IMCa: A High Performance Caching Front-end for GlusterFS on InfiniBand *

Ranjit Noronha and Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{*noronha,panda*}*@cse.ohio-state.edu*

Abstract

With the rapid advances in computing technology, there is an explosion in media that needs to collected, cataloged, stored and accessed. With the speed of disks not keeping pace with the improvements in processor and network speed, the ability of network file systems to provide data to demanding applications at an appropriate rate is diminishing. In this paper, we propose to enhance the performance of network file systems by providing an InterMediate bank of Cache servers between the client and server called (IMCa). Whenever possible, file system operations from the client are serviced from the cache bank. We evaluate IMCa with a number of different benchmarks. The results of these experiments demonstrate that the intermediate cache architecture can reduce the latency of certain operations by upto 82% over the native implementation and upto 86% compared with the Lustre file system. In addition, we also see an improvement in the performance of data transfer operations in most cases and for most scenarios. Finally, the caching hierarchy helps us to achieve better scalability of file system operations.

Keywords: *InfiniBand, System Area Networks, Clusters*

1 Introduction

With the dawn of the internet age, the rapid growth of multi-media and other traffic, there has been a dramatic increase in the amount of data that needs to be stored and accessed. In addition, commercial and scientific applications such as data-mining and nuclear simulations generate and parse vast amounts of data during their runs. To meet the demand for access to this data, single server file systems such as NFS [9] and GlusterFS [1] and parallel file systems such as Lustre [10] over high-bandwidth interconnects like InfiniBand with high-performance storage disks at the storage servers have become common-place. However, even with these configurations, the performance of the file system under a variety of different workloads is limited by the access latency to the disk. With a large number of requests to non-contiguous locations of the disk, the ability of the file system to cope with these types of requests is severely limited. In addition, parallel striping of parallel data provides limited benefit in environments with a lot of small files.

*This research is supported in part by Department Energy's grant #DE-FC02-01ER25506, National Science Foundation grants #CNS-0403342 and #CCF-0702675; grants from Intel, Mellanox, and Sun Microsystems; and equipment donations from Intel, Mellanox, and Sun Microsystems.

To reduce the load on the disk and enhance the performance of the file system, several different types of caching strategies and alternatives have been proposed [5, 4]. Generally, in most file systems, a cache exists at the server side. It might be part of the distributed file system, such as with Lustre [10], or it might reside in the underlying file system such as with NFS. The server side cache will generally contain the latest data. The server side cache may be used to reduce the number of requests hitting the disk, and also provide enhancements when there is a fair amount of read/write data sharing. The server side cache is generally limited in size and shared by a large number of I/O threads. In addition, the limited size of the cache in-concert with policies like LRU can reduce the performance of the server side cache.

In addition to a server side cache, file system protocols like NFS [9] and Lustre [10] also provide a client side cache. A client side cache may provide a large benefit in terms of performance when most of the data is accessed locally, such as in the case of a user home directory. However, client side caches introduce cache coherency issues when there is sharing of data between multiple clients. NFS does not offer strict cache coherency and uses coarse timeouts to deal with the issue. Lustre [10] on the other hand uses locking with the metadata server acting as a lock manger to implement client cache coherency. Writes are flushed before locks are released. With a large number of clients, the overhead of maintaining locks and keeping the client caches coherent increases. GlusterFS [1] does not provide a client side cache in the default configuration.

In this paper, we propose, design and evaluate an InterMediate Caching architecture (IMCa) between the client and the server for the GlusterFS [1] file system. We maintain a bank of independent cache nodes with a large capacity. The file system is responsible for storing information from a variety of different operations in the cache. Keeping the information in the cache bank up-to-date is achieved through a number of different hooks at the client and the server. Through these hooks, the client attempts to fetch the information for different operations from the cache, before trying to get it from the back-end file server.

We expect multiple benefits from using this architecture. First, the file system clients can expect to retain the benefits of a client side cache with a small penalty bounded by network round-trip latency. With the advent of low latency, high-performance networks like InfiniBand which offer low latency messaging, the penalty associated with this is likely

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.84

to be low. Second, since the number of caches is small in comparison to the number of clients, and these caches are lockless, keeping the caches coherent is considerably cheaper. Finally, we expect to reap the benefits of a client cache without the associated scalability and coherency issues.

Our preliminary evaluations shows that we can improve the performance of file system operations such as stat by up to 82% over the native design and upto 86% over a filesystem like Lustre. In addition, we also show that the intermediate cache can improve the performance of data transfer operations with both single and multiple clients. Finally, in environments with read/write sharing of data, we can see an overall improvement in file system performance. Finally, IMCa helps us to achieve better scalability of file system operations.

The rest of this paper is organized as follows. Section 2 describes the background work. After that, Section 3 tries to motivate the need for a bank of caches. In Section 4 we discuss the design issues. Following that, Section 5 presents the evaluation. Section 6 looks at related work. Finally, conclusions and future work are presented in Section 7.

2 Background

In this section, we discuss the file system GlusterFS and the dynamic web-content caching daemon MemCached.

