time, response time, as well as throughput can be monitored. SLEDS monitors I/O service time; Facade monitors a number of statistics including I/O arrival and completions, average latency, and read/write request arrival rates.

Enforcer—The *Enforcer* is the mechanism that shapes bandwidth by throttling resource consumption. To manage bandwidth by rate-limiting, the system must have the ability to place caps on overdemanding clients. Facade uses an I/O scheduler, and SLED uses leaky buckets to control the rate that SCSI commands can be issued.

Controlling Technique—The *Controlling Technique* decides how much and when to throttle the bandwidth of over-limit clients as well as how much and when to release the throttling. Triage [10] uses a control-theoretic approach; Facade and SLED uses heuristic-based approaches.

Figure 2 shows the components of this throttling-based approach. These systems work by allowing specifications to be associated with workload streams. The actual performance is monitored. If some entities are not receiving their desired bandwidth, the enforcer throttles back the bandwidth of competing over-limit entitites' streams in order to boost the bandwidth of the under-limit entities. The details of the throttling are determined by the controlling technique.

3. QoS issues in *Ceph*

QoS issues are closely intertwined with performance issues, and any component that can impact the performance should potentially be made QoS-aware. Instead of just maximizing the performance, capabilities should exist that allow the performance to be *managed*, with an overall framework that ties everything together. The following are the major issues we are facing and the directions we are taking in the development of Ceph. Many of the issues are interrelated and complementary.

End-to-end QoS assurance—An overall QoS framework must consider all of the components of the system in an end-to-end fashion [2]. At the very least both the interconnect network QoS and the underlying disk QoS must be considered. We are addressing this issue with simultaneous top-down and bottom-up approaches. In the top-down approach we start with user QoS goals and decompose the goals into sub-goals for the different underlying components. In the bottom-up approach we are constructing the QoS framework following the object-based storage methodology, we encapsulate the QoS complexity in the OSDs and use them as our fundamental QoS building blocks.

Specification—Facilities must be provided to enable the specification of QoS requirements. Research exists in the area of user specifications. The issue we are focusing on is the mapping of the user-level QoS requirements to system-level QoS requirements. We are taking the top-down approach described earlier to determine what individual system components must do to support the user-level QoS goals.

Load balancing—Ceph achieves redundancy by replicating objects across different OSDs. In the case of a read operation, if a single OSD cannot satisfy the timing constraints, higher performance may be achieved by splitting the read request across replicas of the same objects. In addition, to access a specific OSD, there may be multiple paths from the client to the particular OSD. If the link is the bottleneck, the data transfer may be balanced across multiple paths to spread the loads over the path, and also to increase the total transfer rate. The links may also have different performance characteristics.

Latency—While the throughput experienced by the clients are important, latency cannot be overlooked. Although the MDS cluster is decoupled from the data path, it still affects request latency, which includes both the responsiveness of the MDS cluster and the initial transfer by the OSDs. The number of MDSes in the cluster and its performance should be taken into account in the overall end-to-end QoS assurance.

Object replication and distribution—The Ceph object distribution algorithm [7] determines on which OSD each object is located. It distributes objects evenly across OSDs to avoid hot spots even as new

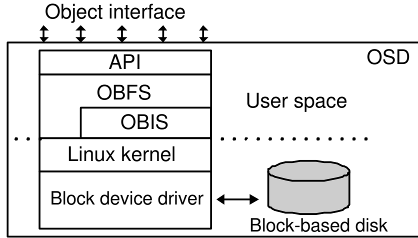

Figure 3. Internal organization of the OSD.

OSDs are added. QoS issues may be incorporated as a parameter to the distribution algorithm. Objects with timing constraints should be distributed widely to avoid concentration. Also, because of the scale of the system, it is likely that the OSDs will be heterogeneous in nature with varying performances. In this case it would be feasible to consider the performance characteristics of the underlying OSDs in determining the location of new objects and replicas. For example, primary copy of objects or objects with timing constraints could be placed on higher performance (or more reliable) OSDs while replicas placed on lower performance (or less reliable) ones.

Recovery—The fully distributed recovery mechanism [20] enables OSDs to replicate objects against possible OSD failure. The resulting recovery traffic may impact the load of involved OSDs as well as the network interconnect. The recovery traffic can be assigned lower priority to lessen the impact on normal traffic. However, the time to recover may be crucial for a critical recovery, in which case it would be more beneficial to assign higher priority to the recovery traffic.

