Exploring Lustre Overstriping For Shared File Performance on Disk and Flash

Michael Moore
Cray, Inc.
Austin, TX, USA
mmoore@cray.com

Patrick Farrell
Whamcloud
St. Paul, MN, USA
pfarrell@whamcloud.com

***Abstract*— From its earliest versions, Lustre has included striping files across multiple data targets (OSTs). This foundational feature enables scaling performance of shared-file I/O workloads by striping across additional OSTs. Current Lustre software places one file stripe on each OST and for many I/O workloads this behavior is optimal. However, faster OSTs backed by non-rotational storage show individual stripe bandwidth limitations due to the underlying file systems (ldiskfs, ZFS). Additionally, shared file write performance, for I/O workloads that don't use optimizations such as Lustre lock ahead, may be limited by write-lock contention since Lustre file locks are granted per-stripe. A new Lustre feature known as 'overstriping' addresses these limitations by allowing a single file to have more than one stripe per OST. This paper discusses synthetic I/O workload performance using overstriping and implications for achieving expected performance of next-generation file systems in shared file I/O workloads.**

Keywords-Lustre; Performance;

I. INTRODUCTION

Lustre [1] is a large scale, distributed file system which presents a POSIX file system to user applications, maintaining the same consistency semantics as a local file system. A Lustre file system back end consists of metadata targets (MDTs) and data targets (OSTs), which are served by metadata servers (MDS) and data servers (OSS). There is also a management server or MGS used for configuration. The basic model for scaling a Lustre file system is adding additional metadata and data targets.

Performance scaling in a parallel system is generally divided into vertical scaling, increasing the performance of individual components, and horizontal scaling, increasing the number of components in the system. Lustre enables horizontal scaling by distributing data across OSTs, including within individual files by striping the file contents across many OSTs. This capability – present since the earliest releases – allows scaling of both multi-file and single file workloads.

This simple approach works well for many scenarios and allows Lustre to scale from small clusters to the largest machines, but Lustre currently imposes some restrictions around striping. In particular, there is a one-to-one relationship between file stripes and OSTs, with only one stripe of a file placed on each OST. At the on-disk file system level, each file stripe is stored in a single object. A Lustre file is composed of one or more stripes and the data of each stripe is stored in an object in the on-disk file system. The term object will be used when referring to local file system behavior and stripe will be used when referring to the Lustre file. A shared file with a single stripe per OST is limited to a single object per OST; whereas with multiple files there are multiple objects per OST since each file has a different stripe.

The bandwidth available to a single object is restricted by limitations at several layers of Lustre, this is why peak I/O performance on new systems generally requires multiple files per OST. These limitations have been present for a long time, but new storage technologies have made them more acute.

A. Emerging Issues

For many years, growth in CPU frequency, which is the primary driver of single object and single stream performance, kept pace with growth in storage speeds. The slowing of Moore's Law and the advent of SSDs have altered these trends significantly. New distributed parity raid technologies such as GridRAID [2] and DCR [3] have also enabled extremely large arrays further exacerbating the problem. The result is that the fastest, modern OSTs are capable of 10 GB/s write bandwidth and trends suggest near future systems will be capable of several times that.

The limitations for single object performance fall in to two main categories, which we will discuss separately (1) local file system performance (2) shared file Lustre distributed locking (LDLM) contention

1) Local File System Performance

Each Lustre stripe corresponds to a single on-disk object in the on-disk file system of an OST. Lustre supports either the ext4 derived ldiskfs or ZFS as the on-disk file system, which leads to different specific limitations for each, but the general problems are similar. When adding data to an object in an on-disk file system, blocks must be allocated, local metadata updated, and, if page caching is in use, pages must be tracked in either the Linux page cache (ldiskfs) or ARC (ZFS).

Both the ARC and the Linux page cache have limitations to how much data they can add to a single object at a time. This limit is partly because very high bandwidth

applications require freeing pages to add pages, but it is also due to the general overhead of page tracking and data copying, all of which are CPU bound. In practice, both have single object limitations in the single digit GB/s range. Specific rate limitations vary depending on kernel version and hardware but are generally 5-10 GB/s for modern systems. Work is ongoing to improve performance in the Linux page cache, but the page cache and ZFS ARC implementations are already heavily optimized, so the scope for large improvements is limited.

Leaving aside caching, which can be disabled in some scenarios, we are left with the on-disk file system limitations. The upper limits of bandwidth to a single file in each file system have not been well explored, but there is reason to believe they are below the 10s of GB/s expected of future OSTs. These limits stem from the work and locking required to allocate blocks/extents and mange other local file system metadata.

In both the case of the page cache/ARC and the local file system limitations, it is possible and desirable to raise the limit by applying engineering effort, but each limitation represents significant engineering effort to overcome. Additionally, certain fundamental requirements mean that while the limitations can be increased, they will never be removed entirely.

2) Shared File Lustre Distributed Locking Contention

To manage file and object access between different clients, Lustre uses a distributed locking mechanism known as the Lustre distributed lock manager (LDLM). LDLM manages locking between distinct Lustre clients, and is fundamental to how Lustre presents a standard POSIX file system abstraction in a distributed environment with concurrent updates from many clients. This section is effectively an abbreviated version of the discussion in our previous paper [4], those looking for more information are encouraged to review the Lustre Locking Behavior section of that paper.

For this discussion, understand that LDLM extent locks are range locks granted by the OSTs upon request from clients. It is impossible for two clients to hold write locks on the same range, so when the server receives a conflicting write request, the existing lock must be cancelled before the new lock can be granted.

To write to a file a Lustre client must have a write lock covering the file range for the intended write. The Lustre client determines which stripe contains that portion of the file, and, if it does not already have the required lock, sends a lock request to the OST which contains that stripe. This lock request covers only the region required for the write.
While the client only asks for the region strictly required, it is inefficient to request a lock for every write, so the server attempts to return the largest non-conflicting lock. If an existing lock conflicts with the actual request from the client, that existing lock must be cancelled, and is cancelled before determining the "largest non-conflicting lock". Generally, this means clients acquiring write locks will acquire write locks on the entire stripe. This is desirable behavior in most scenarios, where the client will write repeatedly to the same stripe but can be problematic when more than one client wants to write to the same stripe as depicted in Fig. 1.

Multiple writers to a single file generally write in a strided pattern, where different writers alternate different blocks of the file depicted in Fig 2. Unless the client count and write size are perfectly aligned with the stripe boundaries, this means multiple clients will be writing to the same stripe of the file. Critically, they do not write to the same bytes of the file, so they should be able to proceed in parallel, but the default LDLM behavior prevents this.

