A Performance Study of Lustre File System Checker: Bottlenecks and Potentials

Dong Dai[1], Om Rameshwar Gatla[2], and Mai Zheng[2]

[1]University of North Carolina at Charlotte, dong.dai@uncc.edu
[2]Iowa State University, {ogatla, mai}@iastate.edu

***Abstract*—Lustre, as one of the most popular parallel file systems in high-performance computing (HPC), provides POSIX interface and maintains a large set of POSIX-related metadata, which could be corrupted due to hardware failures, software bugs, configuration errors, etc. The Lustre file system checker (LFSCK) is the remedy tool to detect metadata inconsistencies and to restore a corrupted Lustre to a valid state, hence is critical for reliable HPC.**

Unfortunately, in practice, LFSCK runs slow in large deployment, making system administrators reluctant to use it as a routine maintenance tool. Consequently, cascading errors may lead to unrecoverable failures, resulting in significant downtime or even data loss. Given the fact that HPC is rapidly marching to Exascale and much larger Lustre file systems are being deployed, it is critical to understand the performance of LFSCK.

In this paper, we study the performance of LFSCK to identify its bottlenecks and analyze its performance potentials. Specifically, we design an aging method based on real-world HPC workloads to age Lustre to representative states, and then systematically evaluate and analyze how LFSCK runs on such an aged Lustre via monitoring the utilization of various resources. From our experiments, we find out that the design and implementation of LFSCK is sub-optimal. It consists of scalability bottleneck on the metadata server (MDS), relatively high fan-out ratio in network utilization, and unnecessary blocking among internal components. Based on these observations, we discussed potential optimization and present some preliminary results.

I. Introduction

As one of the most popular parallel file systems (PFSes) in high-performance computing (HPC), Lustre [1] has dominated the market share among the top500 supercomputers [2]. While Lustre is continuously growing in both scale and complexity, it also becomes more vulnerable to hardware failures, software bugs, and administrative errors, etc., which may lead to metadata inconsistencies [3], [4], [5], [6]. For example, in 2016, a hardware failure occurred at the High Performance Computing Center (HPCC) in Texas caused multiple power outages, which in turn caused severe metadata inconsistencies in the Lustre deployed and resulted in catastrophic data loss [7].

To help identify and fix the potential metadata inconsistency, Lustre provides a tool called Lustre file system checker (i.e., LFSCK) [8]. Once triggered, it will scan through the file system to detect and fix issues. Due to the prime importance, great effort has been put to optimize its performance. For example, to encourage system administrators to run it more often, LFSCK has been transformed to an online tool, which means users do not need to unmount the file system to run the checker. In addition, LFSCK is optimized to finish quickly and minimize the disturbance on regular HPC applications.

However, in practice, LFSCK still takes a long time to examine a well-aged Lustre file system, making system administrators reluctant to use it as a routine maintenance tool [9]. Consequently, cascading errors may lead to unrecoverable failures, resulting in significant downtime or even data loss. Moreover, the lengthy checking and fixing period may also make the system more vulnerable, because many repairing operations are not atomic [10]. In fact, although the root cause is still unclear, the unrecoverable corruption at HPCC [7] mentioned above was caused by an unexpected interruption to the lengthy checking of LFSCK. In other words, a slow LFSCK may increase the window of vulnerability of the PFS.

The performance issue is expected to get worse while HPC is fast moving toward Exascale and the Lustre deployment becomes much larger in real systems. However, it still is not clear that whether its slowness is because of the inherent, physical limitations of hardware devices, or because of the sub-optimal design and implementation of LFSCK. This shortage of knowledge prevents us from projecting whether running LFSCK is still feasible for future Exascale Lustre, and also impedes possible redesign or optimizations.

In this paper, we study the performance of LFSCK, with the goals of investigating the bottlenecks and analyzing its optimization potentials. Specifically, we utilize a new aging method based on real-world HPC traces to age the Lustre file system for representative checking. We evaluate the performance of LFSCK with different configurations (e.g., number of nodes) and aging levels, while monitoring the resource utilization. We also instrument its source code to analyze the internal behaviors during checking. Through these systematic experiments and analysis, we are able to identify the root cause of the slowness as well the opportunities for performance optimization. In summary, we draw several key observations:

- LFSCK has neither fully utilized the disk bandwidth nor the network bandwidth;
- LFSCK has a scalability bottleneck on the metadata server (MDS);
- the asynchronous threads and pipelines implemented in LFSCK can easily block each other and slow down the overall performance;
- LFSCK performance can be improved by decoupling the tight binding between layout and namespace checking components.