2.1 Introduction to GlusterFS

GlusterFS [1] is a clustered file-system for scaling the storage capacity of many servers to several peta-bytes. It aggregates various storage servers or bricks over an interconnect such as InfiniBand or TCP/IP into one large parallel network file system. GlusterFS in its default configuration does not stripe the data, but instead distributes the namespace across all the servers. Internally, GlusterFS is based on the concept of translators. Translators may be applied at either the client or the server. Translators exist for Read Ahead and Write Behind. In terms of design, a small portion of GlusterFS is in the kernel and the remaining portion is in userspace. The calls are translated from the kernel VFS to the userspace daemon through the Filesystem in UserSpace (FUSE).

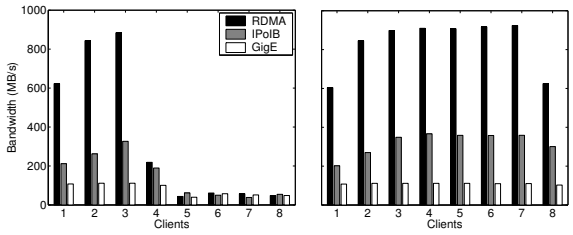
2.2 Introduction to MemCached

Memcached is an objects based caching system [3] developed by Danga Interactive for LiveJournal.com. It is traditionally used to enhance the performance of database application or websites with dynamic content that are heavily loaded. Memcached is usually run as a daemon on spare nodes. Memcached listens for requests on a user specified port. The amount of memory used for caching is specified at startup. Internally, memcached implements Least Recently Used (LRU) as the cache replacement algorithm. Memcached uses a lazy expiration algorithm; i.e. objects are evicted when the cache is full and a request is made to add an object to the cache, or a request to fetch a data element from the cache is made and the time for the object in the cache has expired. Memory management is based on slab cache allocation to reduce excessive fragmentation. Memcached currently limits the maximum size of the object to be stored to 1MB and the maximum length of the key to 256 bytes. The Memcache daemon may be accessed through TCP/IP connections. Clients usually change data elements in memcached through a *T(key, data)* tuple. The API consists of the functions set, replace, delete, prepend and append. A number of libraries are available for accessing memcached daemons; one of them libmemcache is a C based library [2].

3 Motivation

We now consider the motivation for using intermediate caching architecture in a file system. We look at some common problems in file system design that could potentially be solved through the use of a caching layer.

Single Server Bandwidth Drop With Multiple Clients. Protocols like NFS/RDMA attempts to offer the improved bandwidth of networks like InfiniBand to NFS. However, NFS servers usually store most of the data on the disk. The server is constrained by the ability of the disk to match the bandwidth of the network. Since the disk is usually much slower than the network, the benefit from using NFS/RDMA is reduced. The effect of this is shown in Fig. 1(b) and Fig. 1(a), which show the multi-client IOzone Read throughput with different transports, namely NFS/RDMA (RDMA), NFS/TCP on InfiniBand (IPoIB) and finally, NFS/TCP on Gigabit ethernet (GigE). In Fig. 1(a), 4GB server memory is used; in Fig. 1(b), 8GB server memory is used. The bandwidth available to the clients seems to be related to the amount of memory on the server and falls off as the server runs out of memory and is forced to fetch data from the disk.



(a) 4GB (b) 8GB

Fig. 1. Multiple clients IOzone Read Bandwidth with NFS/RDMA [9]

Parallel I/O Bandwidth From Multiple Servers. Parallel I/O attempts to use the aggregate bandwidth of multiple servers. Since the back-end server ultimately uses real disks, the benefits of parallel I/O bandwidth are ultimately mitigated especially for multiple streams that access data spread on different portions of the disk causing increased disk seeking, reducing performance.

Performance For Small Files. Delivering good performance for small files is generally difficult. In data-center environments a large number of small files are used [7]. Data striping techniques generally used in parallel file system are of limited use for small files. Storing files on multiple independent servers can help reduce contention for

small files, but still exposes these files to the limits of the disk on these servers.

Cache Coherency Problems. In file system environments, a client side cache usually provide best performance. Client caches may be coherent such as with Lustre [10] or non-coherent, such as with NFS [9]. Non-coherent client side caches are more scalable but have limited use in environments with read/write sharing. Coherent client side cache may be used in environments with sufficient read/write sharing. However, they have limited scalability.

Server load problems. Reducing the load on the server is generally crucial to improving the scalability of file system protocols. RDMA is generally proposed as a communication offload technique to reduce the impact of copying in protocols like TCP/IP. However, RDMA cannot eliminate other copying overheads such as those across the VFS layer and other file system related overheads. Using an intermediate cache layer may help mitigate the effect of some of these problems. We will now look at the design and implementation of a layer of caching nodes.

4 Design of a Cache for File Systems

In this section, we consider the design of the Intermediate memory caching (IMCa) architecture for the GlusterFS [1] file system. First, we look at the overall block level architecture of IMCa in Section 4.1. Following that, we look at the potential non-data file system operations that could be optimized in Section 4.2. In Section 4.3, we look at the potential optimizations for data operations. Finally, we discuss some of the potential advantages and disadvantages of IMCa in Section 4.4.

