End-to-end performance management for scalable distributed storage

David O. Bigelow, Suresh Iyer, Tim Kaldewey, Roberto C. Pineiro, Anna Povzner,
Scott A. Brandt, Richard A. Golding†, Theodore M. Wong†, Carlos Maltzahn
Computer Science Department, University of California, Santa Cruz
†IBM Almaden Research Center
{dbigelow,suresh,kalt,rpineiro,apovzner,scott,golding,tmwong,carlosm}@cs.ucsc.edu

September 29, 2007

1 Introduction

Many applications—for example, scientific simulation, real-time data acquisition, and distributed reservation systems—have I/O performance requirements, yet most large, distributed storage systems lack the ability to guarantee I/O performance. We are working on end-to-end performance management in scalable, distributed storage systems. The kinds of storage systems we are targeting include large high-performance computing (HPC) clusters, which require both large data volumes and high I/O rates, as well as large-scale general-purpose storage systems.

There are two main issues with performance management in such systems: sharing resources among competing users, applications, or tasks, and maintaining high performance. Sharing a computing cluster and its associated storage among multiple concurrent jobs is one example of resource sharing: each job should be able to work at full speed without interference from the others. When multiple jobs contend for the storage resource, it should be shared according to a specified policy such that each gets the performance it requires. This is in contrast with *ad hoc* schemes based on, for example, the number of requests issued, which may permit a single poorly behaved application to effectively overwhelm the system. The need for high performance comes from the need for storage to keep up with computation: storage is often a performance bottleneck and performance management must not in general come at the cost of performance itself.

We are concerned with end-to-end performance management of data as it moves through the system: from a client's cache, through the network, into a server cache, and onto disk (and *vice versa*). Rather than building independent pieces that do not work together or provide any way to reason about the performance that results when they are composed, we are developing integrated mechanisms that work together to provide overall performance, Our solutions are based on formal real-time principles, allowing us to prove and reason about the guarantees that our system will provide.

Our approach is to *reserve* performance for applications or users centrally, then *manage* the operation of each component in the system according to those reservations in a distributed fashion. Where users or applications care only about *best-effort* performance, the system works to provide the best performance possible within the available resources.

One possible objection to this approach is that application developers and/or system administrators cannot or will not develop performance specifications. We note that in practice people *do* generally think about and manage I/O performance—just in an *ad hoc* way, without support from the system. Every IT installation we know of does capacity planning when acquiring systems, benchmarks the systems before putting them into production to assure they meet the expected performance, and then monitors and tunes performance after installation. Similarly, HPC developers and system administrators routinely monitor and adjust storage performance, and in some systems specifically design the storage resource assignment to ensure that jobs do not interfere with each other (if only by staging data onto and off of cluster-attached storage). Our approach leads to a storage system that has support for performance management built in, ensuring that the mechanisms at each step of the data path work together to maintain the required I/O performance while providing the best overall performance possible.

There are multiple ways to make use of performance management in a large distributed storage system, some of which can work with unmodified applications. One way to introduce performance management in HPC clusters is through the job scheduling system, which can make

