I/O Interference Alleviation on Parallel File Systems Using Server-Side QoS-Based Load-Balancing

Yuichi Tsujita[1(✉)], Yoshitaka Furutani[2], Hajime Hida[3], Keiji Yamamoto[1], Atsuya Uno[1], and Fumichika Sueyasu[2]

[1] RIKEN Center for Computational Science, Kobe, Hyogo, Japan
yuichi.tsujita@riken.jp
[2] Fujitsu Limited, Minato-ku, Tokyo, Japan
[3] Fujitsu Social Science Laboratory Limited, Kawasaki, Kanagawa, Japan

Abstract. Storage performance in supercomputers is variable, depending not only on an application's workload but also on the types of other concurrent I/O activities. In particular, performance degradation in meta-data accesses leads to poor storage performance across applications running at the same time. We herein focus on two representative performance problems, high load and slow response of a meta-data server, through analysis of meta-data server activities using file system performance metrics on the K computer. We investigate the root causes of such performance problems through MDTEST benchmark runs and confirm the performance improvement by server-side quality-of-service management in service thread assignment for incoming client requests on a meta-data server.

Keywords: Lustre · FEFS · MDS · OSS · Data-staging · QoS
K computer

1 Introduction

I/O performance is one of the most prominent contributors in supercomputing, and many parallel file systems, such as GPFS [15] and Lustre [5], have been developed. Meta-data servers (MDSs) are among the most significant performance bottlenecks for Lustre and its enhanced file systems. Since a large number of concurrent file I/O operations at the same storage may lead to high MDS load, poor file I/O performance may be experienced in not only the root-cause user application but also other applications that access the same storage system. In the worst case, unstable file system operation may occur due to such high MDS load. We can easily detect such high MDS loads and root-cause applications through monitoring the associated performance metrics of the MDS, and stable operation can be achieved by terminating such root-cause applications, for instance.

R. Yokota et al. (Eds.): ISC 2018 Workshops, LNCS 11203, pp. 36–48, 2018.
https://doi.org/10.1007/978-3-030-02465-9_3

On the other hand, it is difficult to detect slow MDS response caused by a large number of file accesses under a large stripe count. Once an I/O request is processed by an MDS, the MDS sends associated requests to corresponding object storage servers (OSSes), where the number of requests issued by the MDS is proportional to the stripe count and the number of accessed files. Consequently, setting the stripe count to be large and having a huge number of concurrent file accesses lead to high OSS load, and service threads of the MDS continue to wait for responses from associated OSSes for a long time. As a result, the performance of MDS operations for new incoming I/O requests is degraded, and such a situation leads to poor storage performance.

Knowing the I/O patterns for such slow MDS response is an urgent issue for better file system operation. In order to examine the root causes of slow MDS response, we conducted performance evaluations using MDTEST [6] version 1.9.3. The results indicate that a similar file I/O performance degradation occurs due to the problematic file access patterns. File I/O performance degradation was caused by a mismatched stripe count configuration with respect to the file I/O pattern. Once we introduced quality-of-service-based (QoS-based) service thread management to an MDS, such degradation was minimized.

The main contributions of this paper are as follows:

- Analysis of meta-data accesses to find the root causes of slow storage performance
- Server-side QoS-based management to achieve fair-share service thread allocation on an MDS

Section 2 describes the research background of this research work, including a brief introduction of the K computer, its asynchronous data-staging scheme, and the QoS-based service thread management adopted herein. The analysis of the performance degradation of an MDS is explained in Sect. 3. Section 4 reports the I/O performance evaluation associated with root-cause I/O patterns that led to performance degradation of an MDS and performance improvements using QoS-based management in file system accesses. Related work is discussed in Sect. 5, followed by concluding remarks in Sect. 6.

2 Research Background

This section presents research background and a system overview of the K computer. Since our analysis and I/O performance evaluation were performed using the K computer, we briefly describe the K computer and its file systems before discussing our analysis and performance evaluation.

