Tuning and Analyzing Sonexion Performance

Mark Swan
Performance Group
Cray Inc.
Saint Paul, Minnesota, USA
mswan@cray.com

Abstract—**This paper will present performance analysis techniques that Cray uses with Sonexion-based file systems. Topics will include Lustre client-side tuning parameters, Lustre server-side tuning parameters, the Lustre Monitoring Toolkit (LMT), Cray modifications to IOR, file fragmentation analysis, OST fragmentation analysis, and Sonexion-specific information.**

Keywords – Sonexion; tuning

I. INTRODUCTION

Lustre file systems, and the clients that interact with them, can be quite intimidating and hard to understand. Once a file system is working, though, we begin to want to understand how to get the most performance out of them. Lustre provides many modifiable parameters to allow us to tune client and server performance.

In this paper, we will look at the tools we use to exercise the file system, the tools we use when probing and modifying tunable parameters, the tools we use when looking at file system performance, how data gets from file system clients to the file system servers, where data exists on the file system, how data exists on the file system, and how to take advantage of the nuances of file system characteristics.

II. CRAY SONEXION OVERVIEW

The Cray Sonexion 1600 is composed of a single Metadata Management Unit (MMU) and one or more Scalable Storage Units (SSU).

The MMU consists of four servers and either a 2U24 or 5U84 drive enclosure. The four servers are two cluster management servers and two file system metadata servers (i.e. the MGS and MDS). The SSU consists of two Object Storage Servers (OSS), each with 32 GiB of memory, and a 5U84 drive enclosure.

In the MDRAID configuration, of the 84 drives in the 5U84 enclosure, there are two SSDs, 80 spinning disks arranged into eight Object Storage Targets (OSTs) which are RAID 6 8+2 arrays, and two spinning disks as global hot spares. Each OSS has primary responsibility for four OSTs.

As a general guideline, the performance of a single SSU is said to be 5 GB/s sustained and 6 GB/s peak. The SSU is the building block of the file system.

III. THE TOOLS WE USE

Before immersing ourselves in the details presented in this paper, let's first understand the tools we will be using. Information about these tools is readily available online.

A. *LNET Selftest (lst)*

LST is supplied with the Lustre distribution [1] and can be used to verify the Lustre NETwork (LNET) bandwidth of our system. It can test all segments of the various networks between the file system clients (i.e. compute nodes) and the file system servers. In general, we want to verify that there is more LNET bandwidth than the file system servers can consume or produce so that the network is not a bottleneck.

B. *obdfilter-survey*

The obdfilter-survey tool is supplied with the Lustre distribution [1] and can be used to verify the I/O capabilities of the OSTs without having to worry about any higher level protocols. The tool executes directly on the OSS and communicates with the OSTs using the proper Lustre protocols.

C. *Lustre Control (lctl)*

The Lustre Control (lctl) command is supplied with the Lustre distribution [1] and can be used to show or modify various tuning parameters of Lustre clients and servers.

D. *pdsh*

The parallel shell utility is supplied with many Linux distributions and is used to execute commands on one or more hosts in a cluster.

E. *IOR*

The IOR benchmark [2][3] is available in the public domain and is used to execute I/O operations on many file system clients simultaneously and in a coordinated fashion. Cray has made modifications to this benchmark tool to gather more information [4].

F. *Filefrag*

The filefrag utility is part of the Cray Linux Environment (CLE) distribution and is used to show information about how a file is organized on disk.

G. Lustre Monitoring Toolkit (LMT)

LMT is supplied on the file system servers of Cray Sonexion as well as the Lustre File System by Cray (CLFS). This toolkit gathers a variety of server-side performance information and can store that information into a MySQL database.