Because of the optimization to grant the largest possible lock on each write, multiple clients writing to the same stripe result in false conflicts, where the optimistic locking behavior generates conflicts where none existed.

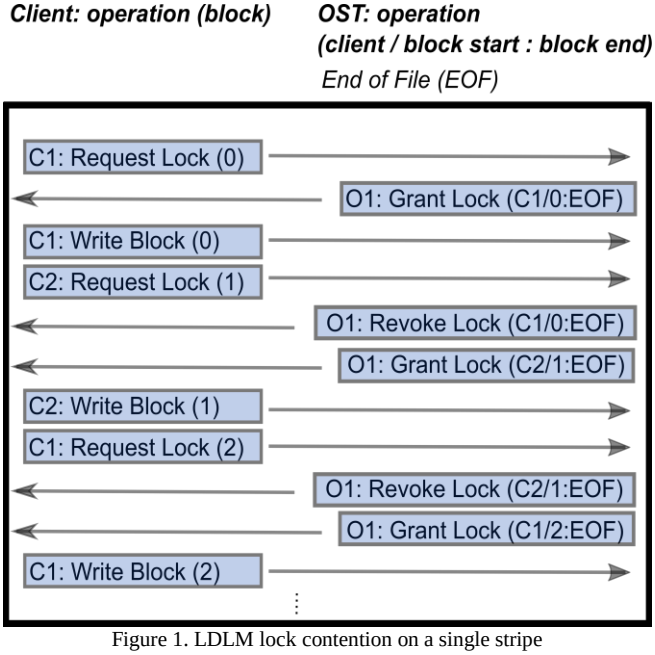

Figure 1. LDLM lock contention on a single stripe

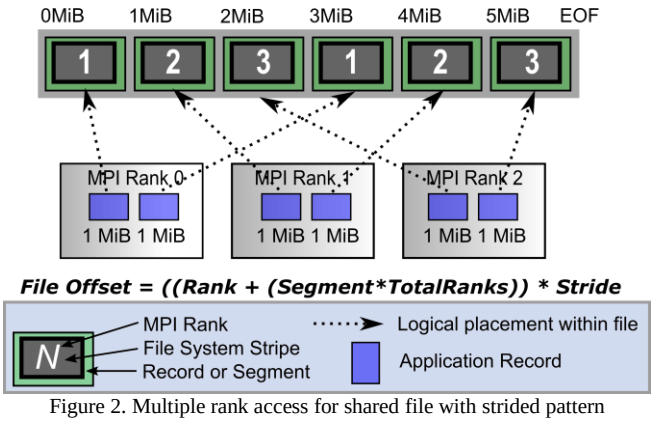

Figure 2. Multiple rank access for shared file with strided pattern

The result of this is extremely poor performance, as the clients essentially take turns waiting for one another. The obvious solution is to disable this behavior, but this does not improve performance [5].

There are only partial solutions currently available for this problem. The first is to make sure that the number of stripes equals the number of clients and align writes such that each client writes only to a single stripe. This works well, and the MPI-IO I/O library allows coercing arbitrary access patterns to this form via collective buffering and aggregation [6]. However, this has limitations and requires the application to use the collective MPI-IO interface. Notably, it means only one client is writing to each OST.

For various reasons, one Lustre client can only write to a single OST at 3-5 GB/s. This is lower than the bandwidth of some OSTs in systems today, and far lower than the per OST bandwidth projected for future systems. This means that directly addressing the LDLM locking problem is necessary to achieve expected single OST performance for shared files. Unfortunately, this is only possible by using complex mechanisms such as Lustre Lockahead [4] or relaxing the POSIX consistency of Lustre by using Lustre group locks, which can risk data corruption. These choices have costs in complexity, time, and effort. It would be better to avoid such complexity and still access the full bandwidth of each OST. This discussion raises a simple question, obvious in hindsight: why do we allow only one stripe per OST?

II. Lustre Overstriping

To extract the maximum performance from an OST on current hardware and software, benchmarks and applications use multiple stripes per OST; the stripes are simply in separate files. The central insight of overstriping is that this solution can be applied within a single file. There is nothing intrinsically necessary about the one-to-one relationship between stripes and OSTs within a file. Relaxing that requirement to allow more than one stripe on each OST is straightforward, and as the benchmark section shows, for many workloads, has benefits similar to using multiple files.

Fig. 3 and Fig. 4 contrasting standard Lustre striping and and overstriping. Normal Lustre file striping places one stripe per OST. A file using four OSTs with a single stripe on each OST creates a layout as shown in Fig. 3. In contrast, with overstriping, the same number of OSTs can be used for more than four stripes. Placing two stripes on each of four OSTs creates a layout with eight stripes as shown in Fig. 4.

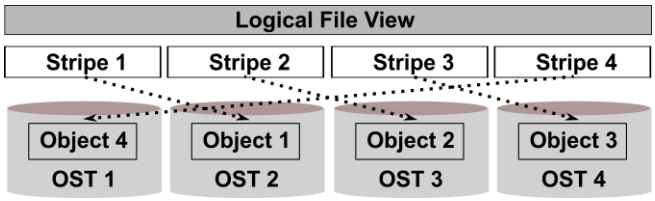

Figure 3. Lustre default stripe layout; 1 stripe per OST

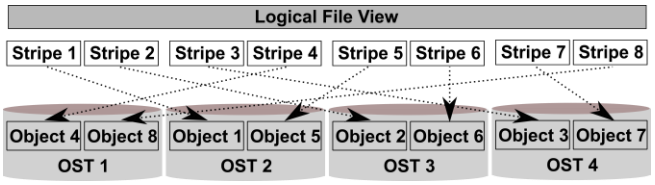

Figure 4. Lustre overstripe layout; 2 stripes per OST

A. *Lustre Overstriping and Compatability*

Due to the flexibility of the Lustre striping implementation, the implementation of overstriping is straightforward. Lustre has a layered design, with responsibilities clearly delineated between different layers, and significant independence between neighboring components. This design means that Lustre does not care which OST a stripe is present on, or even if more than one stripe is present on an OST. Historically, more than one stripe per OST was considered undesirable, so creating such layouts was prevented by sanity checks. However, those checks did not reflect an underlying limitation in the software or architecture. Thus, the implementation of overstriping consisted largely of removing these checks, and creating a userspace interface to express layouts with more than one stripe per OST. This leaves us without much technical detail to discuss directly about the implementation, so instead, we will highlight a few of the complications that emerged.

Overstriping makes it trivial to explore file striping settings that were previously the preserve of only a few extremely large systems. With overstriping it is possible to put 2,000 stripes on a single OST. This exposed several bugs which had not previously been identified. Lustre has a long-standing limit of 2000 stripes per layout component but no deployed system has ever used more than around 1,000 OSTs. With overstriping, it becomes trivial to reach higher stripe counts, and it was discovered values near 2,000 stripes cause a crash due to an incorrect check on the maximum allowed layout size [7].