2.1 K Computer and Its File Systems

The K computer consists of 82,944 compute nodes, where each system rack consists of 96 compute nodes. Figure 1 shows an overview of the K computer system including its two-staged parallel file system. The Fujitsu Exabyte File System

(FEFS), which is an enhanced Lustre file system created by FUJITSU [14,16], has been used in parallel file systems. The layered file system consists of a local file system (LFS) and a global file system (GFS). The LFS, which provides high performance I/O during computation. is intended for performance-oriented use, whereas the GFS is intended for capacity-oriented use, e.g., to keep users' permanent files such as user programs and data files.

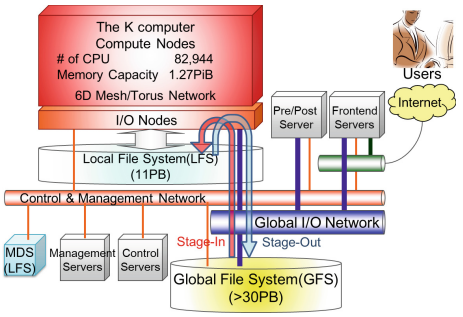
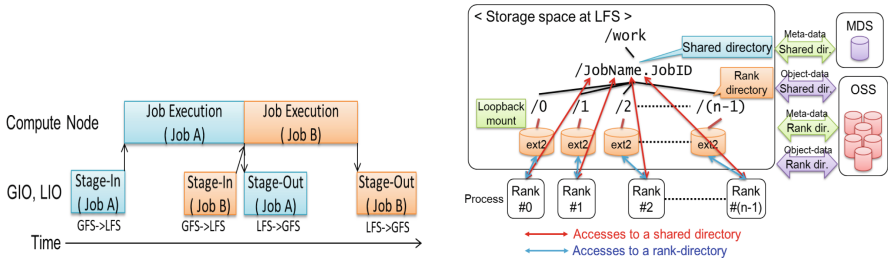


Fig. 1. Overview of the K computer

The number of available OSTs at the LFS is uniquely configured based on the shape of assigned compute nodes according to the I/O zoning scheme [16] of the LFS, where the I/O zoning scheme mitigates I/O interference on OSTs among user jobs by isolating the OSTs assigned for each job. Although the default stripe count of the LFS is 12, users can change the stripe count up to the maximum number of assigned OSTs for newly created files or directories, or can simply set the stripe count to -1, which ensures that the maximum number of assigned OSTs will be used. Moreover, although the default stripe count of the GFS is four, the stripe count of the GFS can be set to the maximum number of OSTs for each target GFS volume.

A subset of I/O nodes, called global-I/O (GIO) nodes, accesses an MDS and the OSSes of each GFS volume via the global I/O network using 4×QDR InfiniBand links. I/O nodes, including GIO nodes, each consist of a single FUJITSU SPARC64 VIIIfx and 16 GB memory. GIO nodes are also responsible for asynchronous data-staging [4] between the LFS and the GFS as shown in Fig. 2(a). This scheme guarantees sufficient I/O performance for programs running on compute nodes and effective job scheduling. An MDS server of the LFS consists of two Intel Xeon E5-2690 CPUs and 256 GB of memory, whereas a subset of I/O nodes, called local I/O (LIO) nodes, is dedicated for an OSS of the LFS. The MDS of the LFS is accessed by GIO nodes and compute nodes in data-staging and local file I/O, respectively.

In order to achieve higher scalability in LFS accesses, loopback file systems, called rank-directories, are created for each rank in the stage-in phase, as shown in Fig. 2(b). Localizing file I/O by rank in each rank-directory can reduce the MDS load.



(a) Asynchronous data-staging scheme (b) Overview of rank-directories

Fig. 2. Data-staging and rank-directories at the K computer

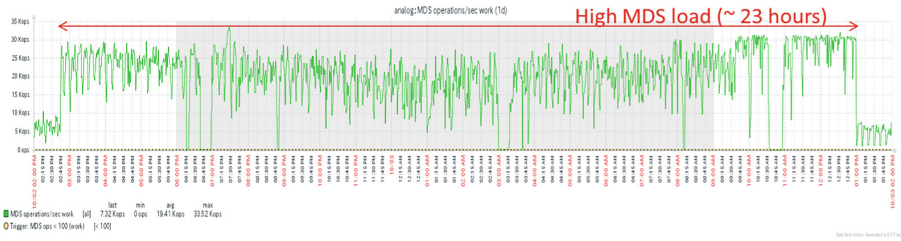
2.2 Performance Problems of File I/O on the K Computer

During the course of our K computer operation, which exceeds file years, we have faced various types of problems related to file systems, as well as problems related to both hardware and software components. Although most of these problems have been addressed, recently we have faced file system problems, including performance degradation due to user applications. Among the file system problems encountered, solutions of the problems of slow file I/O due to inefficient MDS operations is among the highest priorities. In this context, we focus on the analysis of MDS activities and file I/O operations in order to improve file I/O performance.

