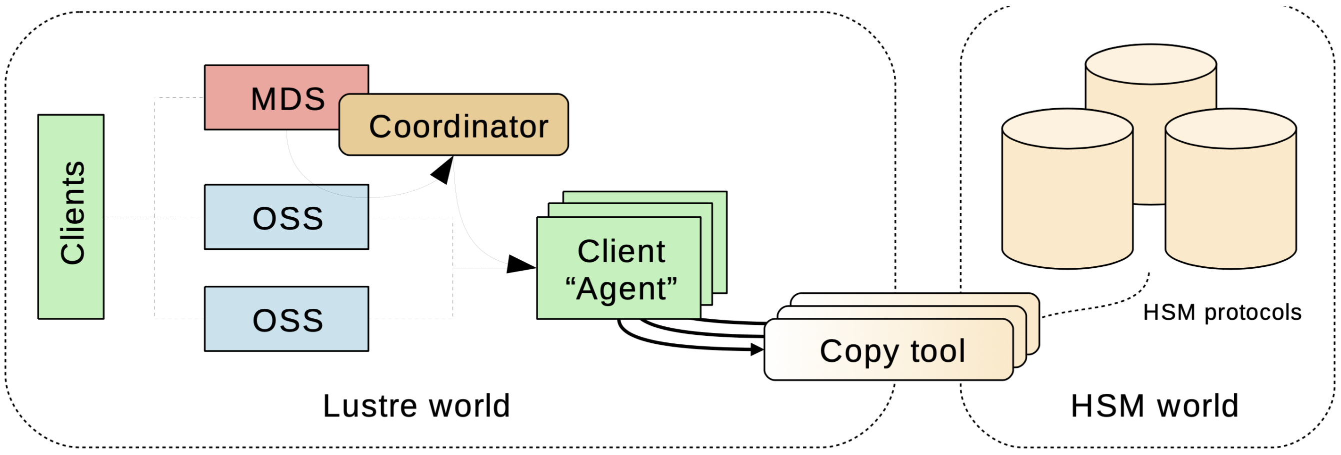
Lustre PCC Investigation and Findings

The purpose of this document is to outline the benefits and drawbacks I have found through the testing and benchmarking of Lustre's PCC functionality. A big important piece of information to keep in mind is that the Lustre filesystem was not tuned at all and is at default settings. In a production environment, better performance might be expected just by nature of fine-tuning the system.

1. Background

In order to understand some of the benefits and drawbacks, it's important to understand how Persistent Client Cache (PCC) functions. Without PCC, when a file is created, the file is created directly on the Lustre filesystem OSS. At the same time, when a file is read, it has to be read from the network filesystem. The whole point of PCC is to speed up the whole read and write processes for files that are accessed or created very frequently. PCC uses a Lustre filesystem feature called Hierarchical Storage Management (HSM). Here is a brief overview of how that system works

1.1. How HSM Works



In essence, the way the HSM system functions is there is something called a "Copy tool" agent running on the Lustre client or whatever system is going to be backing up the data to the HSM. The copy tool "archives" files from the Lustre filesystem to an hierarchical storage system and then when a node in the Lustre network requires access to the file, it is "restored" back to the Lustre filesystem from the HSM system. In addition to the copytool agent being created, the MDT needs to have an HSM coordinator enabled and running.

1.2. How PCC Works

Lustre's PCC re-utilizes the HSM feature for caching. Instead of the Lustre filesystem and an HSM filesystem being the two components, it is the Lustre filesystem and a separate drive on the client to act as a cache. Therefore, with PCC the Lustre filesystem acts like hierarchical storage while the client side drive, which is faster, acts as cache. For example, when a file is created, it gets created on the client's drive first and not on the network Lustre filesystem. However, when the file is created, the metadata for the file is still created on the MDS. The only thing that isn't updated is the OSS with the data of the file itself. When another client on the network requests the file, then the HSM system automatically archives the file from the original client to the Lustre filesystem. Once that process is completed, the new client which requested the file then restores the file from the Lustre filesystem to itself.

