A Simple Measuring Model for Evaluating the Performance of Small Block Size Accesses in Lustre File System

Naveenkumar Jayakumar
Department of Computer Engineering
Bharati Vidyapeeth Deemed University
College of Engineering
Pune 43, India
naveenkumar@bvucoep.edu.in

Anuja Kulkarni
Department of Computer Engineering
Bharati Vidyapeeth Deemed University
College of Engineering
Pune 43, India
anukul2009@gmail.com

***Abstract*—Storage performance is one of the vital characteristics of a big data environment. Data throughput can be increased to some extent using storage virtualization and parallel data paths. Technology has enhanced the various SANs and storage topologies to be adaptable for diverse applications that improve end to end performance. In big data environments the mostly used file systems are HDFS (Hadoop Distributed File System) and Lustre. There are environments in which both HDFS and Lustre are connected, and the applications directly work on Lustre. In Lustre architecture with out-of-band storage virtualization system, the separation of data path from metadata path is acceptable (and even desirable) for large files since one MDT (Metadata Target) open RPC is typically a small fraction of the total number of read or write RPCs. This hurts small file performance significantly when there is only a single read or write RPC for the file data. Since applications require data for processing and considering in-situ architecture which brings data or metadata close to applications for processing, how the in-situ processing can be exploited in Lustre is the domain of this dissertation work. The earlier research exploited Lustre supporting in-situ processing when Hadoop/MapReduce is integrated with Lustre, but still, the scope of performance improvement existed in Lustre. The aim of the research is to check whether it is feasible and beneficial to move the small files to the MDT so that additional RPCs and I/O overhead can be eliminated, and read/write performance of Lustre file system can be improved.**

Keywords-Big Data; Metadata; Lustre; Active Storage; Small File

I. INTRODUCTION

The quantity of data generated and consumed by HPC (high-performance computing) applications is increasing exponentially. Current I/O paradigms and file system designs are often overwhelmed by this deluge of data. To improve the I/O throughput to a certain extent, parallel file systems incorporate features such as data striping, sharing resources, etc. File systems such as Lustre, AFS, NFS use a single metadata server to manage globally shared file system namespace. While simple, scalability can't be achieved by this design, having as result the metadata server to become a bottleneck and a single point of failure. In big data environment, most of the files (70%) are small, and most data (nearly 90%) is placed in big files. The number of small files is big though used space is not. Small files consume more resources and produce big slowdown. Also, the latency of access to small files is important. The problem is studied by considering Lustre file system as a use case. Lustre file system is open source, most widely used in HPC environment and easy to implement.

Lustre is a General Public Licensed (GNU), open-source distributed, parallel file system. It is developed and maintained by Sun Microsystems Inc. Lustre is supported by Linux operating system, and it presents a POSIX interface to its clients with which shared file objects can be accessed in parallel. The key features offered by the Lustre file system are, among others, scalability, high-performance, POSIX compliance, high availability, interoperability. Lustre is an object-based file system with three main components: Object Storage Servers (OSSs), Metadata Servers (MDSs), and clients. When a client wants to write a file to the Lustre file system, first it communicates with the MDS with a write request. The MDS checks for the user authentication and the intended file location. Depending on the file system settings or the directory settings, the MDS sends back a list of OSTs that can be used by the client to write the file. Once the reply is sent by the MDS, the client can now directly interact with the assigned OSTs without any contact with the MDS. This is applicable for any file regardless of size, whether it's a few bytes or a few terabytes [1]. The main advantage of Lustre file system over Storage Area Network (SAN) file system and Network File System (NFS) is that it provides: a global name space, the ability to distribute very large files across multiple storage nodes, wide scalability, in performance as well as storage capacity. Since large files are distributed across many nodes in the cluster environment, Lustre file system is best suited for high-end HPC cluster I/O systems. Lustre servers are equipped with multiple storage devices which provide high-availability.

