M.S. THESIS

An Efficient Journaling Mechanism in Lustre File System for Fast Storage Devices

고속 스토리지를 위한 Lustre 파일 시스템의 효율적인 저널링 머캐니즘

AUGUST 2018

서울대학교 대학원
컴퓨터공학부

Hangqi Cui

An Efficient Journaling Mechanism in Lustre File System for Fast Storage Devices

고속 스토리지를 위한 Lustre 파일 시스템의 효율적인 저널링 머캐니즘

지도교수 염헌영

이 논문을 공학석사 학위논문으로 제출함

2018 년 6 월

서울대학교 대학원

컴퓨터공학부

최 항 기

최항기의 공학석사 학위논문을 인준함

2018 년 7 월

위 원 장	신영길	(인)
부위원장	김명수	(인)
위 원	염헌영	(인)

Abstract

Journaling mechanism is a widely used technique to enhance file system robustness against metadata and data corruptions and minimize file system recovery times after a system crash. However, we found that journaling for the object store in Lustre file system significantly negatively affects the performance. Since it requires that each write request wait for a journal transaction to commit, Lustre introduced serialization to the client request stream and imposed additional latency due to the journal transaction flush for each request. Therefore, threads which handle the I/O operations can only wait for the journal daemon thread to complete journal transaction commit.

In this paper, we provide an efficient journaling mechanism with two schemes to increase the overall efficiency of the Lustre file system. First, we present a parallel journal I/O commit in a cooperative manner with I/O operation threads. This enables the parallel I/O operations instead of the serialized I/O. Second, we use on-the-fly commit to perform the commit processing on the fly without any delayed operations. We implement our schemes in Lustre file system and evaluate them on four 4-core machine with high-performance NVMe SSDs. The experimental results show that our optimized Lustre file system improves the performance by up to 30.1% compared with the existing Lustre file system.

Keywords: Lustre File System, journaling filesystem, ext4, multi-thread, parallel

Student Number: 2016-27542

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel distributed file system is a key part of massively parallel computing environment and widely used in clusters for I/O-intensive parallel applications. Lustre file system is best known for powering seven of the ten largest high-performance computing (HPC) clusters in the world with tens of thousands of client systems, petabytes (PBs) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput [15]. Currently, in the cloud computing era, the performance research of Lustre file system increasingly attracted the attention of industry and research communities [9, 12, 1, 16].

Although Lustre file system provides massively parallel computing environment, Lustre file system can face a performance bottleneck from its journaling technique on fast storage devices. Lustre file system adopts a journaling technique for transaction processing to guarantee the atomicity and durability [13, 11]. To support this journaling, Lustre file system uses an improved

version of the EXT4 file system [7] called *ldiskfs* [2] to store data and metadata. However, *ldiskfs* serializes the transactions for each write request [14] to increase the reliability of Lustre. This strategy can guarantee that the client data is written completely on stable storage after a client receives the reply for write request. Thus, the clients wait for the reply until the transaction is committed to storage completely. It can negatively affect the write performance, especially when we use the fast storage devices like NVMe SSDs [3, 4, 8].

To handle these issues, previous studies [10, 6] investigated these journal I/O operations in parallel distributed file systems. Oral et al. [10] remove the synchronous journal commit by asynchronously replying to the client. Lu et al. [6] propose delayed commit protocol that the requests of committing suboperations are submitted to the commit queue and in the meanwhile the execution flow can be returned back to applications immediately. Our study is in line with these approaches [10, 6] in terms of investigating the journal commit operations. In contrast, we focus on internal operations in the commit procedures.

In this paper, we propose an efficient journaling mechanism for parallel distributed file systems to improve I/O performance for fast storage devices. In our journaling mechanism, we provide two schemes which are parallel journal commit and on-the-fly commit processing. First, parallel journal commit enables journal I/O operations in a parallel and cooperative manner. This allows multiple threads to join and process the commit operations instead of blocking the threads. Second, on-the-fly commit processing enables journal commit to be processed in a on-the-fly manner instead of an exiting piggyback scheme. It can significantly reduce the beginning delay of journal processing. We apply and implement the techniques to transaction processing on Lustre file system

in Linux kernel 3.10.0. The experimental results show that our optimized Lustre file system with our schemes improves the performance by up to 30.1% compared with the existing Lustre file system.