IV. HOW DATA MOVES

1) "max_rpcs_in_flight" and *"max_dirty_mb"*

While there is a max_rpcs_in_flight associated with the metadata client (MDC) and the object storage client (OSC), this section is referring to the OSC tunable value. These two client side tunable parameters are described in the Lustre documentation. As a guideline, the documentation says that *max_dirty_mb* should be four times the value of *max_rpcs_in_flight*. Even though these tunable parameters are associated with each OST in the file system, we typically set all the tunable parameters the same for all OSTs.

By default in the Lustre distribution, max_rpcs_in_flight is 8 and max_dirty_mb is 32. What this means is that, on a per OST basis, if a client has 8 outstanding requests for a particular OST, no more requests will be allowed to be started until one or more of the outstanding requests completes. As a practical example, if a client makes a write requests for 8 MiB to a particular OST, and we know that Lustre is going to chop that request into 1 MiB LNET RPCs, the client can have all 8 of those RPCs in flight. However, if that client makes a write request for 16 MiB to a particular OST, the client will be limited to only having 8 of the possible 16 RPCs in flight.

In order to increase client performance, Cray suggests increasing *max_rpcs_in_flight* to 64 and *max_dirty_mb* to 256. Fig. 1 and Fig. 2 show the effects of these tunable parameters on direct I/O. Similar effects can be seen with buffered I/O as well.

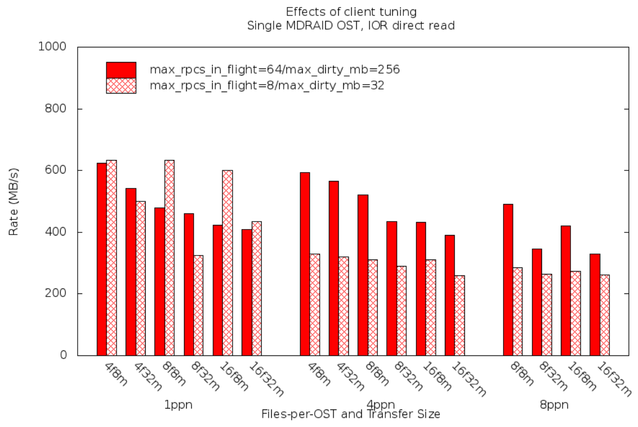

Fig. 1, Effects of client-side tuning on direct reads

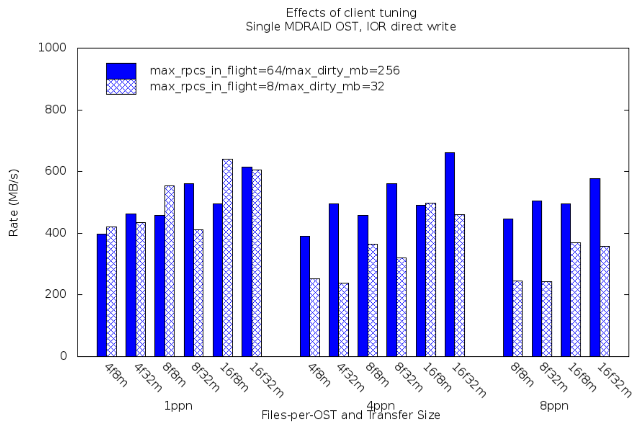

Fig. 2, Effects of client-side tuning on direct writes

2) Lustre 4 MiB Transfers

In releases prior to 2.4.0, Lustre used 1 MiB RPCs. Beginning with release 2.4.0, Lustre can be configured to use 4 MiB RPCs. Not only does this increase the efficiency of RPC transfers but also creates more contiguous data areas on the OSTs. This paper will discuss contiguous data in more detail in the section entitled "How Data Exists In The File System".

V. WHERE DATA EXISTS IN THE FILE SYSTEM

Disk drive manufacturers have documented specifications for the rate of data movement to and from the disk as the different zones of the disk from the "fast edge" to the "slow edge". OSTs are subdivided into what are called "multi-block allocation groups" and each group represents 32,768 blocks of space (where each block is 4,096 bytes). Multi-block allocation group zero (0) is at the beginning of the OST. When the OSS is started (or restarted), it begins looking for free space (in which to write data) at multi-block allocation group zero. As more data is written, the allocator simply continues to advance to higher numbered multi-block allocation groups until the end of the OST is reached at which point the allocator "wraps" and begins at the beginning of the OST again.