Lustre can handle and serve up to tens of thousands of clients. The high-availability mechanism should enable any cluster file system to handle server failures or reboots transparently. The Lustre failover mechanism is robust and transparent, and it allows servers to be upgraded without the need to take the system down. Accessing small files on the Lustre is not efficient. The read/write performance of Lustre file system is currently optimized for large files, i.e. files more than few megabytes in size. To access any file, client has to send initial open RPC to the MDT and after that, to fetch the data from the OSTs, there are separate read/write RPCs to the OSTs. In addition to this, there are separate RPCs to perform disk I/O on MDT and OST. This functionality separation is desirable for large files since one open RPC to MDT requires less execution time compared to total number of read/write RPCs to OSTs, but this affects overall performance of small file significantly when there is only one read or write RPC to the OSTs for accessing file data. One possible way to improve small file performance in Lustre is to put the data for small files only on the MDT where metadata also resides so that additional RPCs and I/O overhead can be eliminated. This research aims to check whether it is feasible and beneficial to move the data from OST to MDT and if yes how much data can be pushed onto the MDT.

II. BACKGROUND

A. *Lustre: Component View On Architecture*

Lustre file system is supported by Linux operating system, and it presents a POSIX interface to its clients with which shared file objects can be accessed in parallel. Lustre is an object-based file system with three main components: Object Storage Servers (OSSs), Metadata Servers (MDSs), and clients. Lustre components are as shown in Figure 1.

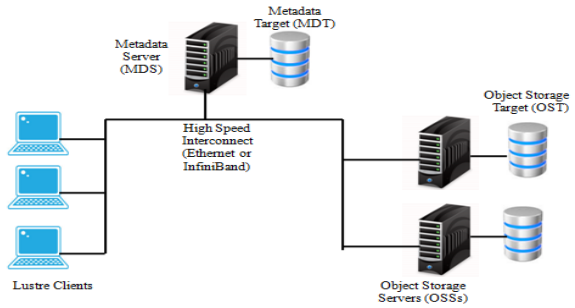

Fig. 1. Lustre: Component view on architecture

B. *Lustre Components*

- Metadata Server (MDS) - Metadata servers provide metadata services. Correspondingly the Metadata Client (MDC) is a client of those services that makes metadata available to the Lustre clients. File metadata, such as file names, access permissions, directory structures, is stored on the Metadata Target (MDT).
- Management Server (MGS) - The management server, stores configuration information for all available Lustre file systems in a cluster. Lustre clients and Lustre target contacts the MGS to retrieve and provide information respectively. The MGS can have a separate disk for storage, or it can share a disk with a single MDT.
- Object Storage Server (OSS) - The OSS exposes block devices and serves data to the client. Correspondingly, Object Storage Client (OSC) is a client of the services. Each OSS manages one or more OSTs (Object Storage Targets). OSTs are used to store file data in the form of objects.

C. *Lustre Functionality*

The collection of MDS/MGS and OSS/OST are referred to as Lustre server front ends and ldiskfs, fsfilt as Lustre server back ends. In Lustre, file operations like create, open, write, read, etc. require metadata information which is stored on MDS. This service provided by MDS is accessed through client interface module, known as Metadata Client (MDC).

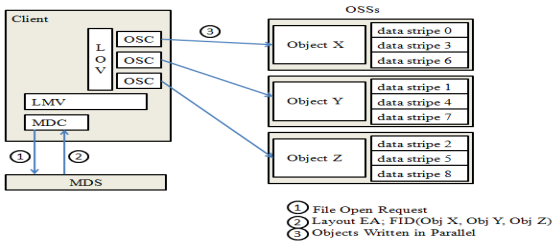

Fig. 2. Basic Lustre I/O operation

From the MDS point of view, every single file is composed of numerous data objects, and these objects are striped across one or more OSTs. Each data object is assigned with a unique object id. MDS stores normal file metadata attributes like inode with some additional information known as Extended Attributes (EA). Extended attribute is used to store file objects layout information (also called as striping EA) which is used to map file object id with corresponding OSTs. Consider one example, if a file P has a stripe count of three, then its EA will be :

EA → <obj id X, ost a>
<obj id Y, ost b>
<obj id Z, ost c>
Stripe size and stripe width

Before reading the file stored on the OSTs, client will communicate with MDS via MDC (Metadata Client) and collect the information about OSTs where the objects of the file are stored. According to the example, the objects of the file are stored on OST a, OST b and OST c. At client side, this information is structured in LOV (logical object volume). Now the client can communicate with corresponding OSTs through a client module interface known as OSC (Object Storage Client). LOV is the software layer in the client stack and it is used to direct the pages towards the correct OSCs and then the

OSCs assemble a vector of pages, group them and send them to OST through PoratlRPC and LNET.