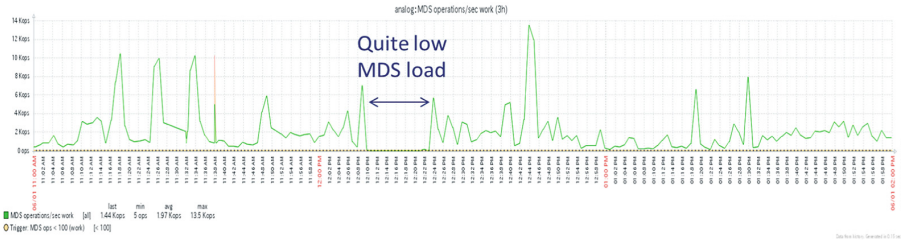
The numbers of MDS operations per second (OPS) observed for typical high MDS load and quite low MDS response cases in the K computer operation are shown in Figs. 3(a) and (b), respectively. The scales of both axes are different between Figs. 3(a) and (b) because we need to clearly show the focused MDS load.

In the high MDS load case in Fig. 3(a), the horizontal axis covers approximately 24 h, which includes the focused MDS activities. A user job with 8,000 processes on 1,000 compute nodes accesses a large number of files on a shared space of the LFS during a 23 h period. (Although 1,008 compute nodes were assigned for the job due to the topological layout, only 1,000 nodes were used for the job.) Such a file access pattern led to a high MDS load, and consequently there was a tremendous negative impact on the I/O operations of other user jobs due to the high MDS load.

On the other hand, Fig. 3(b) shows the low MDS load case for a period of 14 min near the middle of the graph due to a large number of concurrent file accesses by 12,096 processes on the same number of compute nodes. The horizontal axis of this figure covers approximately three hours in order to focus on the low MDS load. Although this job also accessed a shared space of the LFS, the MDS load was quite low during concurrent file accesses for a larger number of files, as compared with the previous case. As a result, the file access performance of every user job that included this job was degraded during this time.



(a) High MDS load due to a large number of file accesses, continuing for approximately 23 hours



(b) Low MDS response due to congestion between an MDS and the associated OSSes

Fig. 3. MDS activity during slow file I/O at the LFS of the K computer

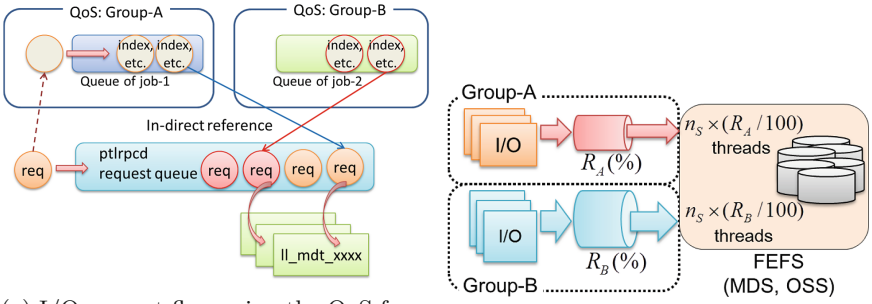
A high MDS load is easily detected by monitoring MDS activities, and stable operation can be achieved by terminating root-cause jobs for example. However, root causes of the slow MDS response cannot be detected only by monitoring MDS activities. Two problematic MDS activities have led to not only performance degradation in file I/O by user jobs but also a large increase in the time required for rank-directory creation prior to stage-in operation. Note that such long times for rank-directory creation have led to delays in start-up of stage-in operation. As a result, slow file I/O or long delays in stage-in operation have led to ineffective job scheduling, despite asynchronous data-staging. Therefore, we need to investigate the root causes of the above MDS performance problems, and finding an effective way to mitigate I/O interference is an urgent problem.

