Minimizing Lookup RPCs in Lustre File System using Metadata Delegation at Client Side

Vilobh Meshram, Xiangyong Ouyang and Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
{meshram, ouyangx, panda}@cse.ohio-state.edu

Abstract

Lustre is a massively Parallel Distributed File System and its architecture scales well to large amount of data. However the performance of Lustre can be limited by the load of metadata operations at the Metadata Server (MDS). Because of the higher capacity of parallel file systems, they are often used to store and access millions of small files. These small files may create a meta-data bottleneck, especially for file systems that have only a single active metadata server. Also,in case of Lustre installations with Single Metadata Server or with Clustered Metadata Server the time spent in path traversal from each client involves multiple LOOKUP Remote Procedure Call (RPC) for path traversal before receiving the actual metadata related information and the extended attributes adding additional pressure on the MDS.

In this paper we propose an approach to minimize the number of Lookup RPC calls to MDS. We have designed a new scheme called Metadata Delegation at Client Side (MDCS), to delegate part of metadata and the extended attributes for a file to the client so that we can load balance the traffic at the MDS by delegating some useful information to the client. Initial experimental results show that we can achieve up to 45% improvement in metadata operations throughput for system calls such as file open. MDCS also reduces the latency to open a Lustre file by 33%.

1 Introduction

Modern distributed file systems architectures like Lustre [5, 3], PVFS [7], Google File System [13] or the object based storage file systems [12, 16] separate the management of metadata from the storage of the actual file data. These architecture have proven to easily scale the storage capacity and bandwidth. However the management of metadata remains a bottleneck [10, 19]. Studies has shown that approximately 75% of all the file system calls access metadata [18]. Therefore, the efficient management of metadata is crucial for the overall system performance. Along with this it will be very beneficial if we can minimize the time spent in communication between the interacting Client and Server nodes in case of file system calls which access metadata.

Lustre is a POSIX compliant, open-source distributed parallel filesystem.Due to the extremely scalable architecture of the Lustre filesystem, Lustre deployments are popular in scientific supercomputing, as well as in the oil and gas, manufacturing, rich media, and finance sectors. Lustre presents a POSIX interface to its clients with parallel access capabilities to the shared file objects. As of this writing, 18 of the top 30 fastest supercomputers in the world use Lustre filesystem for high-performance scratch space. Lustre is an object-based filesystem. It is composed of three components: Metadata servers (MDSs), object storage servers (OSSs), and clients. Figure 1 illustrates the Lustre architecture. Lustre uses block devices for file data and metadata storage and each block device can be managed by only one Lustre service. The total data capacity of the Lustre filesystem is the sum of all individual OST capacities. Lustre clients access and concurrently use data through the standard POSIX I/O system calls.

MDS provides metadata services. Correspondingly, an MDC (metadata client) is a client of those services. One MDS per filesystem manages one metadata target (MDT). Each MDT stores file metadata, such as file names, directory structures, and access permissions. OSS (object storage server) exposes block devices and serves data. Correspondingly, OSC (object storage client) is client of the services. Each OSS manages one or more object storage targets (OSTs), and OSTs store file data objects.

In our studies with Lustre filesystem, we noticed a potential performance hit in the metadata operation part caused by repeated LOOKUP RPC calls from

Lustre clients to the MDS. Since path LOOKUP is one of the most often seen metadata operations in a typical filesystem workload, this issue may result in performance degradation in Lustre MDS performance.