Another important finding is that these files are not automatically archived by the system. When a file is created, it will remain on the client in PCC forever until it is archived manually, PCC is detached from the file with `lfs pcc detach`, or another client needs to request the file. Once any of those options happens the file is automatically archived to the OSS. If a client requested the file, it gets restored, to the client that initially requested that file. There are options to archive automatically such as using policy engines like Robinhood

The last interesting piece of information about how Lustre works, is that when it is being configured, it can be set to autocache files based on certain parameters. For example, files could be configured to cache files automatically if they are created with a specific user id or a file name or even a project id. For most of the benchmarking performed, all the files were autocached because when configured, any files created with a uid of 0 is set to get cached automatically.

2. Benchmarking

This section will provide some generic background information on the benchmark information that I was able to gather. This section also has a couple of my own opinions on the different benchmark results and how to interpret them. For more specific information, refer to the benchmarking Confluence wiki on my page.

2.1. System Architecture

There were seven nodes: `mawenzi-01` through `mawenzi-07`

2.1.1. Lustre Server Hardware

Mawenzi nodes *01-03* served as different Lustre Servers with homogenous hardware.

CPU - AMD EPYC 7502P 32-Core Processor

- Clock speed: 2.5GHz
- L1 Cache Size: 32KB
- Translation Lookaside Buffer: 3072 4K pages

Memory - Samsung 8x16GiB (128 GiB) ECC DRAM

- DDR4 clocked at 3200 MT/s

Local Storage - Adaptec Smart Storage PQI SAS 1.5TiB Drives

Networking

- 2x HPE InfiniBand HDR/Ethernet 200Gbps 1-port x16 Adapter (ConnectX-6)
- 4x Intel Corporation I350 Gigabit Network Connection

2.1.2. Lustre Client Hardware

Mawenzi nodes *05-07* served as the Lustre Clients with homogenous hardware.

CPU - AMD EPYC 7402P 24-Core Processor

- Clock speed: 2.8GHz
- L1 Cache Size: 32KB
- Translation Lookaside Buffer: 3072 4K pages

Memory - Samsung 8x16GiB (128 GiB) ECC DRAM

- DDR4 clocked at 3200 MT/s

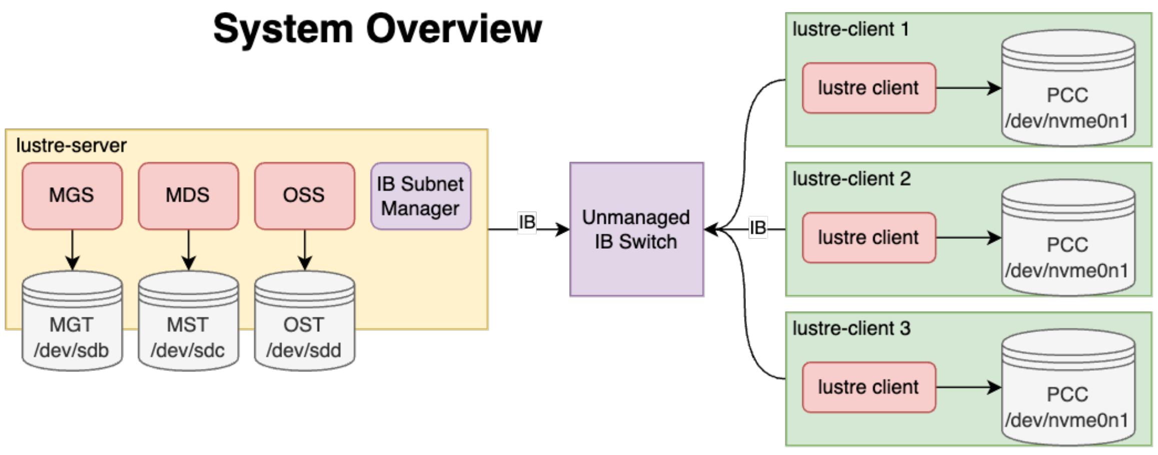
Local Storage - Samsung MZXL51T6HBJR-000H3 1.5TiB NVMe Drive