Previously it was possible to create layouts which exceeded the maximum layout size by, for example, creating a progressive file layout (PFL) with multiple components with large numbers of stripes. If the total number of stripes in a PFL file exceeds approximately 2,700, it is possible to crash Lustre systems. This was not practical to test, nor did it represent a useful configuration, prior to overstriping. The issue was resolved as a prerequisite for overstriping, so versions of Lustre with overstriping will now reject these “too large” layouts, giving an error rather than crashing [8].

It was also noted that Lustre’s handling of extremely large extended attributes (file layout is stored as an extended attribute (xattr)) was inconsistent, with ldiskfs allowing xattrs beyond the maximum supported by Linux, and ZFS limiting xattrs to less than this size. Resolving this required tweaks to the client/server negotiation in order to correctly provide the true maximum size and respect it in the size checks described in the previous paragraph [9].

There are several other examples of unusual issues discovered when pushing the limits of file striping, but these are representative: Issues discovered were significant, but required only straightforward fixes and enhancements to existing code. The main lesson to draw is that when adding new functionality to existing systems, many problems stem not from errors in the new functionality itself, but from latent issues in existing code which are exposed by new usage.

1) Users Application Compatibility

Since overstriping uses Lustre file striping, it is invisible to user applications that do not directly interact with the layout via *lfs getstripe* and *setstripe* subcommands. An application that doesn't interact with Lustre layout information directly will see no change except, hopefully, improved performance. If an application does use *lfs getstripe*, the output format is unchanged except for a change to indicate when overstriping is used. Unless application logic assumes and verifies one stripe per OST the application should not require modifications. For the *lfs setstripe* subcommand, overstriping is accessible via slightly modified versions of the existing arguments. Example usage of current syntax is provided in Section 4.

2) Version Compatibility

Despite the simplicity of the implementation, clients which have not been updated to support overstriped files cannot use them. The sanity checks mentioned in the implementation section will cause older clients to crash when exposed to an overstriped file. The server uses the standard approach for new file layout features (such as PFL [10] and FLR [11]) and does not allow clients to open overstriped files unless they support the feature. This means that both client and server must support the new feature to use it, which is typical. The feature is scheduled for release in Lustre 2.13 from WhamCloud. Availability in Cray CLE and ClusterStor software is planned.

III. I/O Performance

A. Test Environment

Tests of Lustre overstriping were performed using a Cray ClusterStor file system composed of two ClusterStor L300N SSUs and one L300F SSU using the NEO 3.1 software stack. However, the Lustre build included the addition of Lustre overstriping support. Lustre overstriping is not currently available in released ClusterStor software. The L300F SSU was configured in a non-standard, non-redundant manner as a striped, RAID-0 device using ldiskfs. This configuration allows a single OST access to the full disk bandwidth by eliminating parity overhead. This was done to allow evaluation of overstriping and the limitations described in Section 1. The results described in the following sections using the L300F are not representative of a production L300F configuration and are referred to as a "flash OST". The L300N SSUs were configured normally with a 41 disk GridRAID OST using ldiskfs.

A set of 48 heterogenous Lustre clients in an Infiniband cluster were used to perform I/O. The Lustre clients were dual socket Intel Ivy Bridge compute nodes with 64 GB of memory. The same Lustre build was used on both servers and clients. The clients used a CentOS 7.5 operating system.

B. Local Performance Testing

Obdfilter-survey is a synthetic benchmark used to directly measure performance of OSTs. Obdfilter-survey was used to generate load directly on the OSSes to demonstrate the local file system limitations of a single OST. Generating file system requests directly on the OSSes eliminates client locking and network overhead. Obdfilter-survey creates several threads on the OSS to generate I/O across a configurable number of local file system objects which are accessed in configurable size records.

A set of tests were performed on an L300N OST and a flash OST varying the number of threads and objects. A 4 MiB record size was used for all tests corresponding to the typical size of RPCs configured on ClusterStor systems. Each test was performed three times and the median value reported.

The write and read performance of an L300N GridRAID OST is shown in Fig 4 and Fig. 5 respectively. As the number of threads increases both write and read performance plateau with no clear delineation between object counts. Since two or more objects do not significantly improve performance the local file system limitation is above the peak performance of a single L300N OST. Concurrent access to a single object from progressively higher thread counts marginally limits the single object performance. For these results, write and read operation performance tends to improve with higher thread counts so that more I/O requests in flight.

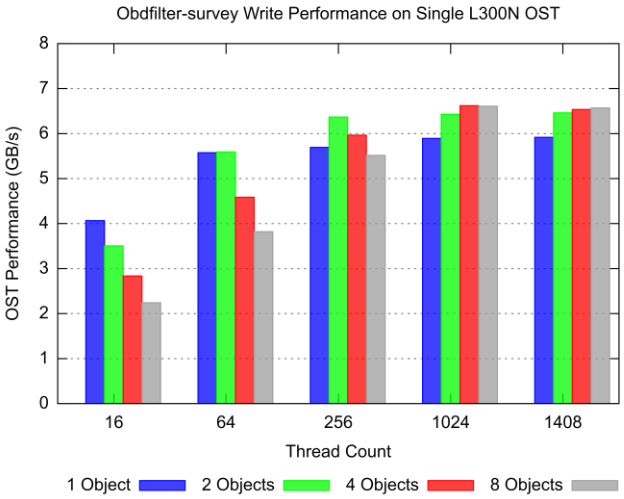

Figure 5. L300N obdfilter-survey write performance

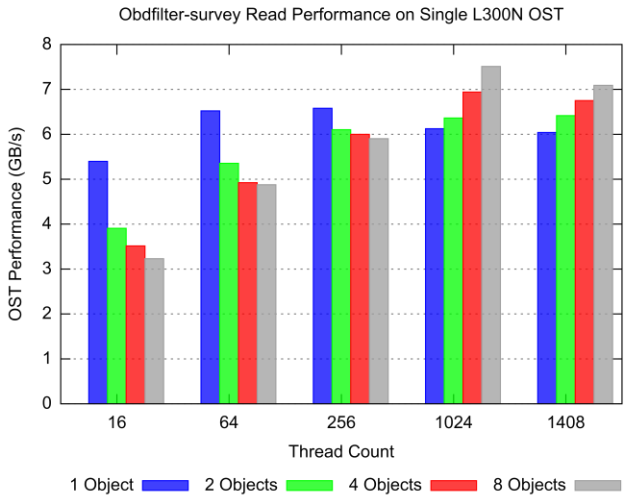

Figure 6. L300N obdfilter-survey read performance

The write and read performance of a flash OST is given in Fig. 6 and Fig. 7 respectively. For the flash OST the performance curve is relatively flat regardless of thread count. The reduced latency of servicing I/O requests using flash devices requires fewer threads to achieve expected performance. The variation in performance across the number of threads is very low compared to the L300N results. Important to our discussion of local file system limitations, the performance of all single object tests is significantly lower than higher object counts. Single object performance is 6.8 – 7.1 GB/s for writes and 7.1 – 7.5 GB/s for reads. All other object counts show expected peak performance of the OST: 11.2 – 11.7 GB/s for write and read. For this flash OST, hardware configuration, and software stack the single object performance limit is 6.8 – 7.5 GB/s. With a single object Lustre is only able to achieve approximately 60% of the expected OST performance.