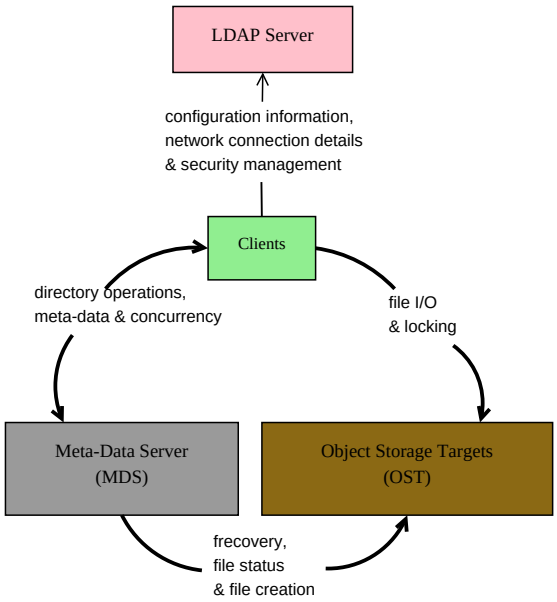

Figure 1: Basic Lustre Design

In this paper we propose a new approach called "Metadata Delegation at Client Side" (MDCS) in this paper. With this new strategy, we are able to minimize the number of LOOKUP RPC [9] calls and hence the request traffic on MDS. We delegate metadata related attributes to the special clients which we call "Delegator Client", which are the client nodes that open/create files initially. In case of increased workload at the MDS, we distribute the load by assigning the ownership to different clients depending upon the client which at first created or opened the file. So in case of an environment where a file is subsequently accessed by many clients at regular intervals, we can distribute the workload from MDS to the Delegator Client and the Normal Client who are subsequently accessing the files will fetch the metadata related attributes from the Delegator Client and hence

In a summary, our contributions in this paper are:

- We have identified and revealed a potential performance hit in Lustre MDS that is caused by repeated LOOKUP RPC calls to MDS.
- We have proposed a new approach (MDCS) to address the potential performance issue.
- We have implemented this MDCS mechanism into Lustre filesystem. Initial experiments show up to 45% improvement in terms of metadata operation throughput over the basic Lustre design.

The rest of paper is organized as follows. In section 2, we describe the background of Metadata operation in Lustre File System and some well known problems in metadata access from MDS. In section 3, we explain how the path name lookup is performed in Lustre and also explain the approach used by us to minimize the number of RPC's by making use of Design presented in Section 4. In section 4, we present our detailed designs and discuss our design choices. In section 5, we conduct experiments to evaluate our designs and present results that indicate improvement. In section 6, we discuss the related work. Finally we provide our conclusion and state the direction of the research we intend to conduct in future.

2 Background

2.1 Lustre Metadata Server and its components

Lustre Metadata Server (MDS) [5, 8, 3] is the critical component of the Lustre File System. Lustre File System has an important component known as Lustre Distributed Lock Manager (LDLM) which ensures cache metadata integrity i.e. atomic access to metadata. In Lustre we also have a LDLM component at the OSS/OST [5, 8, 3] but since in this paper we focus more on the Metadata operation we will consider LDLM which is present at the MDS. In case of a Single Metadata Server deployment a single MDS manages the entire namespace so when a client wants to lookup or create a name in that namespace its the sole owner of the entire namespace. Whereas in case of a Clustered Metadata Server design each directory can be striped over multiple metadata servers, each of which contains a disjoint portion of the namespace. So when a client wants to lookup or create a name in that namespace, it uses a hashing algorithm to determine which metadata server holds the information for that name.

A single MDS server can be a bottleneck. Also the number of RPC calls made in the path name lookup can contribute a significant portion in Lustre File System performance. In case of an environment where many Clients are performing interleaving access and I/O on the same file the time spent in path name lookup can be huge. We have addressed this problem in this paper.

Number of transactions	Transactions per second
1,000	333
5,000	313
10,000	325
20,000	321

Table 1: **Transaction throughput with a fixed file pool size of 1000 files**

Number of files in pool	Number of transactions	Transactions per second
1,000	1,000	333
5,000	5,000	116
10,000	10,000	94
20,000	20,000	79

Table 2: **Transaction throughput with varying file pool**

2.2 Single Lustre Metadata Server (MDS) Bottlenecks

The MDS is currently restricted to a single node, with a fail-over MDS that becomes operational if the primary server becomes nonfunctional. Only one MDS is ever operational at a given time. This limitation poses a potential bottleneck as the number of clients and/or files increase. IOZone [1] is used to measure the sequential file IO throughput, and PostMark [6] is used to measure the scalability of the MDS performance. Since MDS performance is the primary concern of this research, we discuss the PostMark experiment with more details. PostMark is a file system benchmark that performs a lot of metadata intensive operations to measure MDS performance. PostMark first creates a pool of small files (1KB to 10KB), and then starts many sequential transactions on the file pool. Each transaction performs two operations to either read/append a file or create/delete a file. Each of these operations happens with the same probability. The transaction throughput is measured to approximate workloads on an Internet server. Table 1 gives the measured transaction throughput with a fixed file pool size of 1,000 files and different number of transactions on this pool. The transaction throughput remains relatively constant at varied transaction number. Since the cost for MDS to perform an operation does not change at a fixed file number, this result is expected. Table 2, on the other hand, changes the file pool size and measures the corresponding transaction throughput. By comparing the entries in Table 2 with their counterparts in Table 1, it becomes clear that a large file pool results in a lower transaction throughput. The MDS caches the most recently accessed metadata of files (the inode of a file). A client file operation requires the metadata information about that file be returned by MDS. At larger number of files in the pool, a client request is less likely to be serviced from the MDS cache. A cache miss results in the MDS looking up its disk storage to load the inode of requested file, which results in the lower transaction throughput in Table 2.