2.3 QoS-Based Management at an MDS

Although the FEFS is based on Lustre version 1.8, the file system has been still extended to cope with high I/O demands at the K computer. One feature of the FEFS is QoS-based service thread sharing among client nodes or among user jobs on each MDS or OSS. The QoS function limits the number of available threads for multiple pre-defined client groups in one of the two modes, static or dynamic. The static case involves the use of a single rate relative to the total number of threads, whereas the dynamic case involves the use of lower and higher rates to advance the demand-based dynamic assignment scheme. Note that this QoS

scheme prevents the lack of free service threads due to a heavy workload, which occurs in the two above-mentioned problematic cases.

Figure 4(a) shows the I/O request flow using the QoS function. Assume that we have two groups, group-A and group-B, for the QoS control, and that client jobs, Job-1 and Job-2, belong to the former and latter groups, respectively. An I/O request is placed in a queue of `ptlrpcd`, and its associated information is placed in another queue associated with a client job. In this case, a new request from Job-1 is placed in a `ptlrpcd` queue, and its associated information including a reference index, which indicates the position of a corresponding request in the `ptlrpcd` queue, is stored in a queue for Job-1. Therefore, a target request is referenced in an indirect manner from queues for client jobs to the `ptlrpcd` queue. The observer task of the QoS function checks the number of free and working service threads. According to the QoS ratio between groups, an I/O request is dispatched to a service thread.



(a) I/O request flow using the QoS function

(b) Usage example of the QoS function

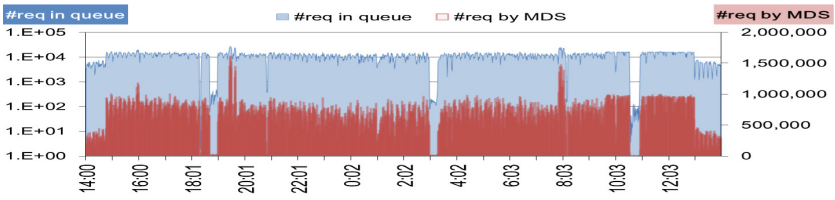
Fig. 4. Schematic diagram of (a) the I/O request flow using QoS function and (b) an example of its usage

Figure 4(b) shows an example of fair-share management among two groups, group-A and group-B, in accessing the same target file system by limiting the maximum number of service threads of an MDS and/or an OSS to each registered group. Assume that we have n_S service threads on each MDS/OSS, the QoS configuration limits up to $R_A\%$ of the n_S service threads for group-A. Thus, group-A can use up to $n_S \times (R_A/100)$ service threads. In contrast, up to $n_S \times (R_B/100)$ service threads can be used for group-B, where its available rate is $R_B\%$. On the other hand, the dynamic configuration can give upper and lower rates in each group.

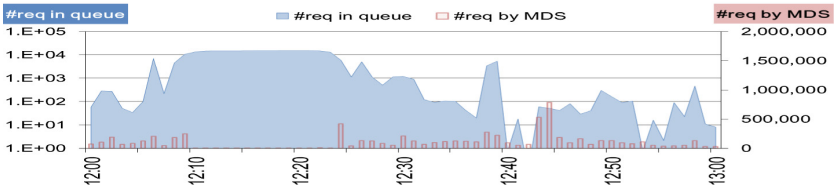
3 Investigation of Internal File Server Activities

High MDS load has been caused by a large number of concurrent file accesses. At present, the K computer limits the maximum number of service threads

to 24 on the MDS of the LFS without any load-balancing, which leads to a high MDS load. Moreover, quite a slow MDS response has been caused by high contention in OSS operations due to a large stripe count in conjunction with a huge number of concurrent file accesses. Such contention is the result of an existing Lustre implementation. Once an MDS receives an I/O request from a client, associated requests to corresponding OSSes are issued by the MDS. The number of associated requests on each OSS is absolutely proportional to the product of the stripe count and the number of concurrent file accesses. Moreover, user jobs specifying a stripe count of −1 use all available OSTs. Once a large number of OSTs are assigned based on the shape of the allocated compute nodes, a large stripe count is provided unexpectedly. Once associated OSSes becomes very busy in managing incoming requests, service threads on an MDS are blocked in order to await responses from the OSSes. During the time of the contention, the MDS has a very difficult time for processing new I/O requests. Therefore, such contention leads to quite slow MDS operation, as shown in Fig. 3(b).