We believe our study is the very first step towards totally new designs and implementations of Lustre file system checkers that are scalable for the future Lustre deployment.

The rest of this paper is organized as follow. Section II introduces the internals of Lustre and LFSCK. Section III further introduces our methodology. Section IV discusses the bottlenecks of LFSCK in different scenarios. Section V reports and analyzes the results of monitoring the internal data structures and implementation of LFSCK. Section VI summarizes this study and discusses the future work.

II. Internals of Lustre and LFSCK

In this section, we introduce the internals of the Lustre file system and LFSCK based on the public documentation as well as our analysis of the source code.

A. Lustre File System

A typical Lustre cluster includes one management server (MGS), one or more metadata servers (MDSes) and many object storage servers (OSSes). The management server (MGS) is very lightweight and normally deployed on one of the MDSes. More than one MDS can be deployed in a Lustre file system. One common way is to use two MDS nodes, where one MDS serves as a standby server to take over the role of the main MDS upon failures. To the best of our knowledge, this is the most widely used setup, so we focus on this setting in this paper. There is a newly introduced feature called distributed namespace environment (DNE), designed to improve the performance of accessing large directories [11]. We leave the study of LFSCK on DNE as the future work.

There are mainly three types of metadata stored in Lustre and they are also what the file system checker needs to check.

First, each entity in the global Lustre file system (such as file, directory, and data object) is represented by an *inode* in the MDT's local filesystem to hold standard POSIX attributes, and additional Lustre metadata is stored in an Extended Attribute (EA) of each *inode*. Currently, Lustre supports two local file systems as its back-end: `ldiskfs` and `zfs` [12]. Here, `ldiskfs` is an extension of the widely used `ext4` [13] file system, hence very popular. In this study, we will focus on `ldiskfs`. Following such a design, it is clear that Lustre needs to maintain the mapping metadata between a global entity and a local *inode*. Lustre utilizes an important auxiliary data structure, called *Object Index* to record the mapping information. Such a data structure exists on both metadata servers (MDSes) and object storage servers (OSSes).

Second, Lustre provides users and applications the POSIX interfaces, which require to maintain certain namespace metadata of the system, such as the tree-structure namespace, the per-file and per-directory attributes, users, groups, and quota metadata. Lustre mainly relies on the metadata servers (MDSes) to store these metadata. As we just described, each file and directory in the global Lustre namespace will be represented by a local *inode* in `ldiskfs` on MDS.

The third important Lustre metadata is the data distribution information or referred as data layout metadata, which is specifically for parallel file systems. In Lustre, clients write files in a RAID-0 manner (striped round-robin) across one or more OST objects, based on the stripe count and stripe size assigned when the file is created. To know the location of each file's data, each MDT *inode* stores the stripe count, stripe size, object identifier(s), and other file layout metadata with each MDS *inode*.

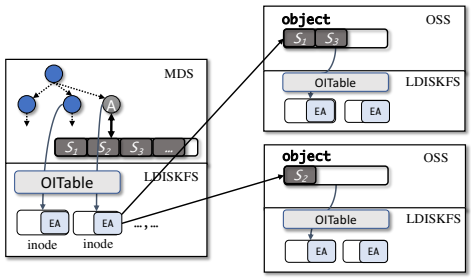

Fig. 1. Lustre Metadata Management Architecture. *Due to space limitation, we only plot one master metadata server and two object storage servers.*

In Fig. 1, we summarize the architecture of Lustre metadata management. As shown in the figure, each global file in Lustre is mapped to a local *inode* in `ldiskfs` of the metadata server (MDS) and striped into multiple internal objects stored on remote object storage servers (OSSes). Each internal object on the OSS is also represented as a single local *inode*. The *Object Index* are stored on both MDS and OSSes to facilitate the mapping between Lustre entities to local file system *inode*.

B. Lustre File System Checker (LFSCK)

Overall, LFSCK includes two stages. In the first stage, LFSCK on the MDS will drive the OSS nodes in the filesystem to conduct checking and possible fixes in parallel. After the first stage on the MDS finishes, all OSSes will start the second stage to resolve orphan and missing objects detected in the file system. Although the repairing could take a considerable amount of time given the number of inconsistency found, normally the inconsistency in an operable file system should be rather small and the first checking phase would dominate the runtime and be more critical to the overall performance. This is true especially when we consider LFSCK as a routine maintenance tool. So, in this study, we only focus on the checking cost by running LFSCK on a healthy Lustre file system. Another reason for doing so is the repairing phase may change its behaviors significantly toward different failure scenarios, measuring and comparing their performance would be hard to be general.