3 RPC Mechanism in Lustre Filesystem

In the following section we discuss about the RPC mechanism in Lustre Filesystem. Our experiments show that nearly 1500-2000 usecs are spent in RPC on a TCP transport.

3.1 Existing RPC Processing with Lustre

When we consider the RPC processing in Lustre we also talk about the how lock processing works in Lustre [5, 8, 3, 19] and how our modifications can benefit to minimize the number of LOOKUP RPC. Lets consider an example where a client C1 wants to open the file /tmp/lustre/d1/d2/foo.txt to read. In this case /tmp/lustre is our mount point. During the VFS path lookup, Lustre specific lookup routine will be invoked. The first RPC request is lock enqueue with lookup intent. This is sent to MDS for lock on d1. The second RPC request is also lock enqueue with lookup intent and is sent to MDS asking "inodebits" lock for d2. The lock returned is an inodebits lock, and its resources would be represented by the file-id (fid) of d1 and d2. The subtle point to note is, when we request a lock, we generally need a resource id for the lock we are requesting. However in this case, since we do not know the resource id for d1, we actually request a lock on its parent "/", not on the d1 itself. In the intent, we specify it as a lookup intent and the name of the lookup is d1. Then, when the lock is returned, the lock is for d1. This lock is (or can be) different from what the client requested, and the client notices this difference and replaces the old lock requested with the new one returned.

The third RPC request is a lock enqueue with open intent, but it is not asking for lock on foo.txt. That is, you can open and read a file without a lock from MDS since the content is provided by Object Storage Target (OST). OSS/OST also had a LDLM component and, in order to perform I/O on the OSS/OST we request locks from an OST. In other words, what

happens at *open* is that we send a lock request, which means we do ask for a lock from LDLM server. But, in the intent data itself, we might (or not) set a special flag if we are actually interested in receiving the lock back. The intent handler then decides (based on this flag), whether or not to return the lock. If foo.txt exists previously, then its fid, metadata information (such as ownership, group, mode, ctime, atime, mtime, nlink, etc.) and striping information are returned.

If client C1 opens the file with the O_CREAT flag and the file does not exist, the third RPC request will be sent with open and create intent, but still there will be no lock requests. Now on the MDS side, to create a file foo.txt under d2, MDS will request through LDLM for another EX lock on the parent directory. Note that this is a conflicting lock request with the previous CR lock on d2. Under normal circumstances, a fourth RPC request (blocking AST) will go to client C1 or anyone else who may have the conflicting locks, informing the client that someone is requesting a conflicting lock and requesting a lock cancellation. MDS waits until it gets a cancel RPC from client. Only then does the MDS get the EX lock it was asking for earlier and can proceed. If client C1 opens the file with LOV_DELAY flag, MDS creates the file as usual, but there is no striping and no objects are allocated. User will issue an ioctl call and set the stripe information, then the MDS will fill in the EA structure.

3.2 How Do We Improve RPC Processing

In the Client side metadata delegation approach during the creation of the file, we cache some information both at the Client and the MDS. At the MDS side we keep track of which Client have created which files and so if needed the metadata information for those files can be delegated to these Clients in future. Such clients are known as "Delegator Clients". Where as the term "Normal Clients" is used for clients who access the file in later runs to perform some I/O. So Normal clients are diverted to the Delegator Clients by the MDS if the load at MDS increases. Along with the caching we also maintain some book-keeping information at the Clients to keep track of the lock related information, the pathname and the inode information which get for the respective dentry object. So apart from the Dentry Cache maintained by the Operating System we have additional book-keeping information to map the file handle to pathname information.