(a) High MDS load



(b) Low MDS response

Fig. 5. Numbers of requests in queue and processed by an MDS over time in the two MDS load cases at the LFS of the K computer

In order to investigate the root causes of the above problem, we have examined the performance statistics of the MDS in detail. Figure 5 shows the average number of requests in a queue and the average number of requests processed by the MDS in one-minute intervals for the two cases shown in Fig. 3. Note that the horizontal axis of Fig. 5(a) covers 24 h, whereas that of Fig. 5(b) covers one hour in order to focus on the target activities. Moreover, the vertical axes on the left-hand side differ scale in order to more clearly show the two operations. We selected the statistics of the MDS in each given interval using the procedure of Lustre `llstat`. In the high MDS load case shown in Fig. 5(a), a very high number of requests are continuously processed over 23 h, whereas, in the slow MDS

response case shown in Fig. 5(b), MDS activity is quite low for 14 min starting from 12:10, when the root-cause job started file I/O.

4 Performance Evaluation

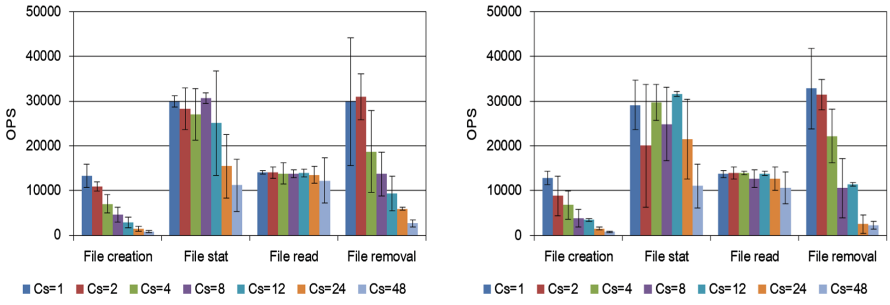
In order to examine the file access patterns that led to slow MDS response and the effectiveness of the QoS function of the FEFS, we conducted a performance evaluation using MDTEST version 1.9.3 at the K computer. In this section, the following three evaluations were conducted in order to examine the MDS of the LFS:

- Stripe count impact on MDS performance using MDTEST
- Impact of QoS management on the MDS for fair-share execution among concurrent MDTEST jobs
- I/O interference alleviation in data-staging using QoS management at the MDS

Through the three evaluations, we demonstrate which types of file I/O access pattern lead to poor MDS response in terms of stripe count configuration and effectiveness of QoS management with respect to MDS performance problems.

4.1 MDS Response Evaluation Using MDTEST

This section discusses performance impact on MDS response regarding stripe count. We evaluated MDS response at the LFS using an MDTEST benchmark code on 192 compute nodes in logical 3D layout of $4 \times 6 \times 8$. We deployed four or eight processes per compute node, and every process accessed 100 files per iteration in each individual directory on the LFS. Mean performance results were obtained from six iterations.



(a) 768 processes on 192($4 \times 6 \times 8$) nodes (b) 1,536 processes on 192($4 \times 6 \times 8$) nodes

Fig. 6. MDTEST evaluation results for various stripe count values, where C_S represents the stripe count

Figure 6 shows the performance results. We have examined seven sets of stripe counts, which are described by C_S in the figure, ranging from 1 to 48 with respect to the target directories on the LFS. The figure shows the mean values of four file-specific operations with variances shown by error bars. Based on the results, we can see a performance degradation in each of the three operations, except for "File read", as the stripe count increased.