In summary, our main contributions are as follows:

- We provide an analysis of synchronous journal commit which significantly affects the I/O performance.
- Based on the observations, we provide parallel journal commit and on-the-fly commit scheme.
- We implement both approaches in Lustre file system and evaluate on NVMe SSDs.
- We demonstrate that our optimized file system improves the performance compared to the existing file system.

The rest of paper is organized as follows. Section 2 describes background and motivation. Section 3 explains our design and implementation. In Section 4, we evaluate the performance impact of our scheme. Section 5 discusses relation works, and Section 6 concludes this paper.

Chapter 2

Background

2.1 Lustre file system

Lustre parallel file system has been widely adopted by the HPC(High-Performance Computing) systems. Lustre is an open-source parallel file system technology and used by many fast HPC systems. Among others, it has been selected by Cray for their products, and will be available on their upcoming petascale systems to be installed in the ORNL National Leadership Computing Facility. The Lustre architecture is a storage architecture for clusters. The central component of the Lustre architecture is the Lustre file system, which is supported on the Linux operating system and provides a POSIX standard-compliant UNIX file system interface.

Lustre as an object-based file system, it composed of three components:

Metadata target (MDT). The MDT stores metadata such as filenames, directories, permissions and file layout on storage attached to an MDS. A meta-

data server (MDS) makes metadata stored in one or more MDTs available to Lustre clients. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs. An MDT on a shared storage target can be available to multiple MDSs, although only one can access it at a time. If an active MDS fails, a standby MDS can serve the MDT and make it available to clients. This is referred to as MDS failover.

Object Storage Target (OST). User file data is stored in one or more objects, each object on a separate OST in a Lustre file system. The number of objects per file is configurable by the user and can be tuned to optimize performance for a given workload. The OSS provides file I/O service and network request handling for one or more local OSTs. Typically, an OSS serves between two and eight OSTs, up to 16 TB each. A typical configuration is an MDT on a dedicated node, two or more OSTs on each OSS node, and a client on each of a large number of compute nodes.

Lustre Clients. Lustre clients are computational, visualization, desktop, or data transfer nodes that are running Lustre client software, allowing them to mount the Lustre file system. The Lustre client software provides an interface between the Linux virtual file system and the Lustre servers.

When a client wants to read from or write to a file, it first sends a request to MDS to fetch the layout EA(extended attributes) from the MDT object for the file. The client will then use the acquired information to perform I/O on the file, directly interacting with the OSS nodes where corresponding objects are stored.

2.2 Lustre Backend File System – ldiskfs

The Lustre file system uses a heavily patched version of the Linux EXT4 filesystem to store data and metadata. This version, called *ldiskfs*, is a superset of Linux EXT4 filesystem and used by Lustre MDS and OSS as an underlying local filesystem. Instead of using the existing EXT4 filesystem I/O path, the I/O path of *ldiskfs* has been optimized for improving performance by OFD/OSD module. Therefore, there are several differences between *ldiskfs* and EXT4 filesystem. The main differences between the EXT4 and *ldiskfs* filesystems in terms of their respective I/O paths are like follows:

- In *ldiskfs*, allocation/block lookup is done up front with inode lock held, but then inode lock is dropped while the I/O is being submitted to disk. This greatly improves I/O concurrency because inode is locked for a short time.
- In *ldiskfs*, allocation is done for the entire RPC(1MB) at one time, instead of 4KB block-at-a-time lick VFS does. This avoids lots of overhead in repeated bitmap searching and also allows more efficient extent finding.
- In *ldiskfs*, writes are fully synchronous to the client, which means no reply until data and metadata are on disk.
- In *ldiskfs*, currently journal flush is forced after every write RPC.

These features may not affect buffered writes but have a huge impact on direct writes. In this case, the number of journal commits is even more than data commits.