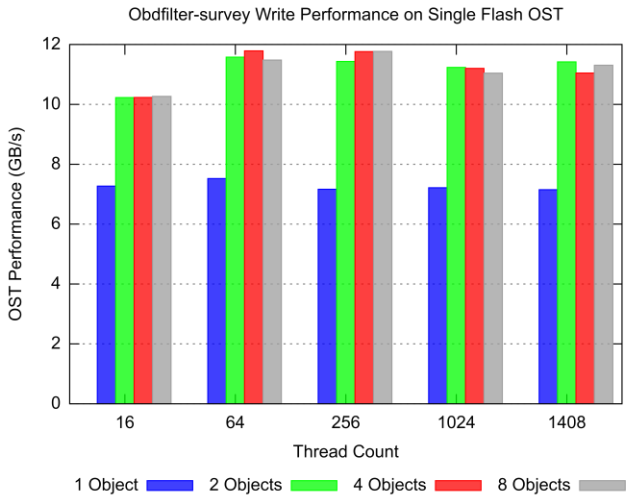

Figure 7. Flash OST obdfilter-survey write performance

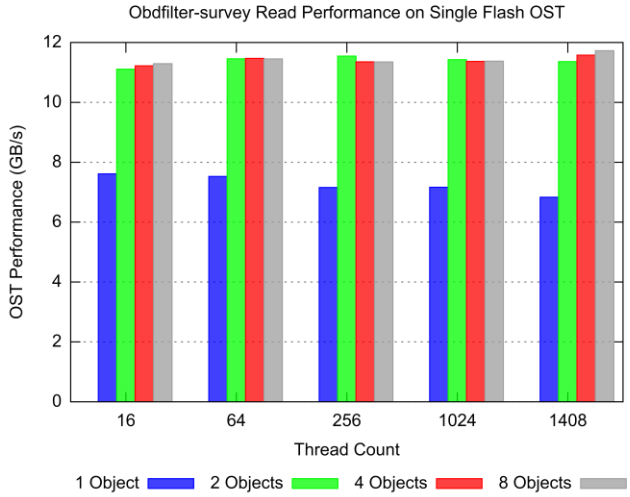

Figure 8. Flash OST obdfilter-survey read performance

Obdfilter-survey was used to evaluate the current local file system limitation described in Section 1 on a disk and flash based OST. Test results demonstrated that the local file system performance limit in the test environment is around 7 GB/s for a single object. The potential performance of the flash OST exceeded 7 GB/s and, as expected, multiple objects were required to achieve peak bandwidth. The L300N OST was able to achieve near peak performance with a single object indicating there was not a local file system limitation, although additional objects saw marginally improved performance at high thread counts. In the next section, we evaluate the ability of overstriping to address the second limitation, LDLM contention.

C. Lustre Client Performance Testing

IOR [12] is a synthetic benchmark used to generate specific I/O access patterns. Due to time and system constraints IOR is used as a proxy for application workloads. Each IOR test writes a fixed amount of data, 64GB per node, which is equal to the amount of node memory. Between each write and read test the client cache is flushed to eliminate any client cache effects. For all tests, if more than one MPI process per node is used the MPI processes were packed on a node e.g. rank 0 – 15 are placed on node 0. The access pattern is a shared, strided pattern where each process accesses multiple segments at a fixed stride throughout the file as depicted in Fig. 2. The Lustre stripe size was set equal to the segment size, known as the block size in IOR, and to the record size, known as the transfer size in IOR. Many applications access data in a similar pattern either directly or through higher-level I/O libraries. For shared file workloads, parameters such as node count, OST count, record size and Lustre stripe size all interplay to impact performance. These tests focus on record sizes equal to the Lustre stripe size with all nodes accessing a single OST to demonstrate the effect of LDLM contention. The term “overstripe count” generally refers to the total

number of stripes of a file when using more than one stripe per OST. In the following discussions, since a single OST is used, the overstripe count and number of stripes per OST is equal.

1) Disk Shared File Performance

Initial tests used an L300N disk-based OST focused on the performance of the shared, strided access pattern while scaling the number of Lustre stripes on a single OST. Local file system testing of an L300N OST didn't require multiple objects to achieve expected performance and since LDLM contention doesn't affect read performance, read data is not presented. Fig. 9 and Fig. 10 show the write performance of varying compute node counts for one process per CPU core (16) using 1 MiB and 16 MiB record sizes respectively. Generally, the performance for this specific workload, even with overstriping, fails to approach optimal OST performance. However, there are significant improvements in write performance using overstriping. All node counts show at least a doubling of performance comparing the write performance of a single stripe and optimal Lustre stripe per OST count. This improvement is due to more stripes providing additional locking domains since each stripe can grant a single lock. The incremental improvement in write performance from left to right in the figures illustrate the reduced LDLM contention, and increased concurrency, allowed by multiple stripes. The cause of the dramatically improved performance in the 4 node case at 16 or 32 stripes was not investigated further.

Comparing 1 MiB and 16 MiB record sizes shows improved write performance with the larger 16 MiB records and corresponding 16 MiB Lustre stripe size. Although 16 MiB records improve performance the rate is still below peak file per process rates which indicates LDLM contention is still reducing performance for this workload. With a larger Lustre stripe size more data is written per granted lock and fewer locks are required to cover the same amount of data. Additional stripes per OST, beyond 32, may provide further performance improvements but were not investigated.