The essential workload for Lustre file system checker is to scan and check the metadata of all stored data files. The metadata checking is based on the redundant metadata stored in different places. As we have described, Lustre has three types of metadata: mapping metadata, namespace metadata, and data layout metadata. For each of them, there will be corresponding reference metadata to be used for checking:

- For *mapping metadata*, each mapping from a Lustre File Identifier (FID) to a local file system *inode* will have a corresponding metadata stored in the local *inode* to point back to the Lustre entity it represents.

- For *namespace metadata*, each directory or file *inode* will have a corresponding `linkEA` stored in the *inode* pointing back to the parent directory FID.
- For *data layout metadata*, on metadata server, each Lustre file needs to store the locations of all its stripes; while on the object storage server, each object will store the FID of its corresponding Lustre file as redundant reference.

To check these redundant metadata, LFSCK runs on both metadata servers (MDSes) and object storage servers (OSSes) to check the mapping metadata; runs on MDSes to check the namespace metadata; and runs on both MDSes and OSSes to check the data layout metadata.

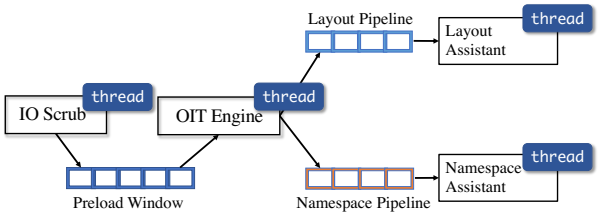

Fig. 2. Lustre file system checker (checking phase) on MDS.

Lustre designs and implements LFSCK as several kernel threads connected with kernel buffers. We show the overall flow of the checking phase on MDS in Fig. 2. There are in total four kernel threads running concurrently. The *Scrub* thread is to check the mapping metadata on all servers. To maximize the performance, it is designed to sequentially scan the local disks and iterate all the local *inodes*. The *OIT Engine* thread is the main LFSCK engine. It also iterates all the local *inodes*. But, it is kept always slower than the *Scrub* thread. The maximal difference between the iterators of *Scrub* and *OIT Engine* is defined as a constant `SCRUB_WINDOW_SIZE` (default value is 1024). If the *Scrub* thread goes too fast, it will stop and wait for *OIT Engine*.

While *OIT Engine* iterates all the local *inodes*, it will detect whether it is a file or directory and start layout checking or namespace checking respectively. For either case, the engine will formulate a checking request and put it into the layout pipeline buffer or the namespace pipeline buffer. The layout and namespace assistant threads are both started by the main engine and run asynchronously. They will retrieve the checking request from the pipeline buffers and conduct the real checking. For the layout checking, it indicates sending the RPC requests to corresponding OSSes and checking the back-reference metadata stored on the striped object's metadata. For the namespace checking, it indicates reading subdirs and subfiles' metadata from local disk and check them. The size of layout and namespace pipeline buffers is by default initialized from a constant `LFSCK_ASYNC_WIN_DEFAULT` (default value is 1024), but can be modified when start a LFSCK run. The buffers are managed in a classic consumer-producer way. It will block the producer or the consumer given the buffer is full or empty.

The flow of the checking phase on OSS is much simpler. It runs the *Scrub* thread as the MDS does, while at the same time, serves RPC requests from MDS about random objects' back-reference metadata.

As we described in this section, the Lustre metadata management and LFSCK are rather complex. During checking, local disks, network, remote disks are all used and potentially limit the performance. This makes identifying the bottlenecks and optimize LFSCK quite challenging.

III. Methodology

A. Evaluation Platform

To make our evaluation and analysis meaningful and repeatable to the community, we utilized the NSF CloudLab platform (i.e., Wisconsin Cluster) instead of a particular in-hourse HPC cluster to conduct all tests [14]. We started the experiment using *c220g1* nodes, each of which has two Intel E5-2630 CPUs, 128GB memory, two 1.2 TB 10K RPM hard disks, one Intel DC S3500 480GB SSD, and 10GB Intel NIC network connections. To run the experiment, we also created a profile and needed disk images in CloudLab. We also plan to release the profile for others to repeat the experiment. The new disk image has CentOS 7.3 and Lustre 2.10.4 installed. In all the experiments, we deployed the Lustre cluster with one metadata management server (MDS) and eight object storage servers (OSSes). Although the scale could be larger, from our evaluation results, we believe such a scale is already capable of revealing the performance characteristics and potential bottlenecks of LFSCK. Also, using smaller cluster can minimize the resource consumption as the aging and testing may take a considerable amount of time.