2.3 Journaling Block Device

The Journaling Block Device (JBD) provides a filesystem consistency by journaling mechanism. EXT4 filesystem uses a fork of JBD, and this version called JBD2. Due to the *ldiskfs* based on EXT4 filesystem, it also uses JBD2 for keeping filesystem consistency. EXT4/JBD2 provides three journaling modes: write-back, ordered, and data journaling. However, *ldiskfs* does not provide data journaling mode, since this mode would double the amount of data being written to disk and cut the write performance by 50%, which is not something that users want.

JBD2 adopts a single compound [4, 5] transaction. There is only one running transaction that absorbs all updates and on committing the transaction at any time. An OST io thread starts a running transaction for each update and associates the update with the transaction. The running transaction has a linked list, which has the pointers to the modified blocks. Due to the *ldiskfs* will force a journal flush after every write RPC in direct write operation, a journal thread changes the state of the transaction to *committing*. and writes the blocks associated with the transaction into the journal space. After the transaction commits, the transaction is marked as checkpoint. The committed blocks are written back to the original area by an application thread during the checkpoint operation. Then, the journal area is reclaimed via the checkpoint operation.

The following list summarizes the steps taken by *ldiskfs* for a Lustre journal commit in the default direct write with JBD2 daemon thread. The sequence of events is triggered by a Lustre OSS when it gets a write request from a client.

1. OSS opens a transaction on its back-end file system.

2. OSS updates file metadata in memory, allocates blocks and extends the file size.

3. OSS closes transaction handle and obtains a wait handle.

4. OSS OSS writes pages with file data to disk without waiting.

5. After current running transaction is closed, server flushes updated metadata blocks to the journal device and then marks the transaction as committed.

6. Once transaction is committed, server can send a reply to client that the operation was completed successfully and client marks the request as completed.

Although Step 3 minimizes the time a transaction is kept open, the above sequence of events may be sub-optimal. For example:

- The write I/O operation for a new thread is blocked on the currently closed transaction which is committing on.
- The new running journal transaction has to wait for the previous transaction to be closed.
- New I/O RPCs are not formed until the completion replies of the previous RPCs have been received by the client creating yet another point of serialization.

2.4 Journal Commit Transaction

We focus on the I/O operations in transaction processing, and optimization of the performance in fast storage like NVMe SSD. To commit a transaction, a

JBD2 daemon thread wakes up and processes a commit procedure by calling `jbd2_journal_commit_transaction` which is the main function of journal commit transaction in JBD2 daemon thread. In this function, it first changes the running transaction to a committing transaction and its state to committing by acquiring the state lock. Then, the JBD2 daemon thread waits for other application threads to finish their metadata updates in current transaction by checking the `t_updates` variable. If the jh is already associated with a running transaction, the jh must be moved to a committing transaction. In the meanwhile, the committing transaction does not accept any new thread join this transaction, and the next modification triggers the creation of a new running transaction. With the committing transaction, the JBD2 daemon thread prepares for the journal I/Os by creating an I/O list, which is used to wait for the completion of I/Os. Then, the JBD2 thread fetches the jh from the head of the transaction metadata buffer list and creates a copy of its buffer to a newly allocated journal space block which points to the journal space block while metadata buffer point to the original block address. At this time, JBD2 thread removes the jh from the metadata list by updating the head of the list to the next of the head and inserts the jh into the shadow list under the list lock.

To perform a batched journal I/O, the jbd thread aggregates the I/O buffers by inserting it into a write(wbuf) and the IO list. If the number of inserted buffers(bufs) is more than the maximum number of buffers allowed at one time or the metadata buffer list empty, the JBD2 thread issues I/Os to the journal area by calling `submit_bh()` and prepares for the next I/Os. After issuing all the I/O buffers for journaling, the JBD2 thread waits for the completion of the metadata I/Os. And then removes the jh from the shadow list and inserts it into the forget list under the list lock. After all the I/Os are completed, the journal

thread writes the commit block for the transaction atomicity; if a crash occurs, the file system can replay or discard the transaction according to the existence of the commit block of the transaction in journal space. Then the journal thread makes a checkpoint list with the buffers that are not freed and still dirty in the forget list under the list lock. The committed transaction is inserted into the tail of a checkpoint transaction list for checkpointing by acquiring the state and list locks. Finally, the JBD2 daemon thread will wake up all sleep waiting OST I/O threads by calling `wake_up()` function.