Power-aware QoS—The very large number of disks involved in building Ceph brings the issue of energy consumption to the forefront. Power consumption is a pertinent issue for modern day computing installations. In fact at many sites the computing power is limited by the amount of electrical power that can be provided to the location and the difficulty of dissipating the resulting heat. Economic, environmental, and geographical issues exacerbates the problem. Considering the power characteristics brings a whole new dimension to the QoS framework. Power-aware object distribution, replication, and load balancing are all important issues that should to be addressed. This also brings up the interesting research question that if the power is limited and not all OSDs can be powered on at the same time, how can we prioritize and manage power.

4. Basic QoS Support in OSD

Now that we have examined how QoS is done with the throttling approach in existing distributed storage systems and outlined some of the high-level QoS issues in Ceph, we turn our focus toward how to enable basic QoS support in the Ceph system. One fundamental capability that all QoS-aware systems must have is the ability to shape disk bandwidth (the *Control* component in the model). All the higher level QoS goals (such as those described in Section 3) depend on having this ability, as those goals can be decomposed into why, when, how, which, and how much to shape the disk traffic. The rest of this paper focuses on how this capability can be provided by the OSDs, forming the fundamental building block in our overall QoS framework.

Each OSD is an intelligent and autonomous storage device that offloads disk block allocation and scheduling functions from the traditional storage server and provides an abstract data object interface to the client system software. All local storage issues are managed within the OSD. Each OSD consists of a CPU, network interface, local cache, and storage device (disk or small RAID configuration). For the Ceph system, we are currently focusing on building OSDs from commodity block-based hard drives.

In the Ceph system, each file is composed of one or more objects striped across different OSDs. The block-based hard drives in the OSD are managed by Object-Based File System (OBFS) [16], a small and efficient file system designed to optimize disk layouts based on the expected workloads of Ceph. For simplicity, our current implementation of OBFS is done in user space. Another version of the file system (EOBFS, or Enhanced OBFS) is implemented in the kernel. OBFS is designed for use within OSDs built with traditional block-based disk drives. OBFS bypasses the kernel page cache and manages the raw device directly. It uses a traditional elevator algorithm for the scheduling of disk requests.

An optional timing constraint can be specified at the file level, or it can be specified by the administrator on the basis of users, user groups, file type, or

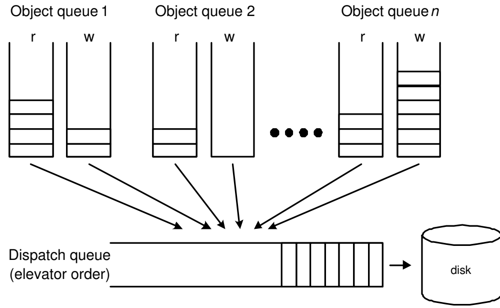

Figure 4. Structure of the object disk scheduler.

application type. As the file is broken up into objects, the object inherits the information in its *attributes* [3]. QoS requirements can either be none or in the form of $< p, b >$, indicating the object would require b amount of data in a time period of p. The data rate can be specified separately for read and write. For a read request, the OSD ensures that the data are available for delivery out of the OSD at the required rate for each client. For write requests, it ensures that it can consume the data at the required rate for each client. Objects without the timing specification are considered to be best-effort objects.

Upon query, the OSD can provide information to the MDS cluster on device capabilities and current load condition, as well as performing admission tests. The MDS can decide whether to use these information for its purpose. Admission control can be either enforced or advisory only; it is an option selectable by the MDS. If the MDS does not respect the admission control and places more load on the OSD than it can handle, the OSD will throttle traffics with its object disk scheduler (details to be described in later section).

The basic underlying QoS support is provided by the OSD's capability to assure rates. This capability is provided by QoS-aware disk schedulers. The next two sections describe disk schedulers in general and our QoS-aware disk scheduler design.

5. Disk Scheduling

The physical layout of a disk and its mechanical nature necessitates the use of disk schedulers in determining the order of I/O requests to dispatch to the disk. A simple FIFO ordering results in very low performance as the disk head would be moving all over the platter to service requests. The seek time of disk head is significant and is often the bottleneck in the data path. To minimize head movement, requests can be sorted in elevator order. This maximizes the total throughput at the expense of response time, as heavy activities at one area of the disk platter can starve requests at another area of the disk. Disk schedulers attempt to balance the two conflicting goals of low response time and high throughput.