A. mb_last_group

Beginning with NEO release 1.2.3, we are able to view and change the position at which the allocator will begin looking for free space. This value is stored on each OSS in "/proc/fs/ldiskfs/md*/mb_last_group". We can simply "cat" that file to see the value or "echo" a new value into it. Fig. 3 shows the performance of the obdfilter-survey tool as the allocation position is moved from the fast edge to the slow edge of an OST.

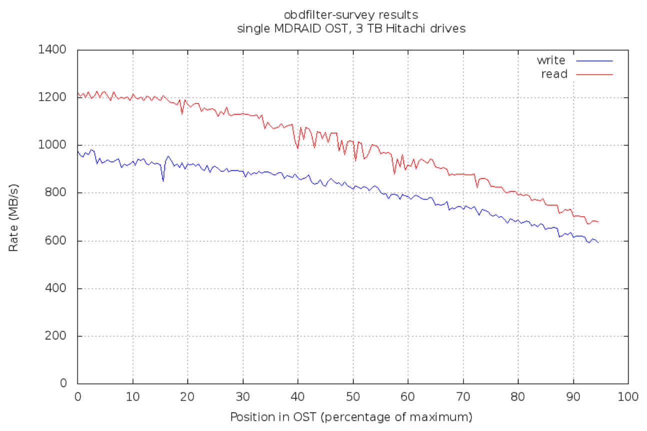

Fig. 3, OST speed zone differences

While the allocator begins looking for free space at the beginning of each OST when it is started (or restarted), OSTs quickly get “out of sync”. Different amounts of data are written to different OSTs and, very soon, the values of *mb_last_group* can appear to be random on a file system. Fig. 4 shows a snapshot of the different allocation positions on each OST of a file system.

B. *mb_groups*

Information about each of the multi-block allocation groups also exists on the Sonexion OSS. Alongside the “proc/fs/ldiskfs/md*/mb_last_group” special file is one named “mb_groups”. This file contains one line for each multi-block allocation group on that OST and each line consists of several pieces of information. One of the interesting pieces of information is a column indicating how many blocks are free in that multi-block allocation group. Fig. 5 shows a plot of this information for a single OST. Visualizing this information, as well as noticing where the *mb_last_group* value is, can indicate where (in the OST) data already exists and how dense that data is. Just as the *mb_last_group* value differs from OST to OST, so does the placement and density of information of each OST.

Even though the OST shown in this plot is more than 86% free, there are areas where the data is quite dense. The vertical blue line at 13,164 is the value of *mb_last_group* and represents where the allocator will begin looking for free space when writing data.

For a file system containing hundreds or thousands of OSTs, it becomes somewhat impractical to view and analyze one plot for each OST. My colleague Doug Petesch and I are continually attempting to find ways to visualize the space usage of an entire file system. Fig. 6 and Fig. 7 are examples of viewing the free space of an entire file system as a “heat map”. Lighter colors represent free space while darker colors represent used space.

