User Level Implementation of Scalable Directories (GIGA+)

Sanket Hase, Aditya Jayaraman, Vinay K. Perneti, Sundararaman Sridharan,
Swapnil V. Patil, Milo Polte, Garth A. Gibson

CMU-PDL-08-107
May 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

High performance computing applications are becoming increasingly widespread in a large number of fields. However the performance of I/O sub-systems within such HPC computing environments has not kept pace with extreme processing and communication speeds of such computing clusters.

The problem of high performance is tackled by system architects by employing a variety of storage technologies such as Parallel File Systems. While such solutions serve to significantly alleviate the problem of I/O performance scaling they still a lot of roam for improvement because they not endeavor to significanly scale the performance of meta-data operations. Such a situation can easily arise in database or telecommunication applications that create thousands of files per second in a single directory. When this happens, the consequent performance degradation effected by the slowdown of meta-data operations can severely slowdown the performance of the overall I/O system.

GIGA+ affords a potential solution to this crucial issue of meta-data performance scaling in I/O sub-systems. GIGA+ is a scalable directory service that aims to scale and parallelize meta-data operations.

Acknowledgements: We would like to thank Prof. Garth Gibson for giving us the opportunity to work on GIGA+. The numerous discussions that we have had with him through this project have really been enlightening. We would also like to thank Milo Polte and Swapnil Patil. They have been of invaluable help to us.

We would also like to thank Wittawat Tantisiriroj without whom we wouldn't have been able to run most of our experiments.

Special thanks to Garth for giving us the opportunity to meet with Rob Ross and Sam Lang from Argonne National Labs. Discussing our project with them was a valuable experience.

Finally we are also grateful to have the opportunity to present our work at PDL visit day on the 15th of May at Carnegie Mellon University.

Keywords: GIGA+, Huge Directories, File-system using FUSE, Meta data performance scaling.

1 Introduction

Large scale and High Performance computing often requires significant computational capability and involves large quantities of data. High performance computing applications are becoming increasingly widespread in a large number of fields. However the performance of I/O sub-systems within such HPC computing environments has not kept pace with extreme processing and communication speeds of such computing clusters. The performance degradation induced by shortcomings in I/O capability can severely impede overall cluster performance [8].

The problem of high performance is tackled by system architects by employing a variety of storage technologies and high-speed interconnects. One of the principal tools they employ for this purpose are Parallel file systems, quite simply because of the greater shared access to data in parallel that they enable. In addition, their ability to fulfil critical I/O subsystem requirements ensures that Parallel File systems such as PVFS, Lustre, GPFS and PanFS^r are well suited for HPC cluster environments.

By employing strategies such as data stripping across I/O servers, parallel file systems easily scale bandwidth and improve performance by parallelizing data operations. However, most parallel file systems do not provide the ability to scale and parallelize meta-data operations because it is inherently more complex than scaling the performance of data operations. [9] PVFS provides some level of parallelism through distributed meta-data servers that manage different ranges of meta-data. This implies that meta-data operations are distributed in the sense that depending on the object being operated on, a particular meta-data server is contacted. This affords some performance boost but a significant problem still arises if a single directory on a server becomes a "hot-spot" for meta-data operations. Such a situation can easily arise in database or telecommunication applications that create thousands of files per second in a single directory. When this happens, the consequent performance degradation effected by the slowdown of meta-data operations can severely slowdown the performance of the overall I/O system.

In such environments scaling meta-data performance and achieving true parallelism with respect to meta-data operations becomes critical. One such challenge is building scalable directories for cluster storage – i.e., directories that can store billions to trillions of entries and handle hundreds of thousands of operations per second.

GIGA+ affords a potential solution to this crucial issue of meta-data performance scaling in I/O sub-systems. GIGA+ is a scalable directory service that aims to scale and parallelize meta-data operations. GIGA+ partitions a directory into a scalable number of servers. In this world such directories are referred to as Huge Directories. These Huge Directories are partitioned in a manner that ensures load balancing across all servers. GIGA+ achieves this load balancing by employing an extensible hashing scheme.

A scalable directory service such as GIGA+ can be developed in an existing parallel file system such as PVFS, pNFS or PanFS^r. However working with huge system like PVFS to build a non-trivial scalable meta-data service is a significantly complex endeavour. Furthermore such an implementation would not be portable across multiple file systems.

A simpler solution to implement GIGA+ would be to develop a standalone system that runs on top of a parallel file system like PVFS. By implementing GIGA+ as a user space application portability across multiple file systems can be achieved. Such a tool could easily be ported to run on top of any file system such as PVFS, pNFS and others and could serve as a useful tool for research in this area.

The following sections describe the design and implementation of our user land GIGA+ huge directory service. The next section provides a background into what GIGA+ is and a background into the tools that were used, specifically FUSE and ONCRPC. This is followed by a detailed design of our user land implementation. Lastly we provide a detailed section on the various experiments that were conducted and analyses of the results. The various results in this section not only demonstrate the proof of concept along with performance of our user land implementation but also the insignificance of the FUSE overhead.

2 Background

"*Much of this section has been adopted from the GIGA+ paper [2]. Thanks to Swapnil for the same*"

2.1 GIGA+

GIGA+ is a new directory subsystem that can scale, both the capacity (i.e., index large number of files) and the performance (i.e., provide high throughput). It was designed with 3 primary goals:

(1) High scalability through more concurrent and unsynchronized growth
(2) High concurrency through minimal synchronization overhead
(3) Maintain UNIX file system semantics

GIGA+ divides each directory into a scalable number of fixed-size partitions that are distributed across multiple servers in the cluster. The set of servers that store the partitions of a directory is called the server list. Each server holds one or more partition of a directory; and, the storage representation of a partition are managed as a directory by the server's local file system. The number of partitions (of a directory) grows with directory size. A directory starts small and is represented using a single partition that holds all the keys. As more entries are added in the directory, the partition becomes full and it splits half into a new partition on a new server.

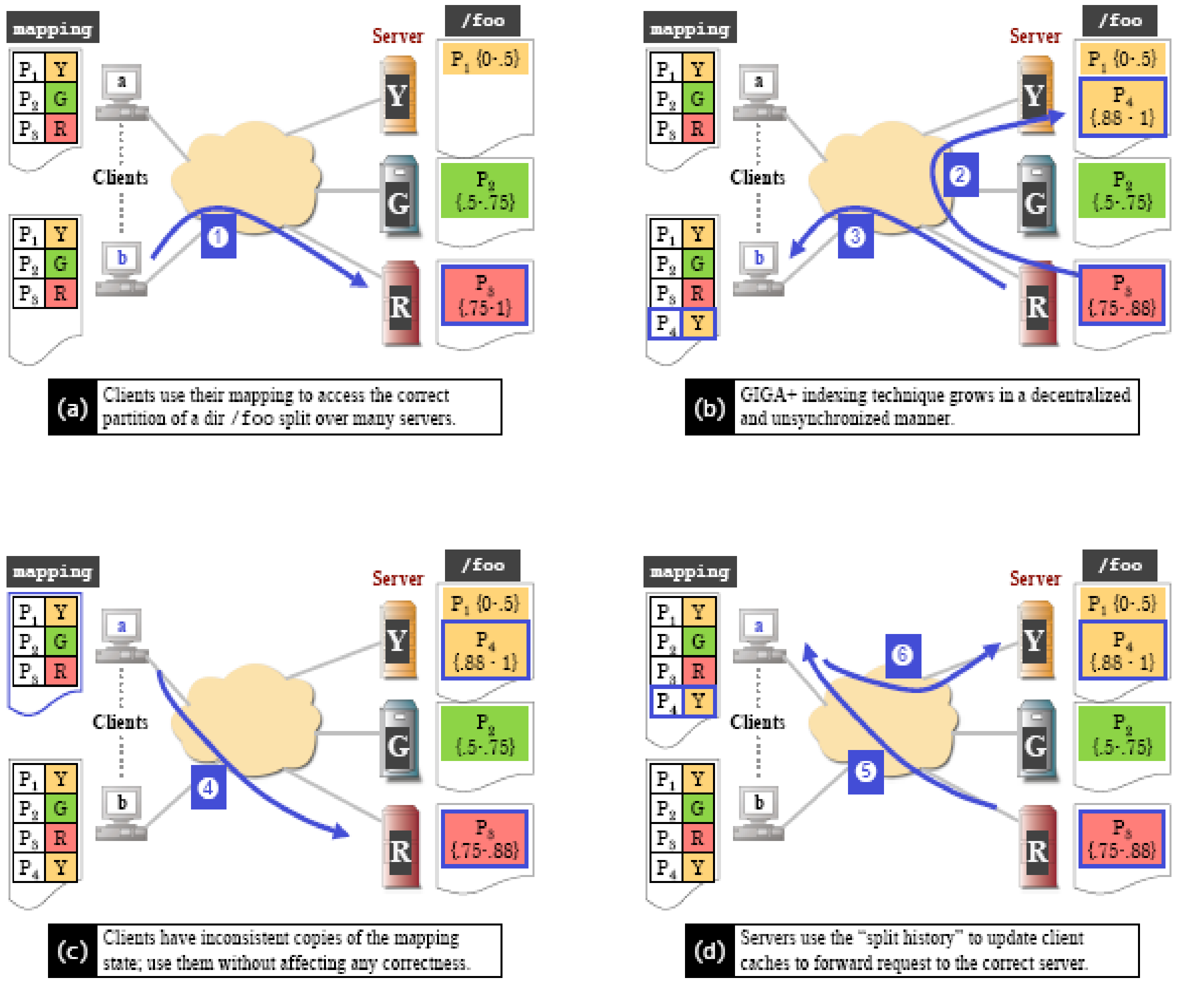

mapping
Server
/foo
Clients
a
b
Y
G
R
P_1 Y
P_2 G
P_3 R
P_4 Y
P_1 {0-.5}
P_2 {.5-.75}
P_3 {.75-1}
P_4 {.88 - 1}
P_3 {.75-.88}
1
2
3
4
5
6
(a) Clients use their mapping to access the correct partition of a dir /foo split over many servers.
(b) GIGA+ indexing technique grows in a decentralized and unsynchronized manner.
(c) Clients have inconsistent copies of the mapping state; use them without affecting any correctness.
(d) Servers use the "split history" to update client caches to forward request to the correct server.

Figure 1: Overview of GIGA+. Directory '/foo' divided into partitions that are distributed on three servers. Clients use their partition-to-server mapping to lookup the correct partition and server. Steps involved in an insert and lookup in GIGA+: (1) Client sends a request to insert a file in a partition of the directory. (2) If the partition is full, GIGA+ uses its decentralized indexing technique to create a new partition and record this info at the old server. (3) Server sends the response the client, who updates its mapping state. (4) Clients have inconsistent copies of the mapping information but they continue to use it for directory operations. (5) An "incorrect" server receives a clients' request; it uses its up-to-date view of the world to update the client's stale cache. (6) And then clients then send their request to the "correct" server

The directory, i.e., the number of partitions, grows or shrinks dynamically using the decentralized and parallel GIGA+ indexing technique. The technique ensures that each partition contains different entries, i.e., the partitions don't overlap with each other. As a result, any operation (insert, lookup or delete) on a directory entry only needs to look at one partition - the partition that holds that entry. However, to perform a directory operation, clients need to determine the correct partition and the server that stores it. In GIGA+, clients cache the partition-to-server map (P2SMap) created from the indexing mechanism. A naive version of the P2SMap might look like a simple partition-server table, as shown in the clients in Figure 1. As the directory grows with usage and new partitions are created on the servers, this P2SMap changes – and, at high insert rates, it changes rapidly.

While this would cause high overheads due to synchronization and serializations, GIGA+ avoids a lot of such issues by allowing clients to continue using the stale P2SMap information for all its operations. GIGA+ guarantees the correctness of all client operations in spite of old, cached P2SMap information. Clients with an out-of-date P2SMap send a request to an "incorrect" server that no longer holds the desired key in its partition. The servers then independently use their own view of the world and determine if the client has come to the right place, and if not, reply back to the client with the updated map which the clients merely sync up with their own. While this does result in a few more probes than necessary than the ideal case, and in the worst case could result in logarithmic number of splits, GIGA+'s lack of a need to do explicit synchronization and serialization results in reduction in significant overhead and allows for greater parallelization of concurrent operations.

Allowing for stale mappings in the clients as shown in the example below, is one of GIGA+'s key features that enables both servers and clients to work in parallel.

2.1.1 GIGA+ Indexing

GIGA+ extends on Fagin's original extendible hashing technique [10] which is a dynamic data structure that allows the hash table to grow and shrink with usage. The fundamental principle behind such a hashing technique is controlling how many bits of the hash value are taken into consideration and which bits can be allowed to remain static to maintain backward (in time) compatibility. It uses a two-level structure: it adds a top-level of indirection, called header-table that is an "array of pointers" to the partitions at the second level. This indirection, using a header-table that holds pointers to the partitions, allows the number of partitions to grow gracefully. Each entry in the header-table points to a single partition; however, one partition can be pointed to by multiple entries in the header-table. Entries in the header table are indexed using an increasing radix, which is a r -bit suffix of the hash of the data key K. In GIGA+, the directory entry name serves as the key K and it is hashed using the MD5 or SHA1 algorithm that offers good collision resistance and randomness properties required for load-balanced key distribution.

Essential to the GIGA+ scheme are the concepts of the header table which houses the bitmap and the radix. The bits of the bitmap of a huge directory stores information on how the partition space is split and radix refers to the growing number of bits of the bitmap that need to be considered.