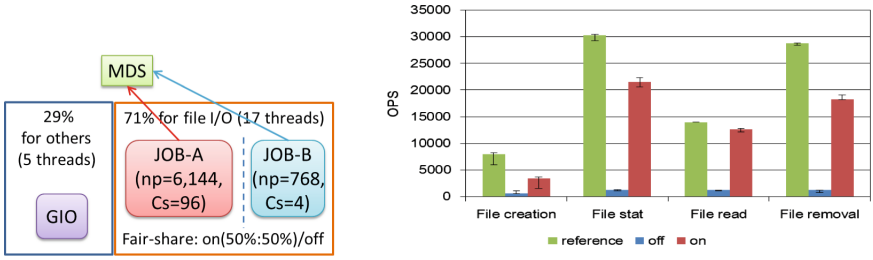
4.2 QoS Impact in Fair-Share Execution Among Concurrent Running Jobs

Fair-share execution using the QoS function was evaluated using two concurrently running MDTEST jobs, JOB-A and JOB-B, on the LFS. JOB-A with 6,144 processes on the same number of compute nodes in a $16 \times 12 \times 32$ layout imitated a root-cause job of slow MDS response under a stripe count of 96. While JOB-B with 768 processes on 192 compute nodes in an $8 \times 12 \times 2$ layout imitated an affected job under a stripe count of four. In both cases, 100 files were created by each process in an individual directory per iteration. We measured the performance of the JOB-B from three iterations during 200 iterations of JOB-A.

Table 1. Configuration of two MDTEST jobs for the examination of fair-share job execution

Notation	Executed jobs	# processes	C_S	# threads	User fair-share
Reference	JOB-B	768	4	17	None
Off	JOB-A	6,144	96		None
	JOB-B	768	4		
On	JOB-A	6,144	96		JOB-A:JOB-B = 50%:50%
	JOB-B	768	4		(up to 90% each if available)

The measurement configuration is summarized in Table 1. The first case, "reference", is the reference case for JOB-B, where the job could achieve peak performance by using all 17 service threads on the MDS without JOB-A. The following two cases, i.e., "on" and "off", represent concurrent execution of the two jobs with and without user fair-share management. Figure 7(a) shows the QoS setup for the MDTEST performance results. In both cases, the number of available service threads was limited to 71% of the total number of service threads (17 threads), while the remaining service threads (five threads) were separated in order to simulate service threads being dedicated to other tasks by GIO nodes to simulate real QoS use. Competition among the two jobs for 17 service threads led to contention on the MDS in the "off" case, whereas in the "on" case, each job could use up to 50% of the 17 service threads. If the counterpart job did not run or did not operate I/O, up to 90% of the 17 threads was available, i.e., up to 15 threads.



(a) QoS setup in MDTEST evaluation for fair-share management

(b) Performance results for the JOB-B group with and without fair-share management during job execution of the JOB-A group

Fig. 7. MDTEST evaluation for fair-share management, where (a) shows the QoS setup and (b) shows the performance results

Figure 7(b) shows the MDTEST performance results of JOB-B for four file-specific operations at the LFS with and without fair-share management, where the bar charts show mean values with bars indicating the maximum and minimum performance values from three iterations. Performance of the "off" case was greatly degraded due to I/O interference by JOB-A because MDS accesses by 6,144 processes under a stripe count of 96 led to slow response of the MDS. By comparing the "off" case with the "on" case, the performance of every operation was improved effectively. For instance, the "on" case was around 16 times faster than the "off" case for "File stat" operation. By comparing the performance of the "on" case with that of the "reference" case, JOB-A also degraded the performance of the "on" case, however the fair-share management mitigated the performance decrease dramatically.

We also found such performance improvements in the JOB-B case in CPU utilization at the MDS, as shown in Fig. 8. With fair-share management, as indicated by "Fair-Share ON," we can see a higher CPU utilization of up to approximately 70%, whereas a lower CPU utilization of approximately 20% is observed without the fair-share management, as indicated by "Fair-Share OFF". Therefore, the high CPU utilization for the "Fair-Share ON" case was due to improved MDS activities for JOB-B.

4.3 QoS Impact in Data-Staging

We measured the times for rank-directory creation in the stage-in phase under high MDS load by an I/O heavy MDTEST job with and without QoS management for MDS service threads. We increased the total number of service threads of the MDS from 24 to 32, two of which were used for file system monitoring. Therefore, the remaining 30 threads were used for data-staging and file I/O. Here, the lower and higher rates of QoS management were 20% and 94%, respectively, for the data-staging job, and 4% and 69%, respectively, for the I/O heavy MDTEST job.