4.1 Overall Architecture of Intermediate Memory Caching (IMCa) Layer

The architecture of IMCa is shown in Fig. 2. The architecture consists of three components: CMCache (Client Memory Cache), MemCached (MCD) array and SMCache (Server Memory Cache). The first component CMCache (Client Memory Cache) is located at the GlusterFS client.

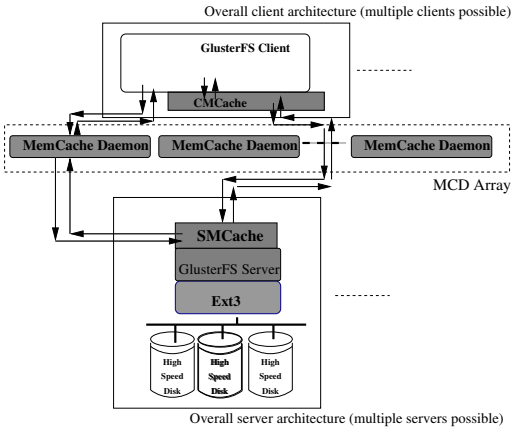


Fig. 2. Overall Architecture of the Intermediate Memory Caching (IMCa) Layer

Client Memory Cache (CMCache): This is responsible for intercepting file system operations at the client. It is implemented as a translator on the GlusterFS client as discussed in Section 2. Once these operations are intercepted CMCache determines whether these requests have any interaction with the caching layer or not. If there is no interaction, CMCache will propagate the request to the server. Interactions are generally in two forms. In the first form, it may be possible to process the request from the client directly by contacting the MCDs. In this case, CMCache will contact the MCDs and attempt to directly return the results for the requests. CMCache communicates with the MCDs through TCP/IP.

MemCached MCD Array (MCD): This consists of an array of MemCached daemons running on nodes usually set aside primarily for IMCa. The daemons may reside on nodes that have other functions, since MCDs tends to use limited CPU cycles. To obtain maximum benefit from using IMCa, the nodes should be able to provide a sufficient amount of memory to the daemons while they are running.

Server Memory Cache (SMCache): This is the final component of IMCa. It is located on the GlusterFS server. SMCache is implemented as a translator at the GlusterFS server. SMCache is divided into two parts. The first part of SMCache intercepts the calls coming from GlusterFS clients. Depending on the type of operation from the GlusterFS client, it may either pass the operation directly to the underlying file system, or perform certain transformations on it before passing it to the underlying file system. The GlusterFS file system uses the asynchronous model of processing requests as discussed in Section 2. Initially, requests are issued to the file system and later when they complete, a callback handler is called that processes these responses and returns the results back to the client. The second part of SMCache maintains hooks in the callback handler. These hooks allow SMCache to intercept the results of different operations and send them to MCDs if needed. SMCache communicates with the MCDs using TCP/IP.

4.2 Design for Management File System Operations in IMCa

We now consider some of the design trade-offs for different management file system operations.

Stat operations: These are included in POSIX semantics. Stat applies to both files and directories. Stat generally contains information about the file size, create and modify times, in addition to other information and statistics about the file. Stat operations are a popular way of determining updates to a particular file. For example, in a producer-consumer type of application, a producer will write or append to a file. A consumer may look at the modification time on the file to determine if an update has become available. This avoids the need and cost for explicit synchronization primitives such as locks. This approach is used in a number of web and database applications [7]. Since the data structures for the stat operations are generally stored on the disk, stat operations usually have considerable latency. It is natural to consider stat functions for cache based functionality. We have designed a cache based functionality for stat. At open, MCD is updated with the contents of the stat structure from the file by SMCache. The key used to lo-

cate a MCD consists of the absolute pathname of the file, with the string *:stat* appended to it. SMCache uses the default CRC32 hashing function in libmemcache [2] to locate the appropriate MCD. For every read and write operation, the stat structure in the MCD is replaced with the most recent value of stat by SMCache. CMCache then intercepts stat operations, attempts to fetch the stat information from the MCD if available, and return it to the client. If there is a miss, which might happen if the stat entry was evicted from the MCD for example, the stat request propagates to the server.

Create operations: These usually require allocation of resources on the disk. There is not much potential for cache based optimizations. *Create* operations are directly forwarded from the client to the server without any processing.

Delete operations: These operations usually require removal of items from the disk. The potential for optimizations with *delete* operations is limited. Delete operations are forwarded by the client to the server without any interception. When *delete* operations are encountered, we remove the data elements from the cache to avoid false positives for requests from clients.

4.3 Data Transfer Operations

There are two types of file system operations that generally transfer data; i.e. *Read* and *Write*. To implement *Read* and *Write* with IMCa, CMCache intercepts the *Read* and *Write* operations at the client. Before we discuss the protocols for these operations, we look at the issue of cache blocking for file system operations.