The radix increases or decreases with the size of the header-table. The header-table has 2^r entries, each pointing to a single partition; if there are 2^r partitions, every header-table entry will point to a unique partition, and otherwise some entries will point to the same partition. The hash table grows by adding more partitions to store new keys. If the number of partitions exceeds 2^r, the header-table needs more entries to point to the new partitions. This results in increasing r by 1, which "doubles" the header-table to hold 2^r+1 entry. The hash table grows by dynamically allocating more number of partitions. Inserting a key in an over flow partition splits the partition, i.e. a new partition is created and half the keys from the old partition are distributed to this new partition. We use the additional bit (r+1th bit) of the r-bit radix to decide which keys are moved to the new partition; keys with the additional bit "1" are given the new partition.

2.2 FUSE

The main idea behind GIGA+ is to provide performance scalability for meta-data operations in existing file systems. This essentially involves building a scalable meta-data service above an existing file system. As mentioned earlier an easy way to do this is to build a user level application that runs on top of an existing file system. However in order to do this we need the ability to capture and manipulate file system operations in user space. Fuse provides just this ability.

Broadly, FUSE (File System in User Space) is a loadable-kernel module for Linux based machines that allow non-privileged users to create User Space File Systems without ever having to write or edit Kernel code [3]. FUSE achieves this by abstracting Kernel level functionalities in User Space and provides a bridge to actual Kernel interfaces via a FUSE kernel module.

FUSE is particularly useful for creating Virtual File Systems. Unlike traditional Linux based file systems like EXT3 which directly control the management of files on disk, a FUSE based file system typically act as a view or translation of an existing file system or storage device.

FUSE was originally developed to support AVFS [4], but has since become an independent project. There are a number of File Systems [5] implemented that use FUSE.

2.2.1 FUSE Architecture

The Kernel Module provides the necessary capability to hook a VFS intercepted system call such as open(), read() etc to a user space library. This is similar to how a traditional File System like EXT3 would operate. Except that instead of performing its operations on a local storage space such as an underlying Disk, the FUSE Kernel Module forwards the file system request to a procedure in User Space. The FUSE Kernel Module has been merged with the Linux Kernel in 2.6.14+.

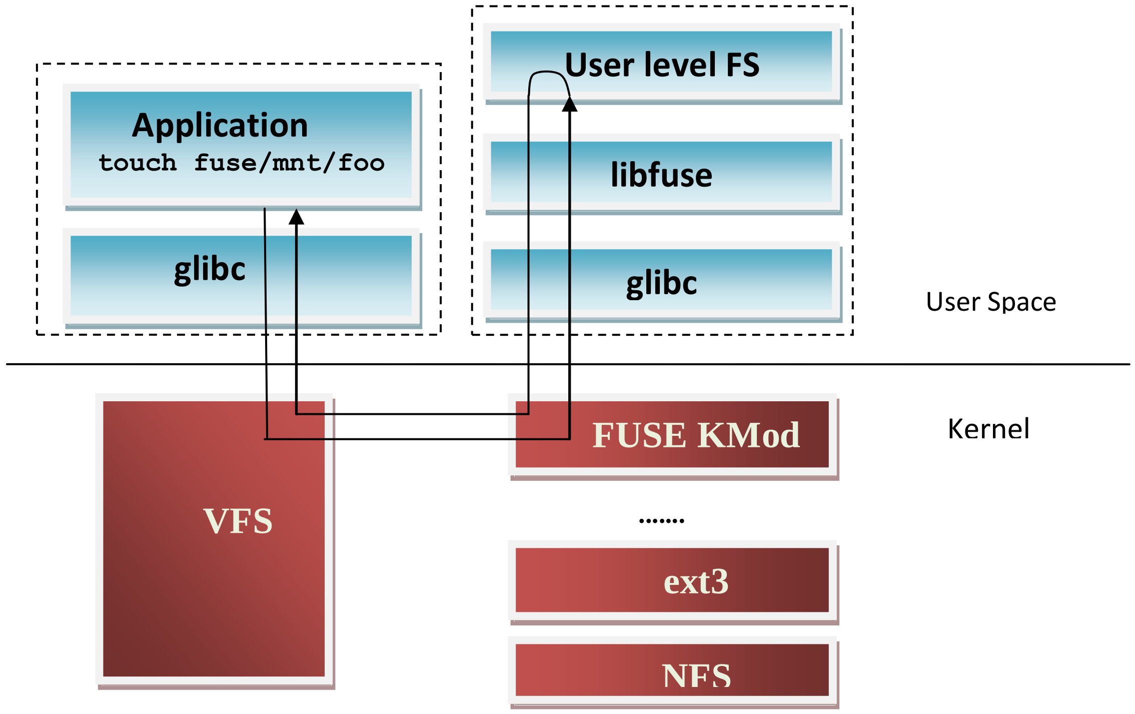


Figure 2: FUSE Architecture. The figure shows the architectural components involved in a FUSE based file system. The two main components are the Fuse Kernel Module and the user space *libfuse* [3] library.

The FUSE User Module (*libfuse*) is a library that provides a set of API's for file system operations. To start up a user space file system that uses FUSE, the program must execute a FUSE mount operation that establishes a bridge between the FUSE user and kernel module. The FUSE user space module provides different API specifications at different abstraction levels thereby providing flexibility for file system developers.

The FUSE kernel module and the FUSE library communicate via a special file descriptor which is obtained by opening */dev/fuse*. This file can be opened multiple times, and the obtained file descriptor is passed to the fuse mount operation, to match up the descriptor with the mounted file system.

Overall, FUSE provides a very easy mechanism to develop a file system in user space that requires absolutely no manipulation to kernel code or having to employ complicated and error prone techniques like LD_PRELOAD able modules, system call hooks and other tricks.

By leveraging the API's provided by FUSE, a capable and portable user space file system can be developed with significant ease and in considerably less time as compared having to write one from scratch.

2.3 ONC RPC

GIGA+ is essentially a distributed file system with clients and servers that run on separate machines. Clients and servers frequently talk to one another exchanging critical file system information. To achieve this, we need an efficient communication protocol for passing information from clients to servers and vice versa. Having to write an RPC communication protocol that provides reliability and multi-thread support is a significant project in itself. Thus, for this purpose we employ ONC RPC.

Open Network Computing Remote Procedure Call is a widely deployed remote procedure call system. ONC was originally developed by Sun Microsystems as part of their Network File System project, and is sometimes referred to as Sun ONC or Sun RPC [6].

ONC is based on calling conventions used in UNIX and the C programming language. It serializes data using the XDR [7], which has also found some use to encode and decode data in files that are to be accessed on more than one platform. ONC then delivers the XDR payload using either UDP or TCP. Access to RPC services on a machine are provided via a *port mapper* that listens for queries on a well-known port, port 111 over UDP and TCP.

The ubiquity of deployment in most Linux deployments lends itself well to the choice of using ONC RPC for the user land implementation. This enables one to use any Linux machine available on the network as a GIGA+ server or client without having to install any special network tools or other software packages.

3 Design

3.1 System Architecture

One of the important design goals of the user land GIGA+ is encapsulation of functionalities in isolated components. Hence, the architecture follows a layering approach with distinct interfaces to communicate amongst the layers.

From a broad view the system can be broken up into two major components – the GIGA+ client and the GIGA+ server.

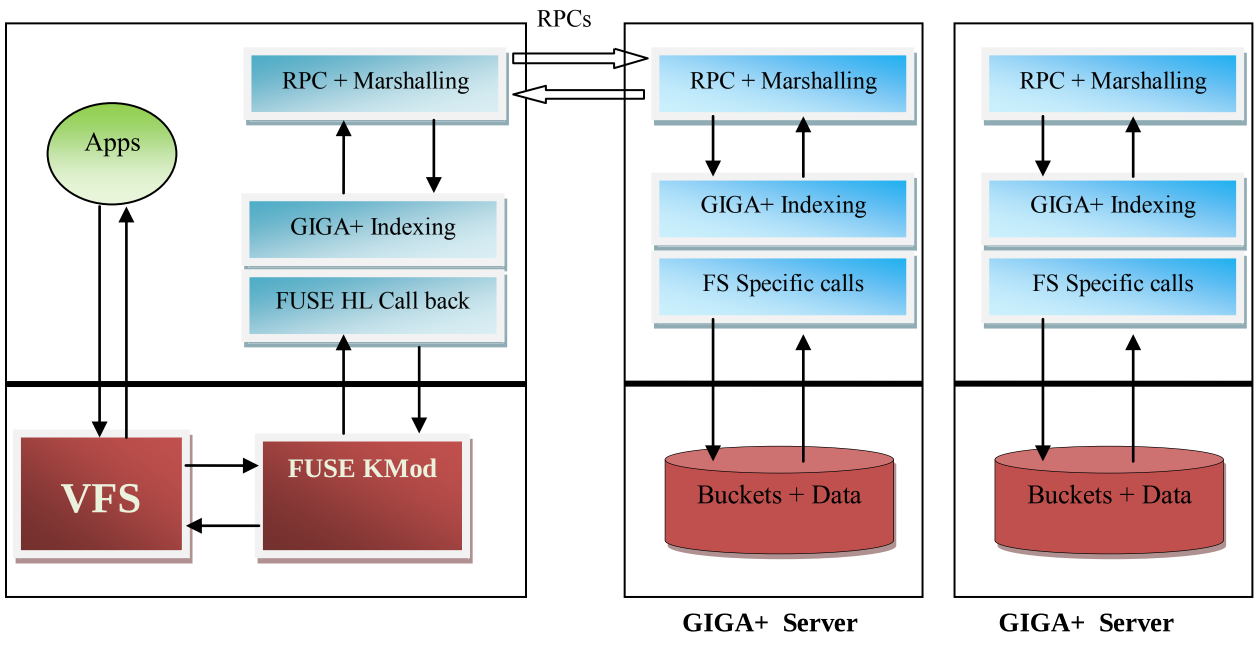


Figure 3: GIGA+ System Architecture

The main reason for adopting a distributed architecture rather than centralizing all the GIGA+ functionality on one machine is to provide scalability of the common storage space shared by the GIGA+ applications. Furthermore having server processes controlling meta-data manipulations as an isolated unit enables fair scheduling of file system operations across a distributed storage space.

This architecture can yield maximum performance when deployed on top a Parallel File system like PVFS. If deployed over a file system such as ext3, it incurs an overhead of migrating file data across servers which is a significant I/O overhead.

3.1.1 GIGA+ Components

3.1.1.1 GIGA+ Client

The GIGA+ client is used to refer to the set of components that begin working once a user application's file system call is handed over to FUSE which results in an appropriate callback being invoked in the GIGA+ client's callback layer. The GIGA+ client sits on top of the *libfuse* library in user space.

When a call like *mkdir* is issued by the application like the shell, the underlying VFS layer detects that the call is under a mount point managed by FUSE and hands off the call to the FUSE kernel module. The FUSE kernel module will then do some processing before finally calling back the code that marks the beginning of the GIGA+ client. This in our implementation would be the giga_clnt_mkdir() which would process the call.

The GIGA+ client consists of three distinct layers each with a designated function.

1) The GIGA+ Indexing component which provides meta-data load distribution across GIGA+ servers. This library is based on extensible hashing. This library caches Huge Directory meta-data information retrieved from GIGA+ servers. This cache informs the client about the distribution of Huge Directories across GIGA+ servers. The client's cache may be stale and updates to the cache are made though regular file system operations.
2) The GIGA+ file system library which serves as the FUSE callback layer and is responsible for requesting file system services from GIGA+ servers. A file system operation issued by an application is sent to this layer by FUSE. This library uses the GIGA+ indexing library to determine which server to contact for the requested operation. Once this information is retrieved an appropriate server side routine is invoked via an RPC interface.
3) The RPC layer that is responsible for network communication between clients and servers. The RPC layer is responsible for marshalling parameters received from the higher layer and sending them out to the appropriate servers and Un-marshalling replies. The RPC layer also ensures reliability of the underlying network transport layer.

3.1.1.2 GIGA+ Server

The GIGA+ Server is a standalone program that serves as the core of the GIGA+ file system. As mentioned earlier the GIGA+ servers are responsible for managing access and manipulating file system objects across the distributed storage space. The GIGA+ servers are only responsible for managing file system data objects under its local storage space. It knows nothing about file system objects maintained on other servers.

GIGA+ servers use the services provide by the underlying file system which may be ext3, PVFS, NFS etc. Servers are only responsible for managing meta-data information of Huge Directories and know nothing of the files that lie beneath them.

The GIGA+ server has three distinct layers, each with a designated purpose.

1) The GIGA+ Server has an indexing library which it creates and manages to perform load distribution of meta-data operations. For this purpose the GIGA+ servers maintain data-structures called Huge Directory Tables or "HD Tables". These HD tables provide directory partitioning information.
2) The File System layer is where the processing of the file system commands happens as and when they are received from GIGA+ clients. When specific calls are received from GIGA+ clients the GIGA+ server calls in to specific file system operation on the underlying file system. This layer also responsible for using the directory partitioning information maintained by the indexing library which is needed to distribute files across servers and thereby provide scaling of directories.
3) The RPC layer is the first layer that receives calls from clients and other servers and sends them off to the file system processing layer.

The Servers HD tables are required for the correct functioning of the GIGA+ indexing on the clients and are hence cached by GIGA+ clients. As mentioned earlier the GIGA+ clients may have stale information in their cache. When this happens clients may send a file system request to and "incorrect" server. At this point it becomes to GIGA+ server's responsibility to identify this situation and update the client's cache with updated data structures.

3.2 GIGA+ Indexing Design

As mentioned before, distribution of buckets for a particular Huge Directory is represented in the form of a bitmap. A partition or a bucket of a Huge Directory is represented using a bit of the bitmap. When a Huge Directory is created the bitmap is initialized.

3.2.1 Concept of Home Server

GIGA+ logic ensures that files in huge directories are spread evenly across available set of servers. But, GIGA+ also ensures even distribution of small directories.

Every directory starts small and it is hosted on its "home server". A home server for a particular directory is determined by hashing directory name alone. This ensures that no single server is overloaded with all small directories. For small directories that have not split, an optimization is performed to utilize distributed resources. Home server hosts the 0th bucket for the directory. All further buckets generated because of bucket splits are distributed across the servers in round-robin manner.