So if we consider the above example and we want to traverse /tmp/lustre/d1/d2/foo.txt, we will perform similar steps and update the book-keeping information. In the first iteration if the file does not exist it is created. For subsequent access of the same file by different clients, for the first round we will have to go through the entire path name lookup so that we populate our cache and the book-keeping information. So if the same file is accessed in an interleaving manner by the clients who have already traversed the path once we can minimize the additional LOOKUP RPC's as we can service them by making use of the book-keeping information. In case of Basic Lustre for the intermediate component in the path we do a LOOKUP RPC whereas in our approach we can service the request for an intermediate component by making use of the book-keeping information and the local cache maintained at the Client. The size of this book-keeping information will be very small and won't consume much of the memory at the Client side.

4 Design of MDCS

Consider a scenario when the Client 1 tries to open a file say /tmp/lustre/test.txt where /tmp/lustre is the mount point. Figure 2 gives a detailed description of the MDCS design. In Step 1, the Client does a LOOKUP RPC to MDS as discussed in detail in Section 3. In Step 2, the processing is done at the MDS side where the Lock Manager will grant the lock for the resource requested by the Client. A second RPC will be sent from the Client to the MDS with the intent to create or open the file. So at the end of step 2 Client 1 will get the lock, Extended attribute information and other metadata details. Conceptually step 1 and step 2 are similar to what we have in the current Lustre Design but in our approach we modify step 2 slightly. In our approach in step 2 we make an additional check at the MDS to see if this is a first time access to the file. First time access means this will be the first time when the metadata related information for this file will be created on the MDS and neither the metadata caches maintained by the kernel have the metadata related information cached. So if this is a first time access then we keep a data structure to keep track of who owns the file and do some validation whether it is a first time access. So at the end of step 2 we get the needed information from MDS to the Client 1. We have written a Communication module,which is a kernel module which will be used for one sided operations. We have also implemented a hashing functionality in the communication module to speed up the lookup process both

a MDS and Client Side. We make use of the communication module for one sided operation like remote memory read, remote memory write.

So in Step 3 we expose the buffers with the information such as extended attributes, etc that will be useful for clients who will subsequently access the file that was open by Client 1. We call such a client who exposes the needed buffer information as the new owner of the file and use the term "Delegation Client" in this paper. Now when the Client 2 tries to open the same file it performs an RPC in step 4 as we do in step 1. The Client accessing the file we call it as "Normal Client" in the paper does a lookup in the hash table at the MDS side that we updated in step 1 and finds that Client 1 is the owner of the file. So instead of spending additional time at the MDS side we return the needed information to Client 2. Now at the end of step 5, i.e., in step 6 the Client 2 will contact Client 1 and fetch the information which was stored in the buffers exposed by the Delegator Client for this specific file. We use our communication module to speed up this process using a one sided operations in step 6 and step 7. Once Client 2 gets the needed information from the Client 1 it can proceed ahead with its I/O operations. The design can also help in minimizing the request traffic at the MDS end by delegating some load at the client. In our approach since we are maintaining a hash table to do lookup; if we see a matching filename we divert the client asking for the metadata information to the Delegator Client. If the hash table also does not have the needed mapping information then it is the first time that the file is being accessed otherwise we return the cached information to this new client depending on the information we cached for the file when the file was first opened or created by the delegator client. This architecture allows the authoritative copy of each metadata item to reside on different clients, distributing the workload. In addition, the delegation record is much smaller, allowing the MDS cache to be more effective as the number of files accessed increases. This architecture will work well when many clients are accessing many files, as the workload will be distributed amongst clients. However, using a delegate in a scenario where all clients are accessing one file moves the hot-spot from the powerful MDS to a client. In future work we can add a mechanism for the metadata server to determine the popularity of a single file. An access counter will be associated with each inode. Each client request will increase the counter by 1, and its value will decay over time. A high access value will be a sign of a very popular file, and the MDS will recall the delegated inode with high access value. Any following request to that file will be serviced by the MDS directly from its cache without network redirection. On the other hand, a delegate client can voluntarily return all inodes it delegates to the MDS if it thinks it is overburdened. In addition, the MDS must be able to recover from client failure by reclaiming the MDS record from the failed node during recovery. We plan to work on this in our future work.