Chapter 3

Design and Implementation

As mentioned above, Lustre uses journals to guarantee metadata consistency, when a client receives an RPC reply for a write request, the data is on stable storage and would survive a server crash. Although this implementation ensures data reliability, it serializes concurrent client write requests because the currently running transaction cannot be closed and committed until the prior transaction fully commits to stable storage. The OST will begin processing and then all these requests will block on sleep waiting for their commits. Then, after all commit done, replies for corresponding completed operations will be sent to the requesting client, and then the client will send its next chunk of I/O requests. If there is only one client requesting service from a particular OST, the inherent serialization in this algorithm is more pronounced, waiting for each transaction to commit introduces significant delay.

An obvious way to solve this problem is to send the responses to clients im-

mediately after the file data portion of a RPC is committed to stable storage. And previous researches have implemented a "*asynchronous journal commits*" technique [10] which has achieved quite good results. However, the implementation of this algorithm boots the dirty and flushed data pages in client memory, and the client releases these pages only after receiving a confirmation from the OST indicating that the data was committed to stable storage. Therefore, such asynchronous journal commit technique cannot work with direct I/O-originated writes, which set `O_DIRECT` flag. In this case, a journal flush is forced.

To achieve higher I/O performance on multi-cores with fast storage in Lustre direct write, we aim to reduce the synchronous journal commit overhead by introducing parallel journal commit in transaction processing. Our implementation provides a parallel journal commits in a cooperative manner, which not only improves journal comit parallelism, but also reduce the performance bottleneck from blocked journal commits

3.1 Parallel Journal Commit in a cooperative manner

In the last chapter we briefly explained how a JBD2 daemon thread processing journal commit transaction. After all other application threads finish their metadata updates, the JBD2 daemon thread will start to prepare I/O lists, which we call metadata I/O preparation step. Initially, JBD2 thread will allocate a new block from journal space for descriptor block, which describes the mapping relations between existing metadata block and journal metadata block. After that, every metadata buffer will be allocated to a new block from journal space, and copy the content of the buffer to the newly allocated block. Then, update the corresponding tag in the descriptor. When all IO lists are prepared,

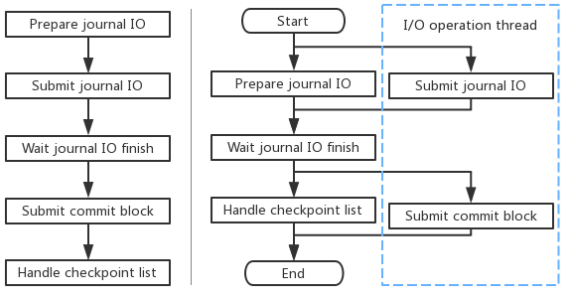

Figure 3.1 Control flow of journal transaction commit process

it calls `starts_journal_io`, which we call the metadata I/O submission step. In this step, JBD2 thread submits all the metadata IO buffers sequentially.

In our implementation, instead of waiting for all metadata I/O preparation step, we utilize the sleeping OST I/O operation thread to assist the JBD2 thread to immediately commit the metadata I/O buffer. We set a count variable in `transaction_s` structure to present a current number of write buffer list. When a new metadata I/O buffer is inserted into the list, the I/O operation threads that are supposed to be sleeping will detect the count change immediately. They fetch the buffers in the transaction buffer list concurrently and write them to the journal space in parallel by calling `submit_bh()`. It overlapped the metadata I/O preparation step and metadata I/O submission step, and significantly reduce the metadata I/O submission time.