4.3.1 Need for Blocks in IMCa

Most modern disk based file systems store data as blocks [6]. Parallel file systems also tend to stripe large files across a number of data servers using a particular stripe width. Generally, the larger the block size, the better bandwidth utilization from the disk and network subsystems. Smaller block sizes on the other hand tend to favor lower latency, but also tend to introduce more fragmentation. IMCa uses a fixed block size to store file system data in the cache. Since IMCa is designed as a generic caching layer and should provide good performance for a variety of different file sizes and workloads; the block size should be set appropriately keeping these limits in mind. It should be kept small enough so that small files may be stored more efficiently. It should also be kept large enough to avoid excessive fragmentation and reasonable network bandwidth utilization. MemCached [3] has a maximum upper limit of 1MB for stored data elements as discussed in Section 2. This places a natural upper bound on the size of data that may be stored in the cache. Depending on the blocksize, IMCa may need to fetch or write additional blocks from/to the MCDs above and beyond what is requested. This happens if the beginning or end of the requested data element is not aligned with the boundary defined by the blocksize. This is shown in Fig. 3. As a result, data access/update from/to the MCDs become more expensive. This is discussed further in Section 4.3.2.

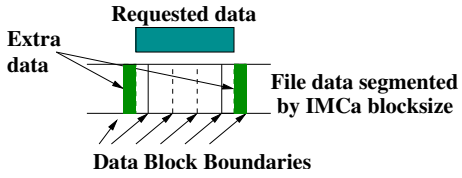


Fig. 3. Example of blocks requiring additional data transfers in IMCa

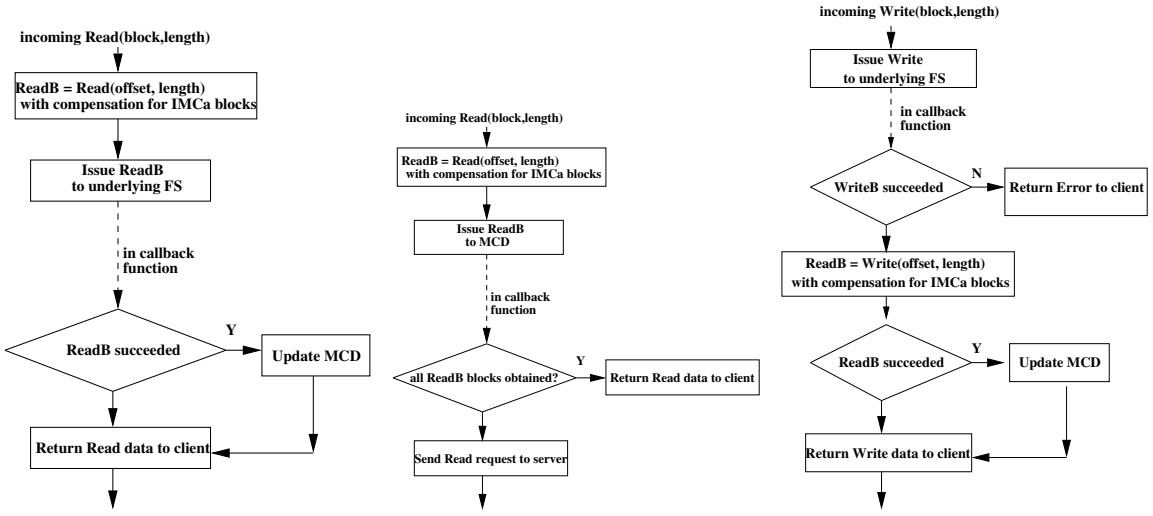
4.3.2 Design for Data Transfer Operations:

We now look at the protocols for Read and Write data transfer operations in IMCa. We also consider the supporting functionality for data transfer operations such as Open and Close.

Open: On the open, in CMCache, the absolute path of the file and the file descriptor is stored in a database, so that this information may be accessed at a later point. At the server, the MCDs are purged of any data relating to the file when the *Open* operation is received.

Read: The algorithm for Read requests in CMCache is shown in Fig. 4(b). On a *Read* operation, CMCache appends the absolute path of the file (which was stored during the Open) with the offset in the file to generate a key. Since IMCa is based on a static block size; the size of the Read data requested from the MCD may be equal to or greater than the current Read request size. CMCache will generate keys that consist of the absolute pathname for the file, that was stored during the open and the offsets from the Read request, taking into account the IMCa blocksize. CMCache uses the keys to access the MCDs and fetch the blocks. If there is a miss for any one of the keys, CMCache will forward the Read request to the GlusterFS server. The cost of a miss is more expensive in the case of IMCa, since it includes one or more round-trips to the MCD, before determining that there might be a miss. The SMCache Read algorithm is shown in Fig. 4(a). Because of the IMCa block size, the *Read* operation may potentially require the server to read additional data from the underlying file system. Once the Read operation returns from the filesystem, the server will append the full file path name with the block offset and update the MCDs with the data. The server may need to send several blocks to the MCDs servers. Using an additional thread to update the MCDs at the server may potentially reduce the cost of Reads at the server.

Write: Write operations are persistent. This means that the Write operations must propagate to the server where they need to be written to the filesystem. CMCache does not intercept *Write* operation. At the server, the Write operation is issued to the file system as shown in Fig. 4(c). When the write operation completes, Read(s) are issued to the underlying file system by SMCache that cover the Write area, accounting for the IMCa blocksize. When the data is available, the Read(s) are sent to the MCDs. Since there may be multiple overlapping *Writes* to a particular record and because of the IMCa requirement of a fixed block-size, neither CMCache nor SMCache can directly send the *Write* data to the MCDs. Write latency may be potentially increased by the additional update of the MCDs at the server. Using an additional thread as with Reads can reduce the cost of this update.