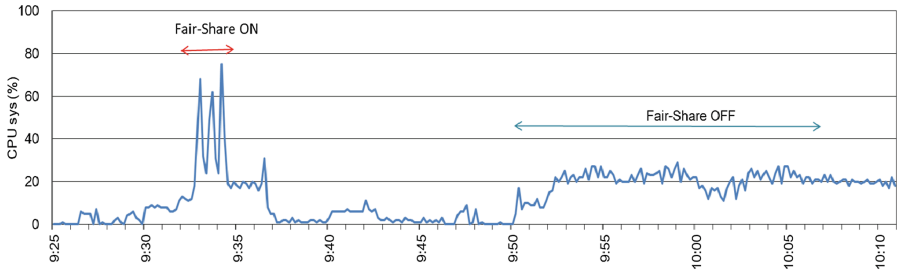


Fig. 8. CPU system use of an MDS during MDTEST runs with and without fair-share management over time

We executed a job for data-staging evaluation by changing the number of compute nodes during a high MDS load caused by an MDTEST job by running 6,144 processes on the same number of compute nodes under a stripe count of 96. Every process accessed 100 files per iteration in an individual directory. Due to the limited time for the evaluation setup without any other user jobs, only one data-staging operation was carried out in each configuration.

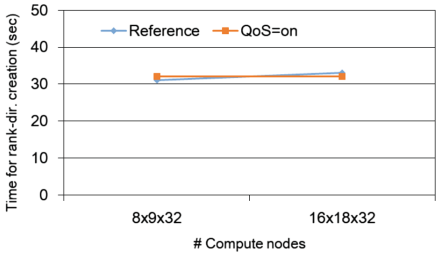


Fig. 9. Times for rank-directory creation in the stage-in phase under QoS control for service threads on an MDS

Figure 9 shows the times for rank-directory creation for two node layouts. The horizontal axis describes the number of assigned compute nodes in the 3D layout. "Reference" indicates the times without the high MDS load job, which were measured as reference values. Note that rank-directory creation in the stage-in phase under the high MDS load job did not finish within five minutes after start-up of the rank-directory creation. In the smaller case of $8 \times 9 \times 32$, only up to four rank-directories were created, none of which was created in the case of $16 \times 18 \times 32$. On the other hand, rank-directory creation with QoS control described by "QoS = on" in Fig. 9 showed approximately the same performance as the "Reference" case.

Based on the results, QoS control is considered to be very effective for stable rank-directory creation in stage-in operation even if high MDS load jobs processed are at the same time.

5 Related Work

Performance optimization in Lustre file systems has been investigated in various research works [2,3,7,12,13]. These works tuned parameters based on empirical data or operation profiles. In such cases, monitoring tools are important in order to clarify activities of MDSs, OSSes, and associated components and future aspects of the target file system [1,17,18]. Uselton et al. [17] demonstrated server-side monitoring using Lustre Monitoring Tools (LMT) [8]. Although LMT monitors the storage server status, such as the CPU utilization, memory usage, and disk I/O bandwidth, the LMT does not provide detailed I/O tracing information, which is useful for systematic analysis such as file system statistics.

A number of studies have investigated load-balancing or contention-aware optimization including QoS management. Zhang et al. [19] introduced machine learning into a QoS parameter setup scheme and implemented the scheme in PVFS2. Qian et al. [11] proposed a dynamic I/O congestion control algorithm for Lustre. They proposed a token bucket filter [10] in the network request scheduler (NRS) framework [9]. However, the token bucket filter does not guarantee free service threads for numerous incoming I/O requests. On the other hand, the proposed QoS-based approach available on the FEFS provides server-side management by limiting the number of service threads to each pre-assigned group for fair-share I/O bandwidth utilization. The QoS function controls RPC request dispatching to service threads based on the IP addresses of RPC request senders or a user grouping scheme. Even in the slow response of an MDS, the QoS function maintains a pre-defined number of free service threads for registered clients, independent of the I/O load by other clients on an MDS. In this context, the QoS approach is more realistic manner for stable file system operation.