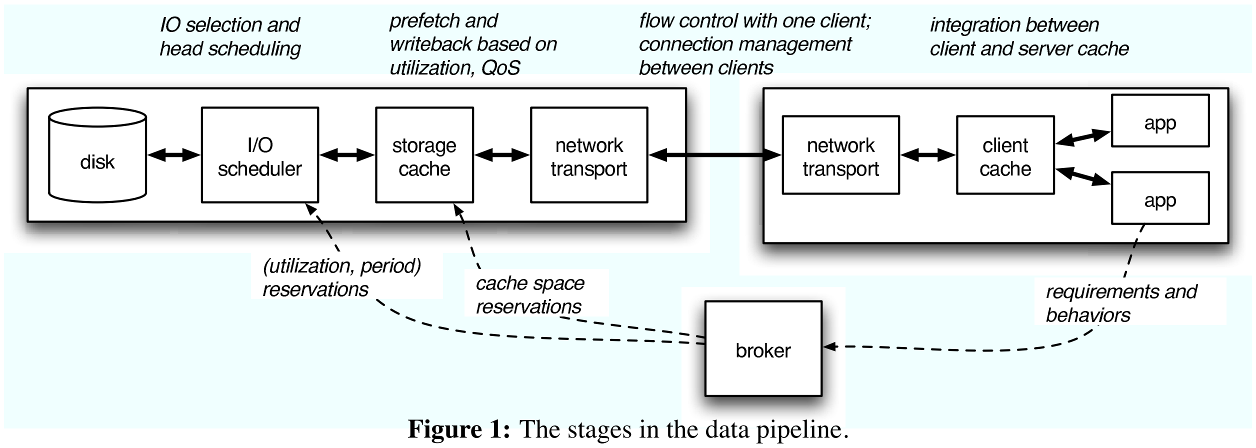

IO selection and head scheduling
prefetch and writeback based on utilization, QoS
flow control with one client; connection management between clients
integration between client and server cache
disk
I/O scheduler
storage cache
network transport
network transport
client cache
app
app
(utilization, period) reservations
cache space reservations
requirements and behaviors
broker

Figure 1: The stages in the data pipeline.

reservations on behalf of an unmodified compute job. Another way is through virtual machines, which can be provisioned with a certain level of storage performance (*i.e.*, a *virtual disk*) on behalf of the software running within the VM.

At the same time, some applications are directly concerned with performance and can benefit from a system that supports performance reservations. Data acquisition and media streaming applications both need to write or read data to storage at a given sustained rate in order to work properly. System housekeeping operations such as backup and data recovery need to proceed at a minimum rate to get their job done within a time window, but without interfering with other concurrently executing applications.

In this position paper, we discuss our research into mechanisms for end-to-end performance management in our Ceph scalable file system. We discuss how to transform application-level requirements and behaviors into I/O reservations and how to guarantee the reservations in each component in the data path from the client to the server and back, including the client cache, the network, the server cache, and the disks.

2 System architecture

Our system is being developed in the context of the Ceph distributed object-based storage system [2]. Ceph is a distributed file system that provides both high performance and scalability. Ceph decouples data and metadata operations, allowing each to be served by specialized clusters and allowing clients direct, high-performance parallel access to storage nodes. Ceph files are striped across *objects*, each of which is stored on a storage device. Ceph's storage devices export a high-level object interface and locally manage low-level storage management including block allocation and request scheduling. EBOFS, the extent-based object file system, runs on each Ceph storage device, providing high-performance object I/O with significant intra-object locality (*i.e.*, few intra-object seeks).

Ceph's overall architecture is representative of many modern distributed file systems. Clients open a file by talking with a metadata server cluster responsible for managing the file system namespace, permissions, and other file metadata. If the requested operation is permitted, clients are told which storage devices contain the data of the specified file. Clients then perform I/O directly with the storage devices containing the regions of the file they wish to access. The tightly coupled cluster of metadata servers maintains a coherent unified file system namespace and directory hierarchy. Object storage is handled by a decentralized set of storage devices, allowing a very high degree of parallelism between clients accessing different files or regions of the same file and the potentially thousands of storage devices containing them.

Adding performance management to Ceph involves managing five key aspects of the data pipeline from the client to the storage devices (Figure 1):

1. *The I/O scheduler* on each storage device. Guarantees on disk I/O are challenging due to their non-preemptivity, stateful and partially non-deterministic response times, and the orders of magnitude differences between best-, average-, and worst-case response times.

2. *The server cache* on each storage device. Providing a buffer between the network (and, further upstream, the client cache) and the disk, the server cache buffers writes destined for the disk and buffers read data prior to transfer to the client.