Consider an example bitmap. The scenario depicted in [figure 4] assumes a configuration with three servers where S[i] represents the server number. N servers are numbered from 0 to N-1.

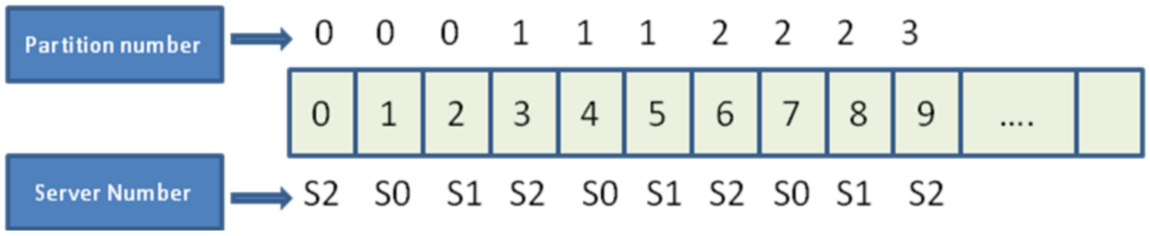


Figure 4: Bitmap and Bucket Distribution

Bucket 0 is stored on server 2 which is the corresponding Huge Directory's Home Server. As the directory grows, further buckets are distributed among the three servers in round-robin order.

As shown, the first N buckets in the bitmap (here 3) represent partition 0 of corresponding servers. The next-round robin set of buckets represents partition 1 and so on.

Now to maintain compliance with the GIGA+ indexing scheme, this home server value is factored into the computation of the final server as follows.

```
server_id = (home_server_id + computed_index) % no_of_servers
```

The server computes the partition_id as,

```
partition_id = computed_index / no_of_servers
```

3.2.2 Split Design

3.2.2.1 Purpose

In most current file systems, there are certain practical limits on the number of entries that a directory can have. Such limits are generally not imposed by the file system itself, but by physical hardware limitations that slow down operations like simple lookups and inserts.

3.2.2.2 Concept

The idea behind splitting is that when a real directory (bucket) on a GIGA+ server grows too big i.e. beyond its configurable capacity, it gets split into two buckets with half of the original entries moving to a new bucket, hosted on another server. Which directory entries remain in the current bucket and which ones get shipped to the new bucket is determined by the GIGA+ indexing logic. This always causes a change in the bitmaps of the directories on both servers and possibly a change in the radix (which determines the number of bits of the hash that are relevant). Thus splitting a directory influences the way the clients find their files.

In GIGA+ the incremental value of the radix is used to determine distribution of files across Servers. Every small directory starts with a radix of zero. This radix is stored in the HD Table data-structure.

When bucket capacity is exceeded, a new bucket is created to share the load of files. This is referred to as a "bucket split". When the 0th bucket splits, the file names are hashed and the last bit of the hash of the filename indicates the target bucket where the file should reside at the end of the split. The split of bucket 0 triggers the extendible Hashing logic employed by GIGA+.

The location of the bucket in the bitmap represents its index. In an extendible hashing scheme, this index is a node in a binary tree. The depth of this node in the tree represents the radix of the index. Following 0th bucket split, further splits involve looking at radix number of bits of the file name's hash to determine the target bucket.

All servers need not be at the same depth of the radix tree. Thus their value of radix for the same huge directory can be out of sync. A reference to a bucket by another server might refer to an empty bucket in which case existing parent of the indexed bucket on the server is found by traversing the bitmap. Bitmap is traversed, in bottom-up order (look at it as a binary tree) until 0th bucket is reached.

3.2.2.3 Indexing with splits

Figure below shows how splitting buckets are indexed. This representation of buckets is complementary with the bitmap representation addressed earlier. As bucket splits, its entries are split in the manner shown below.

Not all buckets that can be indexed with a given radix need to be present. As shown below, certain buckets may not have as many entries as are needed to necessitate a split. As a result, the tree diagram shown is unevenly balanced. This uneven balancing of the splits is the reason clients may land up at wrong servers and may need to be redirected to the closest, correct "parent".

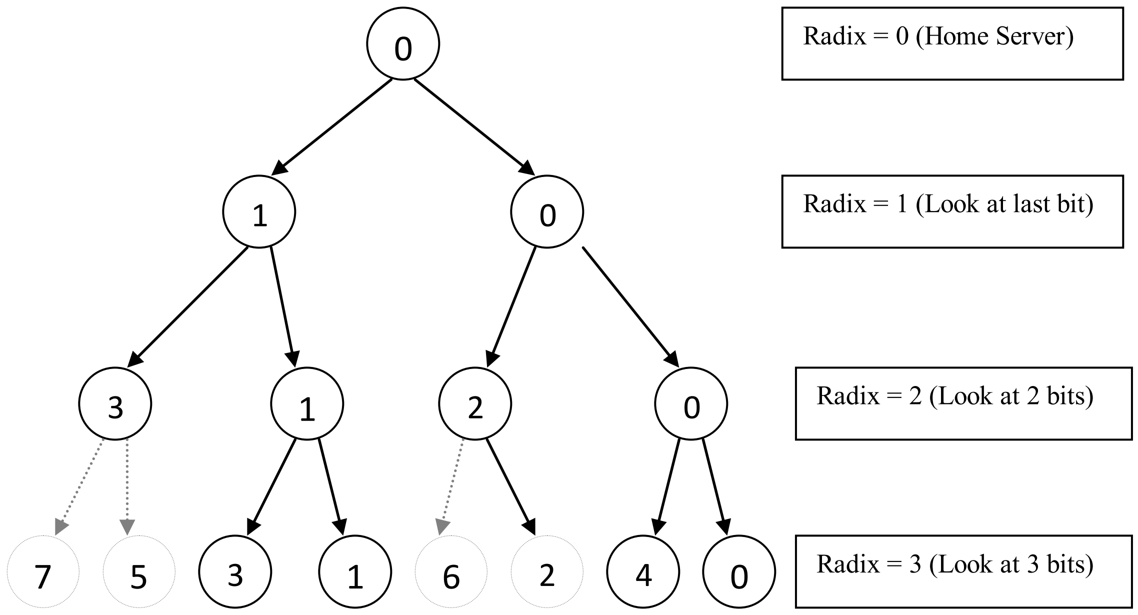

0
1
0
3
1
2
0
7
5
3
1
6
2
4
0
Radix = 0 (Home Server)
Radix = 1 (Look at last bit)
Radix = 2 (Look at 2 bits)
Radix = 3 (Look at 3 bits)

Figure 5: GIGA+ bucket splits and corresponding indices. Every number represents the index into the bitmap which would indicate the presence or absence of that partition. In above example, buckets 2 and 3 have not split yet, so a lookup for index 7 or 5 would seek their parent and land up at 3. Similarly lookups for 6 would end up at 2.

GIGA+ Concurrency Support

The GIGA+ file system supports multiple applications to simultaneously utilize its resources. As a distributed file system, GIGA+ provides the ability to execute parallel I/O operations on files and directories distributed across the global storage space. GIGA+ servers maintain state for Huge Directories, which is required to perform the necessary GIGA+ indexing. To enable support for concurrent applications these global data-structures need to be appropriately protected against simultaneous modifications.

In the current design we attempted to provide as fine grain locks as possible to enable highest level of concurrency as possible.

Thus GIGA+ servers implement buckets level locks. Buckets level locking enables the GIGA+ server to allow simultaneous modifications to a single Huge Directory's meta-data as long the modifications are to different buckets. For instance, if a split is occurring on one bucket of a Huge Directory, then Bucket level locks enable a simultaneous split to occur on another bucket of the same Huge Directory on the same server. This allows for a high level on concurrency.

The GIGA+ clients are inherently multithreaded because FUSE provides a multithreaded interface to file system operations. Concurrency control is relatively simple on the client because applications at the GIGA+ client compete only for access to the HD Table cache.

3.3 FUSE

Scalable huge directories cannot be implemented in user land unless there is a mechanism to capture a user's file system operations on its files. File System in User Space or FUSE provides a mechanism to do just that. Once mounted as a fuse file system, the fuse kernel module will capture all the file system calls made inside its mount point and throw it back to the corresponding user land callback that has been registered with it. It is in this callback that the user land implementation of GIGA+ begins.

FUSE was adopted as the tool to implement GIGA+ in user space for a number of reasons. Since the FUSE kernel module comes installed with Linux kernels 2.6.14+, user space GIGA+ can be completely implemented without having to write or modify kernel code.

Also the FUSE user space library *libfuse* also provides two flexible API's specifications to implement a user space file system. The capability afforded by this API's can be leveraged to develop a powerful file system. Details of the API specification are described herein.

3.3.1 FUSE (libfuse) API

The FUSE user space library *libfuse* provides two sets of API interfaces, namely the FUSE High Level Interface and the FUSE low level Interface.

At the lowest level in user space, the kernel communication layer traps into the FUSE Kernel Module (via VFS) and sends the application's request such as a open(), mkdir() etc. The FUSE User and Kernel module have a highly specialized I/O layer that talk to each other.

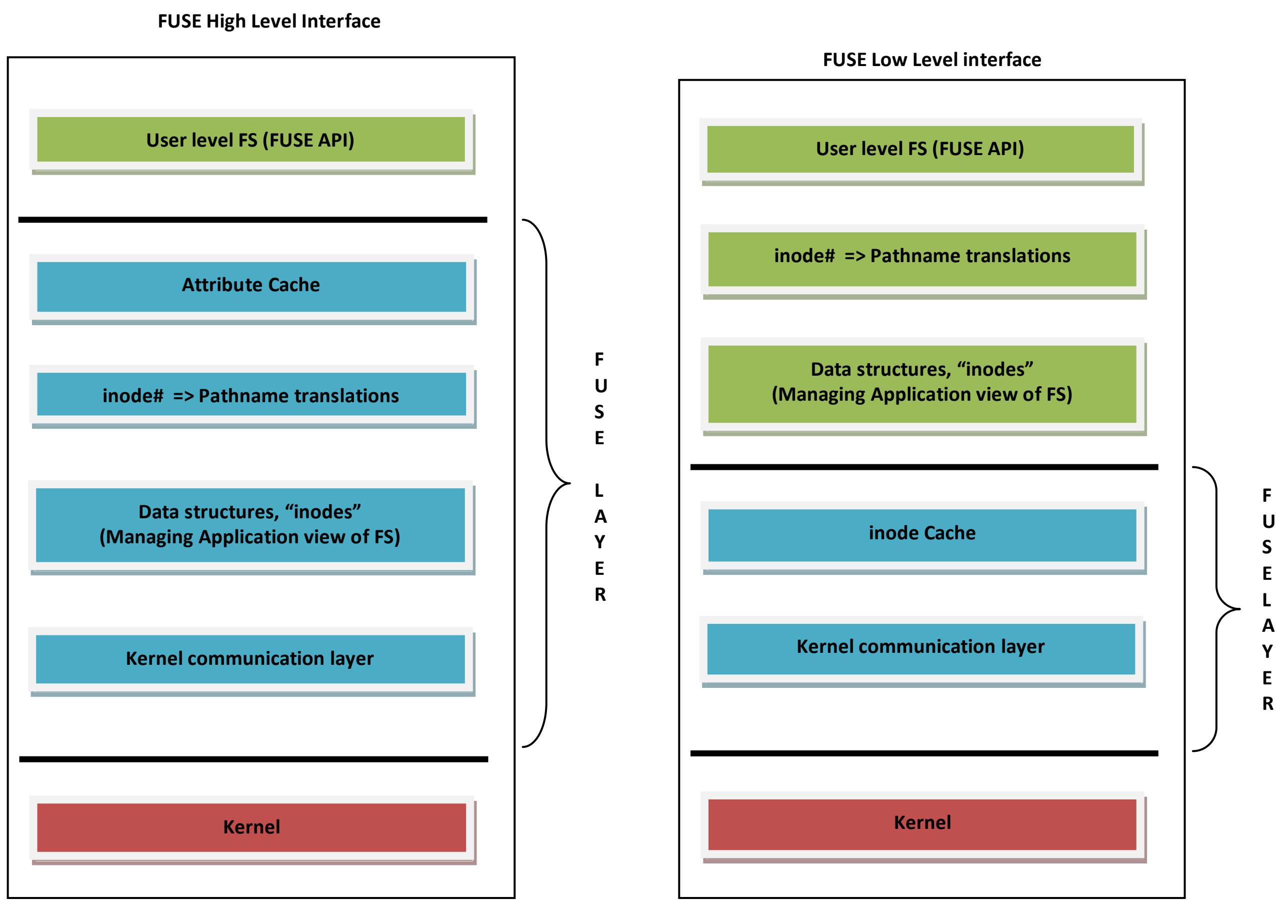

FUSE High Level Interface
User level FS (FUSE API)
Attribute Cache
inode# => Pathname translations
Data structures, "inodes" (Managing Application view of FS)
Kernel communication layer
Kernel
FUSE LAYER
FUSE Low Level interface
User level FS (FUSE API)
inode# => Pathname translations
Data structures, "inodes" (Managing Application view of FS)
inode Cache
Kernel communication layer
Kernel
FUSE LAYER

Figure 6: FUSE High Level and Low Level Interfaces. Figure demonstrates the High Level and low level library available in *libfuse.* As the figure suggests the High Level library clearly provides a richer set of functionalities to the User space file system developer.

The High Level Interface *libfuse*, has data structures to manage the application's view of the File System. Thus for file system objects that are created using the GIGA+ file system, the FUSE High Level library takes the responsibility for managing these objects. It does so by associating inode numbers for file system objects and tracks those by maintaining an in memory tree like data structure that is representative of the applications view of the File System. These data structures are never made persistent and are only maintained in memory. Further, FUSE provides the capability to associate timeout values for system object data structures that are maintained in memory. Once the in memory life-time of a file system object expires the data structures are destroyed. This affords the flexibility to control how large the FUSE in memory data structures can grow.