III. RPC MECHANISM IN NATIVE LUSTRE

In Lustre file system, the communication between client and server is coded as an RPC (Remote Procedure Call) request and response. This middle layer is known as ptl-rpc i.e., Portal RPC which translates the file system request in the form of RPC request and response and the Lustre Networking (LNET) provides network infrastructure to put that down onto the wire.

A. RPC to MDS

Let's assume client C1 wants to open the file /lustre/d1/d2/test.txt to read. Here /lustre is the mount point. The first RPC request is lock_enqueue with lookup intent. This request is sent to MDS for the lock on d1. The second RPC request is also lock en-queue with lookup intent. This request is also sent to MDS asking inodebits lock for d2. The inodebits lock will be returned with its resources represented by the fid of d1 and d2. The third RPC request is a lock_enqueue with open intent, but it is not asking for lock on test.txt. The file content is provided by OST, and hence it can be opened and read without a lock form MDS. Now the lock is requested from an OST.

B. RPC to OSS

After getting the EA structure from the MDS, the client can communicate with OSS. Consider one example with one client and four OSTs. Client C1 reads file P. The file P is striped across four OSTs with data objects P1, P2, P3, and P4. First, client C1 sends lock enqueue requests to all four OSTs in parallel, asking for read lock with intent flag set. If any of the objects client C1 is asking has a blocking request, then the corresponding OSTs don't grant the lock. Instead, they just return the information described by a data structure lvb (Lustre Value Block). lvb contains the information like file size, modification time, etc. If there are no any conflicts, the read locks on the entire file objects will be granted to C1 with lock mode PR.

1) When client C1 wants to access a file stored on OSTs, it sends LOOKUP RPC to MDS
2) At the MDS side, after receiving RPC from client C1, the Lock Manager will grant the lock for the resource requested by the client. Now client C1 sends the second RPC to the MDS with the intent to create or open the file.
3) So, at the end of step 2, C1 will get the lock, extended attribute (EA) information and other metadata details which the client needs to communicate with OSTs.
4) Once the client gets the lock and EA information, it can perform I/O operations.
5) MDS maintain queue to track the allocated resources. When multiple clients try to access the same file then the new client has to wait in the waiting queue till the time the current owner of the lock releases the lock. Once the lock is released, the MDS will then hand over the lock to the new client.
6) For example, client C2 wants to access the same file which was earlier opened by C1 then C2 will be placed in the waiting queue. MDS will send a blocking AST to C1 to revoke the lock granted. On receiving the blocking AST, C1 will release the lock. In some scenario where client C1 is down or something goes wrong, MDS will wait for a ping timeout of 30 seconds after which it will revoke the lock.
7) Once the lock is revoked, the MDS will grant a lock handle and EA for the file to C2. C2 can proceed with I/O once it gets the lock handle and EA information.

C. RPC Mechanism in Proposed System

This research aims to improve the performance of small files by putting small files data only on MDT so that the additional read/write RPCs to the OSTs can be eliminated. This allows the improving of small file performance. The MDT storage is configured with RAID 1+0 which is well-known for high-IOPS. Data on MDT can be used in conjunction with DNE (Distributed Namespace) to improve the efficiency. To store file data on the MDT, system administrators or users must explicitly specify a layout that will allow storing the data on MDT at the time of file creation. The maximum file size for which data can be stored on MDT must be specified by the administrator so that users cannot store large files on MDT which will cause problems to other users. If the layout of a file specifies to the client to store the data on the MDT, but the file size reaches to the maximum size specified by the administrator, then the data will be migrated to OST. In native Lustre architecture, the small file data is present on the OST. Whenever the client wants to access data from small files, it has to send RPCs to both MDS and OSS. As the number of RPCs get increased the overall latency increases which affects the I/O performance. In the proposed system, the data for small files will be stored on MDT instead of OSTs. Since the data and metadata are present on MDT, the client has to send RPC to MDS only. As the number of RPCs decreases the overall latency also decreases and I/O performance for the small files can be improved to some extent.