B. Aging Method

Since LFSCK runs on an aged file system, we need to properly create files and directories on the Lustre file system before running our experiments. This is actually a non-trivial task as the generated directory structure, files number, file sizes, and many other factors may affect how the file system and its checker work [15], [16]. In this study, we leveraged the I/O traces of a real-world supercomputer to age our Lustre file system. Specifically, we utilized the Darshan [17] logs collected from the Intrepid supercomputer at Argonne National Lab in 2013 [18], [19]. It contains the I/O traces of over 50% applications running in Intrepid that year. Each Darshan log file records a summary of the per-file I/O activities done by a single HPC application. We leveraged two key I/O metadata from the logs: 1) `CP_SIZE_AT_OPEN`, which indicates how large the file was when the application opened it; 2) `CP_BYTES_WRITTEN`, which indicates how many bytes were written to the file by this application. By adding this two numbers, we get a rough guess on the size of the file after running the application. There could be some inaccuracy as the newly written bytes do not necessarily add to the file size. But, for our aging purpose, we consider such an error is acceptable, plus that, one file might be accessed multiple times by different applications, the later access will have an accurate `CP_SIZE_AT_OPEN` to correct previous guess and keep the errors controllable.

Darshan anonymized the file's full path for privacy reason. Hence, we can not re-generate the exact directory structure of the file system, and have to generate it manually. Specifically, each time, we selected a random number of files ($1 \rightarrow 100$) from all the files and put them into a directory generated with a random depth ($1 \rightarrow 10$). These parameters are all tunable, so that users can generate the directory structure as they want, such as generating huge directories.

One thing worth noting is that the number and size of files collected from the Intrepid cluster are far too large to fit into our local experimental cluster. For example, there are files that are over 1TB each, while the total capacity of our 8 OSS Lustre cluster totals only 8TB. To age the Lustre filesystem using the collected workload, while limiting the amount of space used by each file, we set the stripe count of files to n*1MB, and all files that are larger than 8MB are striped over all 8 OSTs, with only 8MB of data stored for each file. In terms of metadata management this is equivalent to storing the full amount of data for each file.

In a production deployment, Lustre typically stores files with a stripe count of 1 to 4 for most files, though some large files may be striped over hundreds or thousands of OSTs for performance and capacity reasons. Using the maximum stripe count of 8 for a majority of files allows exploring LFSCK performance bottlenecks more easily.

C. Monitoring Framework

We build a framework to monitor the disk and network usage of all nodes in the Lustre cluster while running LFSCK. The framework leverages `iostat` [20] to monitor the storage devices on all MDS and OSS nodes where Lustre was mounted. Similarly, we use `netstat` to monitor the utilization of the network cards in the cluster used by Lustre [21]. By default, the framework uses 1 second as the interval to output the states of the resource utilization.

IV. LFSCK Bottlenecks: Evaluations and Analysis

In this section, we present our experimental results of running LFSCK and analyze its bottlenecks. We first discuss its scalability issues and then dig deeper into its internal implementation issues.

A. Scalability Bottlenecks

In the first experiment, we keep the number of OSS nodes unchanged, but increase the number of files in the system. In other words, we evaluate the LFSCK performance with an increasing number of *inodes* being used in the system. Specifically, we increase the used inodes from 0.4 million to 2.3 million. As shown in Figure 3, when the used inodes (the x-axis) increases, the execution time of LFSCK increases almost linearly. Note that in production HPC systems it is not uncommon to have over billions of files, the scalability issue would thus be more severe than our results indicate.

In the second experiment, we keep the total number and size of files unchanged, but increase the number of OSS nodes. Specifically, we start with a Lustre with 2 OSS nodes, and then increase the OSS nodes to 4 and 8. For each setup, we maintain the same number of used inodes via our aging tool (i.e., 0.88 million). Table. I shows the execution time of LFSCK in the three cases. We can see that as the number of OSS nodes grows from 2 to 8, LFSCK takes much longer time (1.50X) to finish. In this experiment, the number of files and the corresponding used capacity do not change, only the number of file stripes increases as more OSS nodes are included (due to increasing the stripe count to always place all files on all OSTs). This indicates that striping all files over all OSS nodes could slow down LFSCK performance significantly, and is not recommended for production systems. This observation also may imply the potential scalability challenge of running LFSCK in the future Exascale Lustre deployment, where the number of OSS nodes and the number of file stripes could both be huge.