If in the near future some operation is performed on file system objects that FUSE is no longer tracking, FUSE performs a component by component lookup operation trying to rebuild its state for the file system objects. For instance if an application performs *open("foo/bar")* and FUSE knows what "foo" is but not what "bar" is, then it performs a lookup on "bar" (calls *getattr()*) on the GIGA+ file system. Effectively it becomes the GIGA+ file system library's job to remind FUSE of what "bar" is or isn't.

The main reason the FUSE High Level library maintains in-memory data structures tracking file system inodes is to perform pathname translations between inodes and path's. Thus every API in the GIGA+ file system takes a complete pathname as an argument. Thus if an application performs a *read(FD, size)*, the FUSE Kernel Module translates this operation on an open file descriptor to an operation on an inode and sends an appropriate request to libfuse. Libfuse then performs the necessary pathname translation from an inode identifier to a complete path. This path is given as argument to the GIGA+ user space file system.

These semantics imply that the GIGA+ user level file system has to never bother with low level details like inode identifiers. Every operation performed by an application under the GIGA+ file system mount point is always translated into a complete pathname.

At the GIGA+ user level file system, file system operation are performed on an underlying Files system such as EXT3, NFS, PVFS or PanFS. Having pathnames at this level simplifies a lot of things for the GIGA+ user level file system.

Since most of the low level work is done by FUSE High Level library itself, much of the low level infrastructural work of building a file system is already provided by fuse.

Thus it provides a complex and powerful set of semantics of developing a file system. One could imagine building a powerful file system by leveraging the capability of the High Level Interface.

The Low Level Interface on the other hand, as figure 6 suggests does not provide nearly the same level functionality afforded by the High Level library. The Interface between the Kernel module and the User library does not change. So as in the case with the High Level library the read request is translated into an operation on an inode. Now the Low Level library in libfuse does not perform and pathname translations of any kind, but simply passes these inode numbers on to the User level File system and expects it to understand what these inode corresponds to.

Specifically, when file create request is issued by an application the user level library is expected to return an inode identifier back to FUSE if and when this file is created. At a later point in time when some application performs an operation of this file FUSE gives the User level file system the same inode that it previously returned.

Effectively FUSE expects the User level file system to maintain its own database which is indexed by inode identifiers. For GIGA+, having to maintain a separate database for managing inode identifiers would impose a significant overhead.

Thus, the Low Level Interface simply provides a set of interfaces that are relatively similar to the file system interfaces provided by a traditional Linux based file system. The user level file system is expected to do most of the work in terms of management of inodes and performing pathname translations. Thus one can imagine that the low level interface is intended for file system applications that are developed from scratch.

For implementing the GIGA+ file system, using the low level interface would mean doing a lot of extra work for management of file system state. Instead by leveraging the functionality of the High Level Library much of work for management of file system state is done by the Interface itself and the GIGA+ user level file system can only deal with the GIGA+ indexing scheme and management of Scalable Directories.

3.3.2 FUSE Interaction with GIGA+ Filesystem

The Figure shows how FUSE controls the semantics of the applications view of the file system. Essentially the FUSE API provides a set of interfaces that need to be implemented based on user level file system i.e. GIGA+ specifications.

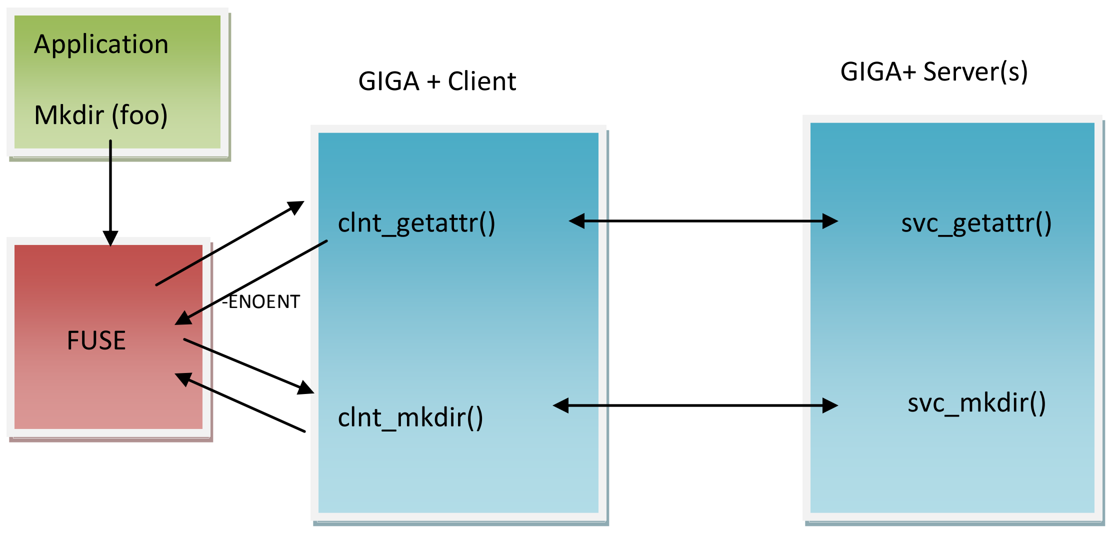


Figure 7: FUSE Semantics

As mentioned previously, the GIGA+ file system has been implemented using the High Level interface provided by the user space *libfuse* library. The FUSE interface is intelligent enough to know exactly the sequence of operations that need to occur when an application issues a file system operation. Thus when an application issues *mkdir()* for example, FUSE knows that for a *mkdir()* to succeed there must not already be file or dir of the same name under the applications current working directory. To determine this for every such operation it invokes the *getattr()* call to our GIGA+ file system library which effectively serves as a lookup operation on an object. Depending on the error code that the GIGA+ file system library returns back to FUSE, it knows whether or not a *mkdir()* can actually be issued.

Note that since GIGA+ is a distributed file system, getattr() followed by mkdir() is not atomic. Even if getattr finds that a file or directory dos not exist, then it is still possible for mkdir to fail because the file or directory can now exist. So when a mkdir call is received we cannot rely on POSIX system call semantics to determine file or directory existence. Since a directory itself is distributed across servers, we need to ensure that a file does exist on a partition on any server before it can be created on a single server.

3.4 RPC Design

GIGA+ user land uses remote procedure calls for communications between the client and server components. As indicated earlier, the architecture uses the layering approach to achieve a certain degree of coherence between the different system components. The RPC mechanism is hence designed as a separate layer of the system which is essentially made up of 3 sub-layers as shown in the figure below.

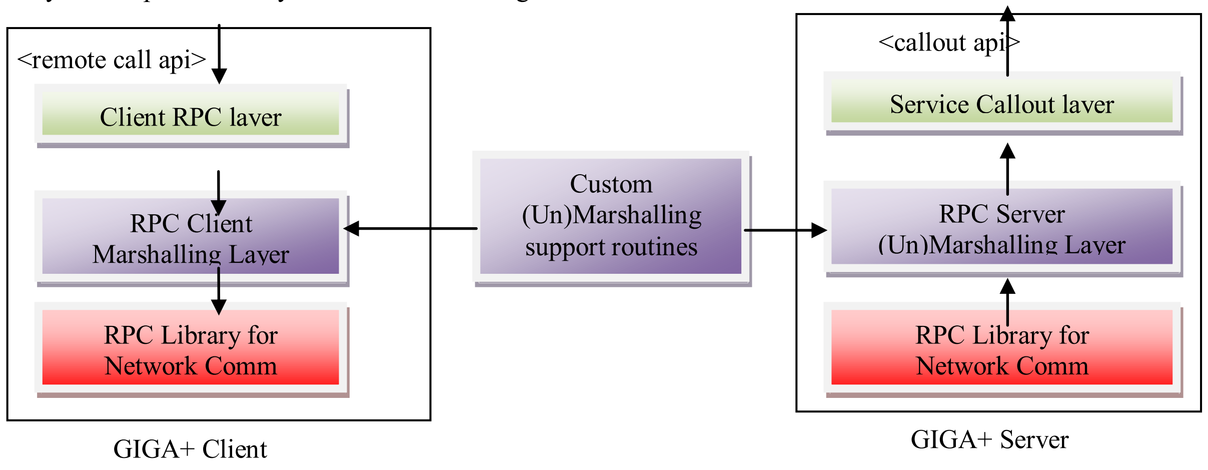

<remote call api>
Client RPC layer
RPC Client Marshalling Layer
RPC Library for Network Comm
GIGA+ Client
Custom (Un)Marshalling support routines
<callout api>
Service Callout layer
RPC Server (Un)Marshalling Layer
RPC Library for Network Comm
GIGA+ Server

Figure 8: Design of RPC Lib. Above diagram shows the path that remote procedure calls traverse after they are issued as a r<call_name>() at the client to when they end up calling out to the corresponding do_<call_name> at the server

To the caller, the RPC layer on the client exposes a set of r<call_name>() function API's. For example, to make a directory on the remote server, the caller merely needs to call *rmkdir(server_number, path, mode, reply_struct);* which will be used by the *client RPC intermediate stub* to first marshall the arguments into a pair of in and out *structs* that are sent to the corresponding, "real" RPC call defined as, say, *rpc_mkdiri(in, ret).*

After this call is made the RPC generated client stub takes over and continues further marshalling before sending the packet out over the network. Now for certain custom data types such as the GIGA+ header table, a custom marshalling support layer needs to be provided which would be used to convert the primitive data types into a standard notation.

Finally once the call reaches the server, the underlying RPC library will invoke the corresponding call to an intermediate layer called as the Service Callout layer. This RPC layer is structured in such a way that every unique RPC call received, merely calls out the corresponding service routine that will process the call. For example when *rpc_svc_mkdir()* reaches the server, the service callout layer will call out *giga_svc_mkdir()* routine of the server code.

This type of design has several advantages such as:

- Abstracting the underlying RPC implementation from the server routine processing
- Support for multi-threading
- Allows for extensions to the RPC layer on the server side to do some call independent processing

Multi-threading support

GIGA+ RPC layer is designed to have full multi-threaded support. This is because GIGA+ servers need to support multiple clients in parallel and this cannot be done if the RPC layer serializes all the calls that it receives. Hence, in current design, every call received at the server spawns a new thread in whose context the call gets serviced.

3.5 Naming

The GIGA+ client program mounts a FUSE file system at specified mount point which is taken as a command line argument to the GIGA+ client program.

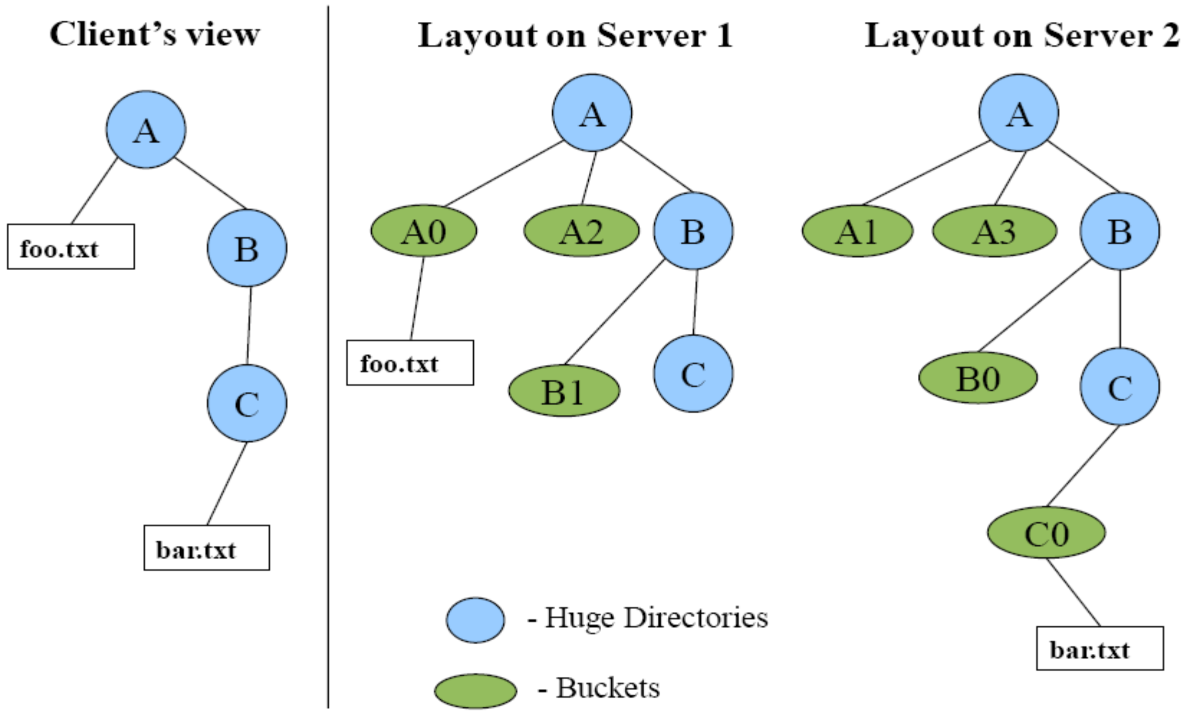


Figure 9: GIGA+ Filesystem Hierarchy

The GIGA+ servers export a mount point where files and directories are physically stored. The mount point exported by each server need not be the same however, when a server starts up, the mount point exported by the server should be empty to avoid any conflicts with the GIGA+ file system.

The GIGA+ file system enforces a common directory structure across all severs. In other words a client issued mkdir is sent to all servers. This is done to enable ease of namespace management. As indicated in the figure below. Huge Directories are replicated across all servers. This enables common pathname manipulations across all servers.

Buckets follow a naming convention of <parent_huge_dirname_special_string_bucket#>. Buckets within the mount point of the file system exported by the server are given a special name which is <ROOT_special_string_bucket#>.