The left part of the Figure 3.1 shows the existing control flow of journal transaction commit. Due to the JBD2 daemon thread execute a single thread journal processing, all processes are performed sequentially. The right part of the Figure 3.1 shows the parallel journal commit flow. In this pattern, journal buffer submission and commit block submission will be executed in parallel by the I/O operations threads which should have been asleep.

In order to distinguish the write operation between synchronous OST write

and other journal commit like mount operation, we set a flag on the OST write path to determine whether the current write operation will flush a journal. If the flag is set to true, the write operation will go into parallel journal commit path. Otherwise, it will go into the existing write path which may not occur any write operation delay caused by journal transaction commit.

To save the CPU utilization and occupation, our implementation will not force all the OST I/O operation go into parallel journal commit path. OST I/O thread will spin locally and not release the CPU occupation to wait for the JBD2 daemon thread to prepare a new metadata buffer. However, due to the metadata I/O preparation processes, not all waiting OST I/O thread can acquire newly inserted metadata I/O buffers, which significantly wastes CPU utilization and occupation. We first use atomic operation to count how many OST I/O operation threads can be used and how many metadata buffers need to be committed. Every OST I/O operation thread will get a unique ticket number according to the order of arrival. We also assign the number of the metadata buffers in the current transaction to each OST I/O operation thread. If we don't need so many OST I/O operation threads, it will release the CPU occupation by calling `wait_event()`.

After issuing all the metadata I/O requests for journaling, JBD2 daemon thread waits for the completion of I/Os. In this step, JBD2 thread will free completed metadata I/O buffers, remove the corresponding buffer from shadow list and insert into forget list. After this step, JBD2 thread starts flushing the commit block that guarantees the atomicity of the transaction. Considering the role of the commit block, it can not start committing until all metadata I/O requests are complete. The first arriving I/O operation thread will wait for the metadata I/O to complete and start submitting the commit block. At this

time, the JBD2 daemon thread will continue the journal commit transaction processing concurrently to finally implement the checkpoint mechanism.

3.2 On-the-fly Commit Processing

During the experiment of the parallel journal commit mechanism, we found that with the increase in the number of I/O threads, the journal transaction commit process present a huge overhead while waiting for the `t_updates` variable [13] which uses to wait all other application finish their metadata updates. In order to better understand the cause of the problem, we analyzed the corresponding code.

We found that there is piggyback mechanism at the end of the journal stop operation. This part of code determines whether the current execution is synchronous transaction batching. If the handle was synchronous, the piggyback mechanism will not force a commit immediately. It yield and let another thread piggyback onto this transaction and waiting for a delta time between current transaction run time and current transaction average commit time. This mechanism is helpful for slow storage device based on SATA interface because it can compound multiple application threads into single transaction, which significantly reduce the number of journal transaction commits. However, in fast storage device, the delay of the transaction will seriously impact the performance of transaction.

The Table 3.1 shows the analysis results of transaction processes. The *Original Lustre* is the default Lustre filesystem and *Parallel Lustre* is the new Lustre with parallel journal commit approach. We perform totally same workload with different Lustre filesystem. We can see that while our Parallel Lustre filesystem

Label	Number of Trx	Performance	Trx startup delay
Existing Lustre	50780 times	16.7 Mb/s	13.9%
Parallel Lustre	68435 times	19.4 Mb/s	13.8%
Optimized Lustre	76547 times	21.1 Mb/s	5.9%

Table 3.1 Transaction commission times and time of waiting for `t_updates`

generates more transactions than existing Lustre, it still has better performance due to the parallel journal commit enhanced journal transaction processing. However, there is still a large transaction delay due to the piggyback mechanism. We tried to disable part of the code and accelerate the start of the transaction. The *Optimized Lustre* shown in the Table 3.1 is the Lustre file system which perform On-the-fly commit, it increases the number of transactions because the *Optimized Lustre* does not wait for other application threads to join the current transaction. At the same time, it reduces the transaction startup delay for better performance.