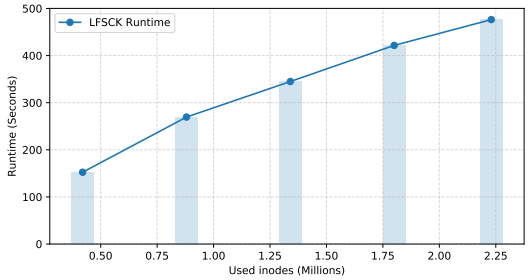

Fig. 3. The trend of LFSCK performance as the number of inodes increases.

TABLE I
Execution time of LFSCK (in seconds) on Lustre with different number of OSS nodes

# of OSS Nodes	2-OSS	4-OSS	8-OSS
Execution Time	144.8	170.5 (**1.18X**)	218.4 (**1.50X**)

B. Internal Bottlenecks

After observing the scalability issues of LFSCK, we further investigate where is the performance bottleneck internally. In this set of experiments, we use a Lustre with eight OSS nodes. We age the Lustre to a level that 2.5 million *inodes* were taken and 4.8 TB of the storage capacity was used. Also, we monitor the disk and network utilization of all Lustre nodes during the entire execution of LFSCK as described in Section III. Due to the space limitation, we only show the results on the MDS and one OSS in this section. Other OSSes have similar behaviors.

We plot the results in in Fig. 4. In this particular run, LFSCK finishes in about 500 seconds by looking at the execution log. This can also be seen through the idle disks and networks from the figure.

In subfigure (a), we can observe that the MDS node has disk read activities until after 500 seconds, while in (b) the OSS node only has read operations in the first 90 seconds. As we described in Section II, OSS plays two roles in file system checking. First, it iterates local *inodes* to verify its *Object Index*. Second, it answers the requests from MDS about layout of a file to conduct layout checking. Therefore, the early

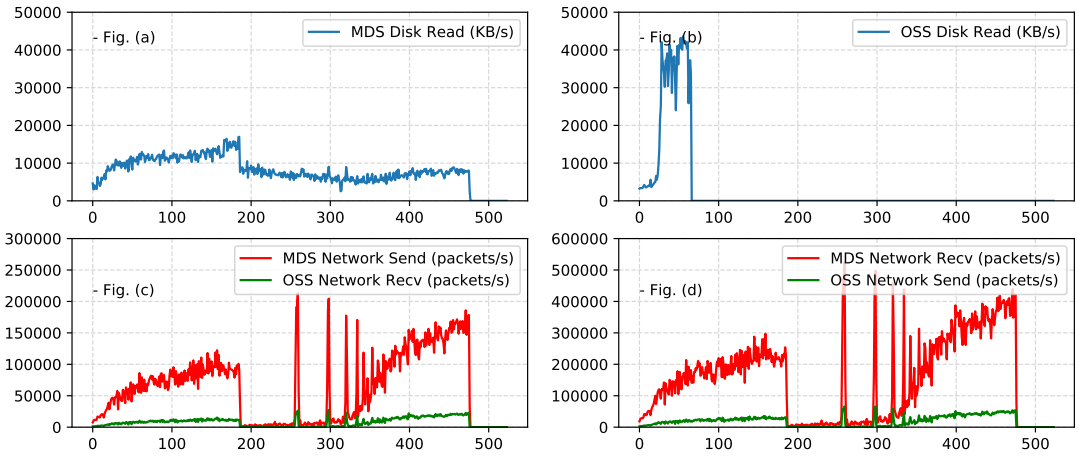

Fig. 4. **Resource utilizations while running file system checker**. *The x-axis shows the time in second, the y-axis for "disk read" shows the I/O bandwidth in KB/s, the y-axis for "network" shows the network packets in packets/s. We did not show the write I/Os because they are not caused by file system checking.*

finish of disk I/Os on OSS implies two points: 1) verifying the *Object Index* could be very efficient, as it finishes quickly on the OSS node and the I/O bandwidth keeps at 40 MB/s; 2) serving the layout checking requests from MDS may need no disk I/Os at all. This is possible as the *inodes* metadata are much smaller than the actual data size. After iterating all of them in previous stage, the server will be able to buffer them in memory, significantly reducing the future disk I/Os.