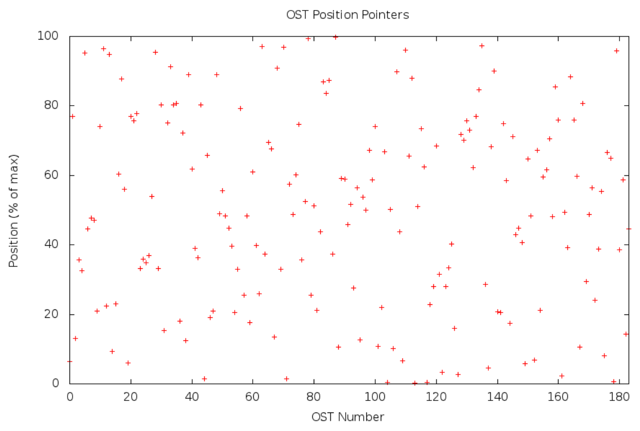

Fig. 4, Random OST allocation positions

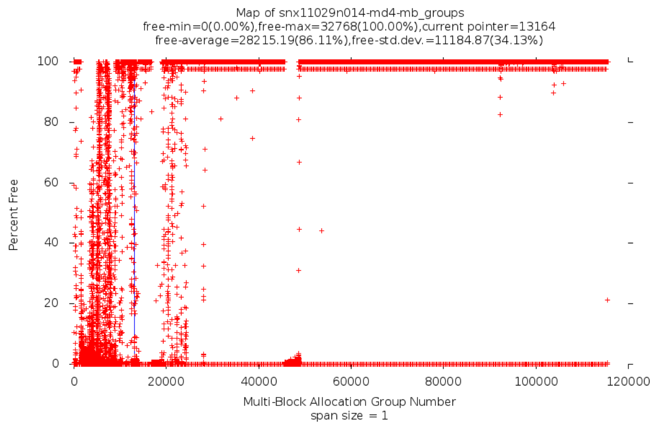

Fig. 5, Multi-block allocation groups

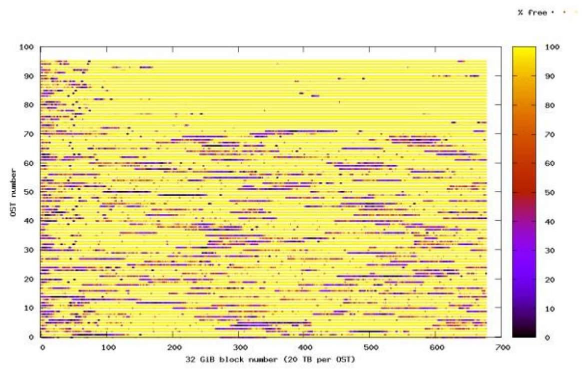

Fig. 6, First example of file system heat map

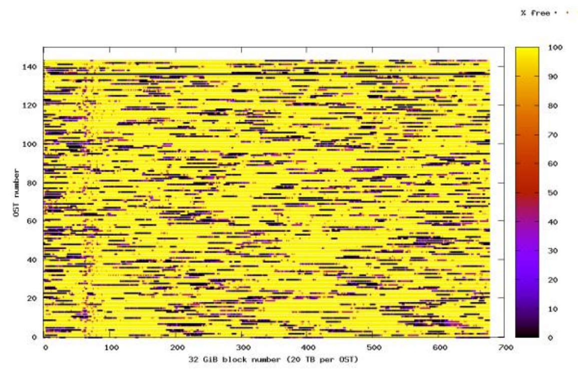

Fig. 7, Second example of file system heat map

C. read_cache_enable and writethrough_cache_enable

These tunable parameters are discussed in the Lustre documentation [1]. By default, the OSS cache is enabled. In summary, these parameters enable or disable the use OSS cache for OSTs. From a benchmarking perspective, disabling the OSS cache removes caching effects from the measurement of OST performance. In production, it is generally a good idea to have the OSS cache enabled.

D. readcache_max_filesize

This tunable parameter is also discussed in the Lustre documentation [1]. By default, the maximum amount of OSS cache is used for caching OST data. For most workloads that benefit from cached access to files, leaving this parameter set to the maximum (default) setting may not be the right choice if large files are also streamed through the same OSTs.

With the default setting, the OSS will attempt to cache all files of all sizes. When a workflow writes large amounts of large files to OSTs (i.e. more data than will fit in the OSS cache), any previously cached data will be evicted. When this eviction occurs, future reads of data will need to bring the data into the OSS from OST media.