Chapter 4

Evaluation

4.1 Evaluation setup

We used four 4-cores machine with one Intel(R) Core(TM) i7-4790 CPU(support Hyper-threading), 32GiB DRAM, 10GbE network card. One for MDS, two for OSS and one for client. Each OSS has one 375 Gib Intel(R) Optane(TM) SSD DC P4800X for OST. The machines run CentOS 7.2 distribution with a Linux kernel 3.10.0. We evaluate the existing Lustre file system and fully optimized Lustre (O-Lustre) file systems in the default mode. To present a performance breakdown, we also evaluated an optimized Lustre file system with our Parallel Journal Commit (P-Lustre) feature. We run the workloads with the direct flag set, and we also evaluated the workloads with *fsync* intensive which perform synchronous I/O operation. We vary the number of CPU cores from 2 to 16, and run each test several times and report the average.

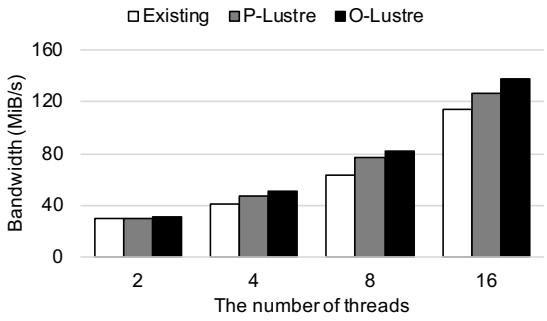

Figure 4.1 Fio benchmark performance with number of threads

4.2 Performance results

4.2.1 FIO performance

We present the results of performance with FIO benchmark. We configured FIO to issue sequential write operations using a variety of threads, 1 GiB file size per thread, 4 KiB block size and direct write.

In the case of two thread shown in the left of Figure 4.1, for each OSS only one FIO thread assign to the server. Since there is no any other application thread can be piggyback to the current transaction, the performance of O-Lustre and P-Lustre showing similar results. As the number of threads increase, the performance of P-Lustre shows better performance due to the Parallel Journal Commit make the journal commit processing in parallel. At the number of thread is 8, it shows the best performance of 22.0% increment than existing Lustre. Except the two thread benchmark, The performance of O-Lustre is better than the performance of P-Lustre. It is due to the O-Lustre have optimized piggyback mechanism with On-the-fly commit which will not occur to much journal transaction delay caused by other application thread join into current

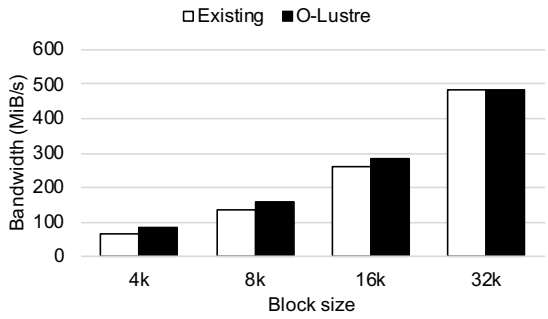

Figure 4.2 Fio benchmark performance with number of block size

running transaction. And it shows the best performance when the number of threads is 8, in this case, the performance of O-Lustre is 30.0% better than existing Lustre.

Figure 4.2 shows existing Lustre and O-Lustre in terms of FIO performance with different block size. As the size of block increases, the performance of both Lustre increases exponentially. Compare to the existing Lustre, our O-Lustre shows the best performance when the size of block is 4k. As the size of block increases, the performance improvement of O-Lustre decreases slightly compared to existing Lustre. This is because that as the size of block increases, there will be less journal transaction should be committed with a same workload. The best improvement occurs in 4k block size, in this configuration, it could maximize the file system overhead by synchronous journal commission.