(a) Read (SMCache) (b) Read (CMCache) (c) Write (SMCache)

Fig. 4. Logical Flow for Read and Write operations in IMCa

Close: Closes propagate from the client directly to the server without any interception. When the close operation is intercepted by SMCache, it will attempt to discard the data for the file from the MCDs.

4.4 Potential Advantages/Disadvantages of IMCa

In this section, we discuss the potential advantages and disadvantages of IMCa.

Fewer Requests Hit the Server: The data server is generally a point of contention for different requests. In addition to communication contention, there may be considerable contention for the disk. IMCa may help reduce both these contentions at the server.

Latency for Requests Read From the Cache is Lower: With considerable percentage of Read sharing as well as Read/Write sharing patterns, a large number of requests could potentially be fielded directly from the MCDs. This might help reduce the latency for these patterns, in addition to reducing the load on the server.

MCDs are self-managing: Each cache in the MCD implements LRU. As the caches fill up, unused data will automatically be purged from the MCDs. There is no need to manage the cache by the client or the server. This reduces the overhead of IMCa. Additional caching nodes can be easily added. IMCa can transparently account for failures in MCDs.

Failures in MCDs do not impact correctness: Writes are always persistent in IMCa and are written successfully to the server filesystem before updating the MCDs. Irrespective of node failures in the MCDs, correctness is not impacted.

Additional Nodes Elements Needed Especially For Caching: MCDs needs an array of nodes on which to run the daemons. These nodes might be used for other purposes such as storing file system data or running web services.

Cold Misses Are Expensive: Reads on the client require one or more accesses to the MCDs depending on the blocksize and the requested Read size. If any of these accesses results in a miss, the Read needs to be propagated to the server. As a result, misses are more expensive than in a regular file system.

Additional Blocks/Data Transfer Needed: In IMCa data is stored in blocksizes to act as a tradeoff between bandwidth, latency, utilization and fragmentation. If the block size is set too large, small Read requests will be penalized, requiring additional data to be transferred from the MCDs. If the block size is set too small, large requests might require multiple trips to the MCDs to fetch the data.

Overhead and Delayed Updates: IMCa hooks into both Read/Write functions at the server through SMCache. Read/Write data from the server needs to be fed to the MCDs before it is returned to the client in non-threaded mode. This may result in additional overhead at the server and updates from the MDCs being delayed.

5 Performance Evaluation

In this section, we attempt to characterize the performance of IMCa in terms of latency and throughput of different operations. First, we look at the experimental setup.

5.1 Experimental Setup

We use a 64 node cluster connected with InfiniBand DDR HCAs. Each node is an 8-core Intel Clover based system with 8GB of memory. The GlusterFS server runs on a node with a configuration identical to that specified above; it is also equipped with a RAID array of 8-HighPoint Disks on which all files used in the experiment reside. IP over InfiniBand (IPoIB) with Reliable Connection (RC) is used as the communication transport between the GlusterFS server and client; as well as between the components of IMCa namely SMCache, CMCache and the MCD array. The MCDs run on independent nodes and are allowed to use upto 6GB of main memory. Unless explicitly mentioned, SMCache and CMCache use a CRC32 hashing function for storing and locating data blocks on the MCDs. For comparison, we also use the default configuration of Lustre 1.6.4.3 with a TCP

transport over IPoIB. The Lustre metadata server runs on a node separate from the data servers (DS).

5.2 Performance of Stat With the Cache

We look at the performance of the *stat* operation with IMCa as discussed in Section 4.2.

Stat Benchmark: The benchmark used to measure the performance of stat consists of two stages. In the first stage (untimed), a set of 262144 files is created. In the second stage (timed) of the benchmark, each of the nodes tries to perform a stat operation on each of the 262144 files. The total time required to complete all 262144 stats is collected from each of the nodes and the maximum time among all of them is reported.

Performance With One MCD: The results from running this benchmark is shown in Fig. 5. Along the x-axis the number of nodes is varied. The y-axis shows the time in seconds. Legend *NoCache* corresponds to GlusterFS in the default configuration (no client side cache). Legend *MCD (x)* corresponds to GlusterFS with x MemCached daemons running. From Fig. 5, we can see that without the cache, the time required to complete the stat operations increases at a much faster rate than with the cache nodes. With a single MCD, the time required to complete the stat operations increases at a much slower rate. At 64 clients, with 1 MCD, there is an 82% reduction in the time required to complete the *stat* operations as compared to without the cache. GlusterFS with a single MCD outperforms Lustre with 4 DSs by 56% at 64 clients.