By changing this parameter's value to 64 MiB, for example, the OSS will attempt to cache any files that are 64 MiB or smaller. The OSS will not attempt to cache data for files larger than 64 MiB. Understanding the usage patterns of a workflow, and adjusting this tunable parameter appriopriately, can provide great benefit.

VI. How Data Exists In The File System

Understanding how data exists in the file system can be just as important (and more complicated) than understanding where the data exists. The data in the *mb_groups* file indicates the density of user data in the OST but it does not indicate the density of data for any particular file. For that, we look at information from *filefrag*. Fig. 8 is a representation of how a set of four files are stored on an OST. These files were created with a simple IOR job using direct I/O to a single OST, 4 files per OST, and 1 GiB per file. Each color represents one of the 4 files. To see more details of the files, we zoomed into the first 50 MiB in Fig. 9. Notice the white column representing a pre-existing 1 MiB of data. Fig. 10 shows the distribution of contiguous data sizes for the files. There are two interesting observations.

1. If tightly packed, the data for all files should consume exactly 4 GiB of space on disk. Details of the *filefrag* output for the job revealed that it took only an extra 6 MiB to represent the 64 GiB on disk (i.e. there was only 6 MiB of pre-existing data on disk where we were writing data).
2. The histogram of contiguous data sizes tells us that the vast majority of data on disk are single 1 MiB chunks for any particular file. In other words, there are very few portions of files with 2 or more MiB of contiguous data.

Referring back to Fig. 5, if the jobs had been attempting to write to areas of the OST where a lot of data already existed, our first observation could have been quite different. There could have been dozens or even several hundred GiB of "distance" between the start of our first file and the end of the last.

Our second observation leads us to look at how blocks of data for individual files are stored.