3. *The network connection* between an individual client and a storage device with which it is performing I/O. An individual network connection must transfer data between the client cache and the server cache so as to

guarantee the overall I/O performance without overflowing either buffer or unnecessarily stalling the application.

4. *The set of concurrently active network reservations* between potentially hundreds of thousands of clients and tens of thousands of storage devices. The aggregate of all reservations in the system cannot exceed the aggregate bandwidth of the network, nor can the demand on any single link exceed its capacity.

5. *The client cache* on each compute or I/O node. The client cache provides the first layer of performance management, buffering client writes, storing data from client reads, and interacting with the network and server cache managers to coordinate transfer of data to and from the storage devices.

3 Reservations

We define a "performance reservation" simply and generally as a guaranteed portion of the available resources in each component of the I/O system that ensure a certain overall level of performance to an application's I/O requests. We are investigating whether reservations should be hierarchical or not, whether they should be associated with users, sets of files, jobs, or something else, how persistent they should be, and so on. We ignore these questions in this discussion and talk in general about reserving performance for "applications," whatever they may be.

Our reservation mechanism has two goals:*generality* and *device-independence*. Applications need to make reservations in terms meaningful to them, recognizing that different applications may specify their needs in different ways. For example, a real-time data acquisition application may need to write some amount of data every period, while a transaction-processing system may need to perform a minimum number of transactions per minute. And reservations should not depend upon applications having detailed knowledge of the devices, since non-trivial scalable systems are generally composed of heterogeneous populations of storage devices.

We separate the reservation problem into two parts: *translation* of application requirements into a common representation and *admission control*—testing for feasibility and permissability and allowing or denying the reservation—on the basis of that common representation.

Based on the RAD model [1], our reservations have two components: a *budget b* specifying an amount of resource required and a *period p* specifying a time granularity with which it is required, *e.g.*10 Mbit/minute or 100 Gigabytes/hour, *etc*. Given knowledge about the device serving the reservation, the system can translate this

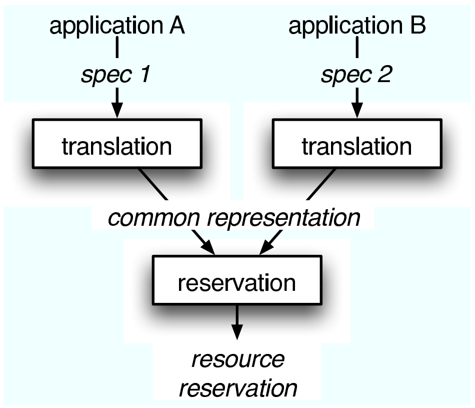


Figure 2: Structure of the broker for translating application requirements into resource reservations.

reservation into a required *utilization* $u = b/p$ and a set of *deadlines* d at which that utilization must hold, each one period apart. Explicitly specifying and controlling these two quantities allows better control over the quantity and timing of the resources delivered to applications and enables greater overall performance because we no longer have to guess at application resource needs or constraints, *e.g.*by scheduling a disk request simply because we *think* the application that issued it might starve if we wait just a little longer.

We are developing a library of algorithms to translate different application requirements into this uniform representation. Key to the translation is knowledge about application behavior, which is often available for those applications, like multimedia, transaction processing, or scientific applications, that require performance guarantees: transactions generally involve small, randomly located requests, multimedia data is highly sequential, and scientific applications generally have very well-characterized access patterns.

As an example, consider a rate-oriented soft real-time specification of i I/O requests per second on average. Given knowledge about the application's I/O locality, burstiness, allowed request latency, *etc*., this can be turned into a concrete specification of the application resource reservation r consisting of a desired number of I/O's n, period p, and buffering capacity c. A mean execution time MET can be derived from the locality and the characteristics of the disk in question, allowing us to derive the desired disk utilization $u = n * \text{MET}/p$. The translation also knows a worst-case execution (seek and rotate) time WCET from the drive's characteristics and adds a small overhead factor to account for 2 inter-stream seeks per period + 1 extra worst-case execution time to ensure that the full desired utilization will always be granted. The reserved disk utilization is thus

$$u\prime = \frac{2 \cdot \mathrm{WCET} + \max(\mathrm{WCET}, (n-1) \cdot \mathrm{MET} + \mathrm{WCET})}{p} \quad (1)$$

Our current broker assumes homogeneous disks; we are working to extend it to handle heterogeneity based on knowledge about the location of the data being accessed.