This figure also exposes the performance issue of LFSCK internally. First, in both MDS and OSS nodes, the network and disk bandwidth are not fully utilized. On our evaluation platform, all nodes are having the same hard disk, which easily provides over 100 MB/s read/write performance towards sequential I/Os when formatted as `ext4`. However, during LFSCK, the I/O bandwidth on MDS never exceeds 20 MB/s and on OSS nodes barely reach 40 MB/s. As previous description, in the first stage of file system checking, the OSSes basically run the Object Index Scrub (OI-Scrub) to iterate local *inodes* sequentially. The slowness might just come from the fact that the *inode* tables are small and discontiguous on the disks. But, the disk accesses on MDS is even slower than the OI-Scrub. Based on LFSCK implementation shown in Fig. 2, this should because of the slower *OIT engine*, which is blocked by one of the connected two pipelines (layout and namespace). Hence, either layout checking or namespace checking should be slow in this case and drag down the overall performance.

C. Layout Checking Bottleneck

In LFSCK, the layout checking is driven by the metadata server. MDS sends requests to all OSSes and collects their replies for checking. This can be seen in Fig 4 (c), where the MDS sent around eight times more packets than one OSS received. Such a multiple may further increase when more OSSes are deployed and more larger files are stored in the system, which leads to potential scalability issue. In addition, from Fig. 4 (c) and (d), we can see that MDS receives as many as twice packets than it sends out. This is because that multiple OSSes are replying to the MDS at the same time. Such a fan-out ratio can saturate the MDS network even earlier (on the receiving side) when the Lustre scales.

To exactly show the network bottlenecks of layout checking, we measure the network transmission speed of LFSCK again, but with all metadata on MDS and OSS nodes were buffered in memory already. As shown in Fig. 5, LFSCK took around 130 seconds to finish in this ideal case (i.e., all metadata are in memory). During the execution, both send/recv packets per second are kept as a constant, relatively high number on the MDS node. Particularly, it received over 600K packets per second (the red curve) and sends over 250K packets per second (the blue curve). Similarly, in this case, MDS received double packets compared to what it send. This may lead to a situation where the sending bandwidth of MDS is still far way from its upper bound, but the receiving bandwidth is saturated. As a result, the layout checking could be blocked despite of the under-utilized sending bandwidth, hurting the overall performance of LFSCK.

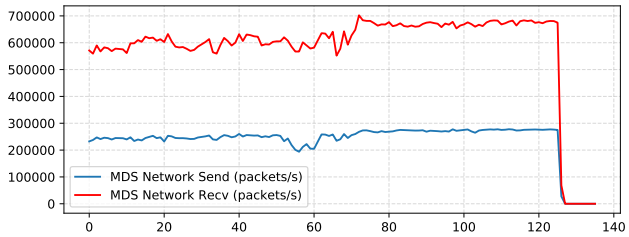

Fig. 5. Network packets on MDS with metadata buffered.

By further looking at the network transmission speed on MDS in Fig. 4 and Fig. 5, we can see that during the particular LFSCK run in Fig. 4, the network has not become a bottleneck yet. Both incoming and outgoing bandwidth are far below its physical limitation. However, this under-utilization of network bandwidth could change if there are more OSSes deployed or more large files stored. For example, if there are 16 OSSes and 1 MDS, and all files are striped over all 16 OSSes, during time frame $400s \rightarrow 500s$, the MDS will probably receive

double packets (more than 700K packets/sec) to keep up with the *OIT Engine*. Such a throughput is more than the network transmission limit shown in Fig. 5 (i.e., 600K packets/s). In this case, the OIT engine could be slowed down by the network transmission.

D. Namespace Checking Bottlenecks

In LFSCK, the namespace checking only happens on the metadata server and involves disk accesses to local *inodes* if needed. Fig. 4 already shows that the disk read bandwidth on MDS was very low (e.g., lower than 7.5 MB/s for about 300 seconds) while on OSS nodes, the disk read bandwidth keeps around 40 MB/s. This may be caused by the slow namespace checking.