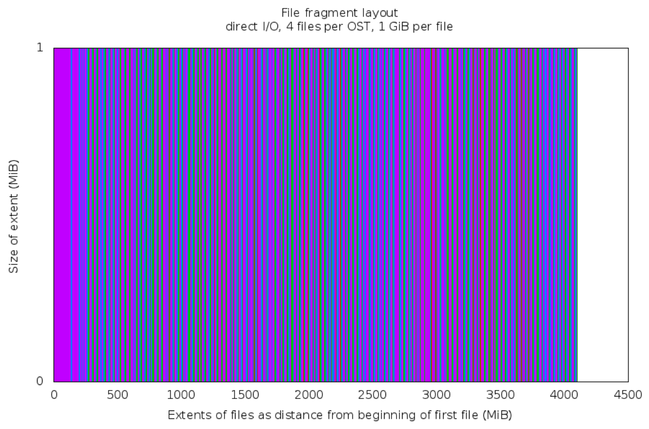

Fig. 8, File fragmentation

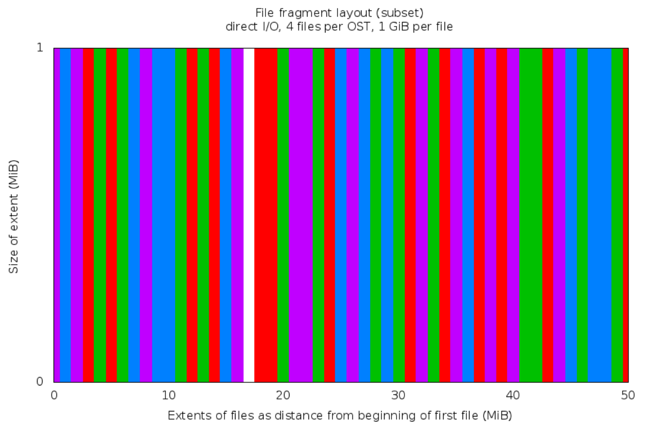

Fig. 9, File fragmentation subset

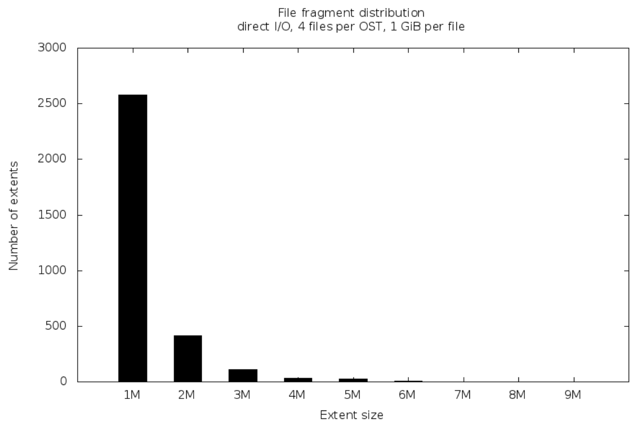

Fig. 10, File fragmentation distribution

A. *OST Pre-Allocation Size*

When the OSS receives a 1 MiB chunk of data from a client to be written to an OST for a particular file, it allocates, by default, a 1 MiB area in which to write. Since the allocated area matches the size of the data from the client, the OSS can simply write that data, one after the other, to the OST. No seeking is involved and writing is very efficient. However, the result is often what we saw in Fig. 8, Fig. 9, and Fig. 10, in which there are very few portions of files that have two or more MiB of contiguous data. In most cases, those 1 MiB chunks of data from multiple files are interleaved.

With data in our files interleaved in this way, reading files becomes inefficient. When receiving a request from a client to read a large chunk of data from a file (e.g. 32 MiB), it may take as many as 32 seeks to fulfill that request. Especially when multiple files are actively being read from an OST, the read requests are nearly always received in a different order than the write requests were.

On way to alleviate this high rate of read seeking is to create more contiguous areas of data for each file. That's what OST pre-allocation can do. An OSS can be configured such that it will pre-allocate *N* MiB of space for files being written to an OST. Fig. 11, Fig. 12, Fig. 13, and Fig. 14 show the effects of adjusting the OST pre-allocation amount.