Performance With Multiple MCDs: With an increasing number of MCDs, there is a reduction in the time needed to complete the *stat* operations. However, with an increasing number of MCDs, there is a diminishing improvement in performance. For example, at 64 nodes, there is only a 23% reduction in time to complete the stat operation from 4 to 6 MCDs. The statistics from the MCDs show that the miss rate with increasing MCDs beyond 2 is zero. This seems to suggest that 2 MCDs provide adequate amount of cache memory to completely contain the stat data of all the files from the workload. There is little stress on the MCDs memory sub-system beyond two MCDs. The overhead of the communication protocol TCP/IP is alleviated to some extent by going beyond two MCDs. Using four and six MCDs provide some benefit as may be seen from Fig. 5. At 64 nodes, using GlusterFS with 6 MCDs, the time required to complete the *stat* operation is 86% lower than Lustre with 4 DSs.

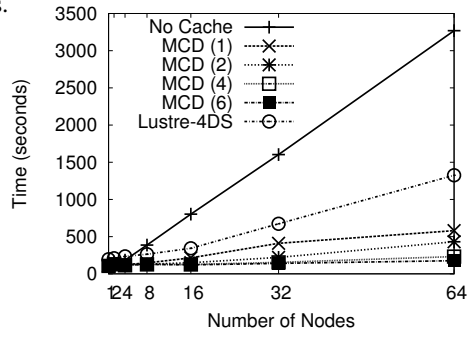


Fig. 5. Stat latency with multiple clients and 1 MCD

5.3 Latency: Single Client

In this experiment, we measure the latency of performing read and write operations.

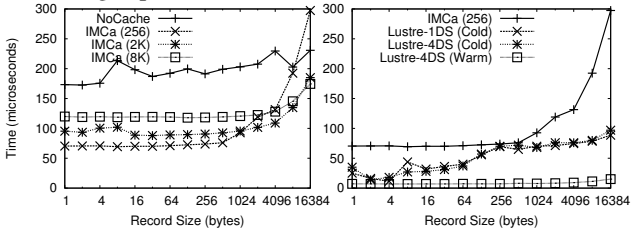
Latency Benchmark: In the first part of the experiment, data is written to the file in a sequential manner. For a given record size *r*, 1024 records of record size *r* are written sequentially to the file. The *Write* time for that record size is measured as the average time of the 1024 operations. We measure the Write time of record sizes from 1 byte to a maximum record size in multiples of 2. In the second stage of the benchmark, we go back to the beginning of the file and perform the same operations for *Read* operations, varying the record size from 1 byte to the maximum record size, with the time for the Read being averaged over 1024 records for each given record size.

Read Latency with different IMCa block sizes: The results from the latency benchmark for *Read* is shown in Fig. 6(a) and Fig. 6(b). For IMCa, we used block sizes of 256 bytes, 2K and 8K bytes. For the Read latency shown in Fig. 6(a), for a record size of 1 byte, there is a reduction of upto 45% in latency using one MCD over using NoCache, with a block size of 2K, and a 31% reduction in latency with an 8K IMCa block size. With an IMCa block size of 256, the reduction in Read latency increases to 59%. As discussed in Section 4, even for a Read operation of 1 byte, the client needs to fetch a complete block of data from the MCDs. So, we must fetch data in multiples of the minimum record size of IMCa. Smaller block sizes help reduce the latency of smaller Reads, but degrade the performance of larger Reads, since CMCache must make multiple trips to the MCDs. This may be seen in Fig. 6(a), where beyond a record size of 8K, NoCache has lower latency than IMCa with a block size of 256 and has the lowest latency overall as the record size is further increased (Fig. 6(b)). Since no Read at the client results in a miss from the MCDs, no read requests propagate to the server. We use a block size of 2K for the remaining experiments.

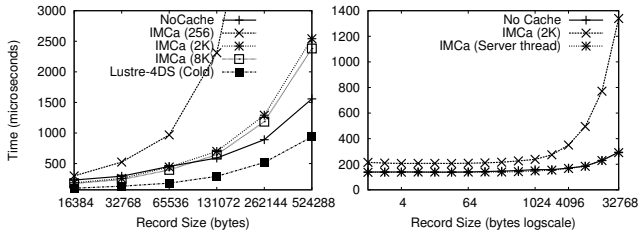
Comparison with Lustre: We use one and four data servers with Lustre, denoted by 1DS and 4DS respectively. Also, we use two different configurations for Lustre, warm cache (Warm) and cold cache (Cold). For the warm cache case, the Write phase of the benchmark is followed by the Read phase of the benchmark without any intermediate step. For the cold cache case, after the Write phase of the benchmark, the Lustre client file system is unmounted and then remounted. This evicts any data from the client cache. Clearly, the warm cache case denoted by *Lustre-4DS (Warm)* provides the lowest Read latency in all cases (Fig. 6(a)), since Reads are primarily satisfied from the local client cache (results for larger record sizes with a warm cache are not shown). The cold cache forces the client to fetch the file from the data servers. So, *Lustre-1DS (Cold)* and *Lustre-4DS (Cold)* are closer to IMCa in terms of performance. We discuss these results further in the technical report version of this paper [8].