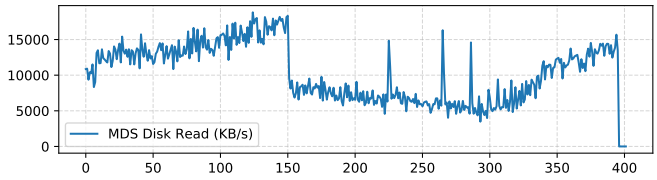

Fig. 6. Disk bandwidth on MDS during namespace checking.

To verify this, we examine the read bandwidth on MDS when only run the namespace checking part of LFSCK. In this case, there is no layout checking, so there is no I/O activities on OSS nodes or network transmissions between MDS and OSS nodes. As shown in Fig. 6, running namespace checking alone takes around 400 seconds and still incurs the similar low disk bandwidth (i.e., lower than 17.5 MB/s and keeps at 7.5 MB/s for a long time). One potential reason for such low performance is that the namespace checking incurs many random I/Os, which can not be served efficiently by the devices. Taking Fig. 4 into consideration, we can conclude that it is the slow namespace checking blocks the *OIT Engine* and slows down the layout checking in this experiment. Note that in our experiment the MDS was running on HDD. As the random I/O performance on SSDs is much higher than that on HDDs, replacing HDDs with SSDs may help alleviate the random I/O bottleneck of the namespace checking.

E. Tight Binding of Layout and Namespace Checking

Since namespace checking only runs on MDS and involves a large amount of random disk accesses, it can easily become the performance bottleneck as shown in earlier test. However, this is not always the case. It depends on how fast the local data accesses are (e.g., using SSD vs. using HDD), how many OSSes are deployed, and how files are stripped in Lustre (e.g. lower stripe count vs. higher stripe count) etc. For example, if there are 16 OSSes instead of 8 OSSes, and the default stripe count is 16 (all files can be allocated on all OSSes), the MDS network might not be able to send/receive enough layout checking requests, hence block the *OIT Engine*. Or, if the metadata are already pre-loaded into memory or the MDS is running on fast NVMe SSDs, the namespace checking could be extremely fast, making layout checking the bottleneck.

In addition, a more complicated scenario is when the performance of layout checking and namespace checking dynamically changes during LFSCK. For example, a namespace checking may encounter a huge directory and block the *OIT Engine*, which consequently starves the layout checking. While, on the other hand, a layout checking request may be handled by a slow OSS or go through a temporally congested network switch. This will starve the namespace checking. In a word, the tight binding between namespace checking and layout checking may lead to frequent blocking, and thus may negatively affect the performance of LFSCK.

As discussed in section II-B, such a tight binding is closely related to two LFSCK parameters: `SCRUB_WINDOW_SIZE` and `LFSCK_ASYNC_WIN_DEFAULT`. Being able to dynamically adjust these two values based on current bottleneck might soften the binding. However, as current Lustre infrastructure does not support real-time inspection of the bottleneck nor online adjusting of these parameters during LFSCK runs, we plan such evaluation in the future work. In this paper, as a preliminary result, we will first show what to expect for the performance improvement if the bind can be completely removed (see Section V-A).

V. LFSCK Performance Potentials

From the experiments in the previous sections, we have identified several bottlenecks of LFSCK: metadata server (MDS) scalability bottleneck, high fan-out ratio in the MDS network, and the tight binding of different internal components. Correspondingly, these would also be the places where potential re-designs and optimization can be applied, such as: 1) the receiving over sending ratio on MDS network can be reduced via data compression when the OSS nodes reply; 2) namespace checking and layout checking can be decoupled while still leverage each others buffers. In the next subsection, we show the preliminary result if the binding between namespace checking and layout checking could be completely removed.

A. Performance with Binding Removed

We expect that the tight binding of different LFSCK internal components could be alleviated by dynamically adjusting two parameters: `SCRUB_WINDOW_SIZE`, `LFSCK_ASYNC_WIN_DEFAULT`. Although current Lustre does not support such feature yet, the potential benefit can actually be validated just using existing LFSCK infrastructure. Specifically, instead of running both namespace and layout checking together, we can run layout checking first and run namespace checking immediately afterward (note that the order is important). In this way, the namespace checking and layout checking are decoupled. Since our metadata server has 128GB memory, it can easily buffer most of the metadata scanned during layout checking, which emulates the effect that if the buffer is effectively managed without having any starving or full.