4 Disk request scheduling

Disk request scheduling ultimately determines the performance available to the applications. Caches can shape the traffic going to the disk, but in the end the disk performance is a fundamental limitation on overall storage system performance.

We have three goals for disk scheduling: that it meet each application's reservation; that it ensure isolation between the requests from different applications; and that it provide good performance.

Our disk request scheduler, Fahrrad, meets these goals. By measuring and accounting for all I/Os issued and seeks incurred, Fahrrad ensures that each application gets the resources reserved for it. As long as each application behaves as it indicated to the broker, this ensures that it will get the performance it desired. If it breaks the "contract" with the broker and behaves otherwise, nothing is guaranteed except that it will not interfere with any other application. By correctly accounting for all seeks, including all "context switches"—seeks between request streams—Fahrrad ensures isolation. By aggressively reordering requests based on explicit knowledge of application requirements, Fahrrad is capable of performance as good or better than best-effort disk schedulers.

Fahrrad works as follows. Requests are segregated into distinct streams, each of which is associated with a reservation with a utilization u and deadlines d_i. Fahrrad temporarily assumes that each requests takes WCET, and moves all requests from each stream that can be accomplished within the available *budget* $b = u * p$ for the current period and before the earliest deadline in the system into a common buffer called the Disk Scheduling Set (DSS). These requests can be executed in any order and can take any amount of time (up to WCET) without jeopardizing any stream's reservation. Requests in the DSS are scheduled as aggressively as possible. As each requests is completed, its actual execution time is measured, and a new request from its stream may be moved into the DSS if sufficient resources remain in the stream's budget for the current period.

Figure 3 illustrates the performance obtained with Fahrrad. It compares a mixed-application workload running on a standard Linux system (a) and one with Fahrrad (b). The workload combines two "media" streams, a transaction processing workload with highly bursty request arrivals, and a random background stream simulating backup or rebuild. Fahrrad meets both the utilization guarantees and throughput requirements of the I/O streams and its throughput exceeds that of Linux by about 200 I/Os per second.

5 Server cache

The server cache stores data as it moves between the network and the disk. Traditionally, it allows reordering requests, speed matching, and write coalescing. It also allows write-behind and read-ahead, allowing the system to trade cache resources for disk performance.

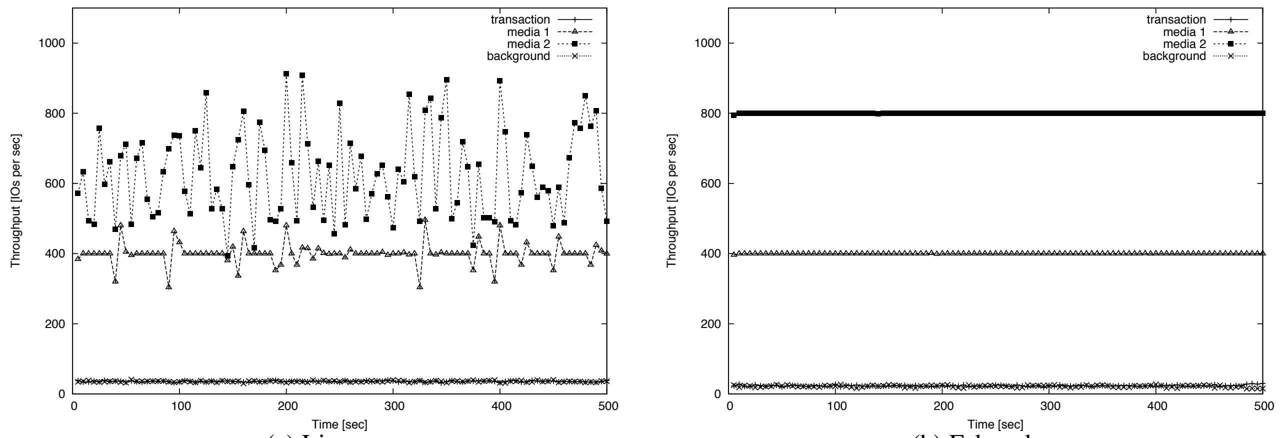
Short periods cause inefficiency by increasing the overhead for per-period inter-stream seeks (the two extra seeks per period in Equation 1). We can lengthen the period of the disk reservation for write-only streams by a factor k by allocating k times as many cache buffers and absorbing that many periods worth of requests—thus formalizing the intuition that write-behind can improve performance. A similar argument applies to read-ahead and can be used to reason about how far in advance to read-ahead, taking into account the probabilistic accuracy of read-ahead.