Write Latency: The Write latency is shown in Fig. 6(c) with an IMCa block size of 2K. *Write* introduces an additional Read operation in the critical path at the server (Section 4). Correspondingly, Write latency with IMCa is worse than the NoCache case. By offloading the additional Read to a separate thread, the additional latency of the Read may

be removed from the critical path and the Write latency can be reduced to the same value as without the cache. IMCa provides little benefit for Write operations because of the need for Writes to be persistent (Section 4.3.2). Correspondingly, we do not present the results for Write for the remaining experiments.



(a) Read (Small)

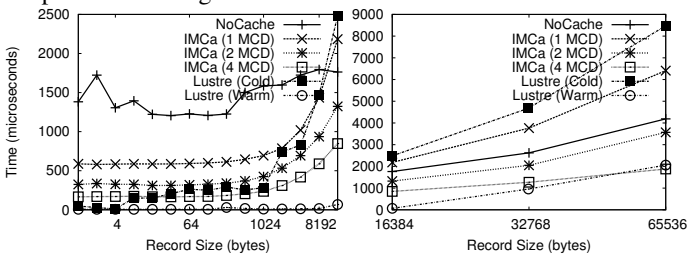


(b) Read (Medium) (c) Write

Fig. 6. Read and Write Latency Numbers With One Client and 1 MCD.

5.4 Latency: Multiple Clients

The multi-client latency tests starts with a barrier among all the processes. Once the processes are released from this barrier, each process performs the latency test (with separate files), described in Section 5.3. The Write and Read latency components as well as each record size for Read and Write is separated by a barrier. The latency for a particular record size is the average of the times reported by each process for the given record size.

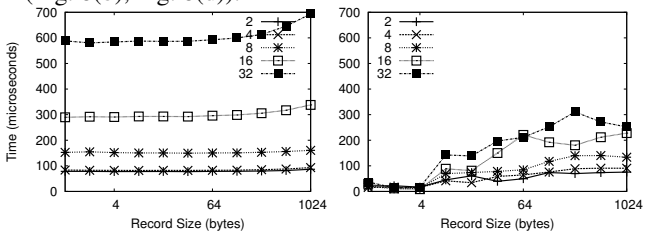


(a) Read (Small) (b) Read (Medium)

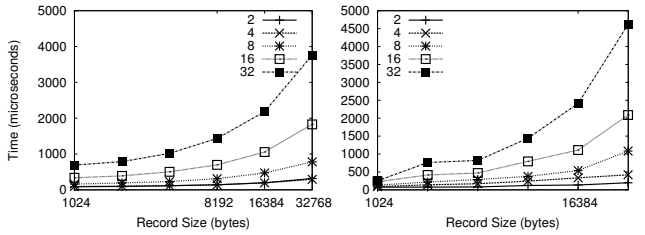
Fig. 7. Read latency with 32 clients and varying number of MCDs. 4 DSs are used for Lustre.

We present the numbers for the Read latency with 32 clients each running the latency benchmark, while the MCDs are being varied. These latency numbers are shown in Fig. 7(a) (Small Record sizes) and Fig. 7(b) (Medium Record Sizes). From the figure, we can see that there is reduction of 82% in the latency when four MCDs are introduced over the NoCache case for a 1 byte Read. Clearly, IMCa provides additional benefit in the case of multiple clients as compared to the single client case. In addition, with 32 clients, and a single MCD, statistics taken from the MCDs show that there are an increasing number of MCD capacity misses. These capacity misses are reduced by increasing the number of MCDs. The trend of increasing capacity misses may be seen more clearly while varying the clients and using a single MCD. These Read latency number are shown in Fig. 8(a) and Fig. 8(c). The Read latency at 32 clients is higher than with one client and increases with increase in record size.

We also compare with Lustre at 32 clients (Fig. 7(a), 7(b)). With a cold cache, for small Reads less than 32 bytes, *Lustre (Cold)* has lower latency than *IMCa (4MCD)*. After 32 bytes, *IMCa (4 MCD)* delivers lower latency than *Lustre (Cold)*. IMCa with 1 and 2 MCDs also provide lower latency than Lustre beyond 8K and 2K respectively. Finally, *Lustre (Warm)* again produces the lowest latency overall. However, the latency for *IMCa (4 MCDs)* increases at a slower rate with increasing record size and at 64K, IMCa (4 MCDs) has lower latency than *Lustre (Warm)*. Similar trends can also be seen with varying number of clients (Fig. 8(b), Fig. 8(d)).



(a) Read (Small)-IMCa (b) Read (Small)-Lustre (4DS)



(c) Read (Medium)-IMCa (d) Read (Medium)-Lustre (4DS)

Fig. 8. Read latency with 1 MCD and varying number of clients

5.5 IOzone Throughput

In this section, we discuss the impact of IMCa on the I/O bandwidth. One of the benefits of a parallel file system with multiple data servers over a single server architecture such as NFS is the striping and advantage of improved aggregate bandwidth from multiple data streams from multiple data servers. This is especially true with larger files and larger record size. Using multiple caches in MCD, it might be possible to gain the advantage of multiple parallel data servers, while using a single I/O server. We use IOzone to measure the Read throughput of a 1GB file, using a 2K block size. We replace the standard CRC32 hash function used by libmemcache [2] with a static modulo function (round-robin) for distributing the data across the cache servers using a 2K block size. We measured the IOzone Read throughput with 1, 2 and 4 MCDs. These results are shown in Fig. 9. From these results, it can be seen that we can achieve a IOzone Read Throughput of upto 868 MB/s with 8 IOzone threads

and 4 MCDs. This is almost twice the corresponding number without the cache (417 MB/s) and *Lustre-1DS (Cold)* (325 MB/s). Clearly, adding additional Cache servers helps provide better IOzone Read Throughput.