We conducted the experiment on the same well-aged Lustre file system as shown in Fig. 4. Our results show that the performance benefit is obvious. By executing the layout checking and the namespace checking sequentially, we are able to reduce the execution time of LFSCK to around 200 seconds,

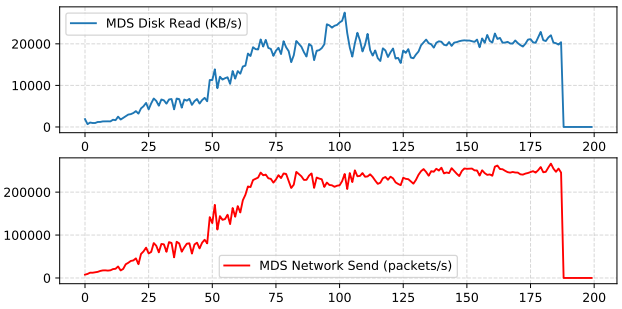

Fig. 7. **Performance of LFSCK with optimization.** The figure shows the disk and network activities when running the layout checking immediately followed by the namespace checking.

less than half of original run (500 seconds), which just ran both checking together. We show the detailed resource utilization in this new run in Fig. 7. It shows the performance benefit comes from higher disk I/O and network bandwidths during LFSCK, although there is still room for further optimization. Note that such a performance gain is based on the ideal, extreme case where all metadata can be buffered in memory during layout checking. For a large-scale Lustre deployment, this is unlikely to be realistic. Therefore, we expect dynamically adjusting the two parameters to soften the binding could be a desirable optimization, and we leave it as future work.

VI. Conclusion and Future Work

In this study, motivated by the slowness of LFSCK, we have conducted series of experimental studies on its performance to identify the root cause of the slowness as well the opportunities for performance optimization. Although as an online file system checking tool, LFSCK is designed to work with potential rate limits to avoid affecting the applications, we consider it is still worth exploring how to make LFSCK best utilize the available resources. We believe this study is the first step towards a new design for future Exascale Lustre file system checker. As the future work, we would like to investigate more potential bottlenecks on larger scale production systems, and implement the proposed optimizations in LFSCK/Lustre to improve the performance.

Acknowledgment

We thank the anonymous reviewers and Andreas Dilger (our shepherd) for their insightful feedback. This work was supported in part by NSF under grants CCF-1717630, 1853714, 1718336. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] "Lustre File System," http://opensfs.org/lustre/.
[2] "Top500 Supercomputers: NOVEMBER 2018," https://www.top500.org/lists/2018/11/.
[3] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *Trans. Storage*, vol. 4, no. 3, pp. 8:1–8:28, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1416944.1416947
[4] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu
[5] J. Cao, S. Wang, D. Dai, M. Zheng, and Y. Chen, "PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems," in *Proceedings of the 32nd ACM International Conference on Supercomputing*, ser. ICS'18, 2018.
[6] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey *et al.*, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Transactions on Storage (TOS)*, vol. 14, no. 3, p. 23, 2018.
[7] "HPCC Power Outage Event at Texas Tech," http://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf, 2016.
[8] "LFSCK: an online file system checker for Lustre," http://git.lustre.org/?p=fs/lustre-release.git;a=blob;f=Documentation/lfsck.txt.
[9] LFSCK High Performance Data Division - OpenSFS, http://wiki.lustre.org/images/c/c6/Zhuravlev_LFSCK_LUG-2013.pdf.
[10] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu, "Towards robust file system checkers," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, ser. FAST'18. Berkeley, CA, USA: USENIX Association, 2018, pp. 105–121. [Online]. Available: https://www.usenix.org/system/files/conference/fast18/fast18-gatla.pdf
[11] C. Faber, "Lustre DNE (Distributed Namespace) Basics," *UNIXgr, https://www.unixgr.com/lustre-dne-distributed-namespace-basics/*, 2014.
[12] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, vol. 215, 2003.
[13] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2, 2007, pp. 21–33.
[14] R. Ricci, E. Eide, and C. Team, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.
[15] K. A. Smith and M. I. Seltzer, "File system aging—increasing the relevance of file system benchmarks," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1. ACM, 1997, pp. 203–213.
[16] S. Kadekodi, V. Nagarajan, and G. A. Gibson, "Aging gracefully with geriatrix: A file system aging suite," 2016.
[17] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.
[18] "Intrepid," http://www.top500.org/system/176322.
[19] "Darshan Trace Data, 2013," http://www.mcs.anl.gov/research/projects/darshan/data/.
[20] "iostat(1) - Linux man page," https://linux.die.net/man/1/iostat.
[21] "netstat(8) - Linux man page," https://linux.die.net/man/8/netstat.