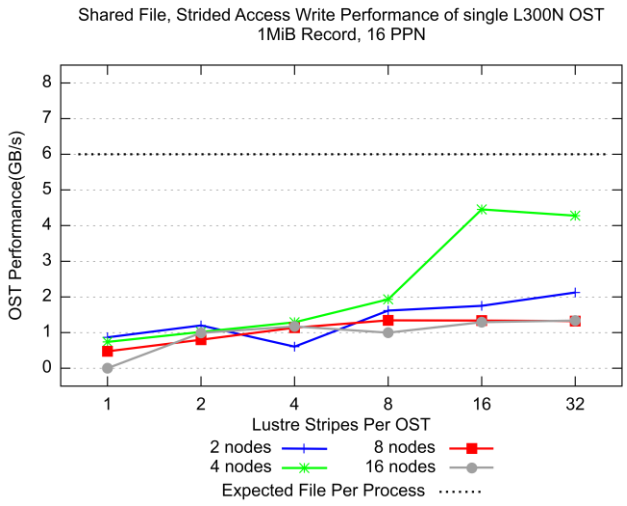

Figure 9. L300N OST shared file 1 MiB record write performance

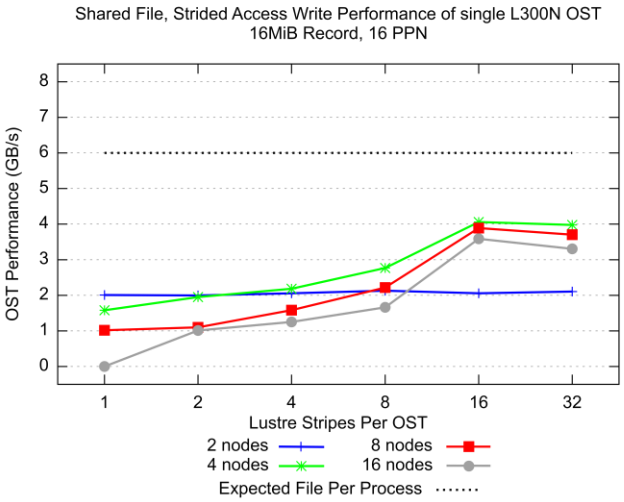

Figure 10. L300N OST shared file 16 MiB record write performance

To illustrate the write performance improvement using Lustre overstriping for an L300N OST Fig. 11 and Fig. 12 show the comparative performance of the current single stripe per OST and the maximum measured performance using overstriping. The "Overstriping" value in the figures represents the highest observed performance for any stripes per OST count tested: 2, 4, 8,16, or 32. The benefits of overstriping is more compelling for both 1 MiB and 16 MiB record sizes as the number of compute nodes accessing the shared file increases. This observation matches expectations that additional Lustre clients introduce more lock contention and by increasing the number of stripes per OST the concurrency increases. In the overstriping tests for 1 MiB and 16 MiB and single stripe tests for 16 MiB, using additional processes per node shows reduced performance when more than 4 nodes are used. This is likely due to increased lock contention due to process placement i.e. each node with 4 processes write 4 records which corresponds to 4 stripes. A single process would allow each client to only write to a single stripe without contention from other nodes. Depending on the node and process count, record and stripe size, and stripe count the alignment of accesses to Lustre stripes can eliminate lock contention by only having a single node access a stripe as described in section 1.

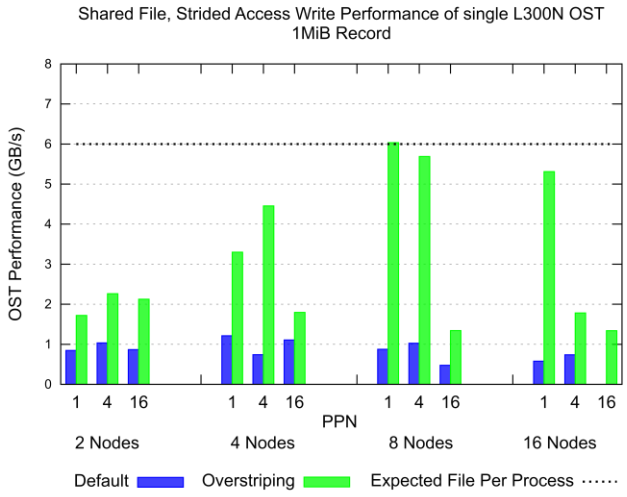

Figure 11. L300N OST Shared file write performance using 1 MiB record

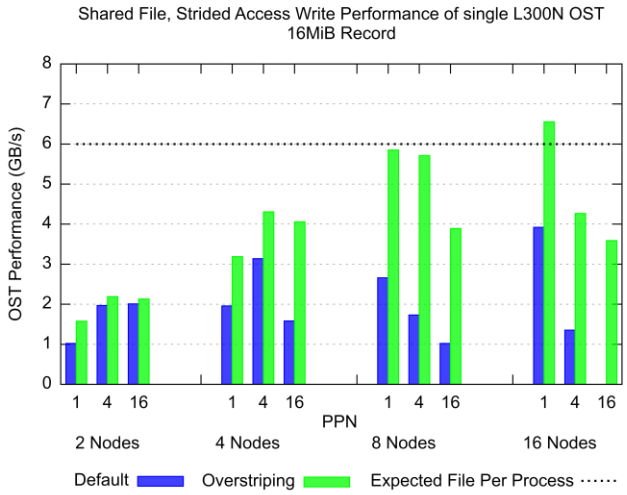

Figure 12. L300N OST shared file write performance using 16 MiB record

Shared file performance on an L300N OST using overstriping showed consistent, significant improvements in write performance. The benefit of additional stripes to reduce LDLM lock contention was shown in the incremental improvement of write performance as the number of stripes per OST increased. With adequate compute nodes to drive performance most tests showed a 50% to 100% increase in performance with some tests showing an improvement of up to 6 times single stripe per OST performance. Most tests showed 16 or 32 stripes per OST as highest performing although using more than 32 stripes per OST was not tested and may yield further improvements for some workloads.

2) Flash Shared File Performance

Tests on the flash OST began with the same workload as the L300N testing, a shared, strided access pattern with one process per CPU core. Due to the higher performance of the flash OST, higher node counts were used in these tests. As seen in local file system tests, multiple stripes are needed to achieve expected performance so read tests were also performed and presented. Fig. 13 shows the incremental write performance increase from 3-6 GB/s, depending on record size, for a single stripe per OST to 11.1 – 11.4 GB/s for multiple stripes per OST which is nearly equal to optimal file-per-process performance. The incremental write performance improvements point to LDLM contention as also seen in testing of the L300N. The performance curve also indicates there isn't a performance penalty, at this scale, for using stripe counts larger than needed to achieve expected performance on a flash OST i.e. performance plateaus once adequate stripes are used. The performance of larger record sizes, with an equal Lustre stripe size, show reduced LDLM contention as seen in the L300N tests. The read tests in Fig. 14 demonstrate the read performance limitation is due to the local file system as all test results show a large increase and subsequent plateau at 11.7 GB/s once more than one stripe is used. Record sizes larger than