3.6 Configuration Files

In the GIGA+ file system, the GIGA+ client and the GIGA+ servers each have their own configuration file which is required for initialization. The GIGA+ client needs to know about all GIGA+ servers and the machines they are running on. The GIGA+ servers also need to know about each other as they need to communicate during a "split".

The mount point exported by the server and the maximum bucket capacity is also specified in the Servers configuration file.

Currently the Client and Server's configuration file need to be manually created before starting the servers and are taken as command line arguments.

4 GIGA+ Implementation Details

This section discusses the parts of GIGA+ that have been implemented in the current system. In this regard we talk about some fine granularities regarding specific implementation details. Further we also discuss certain design and implementation issues. Since this is an on going project, we discuss the current status of this project. Finally we talk about pending work that needs to be accomplished in order to make user space GIGA+ a complete deployable solution.

4.1 GIGA+ Implementation

The current GIGA+ implementation supports multiple GIGA+ clients and multiple GIGA+ servers. The number of Servers is determined by the Configuration File for the client and servers. Currently the GIGA+ servers can be deployed on ext3, NFS or any POSIX compliant file system. The current implementation supports concurrent applications on the GIGA+ client and GIGA+ servers.

4.2 GIGA+ Indexing Data-Structures

Due the close association of the user land GIGA+ indexing scheme with the state maintenance, this section will talk about the data structures used on the client and the server to perform the indexing operations.

GIGA+ clients and servers maintain nearly identical data structures for operation the common sections of which is described in the figure below

As shown in diagram below both clients and servers maintain state for huge directories that they know of in a hash table with lookups done on the relative path names to these directories. The most essential component of these nodes is the header table structure which stores the bitmap, radix and home server that are used in the GIGA+ indexing scheme while locating files. Chaining is used to handle hash collisions. New nodes are always inserted at the end of the chain.

Servers in addition to storing information about the header tables also store an array of pointers to bucket attribute nodes which primarily store the file count to manage the number of files that the buckets can hold before they split and also locking primitive to handle concurrency with bucket level locks.

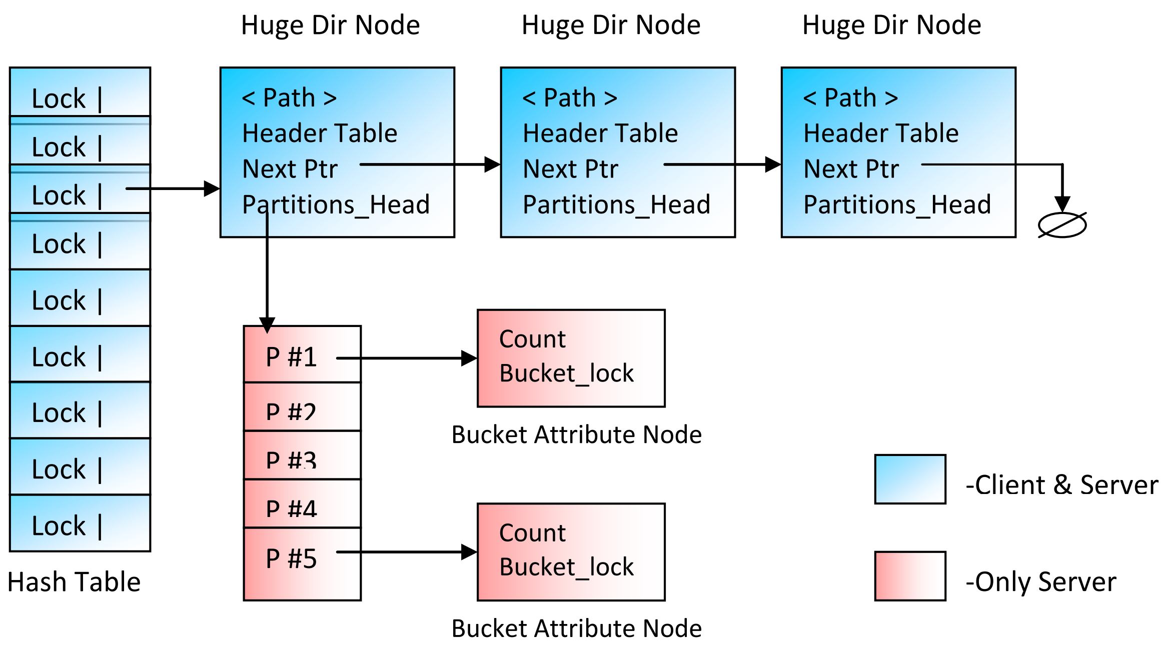


Figure 10: GIGA+ Data Structure. A hash table with chaining is used to store huge directory nodes. Lookups are done based on the hash of the relative path names of the directories. Servers additionally store an array of pointers to bucket nodes that contain locks. This implementation supports a configurable Hash Table size.

4.3 Scalable Directory Support (Splitting)

Current Implementation with configurable values for Bucket Capacity and Bitmap size enables Scalable Directory Support across N GIGA+ servers. We currently use a 160 Bit SHA-1 hash, which easily allows for fair distribution of files across buckets.

Currently we comfortably support a million file entries. However, with the current implementation we support a maximum radix of 32 which can support a maximum of 2^{32} buckets.

4.4 Nested Huge Directories

Current implementation supports arbitrary number of nested huge directories.

4.5 Small Directories

The Directory name is hashed using a 160 Bit SHA-1 hash. Home server for a particular directory is selected using this hash. This enables parallel access to available state of small directories spread over N servers.

4.6 Concurrency

Currently, we support bucket level locks that enable fine grained concurrency control allowing for simultaneous access to different buckets of the same directory. This enable greater level of concurrency as compared to directory level locks.

4.7 Hot-Plug and Play Support for Clients

A new client can be added into the system wide configuration without having to build a lot state during initialization. A new client only pre-fetches knowledge about the partition distribution of the mount points from the Home Server of the mount point. Following this, state for all other existing Huge Directories is built following a lazy approach over a period of time.

4.8 Client Caching

Client caches state only for Huge Directories and not for files. This cache is valid only for the duration of the GIGA+ client's session.

4.9 Filesystem API's Supported

Current implementation supports the following file system API's as in table 1.

API Name	GIGA+ Client Prototype	GIGA+ Server Prototype
Get attributes	giga_clnt_getattr(path,stat)	giga_svc_getattr(path,stat)
Make directory	giga_clnt_mkdir(path,mode)	giga_svc_mkdir(path,mode)
File Open	giga_clnt_open(path,mode)	giga_svc_open(path,mode)
Make Node (File Create)	giga_clnt_mknod(path, mode, dev)	giga_svc_mknod(path, mode, dev)
Read From File (un-tested)	giga_clnt__read(path,buf, size, offset)	giga_svc__read(path,buf, size, offset)
Write to File (un-tested)	giga_clnt_write(path,buf,size, offset)	giga_svc_write(path,buf,size, offset)

Table 1: GIGA+ Filesystem API's

4.10 Implementation Issues

4.10.1 Server State Persistence

In the current implementation the Server does not flush its in-memory data structures to persistence storage to support crash recovery.

4.10.2 Bitmap

Currently we support a fixed sized packet going over the network. This means that the bitmap must always be sent across for all file system operations between clients and servers. While this enable consistent updates for clients, it also incurs an overhead of having to send the bitmap every time. Frequent bitmaps updates ensure that the client's bitmap is not very stale which improves the chance of contacting the right server. However, to support huge number of buckets the bitmap grows significantly. For instance to support a maximum radix value of 32, we need a bitmap of 512MB. As an optimization, we could only send fixed parts of a bitmap with offset and length parameters.

4.10.3 Directory Replication

Currently we replicate the all Huge Directories across all servers. This provides common global namespace which enables ease of path name manipulations. However the need to create directories across all servers on a mkdir imposes an overhead as it requires N serial RPC's.

4.10.4 Synchronized Huge Directory State Across Servers

With current implementation, the bitmap is always sent across using RPC communications. When a sever splits and contacts another server it can potentially synchronize its bitmap with the server it is contacting, thus sharing its knowledge about the distribution of directories across two servers. The splitting server can potentially update the client with this knowledge as well. Overall, this keeps the clients and server more up to date.

4.10.5 FUSE Data Path Optimization for Read/Write Operation

FUSE data path may induce an unnecessary overhead for file i/o. However empirical have proved otherwise. Refer to the experimental section for details.

5 Experiments

5.1 GIGA User-space Benchmarking

User land GIGA+ has been successfully tested across a variety of configurations. There were several goals of these experiments including

- Measuring and comparing the insert rates for files with varying number of servers and bucket sizes
- Correctness and fault-free operation
- Sustained file insert rates

5.1.1 Experimental Setup

Hardware Setup

Experiments were run on machines running Ubuntu Server 7.10 with a 2.6.22.14 Linux kernel. The machines had 36GB hard drive with a 1 GB RAM and a 2 GHz CPU.

Network Setup

All test machines were on a shared Ethernet segment. All machines had their NICs turned on and had a RPC Portmapper running to service RPC requests.

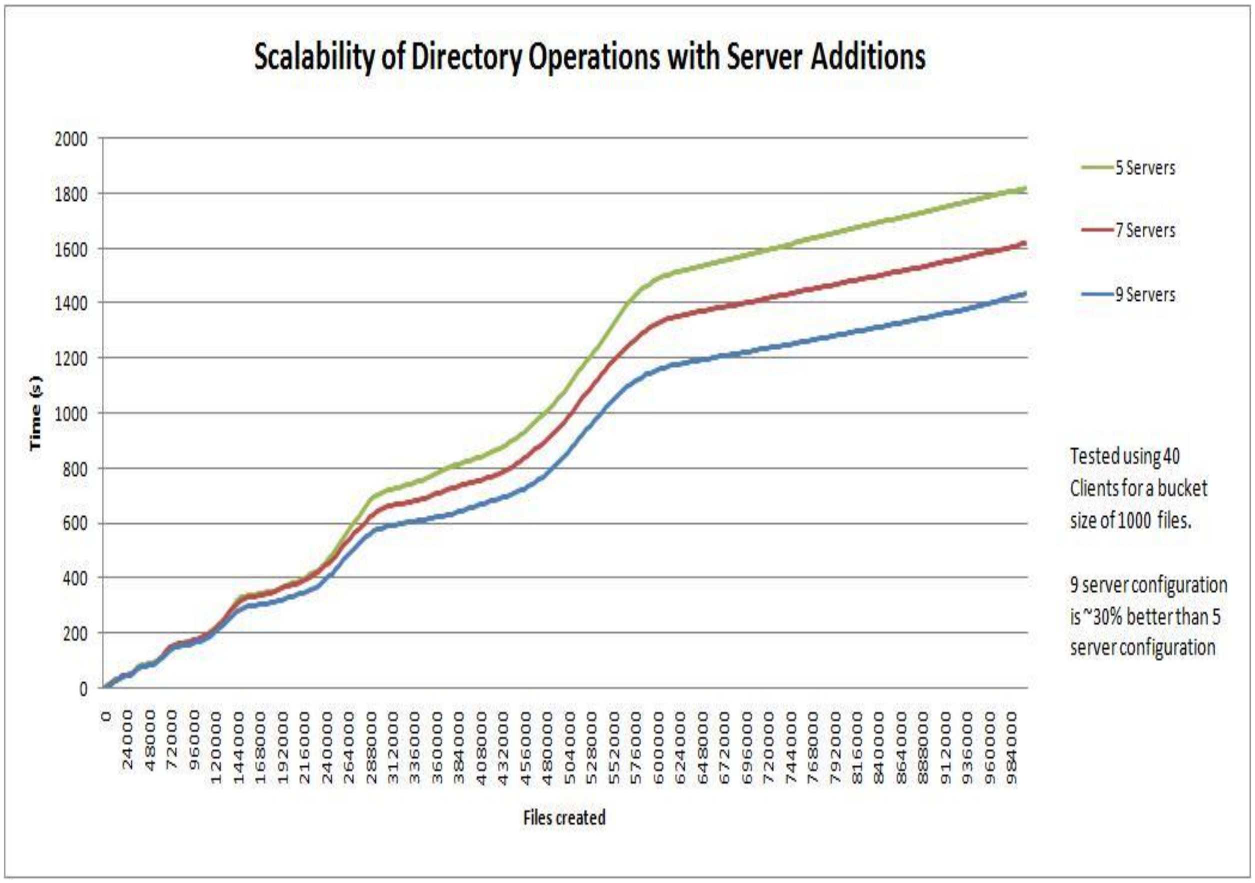
5.1.2 Experimental details

- Inserts were done by a program issuing open calls with random filenames
- The time was measured using gettimeofday() for every **100 files**
- The Number of clients was kept constant at **40**. The reasoning behind this is that these many clients were needed in order to keep the servers busy. We achieved this by using 4 machines each with 10 clients having their own FUSE mount points. Thus we ensure that the client machines are saturated doing nothing but generating metadata requests.
- Two types of experiments were conducted
    - **TEST A**: Varying number of servers (5, 7, 9): This is the most significant large scale experiment that demonstrates the **scaling of directory operations with addition of servers**, bucket sizes were limited to 1000 entries before a split.
    - **TEST B**: Varying number of buckets, keeping servers and clients constant: This experiment was done to demonstrate the **sustained rate of inserts** and the overhead introduced due to splitting.
- All test create 1 million files, a number chosen for mere convenience, as the system is capable of creating much more.