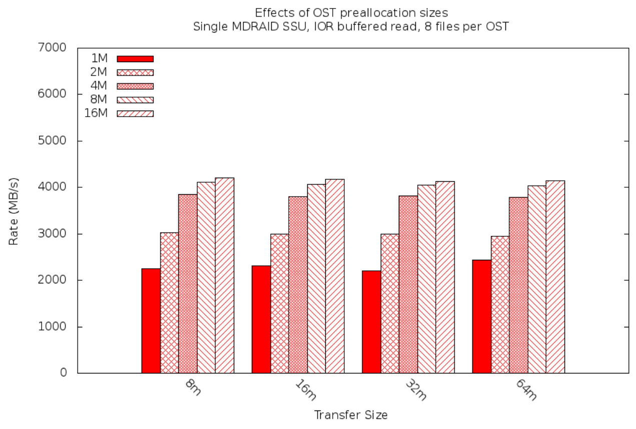

Fig. 11, OST pre-allocation effects, buffered read

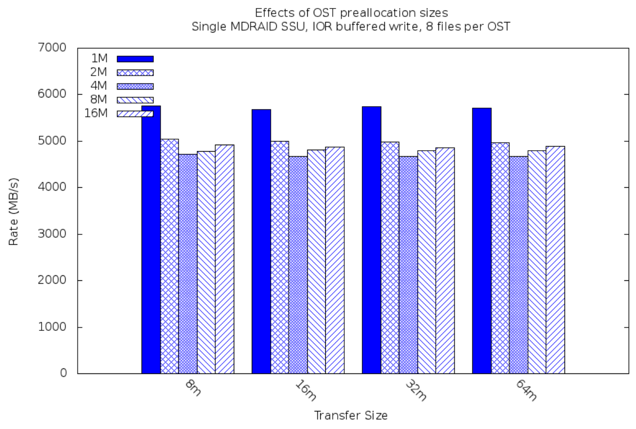

Fig. 12, OST pre-allocation effects, buffered write

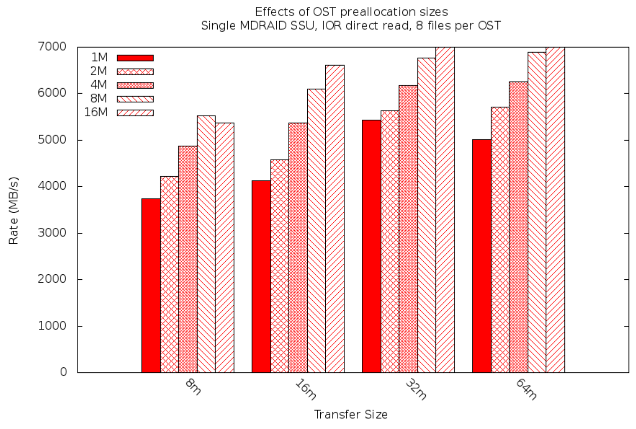

Fig. 13, OST pre-allocation effects, direct read

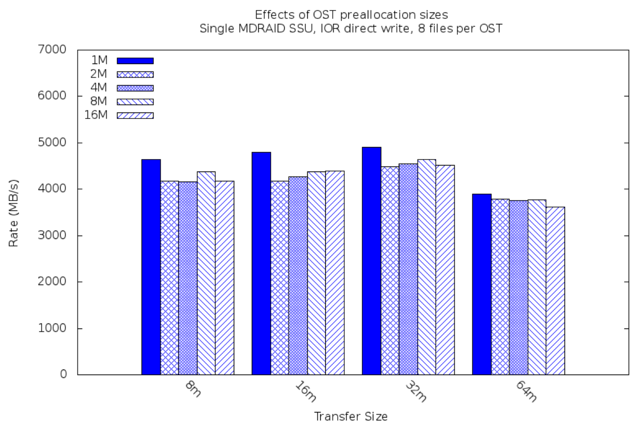

Fig. 14, OST pre-allocation effects, direct write

Notice that as read performance benefits from more contiguous data on disk, write performance decreases. This is because the OST must seek to the correct position in the correct pre-allocated area for a file on the OST in order to place the 1 MiB data chunk coming from the client. The OSS can no longer simply stream the data to the OST, it must properly place it.

B. Lustre Transfer Size (Revisited)

As we have mentioned already, the default transfer size from a Lustre client to a Lustre server is 1 MiB. With release 2.4 of Lustre, this can be increased to 4 MiB. This will increase efficiency of transfers between client and server.

Using 4 MiB Lustre transfers will also solve the problem pointed out with large OST pre-allocation sizes. That is, if we set the OST pre-allocation size to 4 MiB and the server is receiving 4 MiB transfers from the client, the OSS can once again simply stream those 4 MiB chunks out to the OST without seeking. Furthermore, read requests will benefit from the natural increase in contiguous data within files so there will be less read seeking in order to fulfill large read requests. As we see in Fig. 11 and Fig. 13, most of the gains in read performance were realized with even a 4 MiB pre-allocation size.

VII. Future Areas of Research

Cray Inc. will be investigating the use of 4 MiB Lustre network transfers. Also, the Lustre development group of Cray Inc. is also developing tools and methodologies for combatting the fragmentation of data on OSTs.

VIII. Summary

There are many tunable Lustre parameters that affect application and file system performance. There are also many tools and analysis techniques. This paper has presented a set of the most commonly use parameters, tools, and techniques that Cray uses both internally and at customer sites.

References

[1] http://lustre.opensfs.org/documentation

[2] IOR 2.10.3 is available at http://sourceforge.net/projects/ior-sio/

[3] IOR 3.0.0 is available at https://github.com/chaos/ior

[4] D. Petesch and M. Swan, “Instrumenting IOR to Diagnose Performance Issues on Lustre File Systems,” CUG 2013