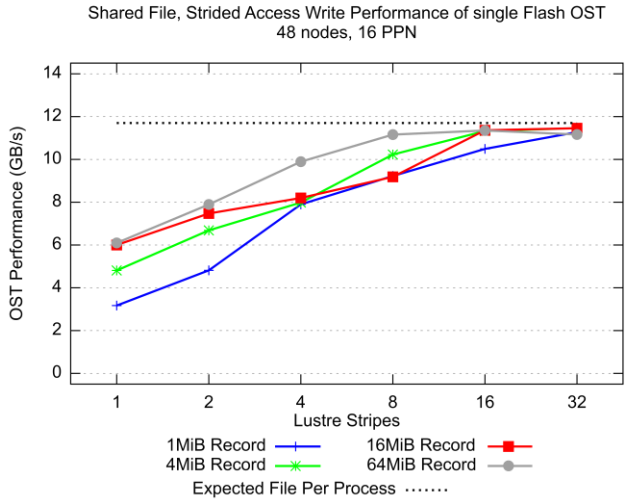

Figure 13. Flash OST shared file write performance, 48 compute nodes

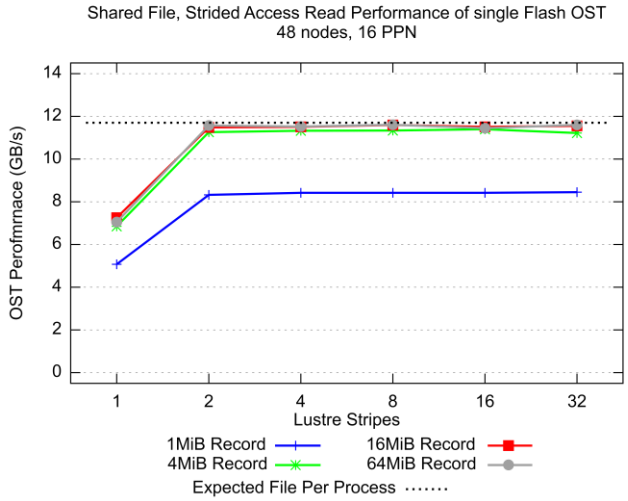

Figure 14. Flash OST shared file read performance, 48 compute nodes

1 MiB match the observed local file system rate in Fig.8.

Second, the effect of varying compute nodes was investigated using the same access pattern and processes per node. The "Overstriping" rate in Fig. 15 and Fig. 16 is the same as described for Fig. 11 and Fig. 12. Fig. 15 and Fig. 16 show roughly similar single stripe performance across all node counts although the 48-node test with a 1 MiB record size is 40% lower than 32 nodes indicating LDLM contention is still an issue, even on the flash OST. The results indicate the effect of LDLM contention on performance for flash-based OSTs is less than disk based OSTs. The reduced latency of servicing requests on flash likely reduces the time a lock is held to service an I/O request. Further microbenchmarks were not performed to confirm this. Additionally, in this testing shared file performance using 32 and 48 nodes achieves file per process write rates which was not possible at the same stripes per OST count in L300N OST testing.

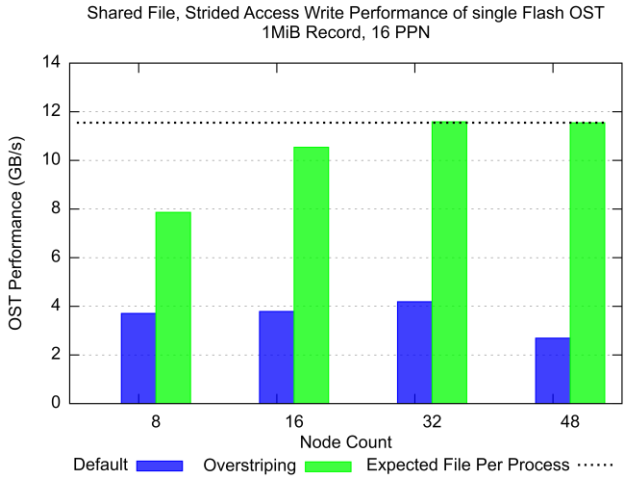

Figure 15. Flash OST shared file write performance, 1 MiB record

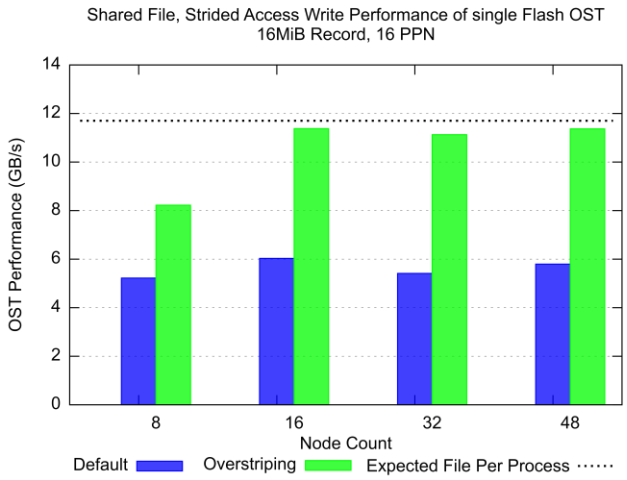

Figure 16. Flash OST shared file write performance, 16 MiB record

Testing using a flash based OST for a shared, strided workload showed the expected incremental write performance improvement as the number of stripes per OST was increased. Comparing the default, single stripe per OST performance to the best Lustre overstriping performance shows an improvement for all tests and up to an improvement of 4 times for specific workloads. Read performance is also improved by using more than one stripe per OST in order to overcome the local file system limitation.

The IOR tests to this point focused on a node with one process per compute core which is a common workload. However, many of the workloads that use this access pattern use collective MPI-IO. The optimizations provided by collective MPI-IO calls, specifically collective buffering, is another important use case for Lustre overstriping which we investigate in the next section.

3) Collective MPI-IO

One of the main application use cases for shared file access is using higher level I/O libraries such as HDF5 that make use of collective MPI-IO. Collective MPI-IO implementations, such as Cray MPICH, use optimizations such as collective buffering and advanced placement of data on aggregators to optimize shared file performance [4,6]. Although collective MPI-IO tests were not performed due to system software and time constraints the following results provide a similar workload from the aggregator point of view to evaluate how Lustre overstriping may benefit collective MPI-IO write performance.

Although a full discussion of collective MPI-IO is beyond the scope of this paper, collective MPI-IO uses a set of MPI processes in collective buffering known as aggregators to perform calls to the underlying file system on behalf of all the ranks involved in the collective operation. We've discussed the challenges of shared file performance and LDLM contention. One optimization used by collective buffering to address the contention is to use a small number of ranks as aggregators. The collective buffering aggregator ranks are performing POSIX shared file accesses which is the same workload used in the benchmarks. The default behavior for Cray MPICH is to use a number of aggregators equal to the number of OSTs placing one aggregator rank per node up to the number of aggregators needed.