5.1.3 Experimental Results

5.1.3.1 TEST A: Scalability of User land GIGA+ directory operations with server additions

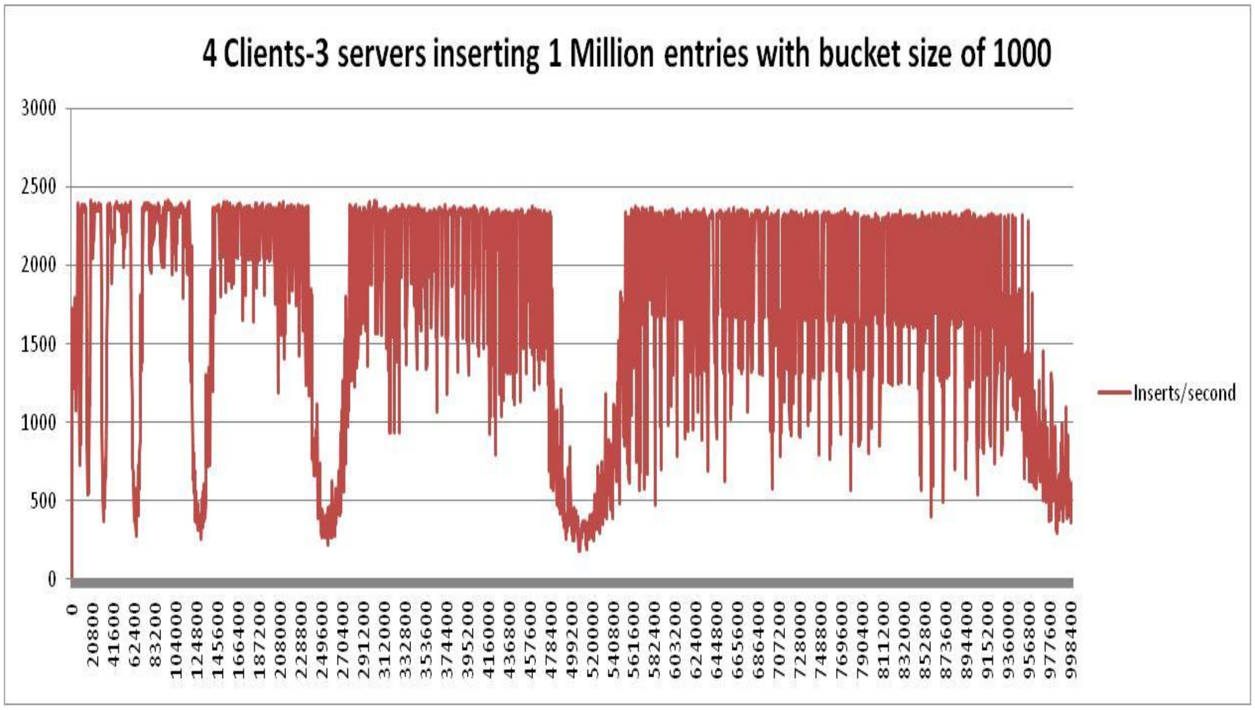
Graph 1 demonstrates how GIGA+ takes advantage of additional metadata servers. It can be seen that it is seen that insert performance linearly improve with server additions. With 9 servers, the time to create a million files is approximately **30% better** than that with 5 servers. The cause behind this effect is that as buckets split, they get spread across a larger number of servers, thereby taking advantage of the increased level of parallelism.



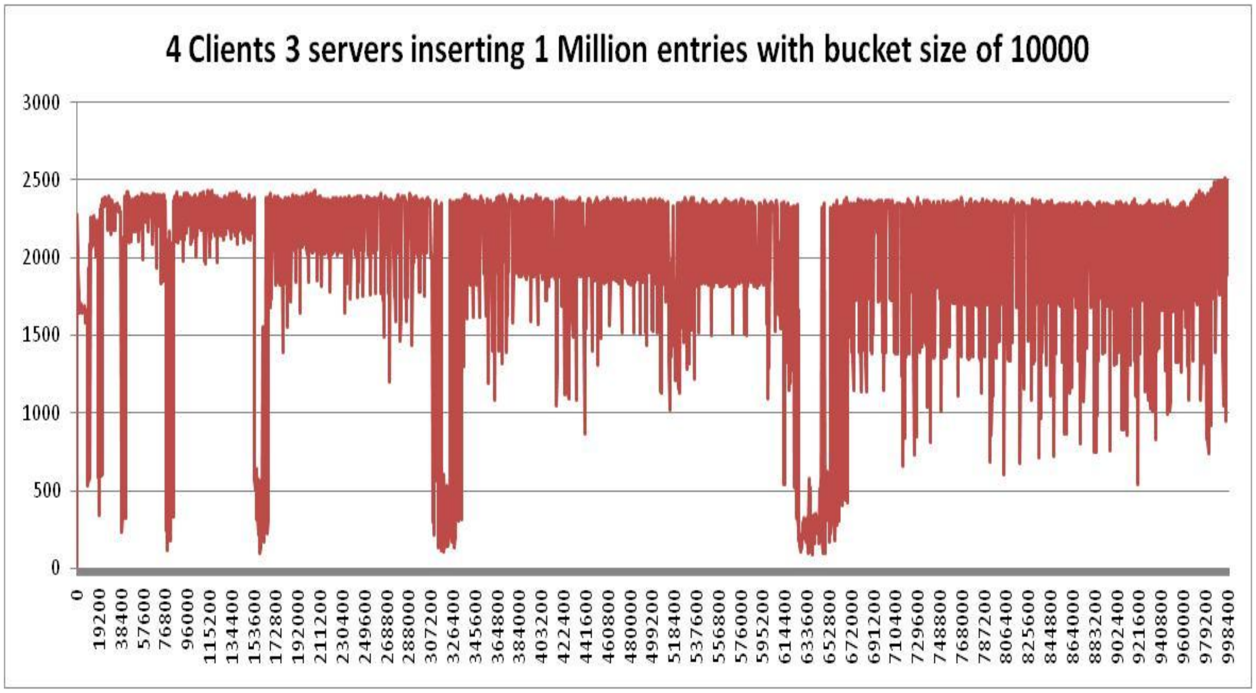
Graph 1: Directory performance scaling with different number of Servers

5.1.3.2 Test B: Insert rate sustenance and splitting overhead

The following experiment demonstrates the sustenance of the client insert rates up to a million file inserts. The following two graphs show the variance of the insert rates as a million files are created. It can be seen that the graphs have periodic "dips" which correspond to the periods where client operations are stalled owing to bucket splitting. One can also observe that in graph 3, as the bucket sizes are increased to 10,000 from 1000 the dips are shorter in duration due to fewer split operations.



Graph 2: Insert rate substance graph. The above graph shows the insert rates achieved for our system with 4 clients and 3 servers. All the 4 clients were started simultaneously and time taken by each client was captured. The dips may have been caused as a result of splitting of buckets in the system.



Graph 3: Insert rate substance graph. The above graph shows the insert rates achieved for our system with 4 clients and 3 servers. All the 4 clients were started simultaneously and time taken by each client was captured. The dips may have been caused as a result of splitting of buckets in the system.

5.2 FUSE Data path experimental setup

The path taken by FUSE has an extra hop for every file system call under the FUSE mount point. The experiments were conducted to measure the overhead introduced by FUSE for this extra hop.

5.2.1 Hardware Setup

Three machines each with 4GB main memory and 160GB SATA Hard drive were used to perform this experiment.

1. One machine where the actual file resides (server)
2. Second machine which is the client where the actual read measurement is being conducted.
3. Third machine is used to perform "dummy" reads. This is done to ensure that the file is cached at the server and hence eliminating the disk I/O bottleneck.

5.2.2 Network Setup

The clients mounted NFS over RDMA, IPoIB and Ethernet.

5.2.3 Test Setup

1. To measure the regular read performance *dd* is issued to read a file on the NFS mount point
2. To compare the read performance with FUSE overhead, *dd* is issued to read a file on a FUSE mount point which uses the NFS mount point as the underlying storage.
   - The server machine exports a folder to the client(s) via NFS-over-RDMA, NFS-over-IPoIB, and NFS-over-Ethernet.
   - The test begins with the "dummy" client issuing multiple reads to fill the cache at the server. This is done to eliminate the disk bottleneck while performing reads from the test client.
   - The test client issues a *dd* command to read files at different block sizes. The first read at the test client fetches the file from the server, consecutive reads use the local FS cache to read the file (If the file fits into the client cache).

Tests were performed with file sizes of 1GB, 2GB and 4GB with different block sizes ranging from 512 bytes to 4MB. **The results show that the over head introduced by FUSE is completely negligible.**

The following section contains a list of graphs comparing the bandwidth measured by issuing reads on a FUSE mount point with the underlying storage being the NFS mount point and the native FS reads on the NFS mount point.

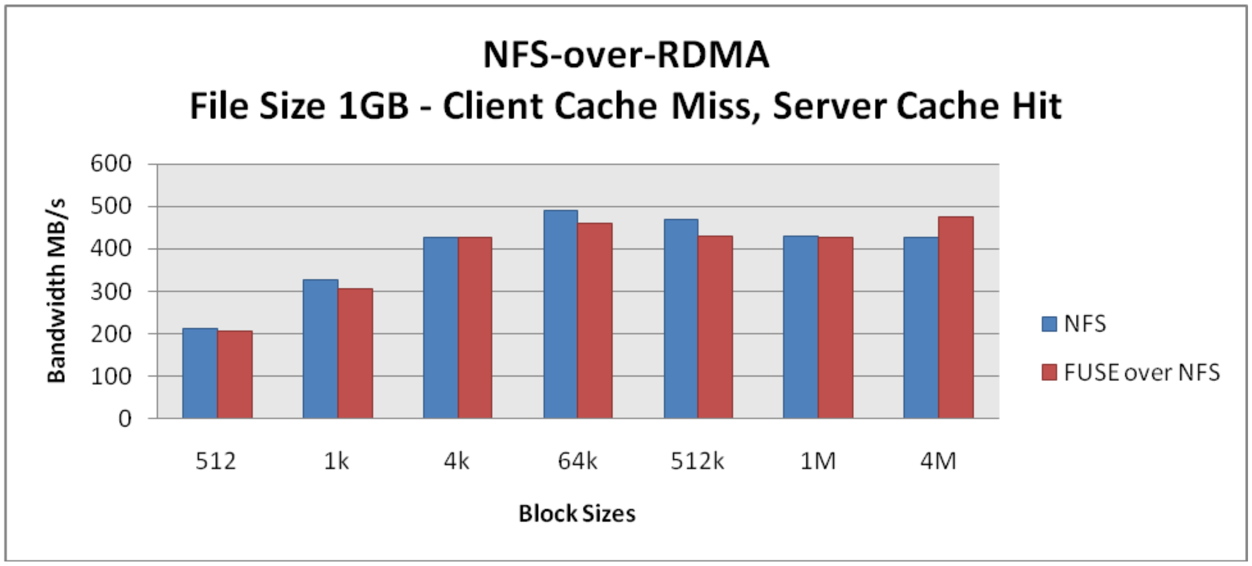
1. **Client cache miss, Server cache hit** – In this setup, the file does not exist in the client's cache. By issuing calls from a "dummy" client to the same file on the server, the server cache is filled. Hence the bandwidth measured is for a file that is being read from the servers' cache. It may be noted that a file that does not fit into the server's cache sees a hit in the read performance as all the reads go to disk when a client requests the file.
2. **Client cache hit** – In this setup, the reads are issued at the client after the first read. The file is serviced from the client's cache, if the file fits into the cache. If the file does not fit in the client's cache, the read requests go to the servers for all the consecutive read requests.

5.2.4 FUSE Data path experimental results (Read Performance)

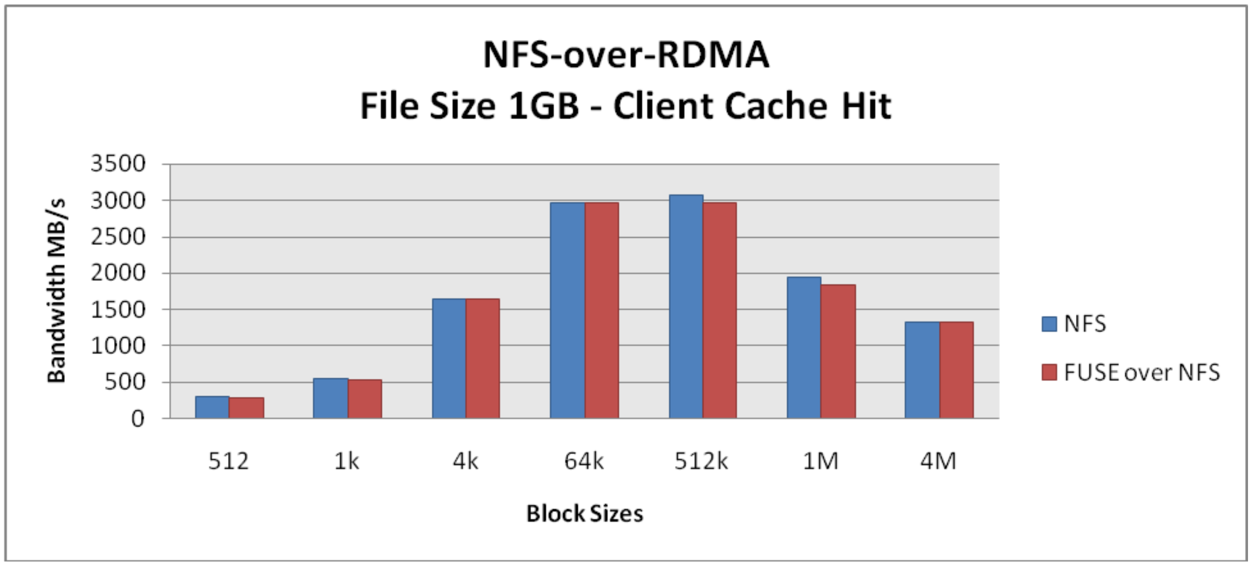
5.1.3.3 NFS over RDMA

File Size 1GB

The following graphs show the comparison of read performance for a 1GB file measured using FUSE mounting NFS-over-RDMA and NFS-over-RDMA directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