5 Experimental Results

We have implemented our design into Lustre-1.8.1.1 to minimize the number of RPC calls during a metadata operation. In this section, we conduct experiments to evaluate the metadata operation performance with our proposed design. One node acts as Lustre Metadata Server (MDS), and two nodes are Lustre Object Storage Servers (OSS). Lustre filesystem is mounted in other eight nodes who acts as Lustre client nodes. Each node runs kernel 2.6.18-128.7.1.el5 with Lustre 1.8.1.1. Each node has dual Intel Xeon E5335 CPU (8 cores in total) and 4GB memory. They are interconnected with 1GigE for general purpose networking. In our testing we configure Lustre to use "tcp" transport in different runs.

In order to measure the metadata operations performance such as *open*, we have developed a parallel micro-benchmark. We have extended the basic "fileop" testing tool coming with the IOzone [1] benchmark to support parallel running with multiple processes on many Lustre client nodes. The extended "fileop" tool creates a file tree structure for each process. This tree structure contains X number of Level 1 directories, with each Level 1 directory having Y number of Level 2 directories. The total level of subdirectories can be configured at runtime. Within each of the bottom level directory Z files are created. By varying the size (fan-out) of each layer, we can generate different number of files in a file tree. We have developed an MPI parallel program to start multiple process on multiple nodes. Each process works on a separate directory to create its aforementioned file tree. After that, each process walks through its neighbor's file tree to open each of the file in that sub-tree. This is to simulate the scenario that multiple client processes take turns to access a shared pool of files. After that the wall clock time on all the processes are summarized and the total IOPS for *open* system call is reported.

In order to perform the tests we created some number of files from a specific Client and those files were accessed subsequently by other clients in an interleaving manner. Using the Postmark benchmark we could

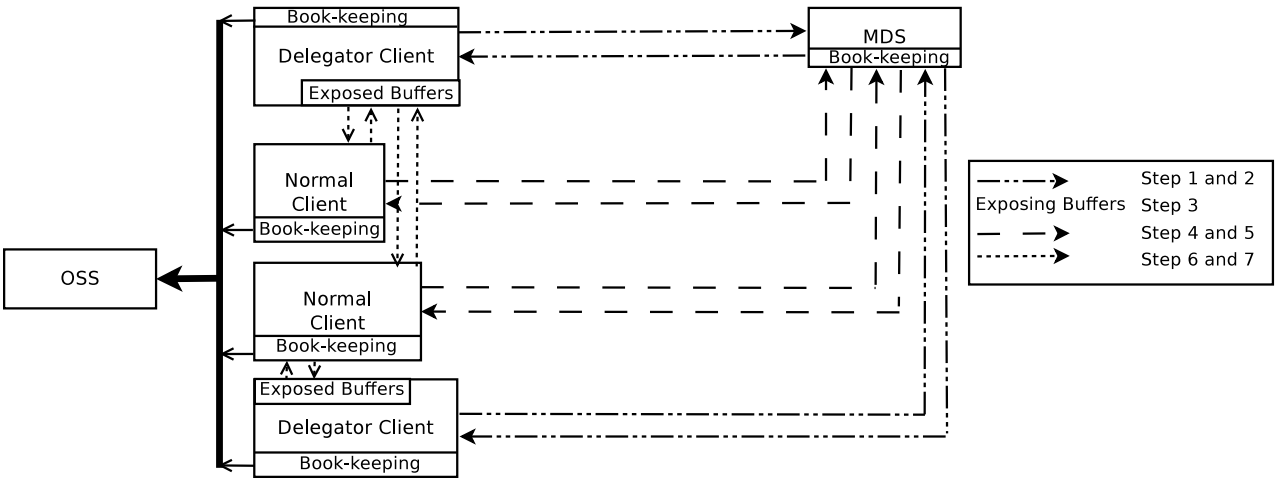

Figure 2: Modified Lustre with Metadata Delegation to Clients

not simulate the kind of the above scenario as in the Postmark benchmark [6] we create some N number of files and as soon as the open/create or read/append operation is complete the file pool is deleted. So we make use of the above mentioned microbenchmark to perform the test and get the experimental results.

In order to see the benefits of the proposed approach in minimizing the RPC we carried out 3 different types of test using our microbenchmark. 1) IOPS in open using our parallel benchmark for different number of client processes. 2) IOPS in open using our parallel benchmark for different number file pool sizes. 3) Time spent in open for varying Path name.