IV. CORE METHODOLOGY

A. Lustre Setup

Figure 2 shows the proposed system architecture in which two virtual machines are created. The virtual machines are formatted with CentOS 6.7 operating system and kernel is patched with Lustre 2.7.0 software release. One virtual machine is configured with MDS (Metadata Server) having single MDT (Metadata Target) and other with OSS (Object Storage Server) with two OSTs (Object Storage Target). Lustre clients are connected to the server over Ethernet connection, and they are also formatted with CentOS6.7 operating system. Lustre file system contains two types of servers, a metadata server (MDS) and one or more object storage servers (OSSs). Since the MDS is the starting point for all POSIX file system operations, it must quickly handle many remote procedure calls (RPCs). To access the file from a Lustre file system, Lustre client needs to query to the metadata target via the MDS to

obtain the file attributes (e.g., file type, owner, access permission) along with the file data layout.

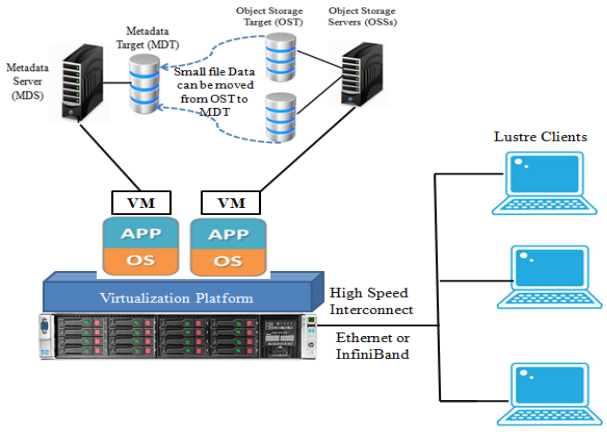

Fig. 3. System architecture

The IO operations on the MDT are mostly small and random. If more data is cached into memory, then the MDS can respond to client queries faster. Therefore, the MDS needs a large amount of RAM to cache the working set of files, and powerful CPUs to handle the simultaneous inquiries. The amount of memory required by the MDS depends on a number of clients and the number of files accessed by them in their working set. Apart from 1 GB memory required by the operating system and 4 GB memory needed for the file system journal, nearly 0.1% of the MDT size is needed for the MDT file system's metadata cache. The remaining RAM is available for caching the file data for the client/application file working set. The file working set cached in the RAM is not used by the clients actively all the time, but it should be kept hot to reduce IO latency and avoid adding extra read IO/s to the MDT which is under load. For the kernel data structure approximately 2 KB of RAM is needed, to keep a file in the cache without a lock. For the LDLM (Lustre Distributed Lock Manager) lock every client requires 3 KB of RAM for each file in the cache.

B. Lustre Benchmarking Tools

Native Lustre performance can be monitored by using different benchmarking tools. The IOzone [2] tool is used to test the read/write performance of small files on Lustre platform. IOR [3] test is conducted to benchmark the speed of read/write operations on MDT and OST. Data Duplicator [4] command is implemented to monitor whether it is beneficial to move small file data on MDT to minimize the number of RPCs. The dd command is used to check the feasibility of the proposed model. It is not possible to put the data on MDT and test the small file performance directly as the file layout must be specified before accessing any file.

1) IOzone: IOzone is the benchmarking tool for the file system. The benchmark measures a variety of file operations (read, write, etc.) and generates the results accordingly. IOzone runs under multiple operating systems. IOzone is useful for performing file system analysis broadly, and it tests file I/O performance for the following file operations: Read, write, re-write, re-read, fread, fwrite, read backwards, random read, read strided, mmap, pread, aio_read, aio_write.