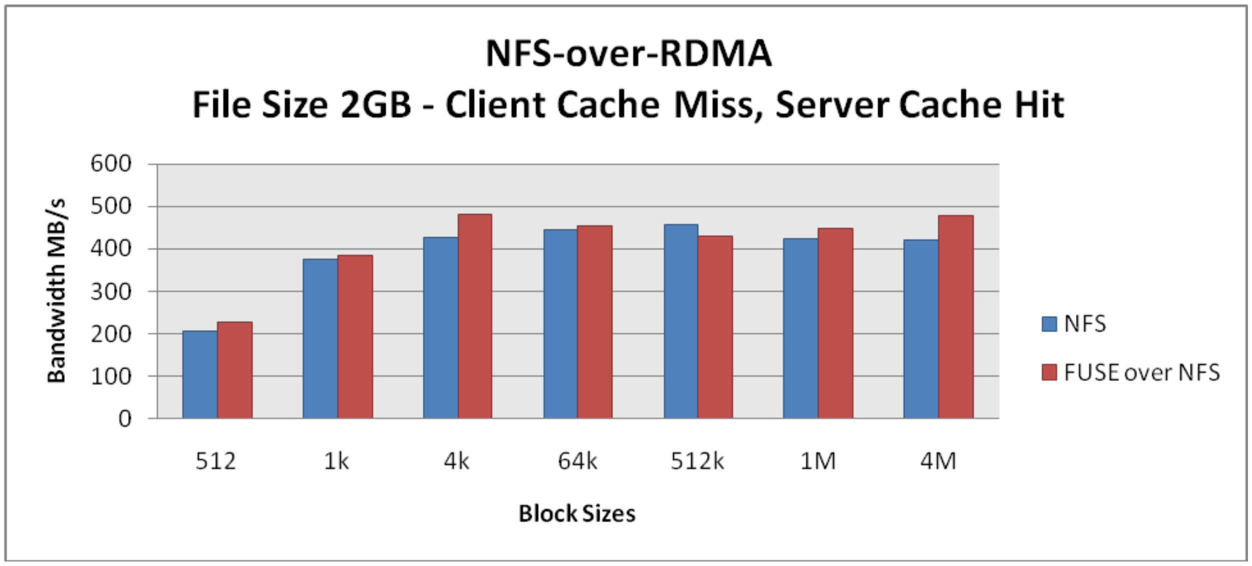
Graph 4: 1GB file read via NFS-over-RDMA for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server.



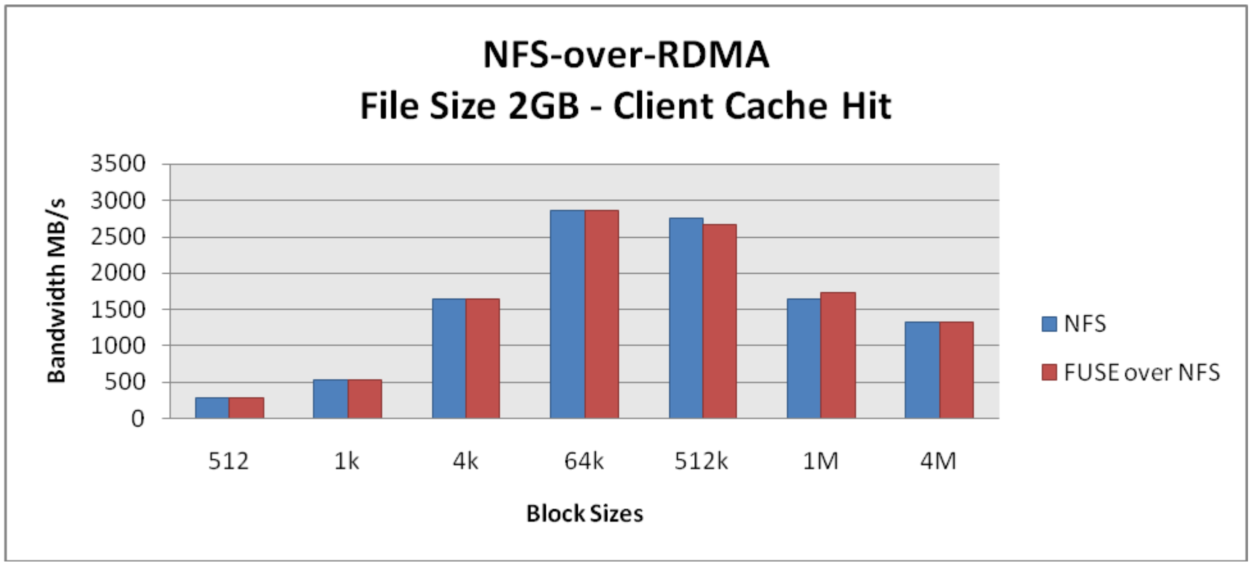
Graph 5: Consecutive 1GB file reads via NFS-over-RDMA for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance.

File size 2GB

The following graphs show the comparison of read performance for a 2GB file measured using FUSE mounting NFS-over-RDMA and NFS-over-RDMA directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



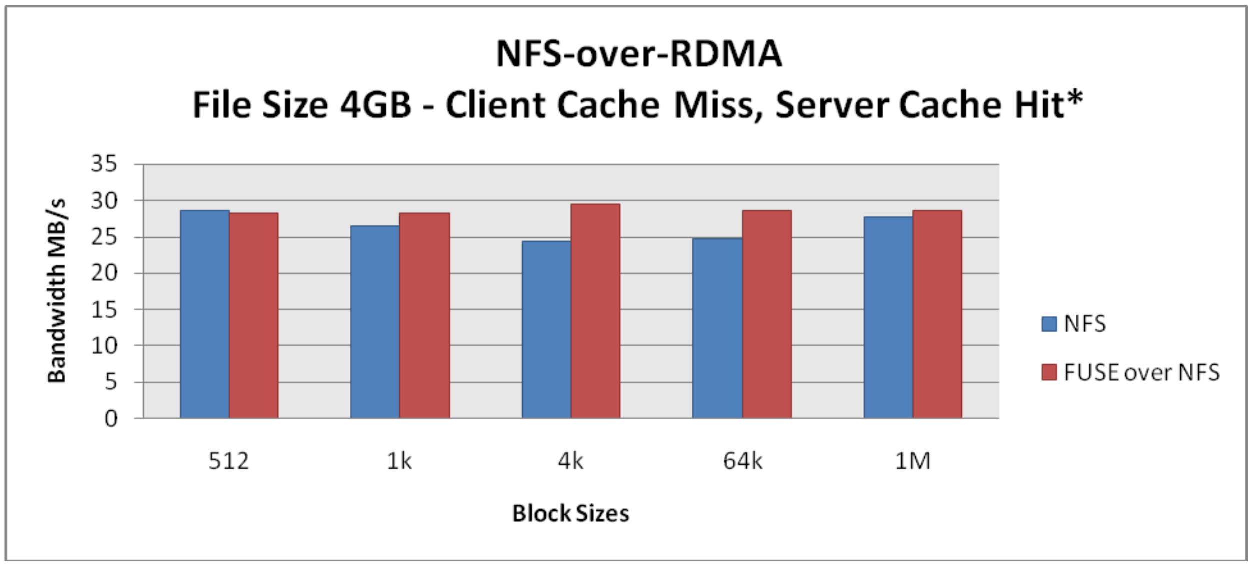
Graph 6: 2GB file read via NFS-over-RDMA for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server



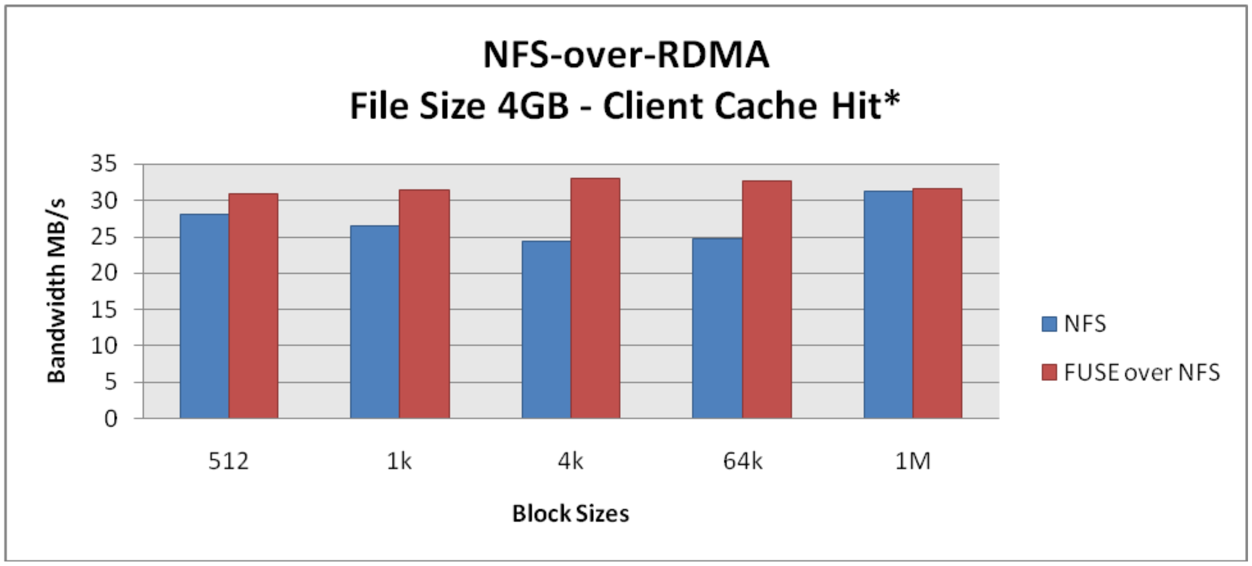
Graph 7: Consecutive 2GB file reads via NFS-over-RDMA for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance

File size 4GB

The following graphs show the comparison of read performance for a 4GB file measured using FUSE mounting NFS-over-RDMA and NFS-over-RDMA directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE. A hit in the read performance when compared with the 1GB and 2GB case may be because the file did not fit into the server's cache and hence servicing the entire file from disk.



Graph 8: 4GB file read via NFS-over-RDMA for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server

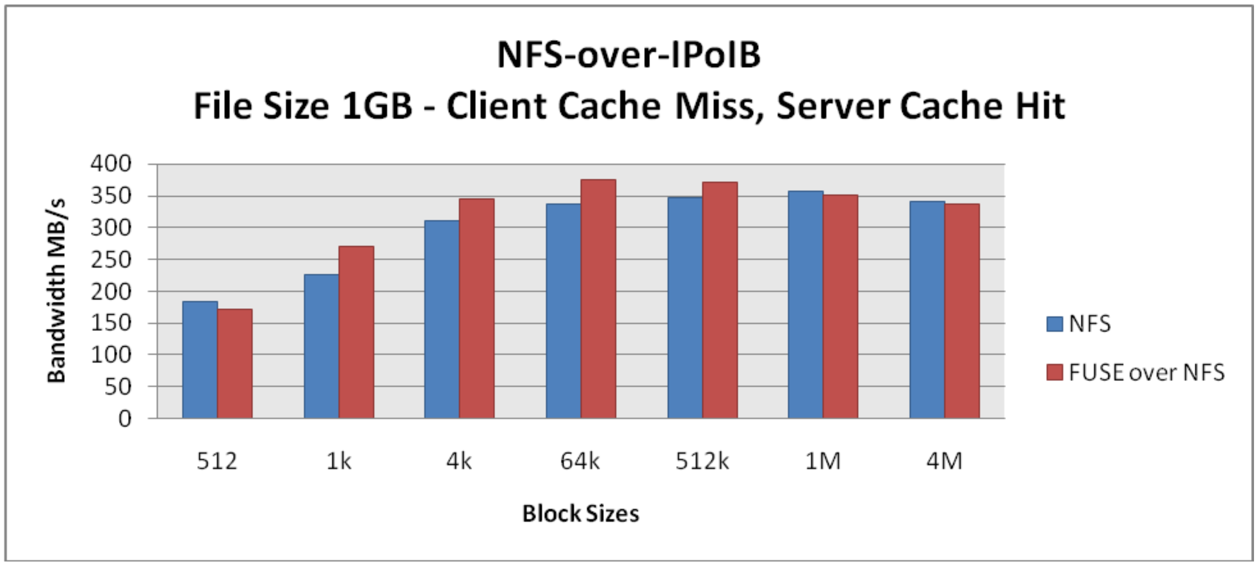


Graph 9: Consecutive 4GB file reads via NFS-over-RDMA for different block sizes from the test client. Since the file size is greater than the clients cache size, the client needs to contact the server for every read. Hence, unlike the previous two scenarios a boost in the performance for a client cache hit is not seen for the case where the file size is greater than the cache size.