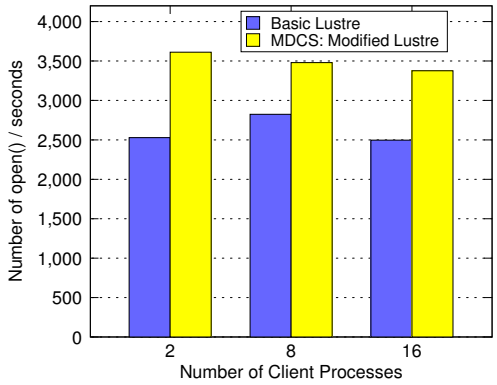

Figure 3: File *open* IOPS, Each Process Accesses 10000 Files

5.1 File Open IOPS: Varying Number of Client Processes

In this test, we first create the aforementioned file tree each containing 10000 files for every client process, then let each process access its neighbor's file tree. Figure 3 shows the aggregated number of IOPS for *open* system call on Lustre filesystem. We vary the number of client processes from 2 to 16 which are evenly distributed on 8 client nodes. With 2 processes, only two client nodes are actually used. With 16 processes, 2 client processes run on each of the 8 client node.

As seen in Figure 3, the modified Lustre with MDCS improves the aggregated IOPS over the basic Lustre significantly. Compared to the basic Lustre, our design reduces the number of RPC calls in metadata operation path, therefore helps improve the overall performance. With two client processes, the new approach (MDCS) promotes file *open* IOPS from 2528 per seconds to 3612 per seconds, an improvement of 43%. The improvements with 8/16 processes are 23% and 25%, respectively. As more client processes access the shared Lustre filesystem concurrently, the MDS sees higher degree of contention for request processing. As a result, the total file *open* IOPS slightly drops for MDCS. With basic Lustre, on the other hand, the Metadata Server has the potential to handle 8 concurrent client processes, given a slightly higher file *open* IOPS with 8 concurrent client processes. When 16 processes are used, however, MDS's performance drops due to the high contention, similar to what we see with MDCS approach.

5.2 File Open IOPS: Varying File Pool Size

In this test we carry out similar basic steps as mentioned in 5.1. But in this test we vary the number of

files in each file tree per process, while using the same 16 client processes. We wanted to understand the significance of this factor while considering the performance aspect into consideration. Figure 4 shows the experimental results. It clearly demonstrates the benefits of our MDCS design. With the capability to minimize RPC calls to the centralized MDS, we can achieve 35% better IOPS in terms of file *open*.

We also observe that, by varying the file pool size for a constant number of processes we don't see a huge deviation in the number of IOPS for the *open*. We speculate this is caused by the fact the file pool size used in our test isn't big enough to stress the memory on MDS, such that most of the files' metadata information are stored in MDS's memory cache. As a result, the aggregated metadata operation throughput remains constant with different file pool size. In our future study we will experiment with larger file pool to push the memory limit of MDS.

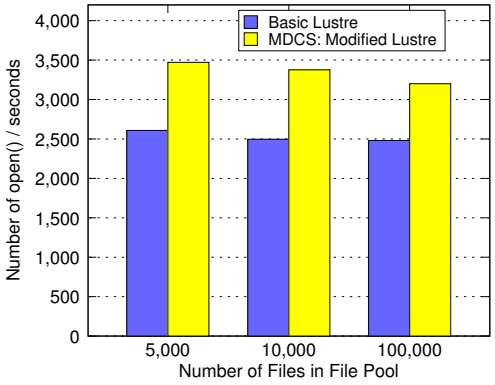

Figure 4: File *open* IOPS, Using 16 Client Processes

5.3 File Open IOPS: Varying File path Depth

In this test we want to measure the performance benefit of the new MDCS approach when accessing files with different File path depth, i.e., number of components in the file path. We start with creating a file tree for each of the client process containing 10,000 files, with file path depth to be 3 or 4. After that each process begins to access files within its neighbor process's file tree.

Figure 5 compares the time spent to open one file, either with basic Lustre or with the MDCS modified Lustre filesystem. First of all, it shows MDCS can significantly reduce the time cost to open one file by up to 33%. We also observe that pathname component factor has a significant importance in the total cost of a metadata operation. Each file path component has to be resolved using one RPC to the MDS, hence the deeper the file path leads to a longer processing time.