To emulate the workload of MPI-IO collective buffering aggregators the same IOR tests were performed using only a single process per node. Although the record sizes tested are 1 MiB aligned, an application doesn't need to issue requests of that size due to collective buffering. The requests will be correctly aligned to Lustre stripe sizes by collective buffering. Fig. 17 and Fig. 18 show the single stripe, per OST performance in the left-hand grouping and the maximum measured performance for multiple stripes per OST in the right-hand grouping.

In Fig. 17, with fewer processes per node, the benefit of overstriping is much larger than previous single OST L300N tests with one process per core. For 4 MiB, 16 MiB,

and 64 MiB record sizes the shared, strided write performance achieves peak file per process performance of 6 GB/s. Additionally, the benefits of overstriping for small record sizes is dramatic, a 5x improvement for 1 MiB record size using 8 or 16 nodes. The tests for the flash OST are similar to results previously seen; the effect of higher process per node counts has less effect on flash-based OSTs than disk-based OSTs. As with previous flash OST results, all record sizes are able to achieve expected file per process OST performance using some stripe count and node count combination.

Emulating the access patterns and process count of MPI-IO aggregator ranks the test results indicate that overstriping is another way to improve shared file write performance for collective MPI-IO using collective buffering. The results show that with proper data placement, where each aggregator accesses a single stripe, shared file performance for collective MPI-IO can achieve near file per process rates even as OST performance continues to increase.

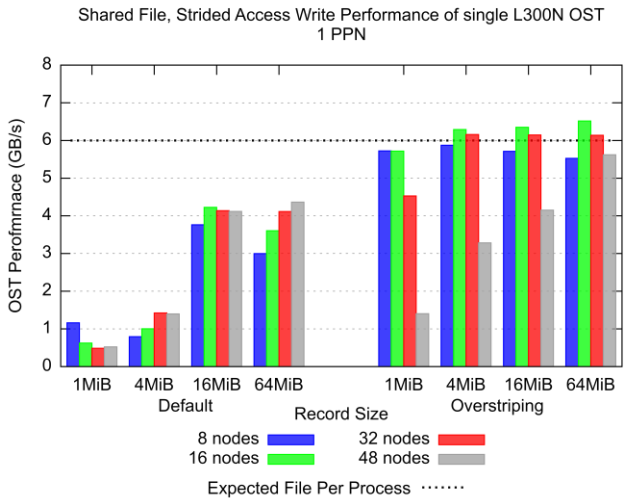

Figure 17. L300N OST 1 PPN shared file write performance

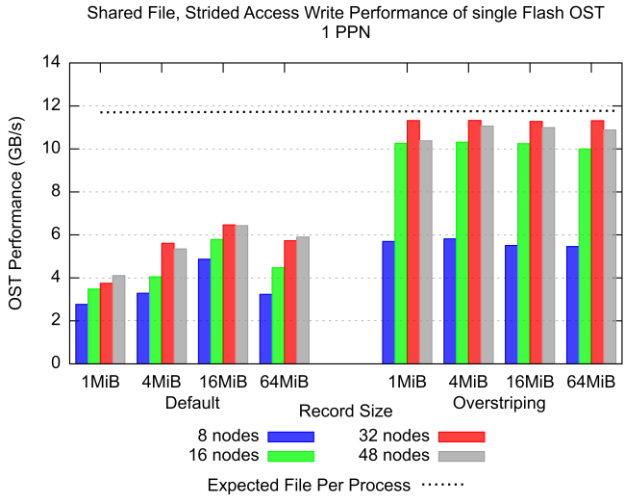

Figure 18. Flash OST 1 PPN shared file write performance

IV. Using Lustre Overstriping in Applications

As described in Section 2, Lustre striping is controlled using the same utility, *lfs*, whether striping is a single stripe per OST or overstriped. Use of the Lustre Library API (llapi) [13] is beyond the scope of this paper but can also be used to programmatically set Lustre striping.

A. Setting Lustre Overstriping

The *lfs setstripe* subcommand is used to specify striping information. There are two methods to specify the stripe layout to use overstriping detailed in Table 1. Note that these options are as implemented in the pre-release version used for testing, they may differ slightly in the released version (details will be available in lfs setstripe help and man pages). Typically, a balanced count of stripes on all OSTs is desired and for that use case the *-C / --overstripe-count* option is sufficient. The command to create a file with overstriping depicted in Fig. 3 is provided in Listing 1. Assuming a file system with 4 OSTs, as depicted in Fig. 3, 8 overstripes will have two stripes placed across the 4 OSTs.

Lustre, by default, will place stripes on all available OSTs while following placement heuristics based on OST capacity utilization or placement within a Lustre pool if that option is provided to the lfs *setstripe* command. Stripe placement policies are still present with overstriping so allowing Lustre to select where stripes are placed can lead to unbalanced placement if OST capacity utilization is unbalanced. However, it's recommended to allow Lustre to handle stripe placement within a pool.

An alternative to Lustre selecting the file stripe layout is manually specifying all OSTs the stripes will be placed on. This may be necessary if a pool with the desired OSTs is not available or for experimentation with unbalanced stripe placement. Continuing the previous example of 8 stripes across 4 OSTs Listing 2 provides an alternate file layout by manually specifying each stripe using the existing *-o / --ost* option which will now support duplicated OST indices.

TABLE I. LFS SETSTRIPE OVERSTRIPING OPTIONS

Overstripe Layout	lfs setstripe Argument
Automatic	*-C / --overstripe-count <count>*
Manual	*-o / -- ost <ost1,ost2, … ostN>*

```
$lfs setstripe -C 8 testfile
```

Listing 1. lfs command for 8 automatically placed overstripe layout

```
$lfs setstripe -o 0,1,0,2,1,2,3,3 \
 testfile
```

Listing 2. lfs command for 8 manually placed overstripe layout

As referenced in section 2 *lfs getstripe* reports the use of overstriping for a file's layout. Specifically, the *lmm_pattern* attribute will indicate overstriping is in use. Continuing the layout example, the corresponding *lfs getstripe* layout information is provided in Listing 3 and Listing 4. In Listing 3 the OST index cycles between the list of 4 OSTs for a total of 8 stripes. In Listing 4 the order of stripes matches the specific OST list order provided to lfs in Listing 2.

```
$ lfs getstripe testfile
testfile
lmm_stripe_count:  8
lmm_stripe_size:   1048576
lmm_pattern:       raid0_overstriping
lmm_layout_gen:    0
lmm_stripe_offset: 8
lmm_pool:          disk
  obdidx   objid      objid          group
  2        39748073   0x25e81e9      0
  0        39840878   0x25fec6e      0
  3        39789909   0x25f2555      0
  1        39826705   0x25fb511      0
  2        39748074   0x25e81ea      0
  0        39840879   0x25fec6f      0
  3        39789910   0x25f2556      0
  1        39826706   0x25fb512      0
```