Networking

- 2x HPE InfiniBand HDR/Ethernet 200Gbps 1-port x16 Adapter (ConnectX-6)
- QLogic Corp. FastLinQ QL41000 Series 10/25/40/50GbE Controller

2.1.3. Physical System Configuration

All ConnectX-6 cards were configured to use IB, rather than ETH, and LNET was configured to use the o2ib protocol, instead of tcp. This allowed the filesystem to utilize the full bandwidth of the cards and not be held back by TCP/IP overhead. All the adapters were connected together using a Mellanox InfiniBand unmanaged switch allowing native EDR IB speeds. In the below figure the system overview is detailed.



2.2. Methodology

Before beginning the benchmarking of all the systems, I had to decide upon which testing tools to use. In the end, I decided `fio` and `ior` would be good based on some resources I found regarding general storage I/O benchmarking and specifically Lustre benchmarking. The first step after deciding on the tools, was to establish a baseline on the drives installed in the systems themselves. After that, `fio` was mostly used to benchmark single client performance with various different options to test how different block sizes and different I/O depths affect performance. `ior` was mostly used to test multi-client configurations with various different configurations to test how caching affected the performance as well as how the number of clients affected performance of the whole system. In addition to those general guidelines, I attempted to consistently use file sizes larger than the amount of memory installed in the system in order to reduce the effects of caching.

All the `fio` benchmarks tested random and sequential read and writes, had a blocksize of 1024k, used the `libaio` I/O engine, and had the `O_DIRECT` API enabled. The only other options that were tested, was using the default block size of 4096, and at other times, adding the `iodepth` parameter and changing that. The reason for changing the block size to the default, which is small, is because, historically, ClusterStor systems struggled with small and random I/O. Therefore, by running benchmarks we can test to see if PCC can help improve those small and random I/O.

The `ior` benchmarks were run with a lot of different options to test how different types of caching may affect performance. Some of the options used can be fairly complicated to explain, so I'd recommend looking through the manual. The "First Steps" section of the IOR documentation explains each of the options I used and tested in great detail with diagrams. In summary though, the options that always varied were `-F`, `-C`, `-e`, and `--posix.odirect=1`. The last option uses the Linux O_DIRECT API, `-e` calls `fsync()` immediately after `write()` to forcefully flush dirty page caches, `-C` then is a way to bypass caching that may be done during the reading process by changing which thread reads from which block (this option is the most complicated and warrants a read at the documentation to understand it in depth), and finally `-F` ensures that every process created by `mpirun` creates its own file and writes to that. With the exception of single client, I used all possible permutations of the above options when running benchmarks with `ior`. The only reason I didn't do the same tests on the single client is because there was already a lot of benchmarking done for single client with the `fio` tool.

2.3. Results

2.3.1. Baseline

Adaptec Smart Storage PQI SAS Drives:

Random Read - 695 MB/s

Sequential Read - 696 MB/s

Random Write - 434 MB/s

Sequential Write - 666 MB/s

Samsung MZXL51T6HBJR-000H3 NVMe Drives:

Random Read - 2374 MB/s

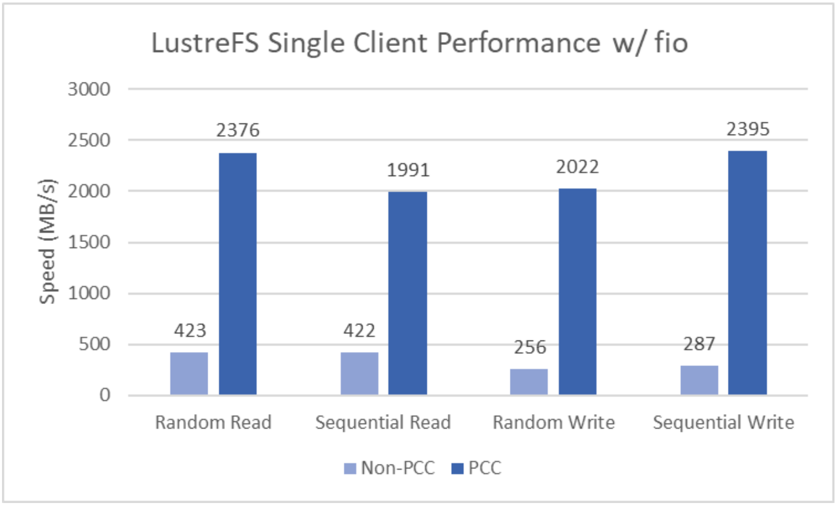
Sequential Read - 2280 MB/s

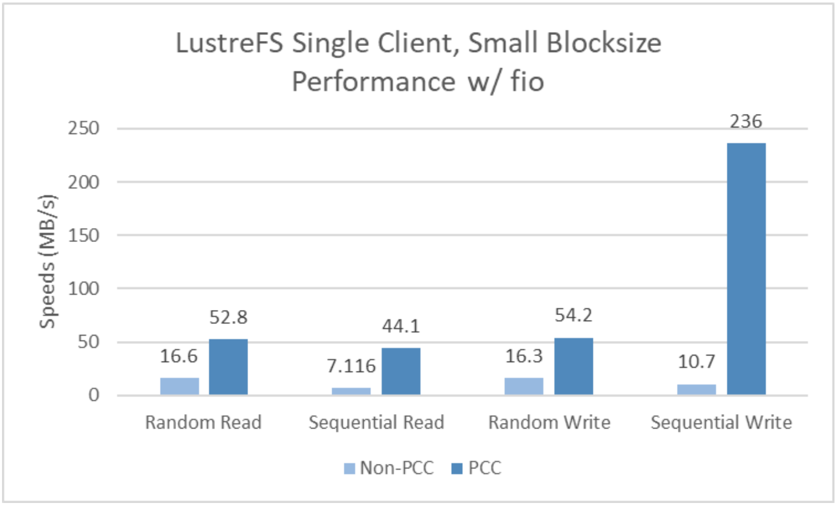
Random Write - 2522 MB/s

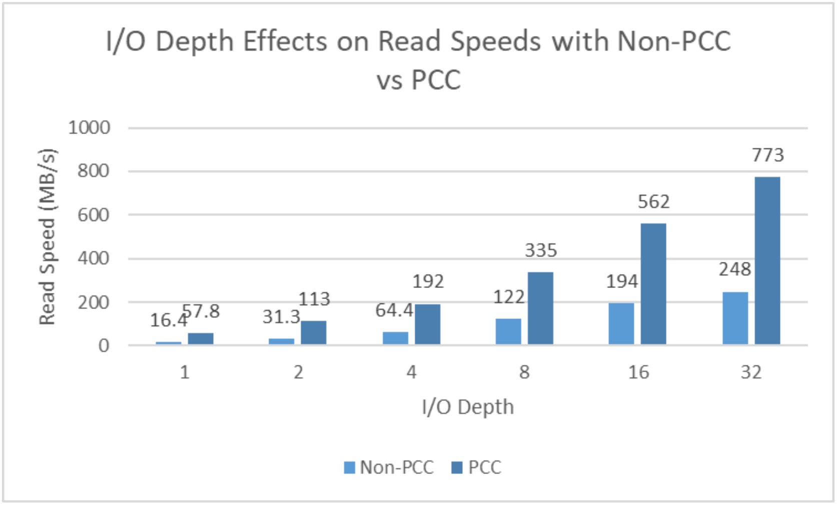
Sequential Write - 2498 MB/s

2.3.2. Single Client Benchmarks

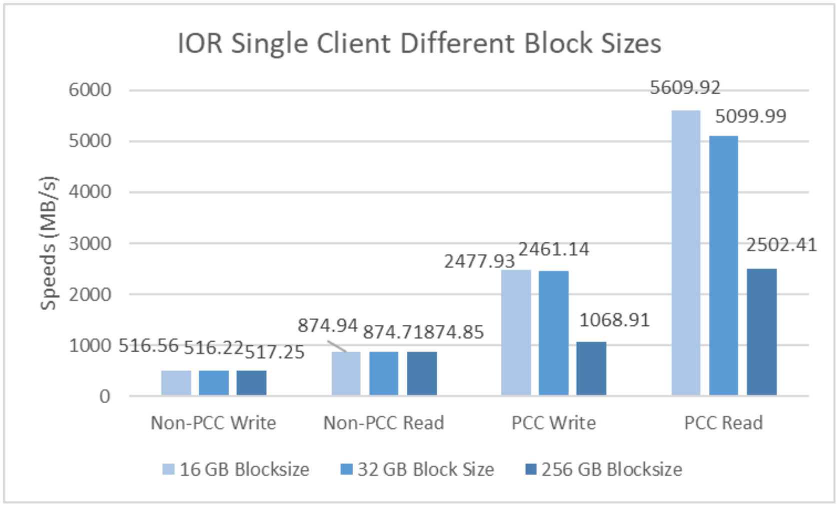
The left most graph is a comparison of the speeds between a client with PCC and without PCC for random and sequential read and writes. Those numbers were achieved with a blocksize of `1024k` , the `O_DIRECT` API being used, and the `libaio` I/O engine. The middle graph changes only the blocksize option to the `fio` default of `4096` and then checks to see how a client with PCC and without PCC behaves with small I/O. The last graph shows how the I/O depth option affects the read speeds between PCC and non-PCC. I/O depth didn't affect write speeds that much as the depth increased. It stayed fairly stagnant. Regardless the results are as follows. For Non-PCC, with an I/O depth of 1, the write speed was 7.491 MB/s and for every I/O depth after that, the write speed was around 11.5 MB/s. For PCC, an I/O depth of 1 had a write speed of 46.7 MB/s and every I/O depth above that had a write speed of around 58.2 MB/s