5.1. Best-Effort Disk Scheduler

Version 2.4 of the Linux operating system uses the Linus elevator that is based on the traditional elevator algorithm. Throughput can be maximized at the risk of starvation if there are heavy activity in some area of the disk. In version 2.6, three new schedulers were introduced: the deadline scheduler, the anticipatory scheduler [9], and the complete fair queueing (CFQ) scheduler. The deadline scheduler is based on the elevator algorithm, but prevents starvation by assigning a deadline (deadline in the sense of aging) to each request. The scheduler will service requests in elevator order unless a requests' deadline expires, in which case it will service any request with an expired deadline first. The anticipatory scheduler is based on the deadline scheduler with the added heuristics of anticipation. When it finishes serving a request, it will not immediately race to the location of the next request, it would sit idle for a short time in anticipation of more requests in the immediate area being dispatched. The CFQ scheduler's goal is to ensures fairness by assigning an individual request queue to each process and using round-robin to serve them.

5.2. QoS-aware disk schedulers

In traditional direct-attached storage as well as distributed storage systems, QoS-aware disk schedulers are used to manage the disk bandwidth in order to provide QoS assurance. These schedulers must satisfy QoS assurance in addition to minimize response time and maximize throughput. They range from special purpose real-time disk schedulers [13, 14] to disk schedulers that can allocate bandwidth amongst different classes of applications [15, 4].

Pure real-time disk schedulers are designed for homogeneous applications such as video-on-demand. In such systems, each disk request is associated with a deadline. The requests are typically ordered based on some combination of real-time scheduling techniques and disk seek optimization schemes (e.g. SCAN-EDF). Some QoS-aware schedulers are designed specifically to address mixed workloads [6, 15, 19]. 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 combined and rearranged by a meta-scheduler. The main goal of these scheduler is to divide up disk bandwidth into different portions.

6. Object-Based I/O Scheduler

The desire for a new disk scheduler was motivated primarily by two factors. First, the layout of data on disk by OBFS is optimized for the object based storage model and differs from the layout of traditional file systems. Second, and equally important, is the desire to provide underlying QoS support in the OSDs. To address these two points, we are developing the Object-Based I/O Scheduler (OBIS). OBIS is intended for use in conjunction with OBFS to manage the access of raw block-based hard disks that serve as the physical storage component within OSD. The rest of this section describes the design of OBIS.

One common theme with many existing disk schedulers described in the previous section is that they all attempt to maximize throughput by arranging requests in elevator-order as much as possible, so long as the other goals of the scheduler are not violated. The *other goals* are objectives such as response time, fairness, or timing constraints. They are typically heuristics implemented to achieve the objectives of the particular scheduler. For example, the deadline scheduler functions as an elevator scheduler unless a request's age has reached the pre-set deadline. Real-time disk schedulers function as elevator schedulers constrained by the deadline of requests.

OBIS is a *mode-changing* disk scheduler. OBIS differs from existing schedulers in that it changes operational modes dynamically to handle different load conditions. The motivation behind the *multi-mode* design of OBIS is that different scheduling techniques have different strengths and weaknesses and are useful under different load conditions. As with the nature of benchmarking and performance evaluation, for any particular scheduler, there will be workloads that can bring the scheduler to its knees. OBIS is designed to operate under a number of different modes. It switches between the different modes according to the current load condition of the disk. A number of parameters are monitored to derive the necessary information in order to select the best operating mode.

Figure 3 shows the internal structure of an OSD with OBIS. For each object stored in the OSD, OBIS maintains two queues, a read queue and a write queue. Queues are located by their object ID via hashing. In addition to the object queues, there is a single dispatch queue. All incoming requests are sorted into the object queues first according to their object ID, then inserted into the dispatch queue in elevator order before being serviced. Conceptually, the insertion of requests into the dispatch queue is done by each object queue (which can be viewed as having individual threads of control). For example, if an object queue is limited to a rate of $< p, b >$, the object queue will try to move requests worth b amount of data to the dispatch queue every p period. If it is able to do this, the object queue is considered to be satisfied. Figure 4 shows the structure of the object disk scheduler. The sorting of incoming requests into different queues by object ID and the determination of which and how many requests from each object queue to move to the dispatch queue forms the basis of the bandwidth assurance framework.

The different operating modes of OBIS determine how requests are moved from the object queues into the dispatch queue. Each queue may or may not have timing requirement information as defined by the attributes of the object. The integration of OBIS and OBFS permits quick retrieval of object information. The scheduler monitors the status and the incoming and outgoing rates of the queues to transition between the different modes. Currently OBIS has four modes as shown in Figure 5.