6 Concluding Remarks

We have investigated the root causes of high MDS load and slow MDS response through analysis of several performance metrics and benchmark runs on the K computer. Our analysis of the two problematic MDS activities has revealed distinct I/O patterns accessing a large number of files on a parallel file system, i.e., the FEFS, on the K computer. A high MDS load originated from a large number of concurrent file accesses, whereas a slow MDS response was caused by a larger stripe count configuration in accessing a large number of files. Such a stripe count configuration led to congestion in sending requests from an MDS to associated OSSes and every service thread on the MDS waited for a long time. Consequently, the MDS could not process new incoming requests. We observed the same situation in MDTEST benchmark runs.

Even in the case of such a slow MDS response, the QoS function of the FEFS mitigated performance degradation in file I/O of the MDTEST benchmark runs. Interference in rank-directory creation during data-staging operation was also dramatically alleviated by the QoS function. In the future, we intended to consider adoption of the QoS function for OSSes of the LFS on the K computer. Such an approach is expected to mitigate I/O interference on OSSes among file I/O by user jobs and the data-staging phase.

Acknowledgment. The results of this paper were obtained using the K computer.

References

1. Brim, M.J., Lothian, J.K.: Monitoring extreme-scale Lustre toolkit. In: Proceedings of the International Workshop on the Lustre Ecosystem: Challenges and Opportunities (2015). http://arxiv.org/html/1506.05323
2. Crosby, L.D., Mohr, R.: Petascale I/O: challenges, solutions, and recommendations. In: Proceedings of the Extreme Scaling Workshop, BW-XSEDE 2012, pp. 7:1–7:7. University of Illinois at Urbana-Champaign (2012)
3. Ezell, M., Mohr, R., Wynkoop, J., Braby, R.: Lustre at petascale: experiences in troubleshooting and upgrading. In: 2012 Cray User Group Meeting (2012)
4. Hirai, K., Iguchi, Y., Uno, A., Kurokawa, M.: Operations management software for the K computer. Fujitsu Sci. Tech. J. **48**(3), 310–316 (2012)
5. Lustre. http://lustre.org/
6. MDTEST. https://github.com/hpc/ior
7. Mohr, R., Brim, M., Oral, S., Dilger, A.: Evaluating progressive file layouts for Lustre (2016). http://lustre.ornl.gov/ecosystem-2016/
8. Morrone, C.: LMT Lustre monitoring tools. In: Lustre User Group 2011 (2011)
9. Qian, Y., Barton, E., Wang, T., Puntambekar, N., Dilger, A.: A novel network request scheduler for a large scale storage system. Comput. Sci. - Res. Dev. **23**(3), 143–148 (2009)
10. Qian, Y., et al.: A configurable rule based classful token bucket filter network request scheduler for the Lustre file system. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 6:1–6:12. ACM (2017)
11. Qian, Y., Yi, R., Du, Y., Xiao, N., Jin, S.: Dynamic I/O congestion control in scalable Lustre file system. In: IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST 2013), pp. 1–5, May 2013
12. Reed, J., Archuleta, J., Brim, M.J., Lothian, J.: Evaluating dynamic file striping for Lustre. In: Proceedings of the International Workshop on the Lustre Ecosystem: Challenges and Opportunities (2015). http://arxiv.org/html/1506.05323
13. Saini, S., Rappleye, J., Chang, J., Barker, D., Mehrotra, P., Biswas, R.: I/O performance characterization of Lustre and NASA applications on Pleiades. In: 19th International Conference on High Performance Computing (HiPC), pp. 1–10 (2012)
14. Sakai, K., Sumimoto, S., Kurokawa, M.: High-performance and highly reliable file system for the K computer. Fujitsu Sci. Tech. J. **48**(3), 302–309 (2012)
15. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002, USENIX Association (2002)
16. Sumimoto, S.: An overview of Fujitsu's Lustre based file system. In: Lustre User Group 2011 (2011)
17. Uselton, A.: Deploying server-side file system monitoring at NERSC. In: 2009 Cray User Group Meeting (2009)
18. Uselton, A., Wright, N.: A file system utilization metric for I/O characterization. In: 2013 Cray User Group Meeting (2013)
19. Zhang, X., Davis, K., Jiang, S.: QoS support for end users of I/O-intensive applications using shared storage systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 18:1–18:12. ACM (2011)