A Novel Network Request Scheduler for a Large Scale Storage System

[1]Qian Yingjin, [2]Eric Barton, [2]Tom Wang, [2]Nirant Puntambekar, [2]Andreas Dilger[2]

[2]Lustre Group, Sun Microsystems, Inc.

[1]National Laboratory for Parallel and Distributed Processing, China

{Yingjin.Qian, Eric.Barton, Tom.Wang, Nirant.Puntambekar, Andreas.Dilger}@Sun.Com

Abstract

This paper presents a novel Network Request Scheduler (NRS) for a large-scale, Lustre TM storage system. It proposes a quantum-based, Object Based Round Robin (OBRR) NRS algorithm that reorders the execution of I/O requests per data object, presenting a workload to backend storage that can be optimized more easily. According to the drawback of static deadlines in large-scale workloads, it proposes a novel, two-level deadline setting strategy that not only avoids starvation, but also guarantees that urgent I/O requests are serviced in a specified time period. Via a series of simulation experiments using a Lustre simulator, it demonstrates that I/O performance increases as much as 40% when using the OBRR NRS algorithm, and the two-level deadline setting strategy can avoid starvation and ensure that urgent I/O requests are serviced in the required time.

***Keywords**: Network Request Scheduler; Lustre; QoS; Large scale; Storage system*

1 Introduction

I/O bottleneck has always been a major obstacle to achieve high performance in high-performance computing (HPC) with the increasing use of HPC platforms and the growing number of parallel I/O intensive scientific applications. The parallel file system has helped ease the performance gap between I/O hardware and processor/memory, but I/O remains an area needing significant performance improvement.

As shown by several studies [2, 4], parallel I/O access uses recurrent, determined patterns based on stride parameters that are good candidates for optimization. Parallel I/O scheduling [1, 5] was proposed to exploit parallel I/O access patterns. LustreTM is a leading technology in parallel I/O technologies and is an emerging open source standard for scalable HPC and cluster computers, running on 7 of Top 10 and 40% of the 100 largest HPC clusters in the world (as of October 2008 [TOP500]). Lustre provides excellent I/O throughput, but further improvements are possible. This paper presents a novel Network Request Scheduler (NRS) framework; a server-side scheduling strategy to optimize throughput with low latency for parallel I/O workloads on large-scale Lustre clusters.

2 Lustre I/O Architecture

Building a Lustre cluster requires a Metadata Server (MDS) and Object Storage Servers (OSSs), each with a backend file system. The MDS manages the name and directories in the file system, is responsible for metadata operations, and maintains file layout attributes containing the references of data objects on each OSS. The OSSs provide file locking and data I/O services. File data may be striped onto many data objects stored on an OSS (in the form of regular files with intelligence on the backend file system), enabling fast, concurrent file write and read capability. A pool of client systems implements POSIX file system interfaces and access servers through one of many supported networks. Each server has a thread pool to handle client requests in parallel, maximizing resource utilization. The service thread count can vary from 2 to 512, depending on server load. Lustre uses the Distributed Lock Manager (DLM) to support fine-grained locking for efficient, concurrent file access. Based on the DLM, it implements client data write-back cache. To access the data, the client obtains the file layout from the MDS, and then transfers data directly to or from the OSS under DLM lock protection, resulting in significantly enhanced performance.

Lustre I/O processing implements the scheduling policy and operates as follows: the client sends an I/O request to the server; the request contains the target data object ID, I/O offset, and count in object, etc. Upon receipt, the request is enqueued, and waits for service. The server then dequeues the request (for execution in the context of a service thread), and writes/reads the bulk I/O data to/from disks through the network. Upon completion of the request, the server sends a reply to the client. Requests are dispatched in default FCFS order.

3 Object Based Network Request Scheduler

In terms of data throughput, the best performance is achieved when disk access is sequential. However, file systems cannot always place and access data sequentially, since various applications have inherent data access patterns and are limited by file system block allocation in sequential space allocation.

To meet the parallel I/O characteristics based on stride from multiple distributed applications, Lustre developed a multiple block allocation (mballoc) [3] for its backend file system that optimizes concurrent block allocations. Mballoc uses pre-allocation technology to reduce the most time-intensive disk seeks by allocating blocks contiguously and reducing file system fragmentation. The pre-allocation region size per object is relatively large, reaching 8MB for a large file. As shown in this paper [3], mballoc offers significant performance improvement, especially for sequential access.