2) IOR: IOR (Interleaved or Random) is designed to measure I/O performance of parallel file system at both the MPI-IO and POSIX level. The IOR benchmark is developed by LLNL (Lawrence Livermore National Laboratory), and it tests system performance by considering parallel/sequential read/write operations. The IOR_survey script is used to test the performance of the Lustre file system. It uses various interfaces and access patterns to test the performance of a parallel file system. MPI is used for process synchronization. Under the control of constants defined at compile time, I/O is done via MPI-IO. The data is written and read using parallel transfers of blocks of contiguous bytes. These blocks are of equal size, and they do not overlap each other. The test consists of three main operations - creating a new file, writing the data in it, then reading the data block. There are two ways to run IOR:

   (a) Interactive command line with arguments. For example:
   IOR -w -r -o filename
   This performs write and read to the specified file.

   (b) Interactive command line with scripts. For example:
   IOR -w -f script.
   This performs all tests in the script to check write data performance.

3) DD (Data Duplicator) Command: The dd command stands for "data duplicator". It is used for data copying and converting. It is a very powerful utility for Linux which is used for multiple applications like –Backup. It can restore a partition or the entire hard disk, backup of Master Boot Record (MBR) and is also used by Linux kernel make file to make boot images. Improper usage of dd command can lead to data loss, hence it must be run by the super user. The syntax for dd command is - dd if=<source file name> of=<target file name> [options]--[4]. n the syntax above ' if 'stands for input-file and ' of ' stands for output-file. The 'source file name' and 'target file name' in the syntax can be disks, partitions, files or devices from which data can be read or written. To test the native Lustre performance, 'dd' command can be run on Lustre client node. 'dd' command is used to monitor the write performance of disk device.

V. RESULT ANALYSIS

A. Small File Performance on Native Lustre Platform

The IOzone test is conducted on Lustre client node to check the I/O performance of read/write file operations in native Lustre platform and also the impact of record size on I/O performance. A set of IOzone tests is executed on the client to find out the influence of record size. To evaluate the read/write performance with IOzone, the file size is set to 2G with stripe size 4M and data striped across 2 OSTs. Figure 4 shows the results of the tests. The vertical axis is the read/write bandwidth (data transfer rate in bits/sec), and the horizontal axis is the record size. The red line shows write performance, and the blue line shows read performance.

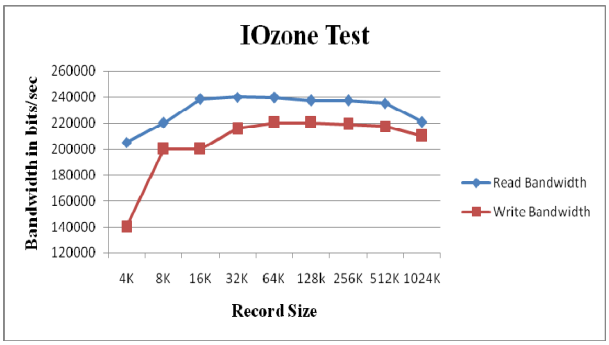

Fig. 4. Small file performance in Lustre

As shown in Figure 4, the trend of the two lines is almost the same, but when the record size is 128K, 256K, and 1M, the performance is better than others. For the record size 4K or 8K, most of the RPC packets contain only one or two pages and hence the I/O efficiency is much lower. From the graph above, it can be concluded that for small record size less number of RPCs are required and the overall I/O performance gets affected when there is only one read or write RPC to the OSTs for accessing the file data. This shows that in the native Lustre platform small files get processed slowly with less record size which will decrease the overall I/O performance.

B. Feasibility Check of the Proposed System

The write performance of OST and the write performance of MDT are compared by running dd command on client node and MDT. As shown in Figure 5, when dd command is run on the client, the latency is more, compared to latency achieved when dd command is run on MDT. This means, when the client accesses small file data stored on OST, it has to send RPCs to both MDS and OSS and thus the time required to perform read/write operation increases. As the client has to send RPCs to both MDS and OSS, the network overhead gets increased.