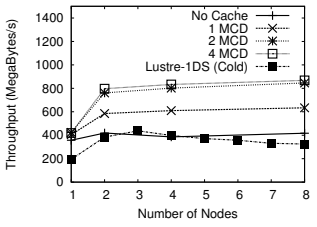


Fig. 9. Read Throughput (Varying MCDs)

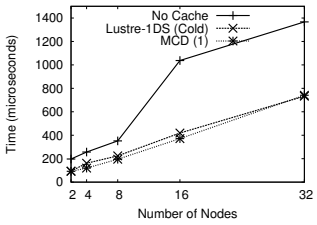


Fig. 10. Read Latency to a shared file

5.6 Read/Write Sharing Experiments

To measure the impact of IMCa in an environment where file data is shared, we modified the latency benchmark described in Section 5.3 so that all the nodes use the same file. In the write phase of the benchmark, only the root node writes the file data. In the read phase of the benchmark, all the processes attempt to read from the file. Again, as with the multi-client experiments (Section 5.4), the Read and Write portions, as well the portions for each record size are separated with barriers.

We measure the read latency, with and without IMCa and compare with *Lustre-1DS (Cold)*. With IMCa, we use one MCD. The read latency is shown in Fig. 10. At 32 nodes, there is a 45% reduction in latency with IMCa over the No-Cache case. Also, as may be seen from Fig. 10, IMCa provides benefit, that increases with an increase in the number of nodes. Since we are using a single MCD, with all the clients trying to read the data from the MCD in the same order, we see that the time even with IMCa increases linearly. With a greater number of MCDs, we expect better performance. Because of space limitations, we do not present the numbers for multiple MCDs here (they are available in the technical report version of this paper [8]). IMCa with 1 MCD provides slightly higher latency compared to Lustre-1DS (Cold) upto 16 nodes. However, at 32 nodes, IMCa with 1 MCD has slightly lower latency than Lustre-1DS (Cold).

6 Related Work

Dahlin, et.al. proposed using client side caching to perform cooperative caching [5]. The client caches are tied together to form a single large file system cache. Our work differs from their work in that we maintain a layer or bank of cache server nodes that are independent of the client side caches. S. Jiang, et.al. [4] proposed enhancements to the buffer caching algorithms on the file system servers. Our work is different from their work in that we propose an intermediate hierarchy of caching nodes that are independent of the file system buffer cache.

7 Conclusions and Future Work

In this paper, we have proposed, designed and evaluated an intermediate architecture of caching nodes (IMCa) for the GlusterFS file system. The cache consists of a bank of MemCached server nodes. We have looked at the impact of the intermediate cache architecture on the performance of a variety of different file system operations such as stat, Read and Write latency and throughput. We have also measured the impact of the caching hierarchy with single and multiple clients and in scenarios where there is data sharing. Our results show that the intermediate cache architecture can improve stat performance over only the server node cache by up to 82% and 86% better than Lustre. In addition, we also see an improvement in the performance of data transfer operations in most cases and for most scenarios. Finally, the caching hierarchy helps us to achieve better scalability of file system operations.

As part of future work, we plan to investigate different hashing algorithms for distributing the data across the cache servers. In addition, we would also like to look at how network mechanisms like Remote Direct Memory Access (RDMA) in InfiniBand can help reduce the overhead of the cache bank and also provide stronger coherency. We also plan on researching how the set of cache servers may be integrated into a file system such as Lustre, where it can potentially interact with the client and server caches. Finally, we would also like to study the relative scalability of a coherent client side cache and a bank of intermediate cache nodes.

References

[1] GlusterFS: Clustered File Storage that can scale to petabytes. http://www.glusterfs.org/.

[2] libmemcache. http://people.freebsd.org/ seanc/libmemcache.

[3] Danga Interactive. Memcached. http://www.danga.com/memcached/.

[4] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[5] M. Dahlin, et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[6] J. M. May. *Parallel I/O for high performance computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[7] S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. Panda. Supporting strong coherency for active caches in multi-tier data-centers over infiniband. In *SAN-03 Workshop (in conjunction with HPCA)*, 2004.

[8] R. Noronha and D.K. Panda. IMCa: A High-Performance Caching Front-end for GlusterFS on InfiniBand. Technical Report OSU-CISRC-3/08-TR09, The Ohio State University, 2008.

[9] R. Noronha and L. Chai and T. Talpey and D.K. Panda. Designing NFS With RDMA for Security, Performance and Scalability. Technical Report OSU-CISRC-6/07-TR47, The Ohio State University, 2007.

[10] R. Zahir. Lustre Storage Networking Transport Layer. http://www.lustre.org/docs.html.