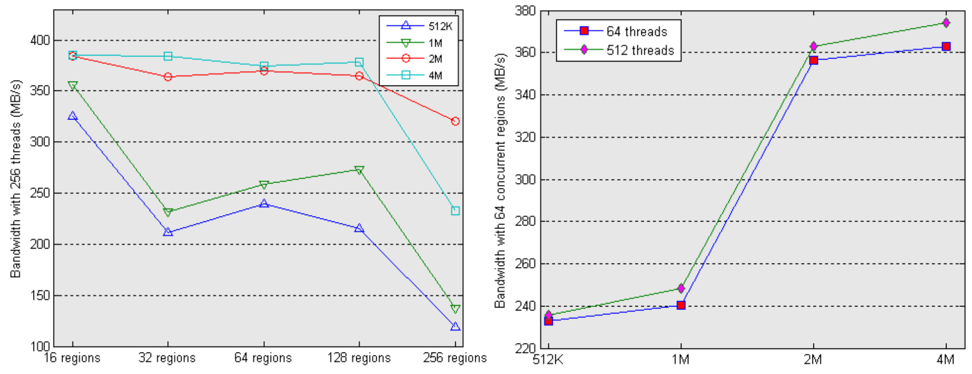

Figure 1 Performance surveys by sgpdd-survey for 2 tires DDN S2A 9550

We conducted considerable research on parallel I/O and observed performance gains with large transfer size and data chunk. Oak Ridge National Laboratory (ORNL) performed a series of surveys using the DDN S2A 9550 as the Lustre backend storage device. As shown in Figure 1, ORNL used sgpdd-survey with different concurrent regions, threads, and record sizes. The graphs illustrate that performance decreases with the growth of concurrent I/O regions. Performance improves slightly with increasing service threads and improves significantly with the growth of record size. For a 4MB record size, performance improves 50% compared with a 512K record size. To improve performance, Lustre breaks the 1MB bulk I/O limitation and implements a 4MB bulk I/O delivery through the network, by aggregating several separated bulk data transfers with 1MB data into a single I/O delivery request to the server. Figure 2 illustrates the paradigm of 4MB bulk I/O delivery. ORNL's surveys show performance improvements as high as 40%, compared with 1MB bulk I/O delivery. We determined that delivering large I/O requests all the way down to the server block layer was required to maximize RAID performance by reducing disk seeks significantly, while 1MB bulk concurrent I/O requests from clients may arrive at servers in a manner that discourages or even destroys sequential access. It was not necessary to transfer bulk data in larger than 1MB chunks to maximize network throughput. Although disk elevator scheduling [6] can optimize I/O requests by sorting and merging them before they are submitted to the disk driver, the disk elevator is implemented at a low level, without a global view of distributed applications accessing the file system. Owing to the limit of the disk elevator's queue depth, relatively few I/O requests are candidates for request scheduling, compared to the huge number of buffered I/O requests on the server.

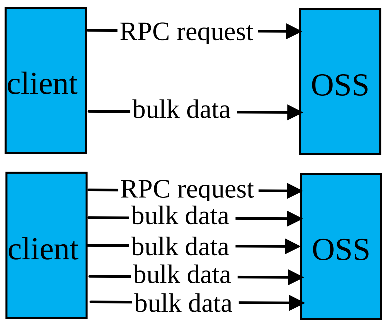

Figure 2 4MB bulk I/O vs. 1MB bulk I/O

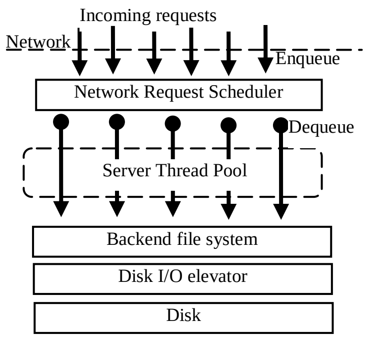

Figure 3 NRS framework

NRS is a high-level scheduling policy, located between the network and backend storage, which provides consistently improved performance. Figure 3 shows the NRS framework. NRS manages incoming requests to a server by reordering request execution to avoid starvation and to present a workload to the backend file system that can be optimized more easily by the underlying disk elevator in the manner of sorting or merging. The work set for scheduling is not the number of service threads, but all of the queued requests on the server. Unlike large 4MB bulk I/O suited only for a specific workload, NRS is an attractive strategy that can be widely used to achieve a similar collective effect on I/O - using NRS at a high level and the disk elevator at a low level. This combined request scheduling works with small I/O requests and when accessing a file shared among different clients.