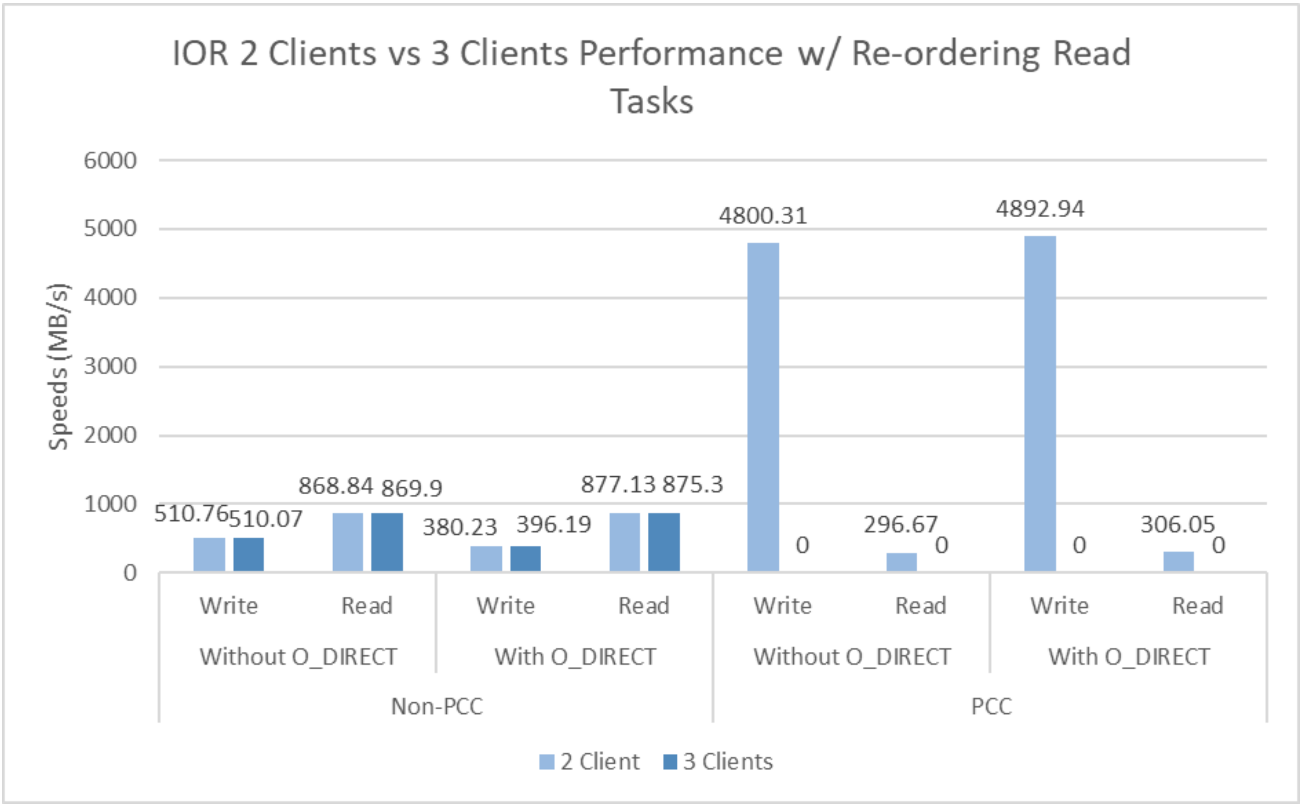


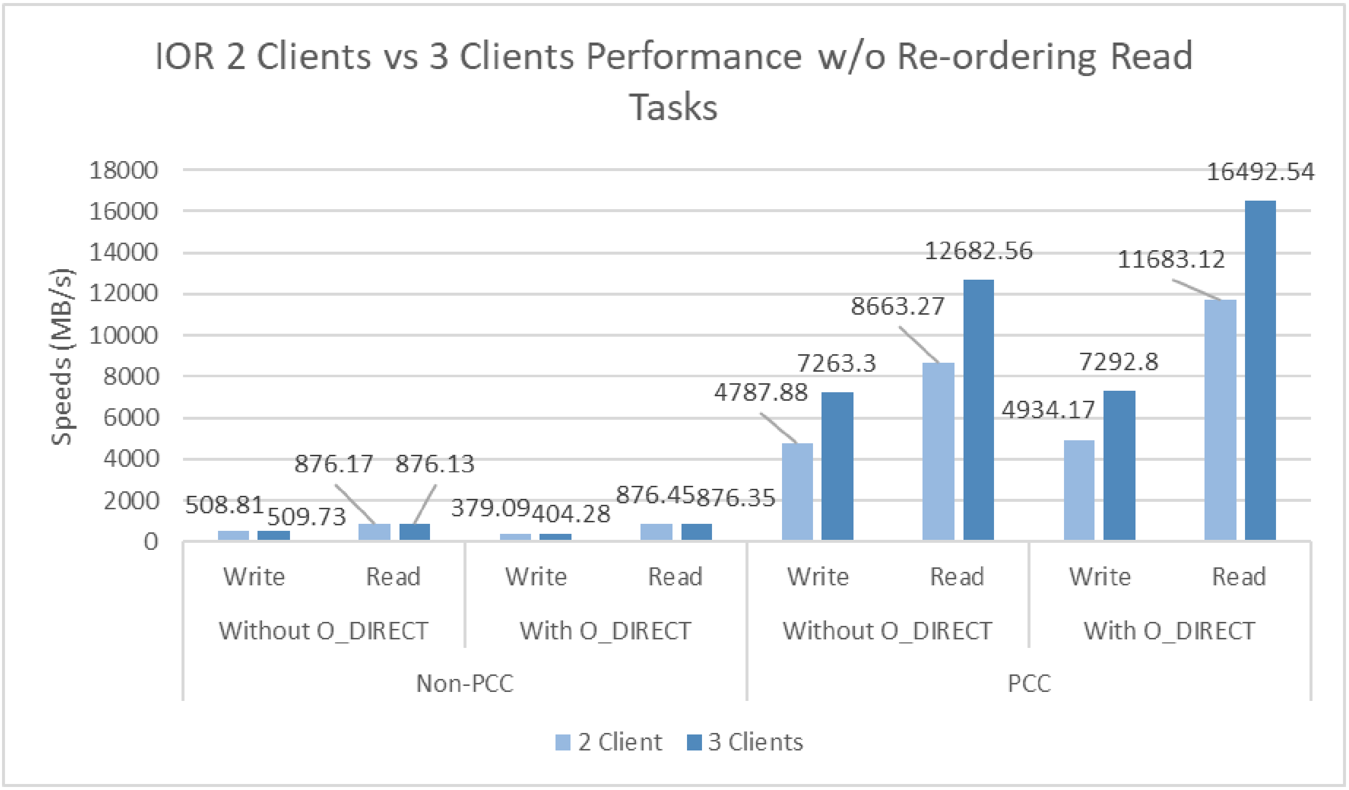
In addition to the above `fio` tests, I ran one test with `ior` testing different block sizes for a large file.