Listing 3. lfs stripe listing for automatic overstriping layout

```
$ lfs getstripe testfile
testfile
lmm_stripe_count:  8
lmm_stripe_size:   1048576
lmm_pattern:       raid0_overstriping
lmm_layout_gen:    0
lmm_stripe_offset: 0
  obdidx   objid      objid          group
  0        39840884   0x25fec74      0
  1        39826711   0x25fb517      0
  0        39840885   0x25fec75      0
  2        39748078   0x25e81ee      0
  1        39826712   0x25fb518      0
  2        39748079   0x25e81ef      0
  3        39789915   0x25f255b      0
  3        39789916   0x25f255c      0
```

Listing 4. lfs stripe listing for manually placed overstripe layout

B. Overstripe Count Tuning

Test results presented in Section 3 indicate that for many workloads, using overstriping to create 16 or 32 stripes per OST provides the best performance for the configuration and workloads tested. However, the use of overstriping does require additional stripes to be created on OSTs and tracked in file metadata. The additional file metadata does consume inodes in the file system and capacity on the MDT. It's recommended overstriping only be used in cases where shared files with a single stripe per OST are a bottleneck, similar to how widely striped files are managed today.

The number of stripes per OST needed to achieve optimal write or read performance will depend on the specifics of the application workload, job size, API, and file system characteristics. In cases where the number of nodes accessing an OST is similar in size to the test environment, results presented in section 3 show using a stripe count per OST equal to or greater than the number of nodes accessing the OST is a good rule of thumb.

In the case of collective MPI-IO using collective buffering with Cray MPICH the number of nodes per OST is specified by the MPI-IO hint *cray_cb_nodes_multiplier*. Although it was not directly, it's expected the multiplier should be equal to the number of stripes per OST. Support for Lustre overstriping in Cray MPICH is planned but is not available in released versions as of this writing. However, Lustre overstriping can be used by setting the Lustre striping using *lfs setstripe*. Using MPI-IO hints to set overstriping on files will require support in Cray MPICH.

The second major factor influencing the number of stripes needed per OST is the record size and Lustre stripe size. As an example, in test results presented for a flash OST, a 16 MiB record size requires only 8 stripes per OST to achieve optimal performance while a 1 MiB record size requires 32 stripes per OST. For disk-based OSTs the number of stripes are higher, presented results indicate 16 stripes per OST are needed for a 16 MiB record and 32 or more stripes are needed for a 1 MiB record.

Of the two limitations motivating the Lustre overstriping feature the local file system limitation will require a minimum number of overstripes to overcome. Testing indicates for current software and hardware an overstripe count of two stripes per OST is adequate given a single stripe performance limit of 7 GB/s and a peak OST performance of 11.7 GB/s. The specific values for the per stripe limit will vary depending on hardware and software. Generally, it's expected that the overstripe count will be driven by reducing LDLM contention.

V. CONCLUSION

Current Lustre striping behavior places a single stripe per OST. The increasing performance of single OSTs due to software and hardware changes has made two issues limiting shared file performance more acute. The first issue, local file system performance limitations, places single stripe performance and subsequently shared file performance below expected OST performance. Experimental results show that multiple stripes on an OST, the contribution of the new Lustre overstriping feature, overcome this limitation so an OST can achieve expected performance for a single file.

The second issue, LDLM contention, has been addressed with other file system features and optimizations in APIs but continues to cause challenges in achieving expected shared file performance across a variety of workloads.

Presented results show the effect of LDLM contention on write performance can be reduced by creating additional stripes on an OST. In all cases, both disk and flash OSTs, shared file write performance was improved by using overstripes. Also, both disk and flash OSTs had combinations of record size, node count, and process per node counts that could achieve file per process rates for a shared, strided workload. Finally, the role of overstriping for collective MPI-IO shared file performance was assessed. Synthetic benchmarks showed aggregators, with a single process per node, can achieve file per process rates for disk and flash based OSTs for specific record sizes and node counts. Future work in evaluating Lustre Overstriping includes (1) testing applications using collective MPI-IO with Cray MPICH (2) testing at larger scale.

ACKNOWLEDGMENT

We would like to acknowledge the assistance of Alexey Lyashkov, Chris Horn, and Justin Miller for Lustre build assistance for the test environment, Bill Loewe, Chris Walker, and John Fragalla for ClusterStor and test environment assistance, Nathan Rutman and Andreas Dilger for guidance on design, and John Bauer and Richard Walsh for manuscript review.

REFERENCES

[1] (2019) Lustre. [Online]. Available: http://lustre.org/

[2] M. Swan, "Sonexion GridRAID Characteristics" presented at CUG 2014, 2014.

[3] (2017) DDN SFA Updates. [Online]. Available: https://youtu.be/jgrIBHSOh7s.

[4] M. Moore, P. Farrell, and B. Cernohous, "Lustre Lockahead: Early Experience and Performance using Optimized Locking," presented at CUG 2017, 2017.

[5] (2015) Shared file performance improvements LDLM lock ahead. [Online]. Available: http://wiki.lustre.org/images/f/f9/Shared-File-Performance-in-Lustre_Farrell.pdf

[6] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in ROMIO," in Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–.

[7] (2019) Remove LASSERT(r0->lo_nr <= lov_targets_nr(dev)) in maintenance branches. [Online]. Available: https://jira.whamcloud.com/browse/LU-11796.

[8] (2019) lfs getstripe buffer overflows with very large stripe counts. [Online]. Available: https://jira.whamcloud.com/browse/LU-11691.

[9] (2019) ZFS ea size limited to 32K. [Online]. Available: https://jira.whamcloud.com/browse/LU-11868.

[10] (2018) Progressive File Layouts – Lustre Wiki. [Online]. Available: http://wiki.lustre.org/Progressive_File_Layouts

[11] (2018) File Level Replication High Level Dsign – Lustre Wiki . [Online]. Available: http://wiki.lustre.org/File_Level_Replication_High_Level_Design

[12] (2019) IOR. [Online]. Available: https://sourceforge.net/projects/ior-sio/

[13] (2019) Lustre Software Release 2.x Operations Manual. [Online]. Available: http://doc.lustre.org/lustre_manual.xhtml.