3.1 Object Based Round Robin NRS Algorithm

Based on the feature of block allocation, we propose an Object Based Round Robin (OBRR) NRS algorithm. The principle is to order the execution of I/O requests that belong to the same data object, by offset, as close as possible to reduce disk seeks. The OBRR algorithm aims to provide scheduling strategies that primarily optimize throughput, but have a concern for fairness and response time.

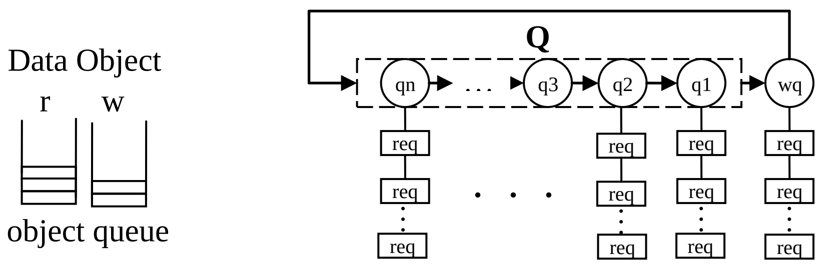

Figure 4 Object Based Round Robin algorithm

The basic concept is illustrated in Figure 4. Each data object has two separate queues (called object queues), which manage its incoming read/write requests, respectively. Each server has a global FIFO queue (called **Q**), which manages the object queue with outstanding I/O requests. When a new I/O request is received, it is added to its corresponding object queue, sorted by an offset in object, and the queue is inserted into the global **Q** if it is not already in it. Each object queue can be considered a sub I/O scheduler in objects based on the elevator algorithm. Under OBRR service, the queue per object is serviced in a round robin fashion. In each round, each queue is provided with a fixed quantum of I/O service. The quantum of service can be defined in terms of the number of serviced I/O requests or the I/O amount in each round. It can be set statically or changed over time to achieve adaptive resource allocation. At the beginning of each service round, the head object queue is removed from **Q**, and becomes the work queue. Requests are dequeued from the work queue for execution until the queue is emptied or its associated quantum is exhausted. If the work queue uses up its quantum but is not empty, it is added to the global **Q** again to wait for the next service round.

The service quantum per object queue is not be identical and if set differently, results in differently weighted services. Lustre is targeted at development of a next-generation cluster file system with 100,000s of nodes. The CPU utilization needed to sort a huge number of requests, imposed by clients at large scale, was the main concern when we designed the scheduling algorithm. In OBRR, the time complexity of sorting is significantly reduced as requests are sorted in the object queue, which is usually a small subset of the total number of queued requests.

3.2 Deadline setting strategies

Executing requests per object queue with a fixed quantum in round robin fashion balances response time and maximizes I/O throughput among objects, but it cannot ensure fairness in objects. As I/O requests to an object continue to arrive, the elevator scheduling policy in the object queue ignores a request, for a long time, because it prefers to handle other requests that are closer to the last served one. To avoid starvation and ensure the fairness in objects, a deadline is introduced to each I/O request.

A request deadline is set and starts ticking upon the request's arrival. Its value is the sum of the request's arrival and expiration times. NRS services requests in elevator order in an object queue unless a request's deadline expires to prevent starvation or to meet the requirement of response time, in which case it first services any request with an expired deadline. To meet different I/O delivery requirements, we designed two deadline setting strategies: a dynamic deadline and a mandatory deadline.

In storage clusters with thousands of nodes, the average time a server takes to handle an I/O request scales approximately linearly with the number of clients contending for the shared resource disk bandwidth and varies over time. In the extreme case, it may even reach hundreds of seconds. Obviously, a static expiration time is no longer suitable for large scale workloads. We designed a scalable dynamical deadline setting strategy for normal I/O requests according to server load. To describe it more exactly, we define a triple $\mathrm{R_i} = (\mathrm{r_i}, \mathrm{bw_i}, \mathrm{s_i})$ where $\mathrm{R_i}$ represents the *i*th I/O count range window; $\mathrm{r_i}$ represents the record size associated with the window; $\mathrm{bw_i}$ represents the corresponding I/O performance with record size $\mathrm{r_i}$ on the backend storage (benchmarked automatically at the time of server setup); $\mathrm{s_i}$ represents the I/O amount of outstanding and servicing requests for which the I/O count is in range $[\mathrm{r_i}, \mathrm{r_{i+1}}]$ and tracked over time as I/O requests arrive and complete. The expiration time **e** is calculated by the following formula:

$$\mathrm{e} = \lambda * \sum s_i/bw_i;$$

where λ is the amplification factor, $\lambda \geq 1$. With a slightly larger reasonable expiration time, it could aggregate and present more contiguous I/O to the backend storage. To reduce sorting, a global dynamic deadline FIFO queue is introduced which includes I/O requests sorted according to their deadline. As the server load decreases, the newly-set dynamic deadline value may be less than the maximum deadline value in the deadline queue. At this time, it simply amends the deadline to the maximum deadline value.