Unlimited—This mode operates when the disk is not saturated given a set of requirements (when no object queue is unsatisfied). This mode is functionally equivalent to the traditional elevator algorithm with

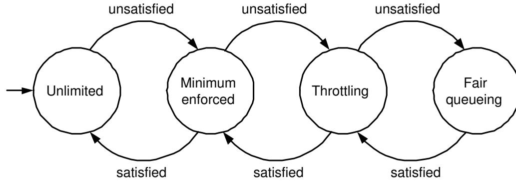

Figure 5. The QoS-aware object disk scheduler handles different load conditions by changing modes.

provisions to prevent starvation. It is similar to the deadline scheduler of Linux. In this mode, requests in the object queues are inserted into the dispatch queue in elevator order as soon as any request enters the object queue. If the age of any request in any of the object queues exceeds the threshold value, those requests would be moved to the dispatch queue immediately. The main objective is to maximize throughput while preventing starvation. The name *Unlimited* in this context indicates that no stream (a stream is a sequence of access to a particular object) is rate limited and the QoS mechanism is not needed.

Minimum Enforced—OBIS monitors the rate of requests entering and leaving each object queue as well as the dispatch queue. This mode is entered when some object queue is unsatisfied (is non-empty and its observed rate is below the required $< p,b >$ rate) for a predefined period. The objective is to limit all object queues with timing specification from going over their specified rate. Objects without timing specification are not limited.

Throttling—When minimum enforced mode can not remove unsatisfactory conditions at all of the object queues, this mode is entered. This mode rate limits all object queues with timing specification at their minimum rate just as in the minimum enforce mode, and the aggregate rate for all object queues without timing specification is throttled to remove unsatisfactory conditions. The throttling is heuristic controlled. The objective of this mode is to give time sensitive traffics more bandwidth by reducing the bandwidth of best-effort traffic.

Fair queueing—This mode is entered when the aggregate rate of object queues without timing specification has been throttled to its minimum but some object queues with timing specification are still unsatisfied. No more throttling would be feasible because of the potential for starvation. At this point the goal is graceful degradation while maintaining fairness. The scheduler employs Weighted Fair Queueing over all of the object queues and moves one request at a time into the dispatch queue. This mode is similar to the Complete Fair Queueing (CFQ) scheduler of Linux. If the queues are given priorities, a combination of throttling and fair queueing can be employed to guarantee some QoS constraints while gracefully degrading others.

The scheduler starts in *unlimited* mode and transition between the different modes depending on the conditions of the queues. During the transition from one mode to the other, there may be times when the total throughput is not maximized. As each object queue inserts its request into the dispatch queue (viewed as having its own thread of control), a separate thread round-robins over all the queues and moves one request from each queue into the dispatch queue. The goal is to keep the dispatch queue as full as possible in order to maximize the data transfer performance through low-level request reordering. The requests in the dispatch queue are removed and executed as fast as the device permits.

7. Conclusion and Future Work

In this paper, we presented a generalized QoS throttling model that allowed us to understand and compare different QoS approaches for traditional distributed storage systems. We outlined the QoS issues for object-based storage in the context of the Ceph system under development at the UCSC Storage Systems Research Center. We focused on how to provide basic QoS support in the OSD, and described the high level design of OBIS. OBIS is a QoS-aware disk scheduler for object storage device that differs from traditional schedulers in that it is a *mode changing* scheduler. OBIS operates in different modes depending on current load conditions. OBIS is intended to integrate with the OBFS file system and serves as an integral part of an overall QoS framework for object-based storage systems. We are currently implementing OBIS as a part of OBFS as well as developing the overall framework.

Acknowledgments

This research was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. We are also grateful for our sponsors Intel Corporation and Veritas Software. Thanks also to Feng Wang, Qin Xin, and Bo Hong for their helpful comments and suggestions.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, 2001.

[2] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Syst.*, 6(3):138–151, 1998.

[3] E. Borowsky, R. Golding, A. Merchang, L. Schreier, E. Shriever, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve qos. In *International Workshop on Quality of Service*, June 1997.

[4] 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.

[5] 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.

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

[7] 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.

[8] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[9] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[10] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. Technical Report HPL-2004-40, HP Laboratories, March 2004.

[11] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *USENIX Conference on File and Storage Technology (FAST'03*, 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] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.

[16] F. Wang, S. A. Brandt, E. L. Miller, and D. D. Long. OBFS: A file system for object-based storage devices. In *Proceedings of 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, April 2004.

[17] F. Wang, Q. Xin, B. Hong, E. L. Miller, D. D. Long, S. A. Brandt, and T. T. McLarty. File system workload analysis for large scientific computing applications. In *Proceedings of 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, April 2004.

[18] 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.

[19] 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.

[20] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 172–181, Honolulu, HI, June 2004.