Our goal is to identify the amount of buffer space required to satisfy a given application's reservation and understand how those buffers can best be allocated across the different application streams. As part of the admission control policy, the minimum capacity that should be allocated for a particular stream must be known in order to decide when to reject a new reservation.

The server cache must be coordinated with the disk scheduler below it and with the network above. Since we are focused on a server cache, where there is a client cache in front that absorbs read hits, we are investigating the value of having the disk scheduler select which cache buffers to process from a stream based on the efficiency of head movement, rather than focusing on cache replacement policies that focus on the likelihood of future buffer re-use. We discuss the interface between server cache and the network in the next section.

6 Network

Data moves between client and server caches over the network. There are multiple problems to solve related to the network: getting quality of service out of the underlying transport connections; managing the transport connections in a large system; and flow control to ensure that



(a) Linux (b) Fahrrad

Figure 3: Behavior of mixed workload during 500 seconds, with and without Fahrrad. Points are the average for 5-second intervals.

the receiving cache always has room for data when it arrives (without interfering with other streams).

Existing storage protocols like iSCSI provide solutions for one-to-one relationships between client and server, but large HPC systems involve many clients—often more than the server's network stack can support at any given moment. In addition, if all the active clients send requests at the same (low) rate, the server will have very little data from each one at any given moment, eliminating any possibility of taking advantage of locality within the requests from an individual client. Both these issues suggest that the server should manage the set of active connections and should get try to get more requests from any one client at a given time while maintaining its guarantees. "Fairness" *per se* is guaranteed by setting appropriate reservations and is thus a concern of the broker rather than the the network or the servers.

We are currently investigating solutions to these issues. Our approach builds on existing network protocols that reserve resources to provide basic quality of service. We are focusing on *flow control* and *connection management*. For flow control, we are investigating a server-driven flow control model where clients provide indications of how much data they have to move, and the server issues time-limited flow control credits based on a prediction of how much cache buffer space the server is expected to have available for servicing those requests in the near future. We are also exploring different ways to get larger batches of requests from individual clients. For connection management, we are investigating an approach where clients send out-of-band connection requests to the server and the server uses a heuristic based on multi-resource job-shop scheduling to select which connections to allow at any given time.

7 Client cache

The client cache stages data, works with the network protocols to move data to and from the server, and provides the opportunity for cache hits. The Ceph file system provides client cache coherence mechanisms, so we focus on the interaction between the client cache and the network and server caches.

The client cache uses flow control credits from the server to pace the requests that it sends. It piggybacks information about the number of read and write requests in cache that need to be processed to help guide the flow control and connection management decisions at the server.

8 Conclusion

We are working on end-to-end performance management for scalable, distributed storage systems. Our integrated solution is based on centralized reservations and coordinated local resource management.

References

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[2] 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.