In Lustre clusters, a client may make urgent I/O requests, i.e., some I/O requests resulting from a lock conflict to clean cache data on the client. During Lustre usage, many timeouts resulting from such kinds of I/O requests have been observed. They must be handled as soon as possible, especially when the server is under heavy load. Otherwise, it causes a cascade of failures and impacts performance. For these I/O requests, we propose a mandatory deadline strategy. The client can indicate the maximum service time on the server to handle I/O requests. The expiration time is mandatory and set to be the same as the indicated maximum service time. In our implementation, all of the maximum service times indicated by clients are the same. A separate global mandatory deadline FIFO queue is used to manage the I/O requests with a mandatory deadline.

When dequeuing a request, the two deadline queues are checked. If there is a request with an elapsed deadline, then that request is serviced first. The two level deadline setting strategy avoids starvation of normal I/O with a dynamic deadline guarantee, and ensures that urgent I/O can be serviced in the required time.

4 Evaluation

We used a Lustre simulator [7] to evaluate the NRS algorithms. We measured principal metrics including I/O bandwidth, disk seeks and response time, etc. The Lustre simulator was developed by Sun as a simulation platform to research scalability, analyze I/O behaviors and design various algorithms at large scale. It simulates disks, the Linux I/O elevator [6], a file system with mballoc block allocation, a packet-level network, and three Lustre subsystems: client, MDS and OSS. The Lustre simulator can simulate concurrent operations by 100,000 clients.

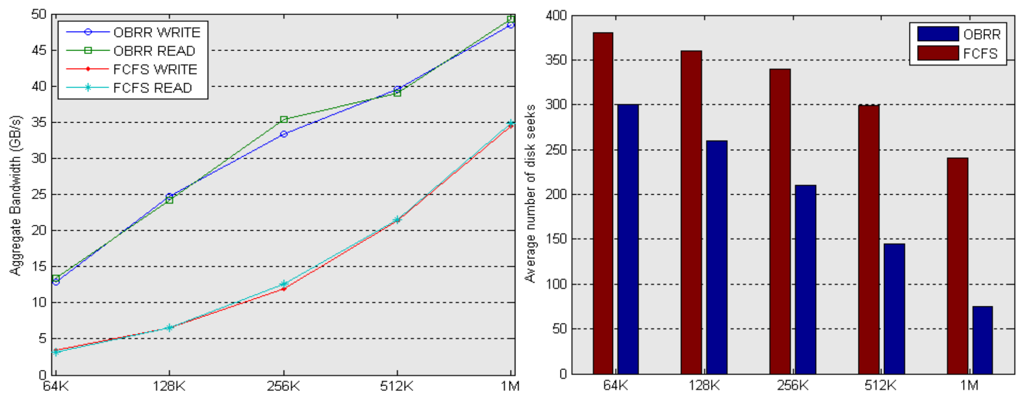

Figure 5 Performance comparison between FCFS and OBRR algorithms

To evaluate the OBRR algorithm, the experiment was designed as follows: raw disk bandwidth 450MB/s with 10ms seek time to simulate the large-scale Jaguar

Lustre cluster, 8000 clients, 144 OSSs, each client writes/reads 32MB I/O, and the quantum was defined to service 8 I/O requests per round. Figure 5 shows the results between the FCFS and OBRR algorithms with transfer sizes from 64k to 1MB using the IOR file per processor access mode. With a 1MB transfer size, FCFS aggregate read/write performance was 34.8GB/s and 34.4GB/s while OBRR performance was 49.2GB/s and 48.6GB/s; average disk seeks dropped dynamically from 240 to 75. Table 1 shows disk request size statistics after merging by the disk elevator with a 1MB transfer size on 1 OSS. The 2MB disk requests improved by 30% and 4MB disk requests increased to 21%. Average disk seeks dropped dynamically, and the disk driver got much larger disk requests using the OBRR algorithm. Aggregate performance improved by more than 40%.