4.2.2 IOR performance

IOR benchmark is a commonly used file system benchmarking tool particularly well-suited for evaluating the performance of parallel file systems like Lustre file system. We configured IOR benchmark option with 4 KiB transfer size, 4 KiB block size, 100 MiB file size per client. We issued sequential POSIX write

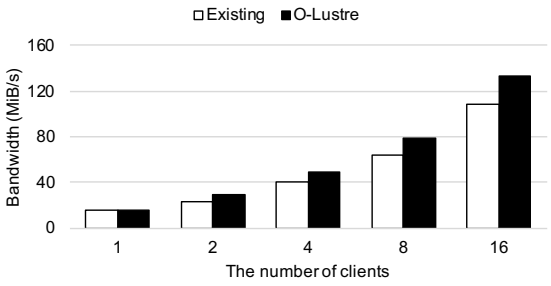

Figure 4.3 IOR benchmark performance with number of threads

operation with O_DIRECT flag set in. We set the Lustre stripe count as 2, which means each file divide into two parts and send to each OSS separately. We experiment the bandwidth under different number of clients which very similar with number of threads.

Under the IOR workload as shown in Figure 4.3, when the number of IOR clients is 2 to 16 the O-Lustre improves the performance compared to existing Lustre by 30.1%, 20.2%, 22.1% and 23.2%. The highest performance still show in the number of clients is 2 which shows 30.1% of improvement.

Bibliography

[1] Alam, S. R., Barrett, R. F., Fahey, M. R., Kuehn, J. A., Messer, O. B., Mills, R. T., Roth, P. C., Vetter, J. S., and Worley, P. H. An evaluation of the oak ridge national laboratory cray xt3. *The International Journal of High Performance Computing Applications 22*, 1 (2008), 52–80.

[2] Chasapis, K., Dolz, M. F., Kuhn, M., and Ludwig, T. Evaluating lustre's metadata server on a multi-socket platform. In *Proceedings of the 9th Parallel Data Storage Workshop* (Piscataway, NJ, USA, 2014), PDSW '14, IEEE Press, pp. 13–18.

[3] Kang, J., Hu, C., Wo, T., Zhai, Y., Zhang, B., and Huai, J. Multilanes: Providing virtualized storage for os-level virtualization on manycores. *Trans. Storage 12*, 3 (jun 2016), 12:1–12:31.

[4] Kang, J., Zhang, B., Wo, T., Yu, W., Du, L., Ma, S., and Huai, J. Spanfs: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 249–261.

[5] Kim, D., Park, J., Lee, K.-g., and Lee, S. *Forensic Analysis of Android Phone Using Ext4 File System Journal Log*. Springer Netherlands, Dordrecht, 2012, pp. 435–446.

[6] Lu, Y., Shu, J., Li, S., and Yi, L. Accelerating distributed updates with asynchronous ordered writes in a parallel file system. In *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012* (2012), pp. 302–310.

[7] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L., and S, B. S. A. A and viver, l. the new ext4 filesystem: current status and future plans. In *In*

Ottawa Linux Symposium. http://ols.108.redhat.com/2007/ Reprints/mathur-Reprint.pdf (2007).

[8] Min, C., Kashyap, S., Maass, S., and Kim, T. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 71–85.

[9] Oral, S., Dillow, D. A., Fuller, D., Hill, J., Leverman, D., Vazhkudai, S. S., Wang, F., Kim, Y., Rogers, J., Simmons, J., and Miller, R. Olcf's 1 tb/s, next-generation lustre file system.

[10] Oral, S., Wang, F., Dillow, D., Shipman, G., Miller, R., and Drokin, O. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2010), FAST'10, USENIX Association, pp. 11–11.

[11] Park, D., and Shin, D. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 787–798.

[12] S Buddy Bland, A., Rogers, J., A Kendall, R., Kothe, D., and M Shipman, G. Jaguar: The world's most powerful computer, 05 2009.

[13] Son, Y., Kim, S., Yeom, H. Y., and Han, H. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, 2018), USENIX Association, pp. 227–240.

[14] Wang, F., Oral, H. S., Shipman, G. M., Drokin, O., Wang, D., and Huang, H. Understanding lustre internals.

[15] Yu, W., Vetter, J., Canon, R. S., and Jiang, S. Exploiting lustre file joining for effective collective io. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)* (May 2007), pp. 267–274.

[16] Yu, W., Vetter, J., Canon, R. S., and Jiang, S. Exploiting lustre file joining for effective collective io. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)* (May 2007), pp. 267–274.