2.3.3. Multi Client Benchmarks

Most of the benchmarks present in this section were run by `ior` with various options. For extremely specific details, refer to all the benchmarking data present in the document at this link. The below two graphs represent the tests performed with all the strict options. So, the leftmost graph has data on 2 client systems and 3 client systems with `-F` , `-C` , `-e` , and `--posix.odirect=1` . However, there is a discrepancy in the graph where the 3 client system has a speed of 0 MB/s. The reason for that discrepancy is the `-C` option which I will explain later. The graph on the right, follows the same options but without the `-C` option.





2.4. Conclusion

As a whole PCC seems to perform better than Non-PCC systems for almost all of the benchmarks, however, there is one important caveat to mention which has to do with the `-C` option for multi-client systems. When that option is used with 3 client systems, the benchmark fails to finish because the server couldn't write the file fast enough and then serve it. Essentially, when PCC is activated and a file is created, the data in the file only exists in the PCC cache of the client that created that file. The moment another client requests access to that file, the copy tool agent running on the "creation" client has to archive that file to the Lustre OSS before it can serve the file to the client that requested it. Therefore, if another client is requesting to read a file and is faster than the server can keep up, it will fail. Therefore, the best workloads for Lustre filesystems with PCC are workloads where a large file already exists on the filesystem and multiple clients are attempting to read it. In addition, it's even more beneficial if multiple clients need to access that large file multiple times. In addition, another use case could be where a client creates a large file that it needs to access immediately but can then be archived to the OSS in the background for other clients to access later. Lastly, workloads which can take advantage of higher I/O depths when reading large files will benefit greatly from Lustre PCC. Conversely, the slowest workloads for a Lustre PCC system is where a file is created on the system by a client and then another client immediately requests to read it while other clients are also requesting to read other files.

3. References

https://lustre.ornl.gov/lustre101-courses/

https://doc.lustre.org/lustre_manual.xhtml#lustrehsm

https://doc.lustre.org/lustre_manual.xhtml#pcc

https://fio.readthedocs.io/en/latest/fio_doc.html

https://ior.readthedocs.io/en/latest/userDoc/tutorial.html

https://wiki.lustre.org/images/4/40/Wednesday_shpc-2009-benchmarking.pdf

https://bpb-us-e1.wpmucdn.com/blogs.rice.edu/dist/0/2327/files/2014/03/Fragalla-Xyratex_Lustre_PerformanceTuning_Fragalla_0314.pdf

4. Definitions

PCC - Persistent Client Cache. A feature that is part of Lustre to help improve the performance of the system

HSM - Hierarchical System Management. Another feature that is part of Lustre to help Lustre systems interact with older, archival forms of storage

MDS - Metadata Server. A core part of the Lustre network that is in charge of tracking and serving the metadata of the files

OSS - Object Storage Server. Another core part of the Lustre network that is in charge of handling the objects in the Lustre system themselves such as storing the data contained in files.