Table 1 Disk request size statistics after merging by the disk elevator on one OSS

request size / algorithm	1MB		2MB		4MB	
	count	percent	count	percent	count	percent
FCFS	1758	99%	1	0%	0	0%
OBRR	485	47%	312	30%	217	21%

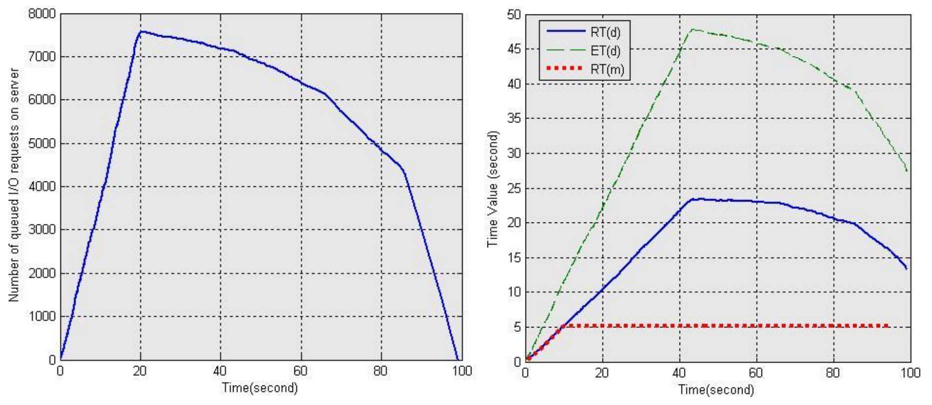

Figure 6 Trace of I/O RPC requests with two-level deadline setting strategy

To evaluate the deadline setting strategy, the experiment was designed as follows: 1000 clients each send 32M I/O with a 1MB transfer size to one OSS; the time skew between clients to launch the I/O is 20s; an extra 50 clients generated 1MB urgent I/O with an indicated maximum service time of 5s per 100ms interval, and $\lambda = 1.5$. In Figure 6, the left graph shows the trace of number of queued requests on the server over time – the maximum number reaches almost 8000. In the right graph, the curve "*RT(d)*" plots by using the requests' completion time as the X axis and the requests' response time as the Y axis. It denotes the variation over time of response time with the dynamic deadline setting policy; the maximal value reaches nearly 25s. Similarly, the curve "*ET(d)*" denotes the variation over time of the corresponding estimated expiration time and shows estimated expiration times are large enough to collect contiguous I/O and setting values almost followed the trend of the requests' response time. The curve "*RT(m)*" denotes the variation over time of response time of requests with a mandatory deadline setting policy. It demonstrates response times are constant at about 5s and urgent requests can be finished in the required time.

5 Conclusion

HPC servers must manage I/O resources efficiently and fairly among many clients. In this paper, we presented an object based NRS for the Lustre file system. We also designed a quantum-based OBRR algorithm with a deadline to schedule intensive parallel I/O workloads on servers. Via experiments based on simulation scaling up to thousands of clients, we demonstrated that this algorithm can significantly maximize I/O throughput by reordering the execution of high-level I/O requests to present a workload to the low-level disk elevator that can be optimized more easily. It maintains fairness, avoids starvation and ensures the response time requirement for I/Os with different urgencies using a quantum based scheduling algorithm per object, together with a two-level deadline setting policy. Based on the simulation, our next step will be to implement and evaluate the NRS on large storage clusters in real-world environments.

References

[1] Adrien Lebre, G. Huard, Y. Denneulin. I/O Scheduling Service for Multi-Application Clusters. Cluster, pp.1-10, *2006 IEEE international conference on Cluster Computing*, 2006.

[2] P. E. Crandall. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[3] Aneesh Kumar K. V. Ext4 block and inode allocator improvements. *Proceedings of the Linux Symposium, Volume One* (July 23rd-26th, 2008 Ottawa, Ontario Canada).

[4] N. Nieuwejaar, D. Kotz. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075-1089, October 1996.

[5] R. Ross. Reactive scheduling for parallel i/o systems, PhD thesis, Dept. of Electrical and Computer Engineering, Clemson University, Clemson, SC, December 2000.

[6] Seetharami Seelam. Enhancements to Linux I/O Scheduling. In *Processing of the Linux Symposium, Volume Two* (July 20nd-23th, 2005 Ottawa, Ontario Canada).

[7] Lustre simulator. https://bugzilla.lustre.org/show_bug.cgi?id=%2013634.