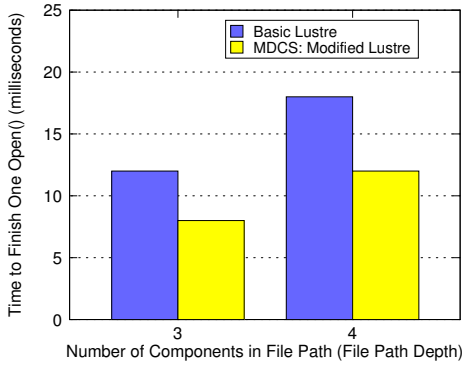

Figure 5: Time to Finish *open*, Using 16 Processes Each Accessing 10000 Files

6 Related Work

The Lustre Design with a Single MDS and a Clustered Metadata [5, 8, 3, 4] are the currently used designs of the Lustre File System. The authors in [20, 17] talk about the evaluation of the performance model of Lustre File System. The authors in [14, 15, 11] explore the interaction of MPI-IO and Lustre File System. The scenario which we focus in this paper i.e. the scenario where the pathname lookup RPC can consume a lot of factor in actual performance numbers can be common in MPI kind of environments. In this paper we propose an approach called as Client Side Metadata Delegation in which we propose a scheme to minimize the number of LOOKUP RPC calls for the interacting clients and hence reduce the request traffic on the MDS.

There is an effort going on in the Linux Community [2] regarding a new system call named name_to_handle to allow a user process to get a file handle (i.e. binary blob returned from a new name_to_handle syscall) from the kernel for a given pathname, and then later use that file handle in another process to open a file descriptor without retraversing the path.

While this wouldn't eliminate the actual MDS open RPC but it could avoid the path traversal from each client saving a considerable amount of time in the pathname lookup. We try to solve the problem in the Lustre Design itself by MDCS.

7 Conclusion and Future Work

In this paper we propose a new approach called as Client Side Metadata Delegation to minimize the

number of RPC and hence to minimize the traffic and load at MDS. In case of Single MDS, when a file is subsequently accessed in an interleaving manner by various client, by minimizing the lookup RPC's can definitely help to improve the performance. This approach proposed in the paper is also applicable to Clustered Metadata design.

We evaluated the potential benefits of proposed approach by using a parallel I/O microbenchmark. Approximately 1500-2000 μsecs are spent in an RPC on a TCP transport.This factor may vary depending on the fan-out of the directory structure and the path elements in the path component. We used varying fan-out and pathname length to perform metadata operations since we are more concerned about the load the the MDS. After analyzing the experimental results we came to the conclusion that the amount of time saved in RPC is approximately equal to (number of path elements) * (number of clients accessing the file).

As part of the future work, we intend to conduct the experiments in larger scale to understand the memory pressure at the MDS by taking into account the Client Side Metadata Delegation approach. We also want to create a Distributed lock management API at the Client side to synchronize the metadata delegated to the Delegator Client and the local copies of the metadata which the Normal Clients have with them.

References

[1] IOZONE. http://www.iozone.org/.

[2] Lustre Devel Queries. http://lists.lustre.org/pipermail/lustre-devel/2011-January/003703.html.

[3] Lustre File System,High-Performance Storage Architecture and Scalable Cluster File System. http://www.raidinc.com/pdf/whitepapers/.

[4] Lustre Networking :High-Performance Features and Flexible Support for a Wide Array of Networks. https://www.sun.com/offers/details/lustre-networking.xml.

[5] Lustre Wiki Page. http://wiki.lustre.org/.

[6] POSTMARK. http://www.shub-internet.org/brad/FreeBSD/postmark.html.

[7] PVFS2. http://www.pvfs.org/.

[8] Sun Microsystems, Inc.,Lustre 1.8 Operations Manual. http://wiki.lustre.org/manual/.

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2:39–59, February 1984.

[10] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Phillip Dickens and Jeremy Logan. Towards a high performance implementation of mpi-io on the lustre file system. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, OTM '08, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119 – 123, 2005.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[14] W.-K. Liao, A. Ching, K. Coloma, Alok Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for mpi-io. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, 2007.

[15] J. Logan and P. Dickens. Towards an understanding of the performance of mpi-io in lustre file systems. In *Cluster Computing, 2008 IEEE International Conference on*, pages 330 –335, 2008.

[16] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8):84 – 90, 2003.

[17] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1 –10, 2007.

[18] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[19] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Tiezhu Zhao, V. March, Shoubin Dong, and S. See. Evaluation of a performance model of lustre file system. In *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, pages 191 –196, 2010.