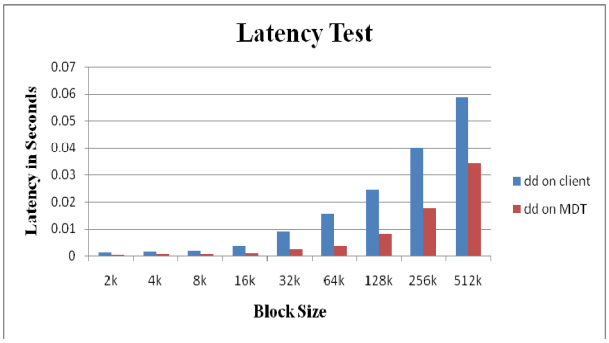

Fig. 5. Latency comparison

When the dd command is run on MDT, the latency is less. This means that if the small files get stored on the MDT, they can be accessed very fast with the minimal number of RPCs. In this case, since the data and metadata will both get stored on MDS, there is no need to send RPCs to the OSS. Considering the results above, it can be concluded that, it is feasible to move small file data on MDT in order to increase small file IO performance. One more important factor that must be considered here is the maximum block size of file for which the data can be moved to MDT. The maximum block size for which data can be stored on MDT can be obtained by evaluating the results of latency test on MDS for different block sizes. Figure 6 shows the results achieved by running dd command on MDT. Here, the latency is less for the block size of up to 8KB. For the block size of 16 KB and above the latency increases exponentially. If the latency is high for the block size of 16 KB and above then it is not beneficial to store the file data on MDT because the small files are more latency sensitive. For the maximum of 8 KB block size, the latency is very low.

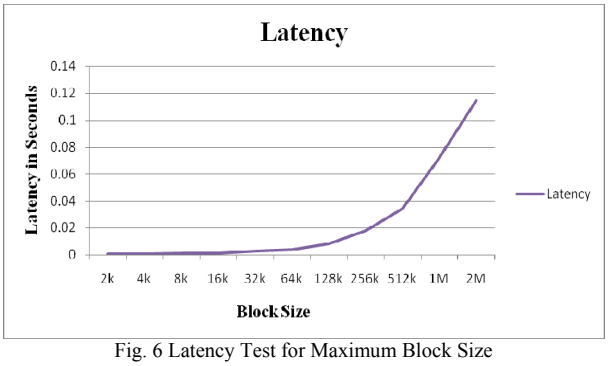

Fig. 6 Latency Test for Maximum Block Size

So it can be concluded that in Lustre file system with 2 GB RAM for MDT, maximum 8 KB record size files can be stored on MDT. These files require minimum latency without impacting the system performance and that way the I/O performance of the small files increases.

VI. CONCLUSION

The read/write performance of Lustre file system is currently optimized for large files. The read/write performance depends on the number of LOOKUP RPCs between client to OST and client to MDT. To access any file, client has to send initial open RPC to the MDT and after that, to fetch the data from the OSTs, there are separate read/write RPCs to the OSTs. This functionality separation is desirable for large files, but this affects the overall performance of small files significantly when there is only one read or write RPC to the OSTs for accessing the file data. Also, the small files are more latency sensitive. One possible way to improve small file performance in Lustre is to put the data for small files only on the MDT where metadata also reside so that additional RPCs and I/O overhead can be eliminated. In the current work, the native Lustre I/O performance is tested by different benchmarking tools (IOzone, IOR) and it is evaluated and proposed that storing small file data on the MDT is feasible and the maximum record size for which data can be stored on MDT with 2 GB RAM is 8KB. The scope of this research is not limited to Lustre file system. It can also be applied to other parallel distributed file systems (for example Ceph, HDFS) that store data and metadata on separate servers. Lustre file system is most widely used in big data environment in which most of

the files (70%) are small, and most data (nearly 90%) is placed in big files. The number of small files is big though used space is not. Small files consume more resources and produce big slowdown. Also, the latency of access to small files is important. This research is very helpful in big data environment to handle small files and to increase the read-write performance of small files.

REFERENCES

[1] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, I. Huang, Understanding Lustre Filesystem Internals, National Center for Computational Sciences, Oak Ridge National Laboratory, 2009

[2] Iozone Filesystem Benchmark, http://www.iozone.org

[3] NERSC, IOR Test, https://tinyurl.com/jnmdo8u

[4] The Linux Juggernaut, 12 Linux dd Command Examples, https://tinyurl.com/ycg6acpm