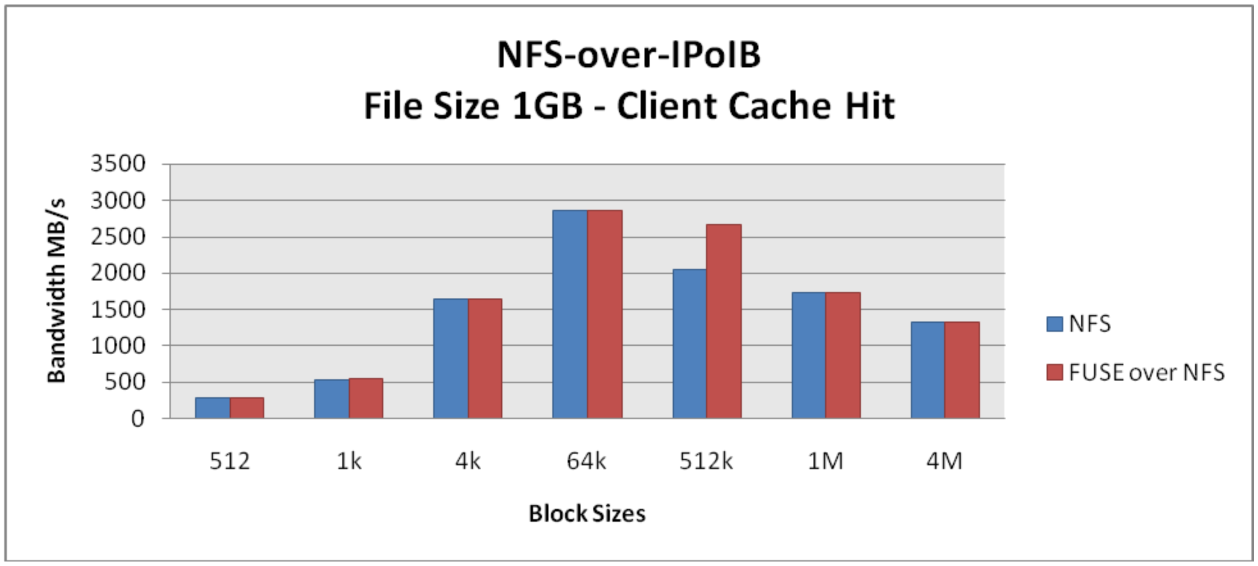
5.3.4.2 NFS-over-IPoIB

File size 1GB

The following graphs show the comparison of read performance for a 1GB file measured using FUSE mounting NFS-over-IPoIB and NFS-over-IPoIB directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



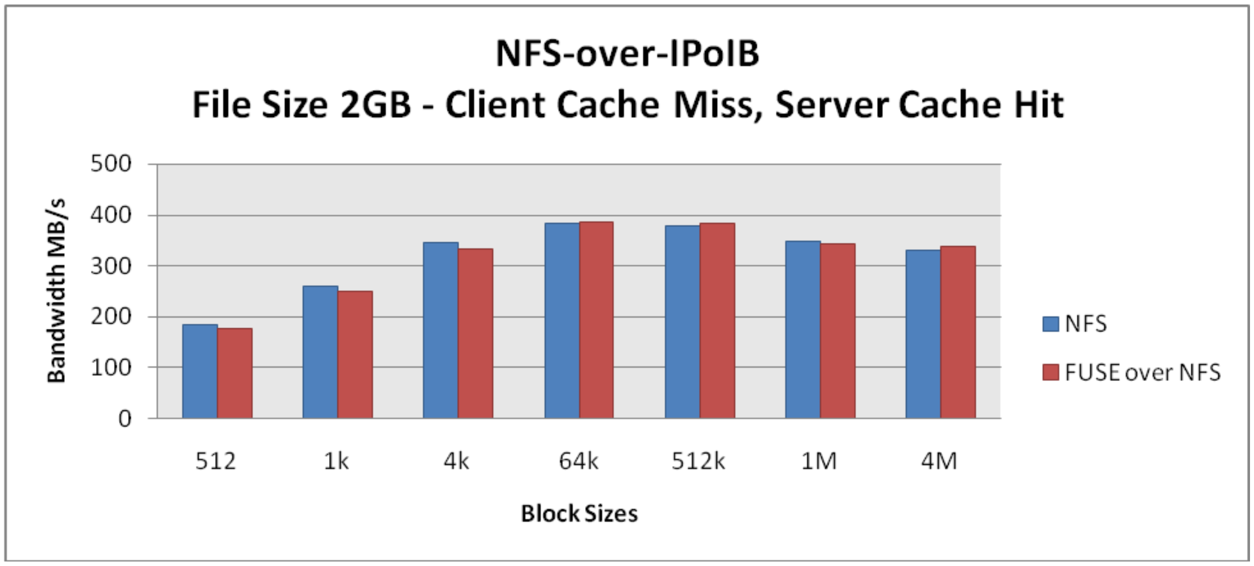
Graph 10: 1GB file read via NFS-over-IPoIB for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server.



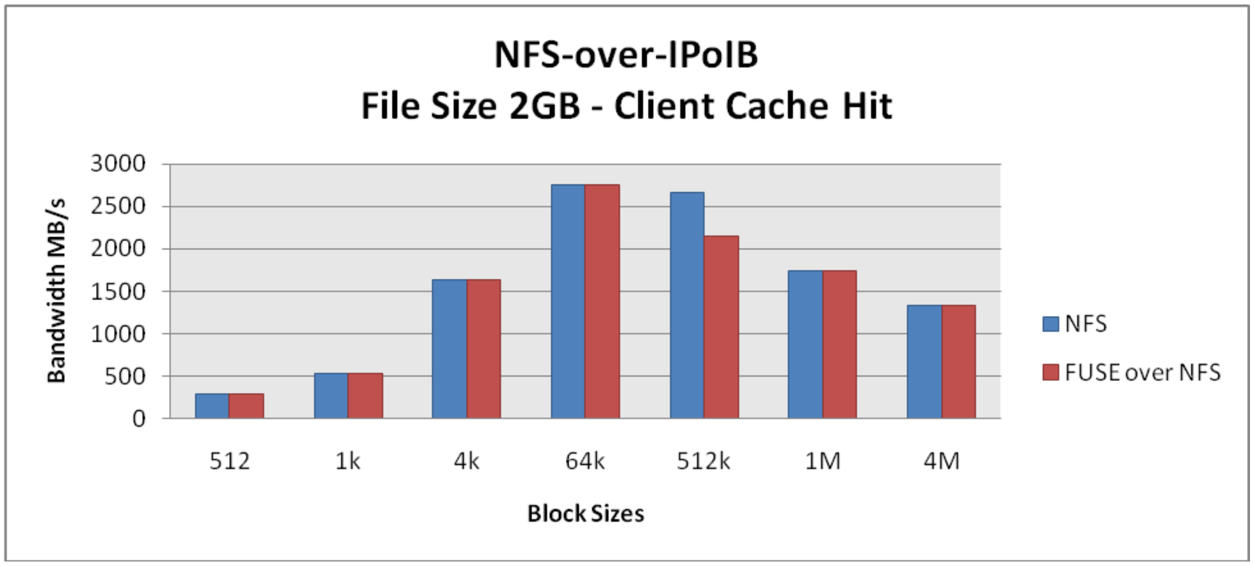
Graph 11: Consecutive 1GB file reads via NFS-over-IPoIB for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance.

File size 2GB

The following graphs show the comparison of read performance for a 2GB file measured using FUSE mounting NFS-over-IPoIB and NFS-over-IPoIB directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



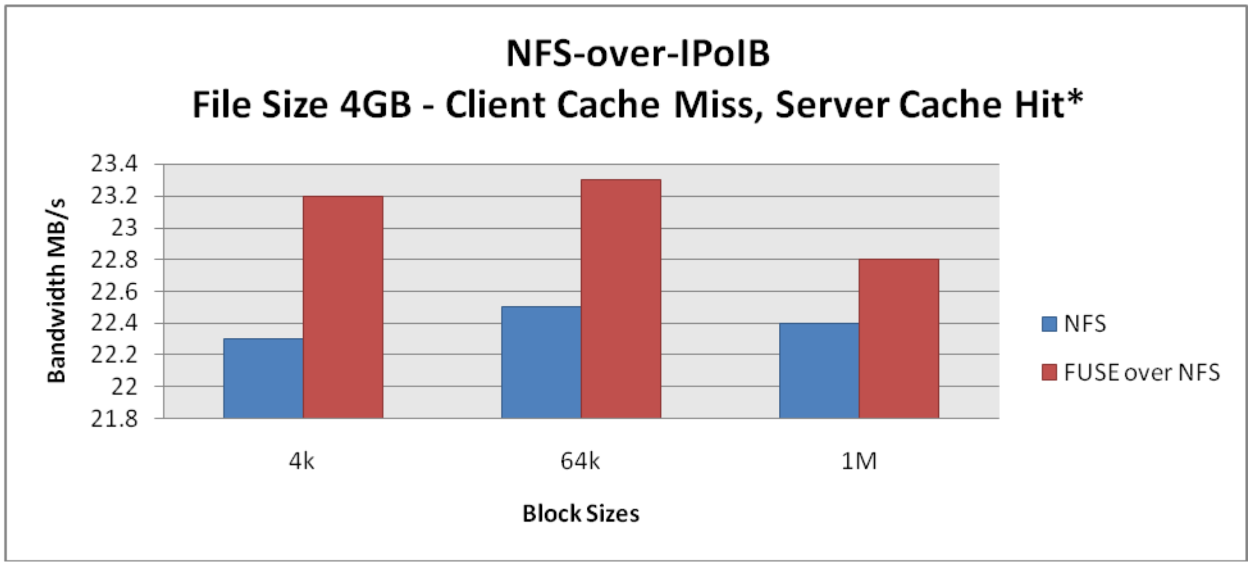
Graph 12: 2GB file read via NFS-over-IPoIB for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server



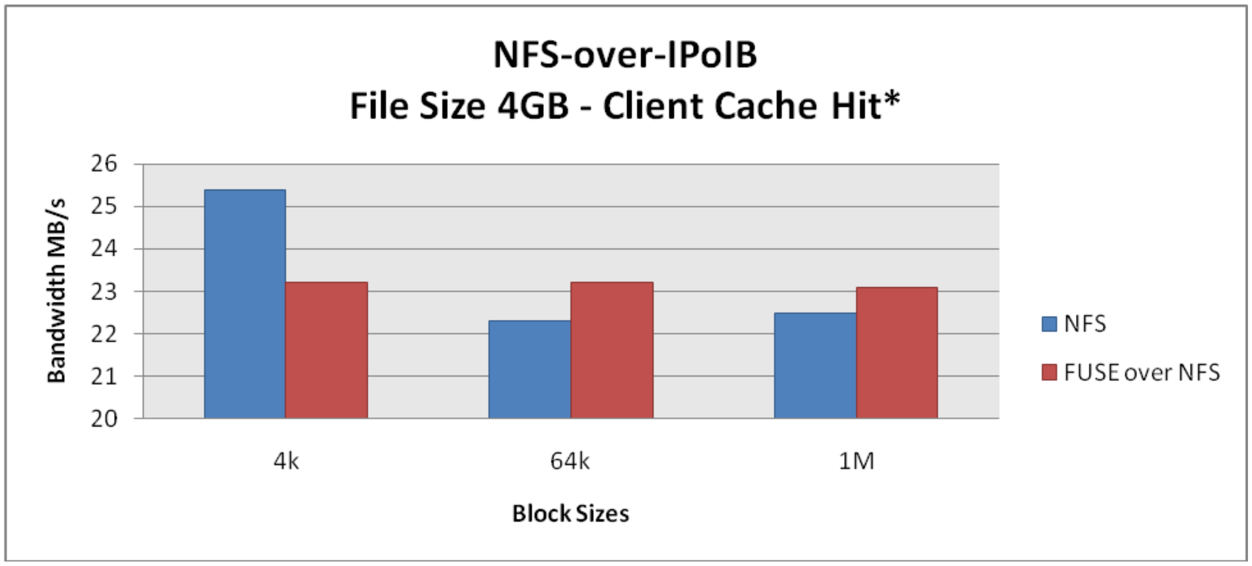
Graph 13: Consecutive 2GB file reads via NFS-over-IPoIB for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance.

File size 4GB

The following graphs show the comparison of read performance for a 4GB file measured using FUSE mounting NFS-over-IPoIB and NFS-over-IPoIB directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE. A hit in the read performance when compared with the 1GB and 2GB case may be because the file did not fit into the server's cache and hence servicing the entire file from disk.



Graph 14: 4GB file read via NFS-over-IPoIB for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server

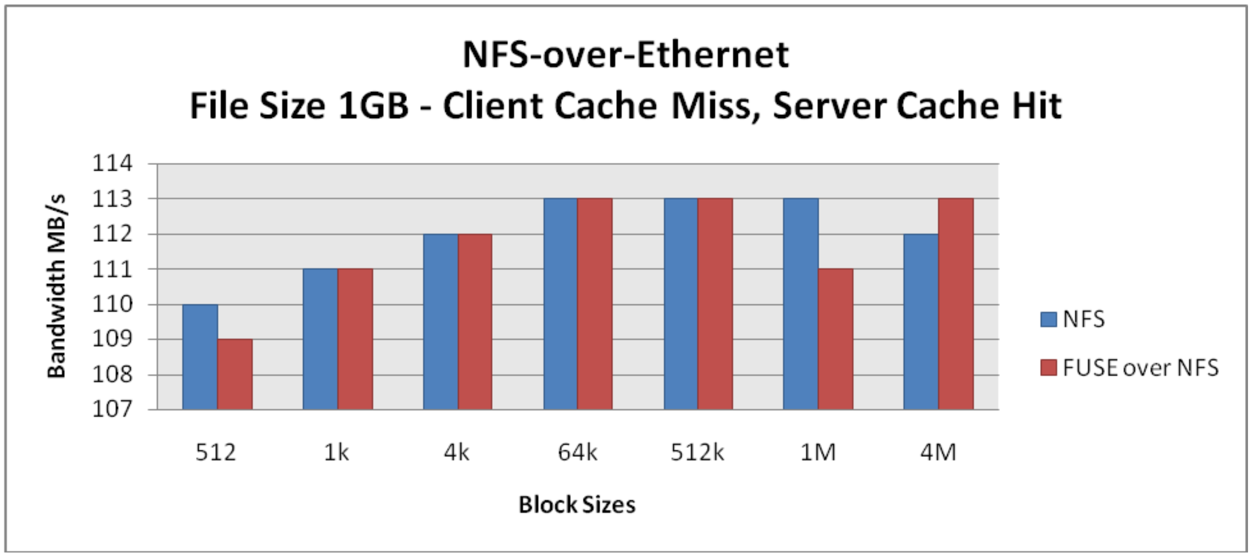


Graph 15: Consecutive 4GB file reads via NFS-over-IPoIB for different block sizes from the test client. Since the file size is greater than the clients cache size, the client needs to contact the server for every read. Hence, unlike the previous two scenarios a boost in the performance for a client cache hit is not seen for the case where the file size is greater than the cache size.

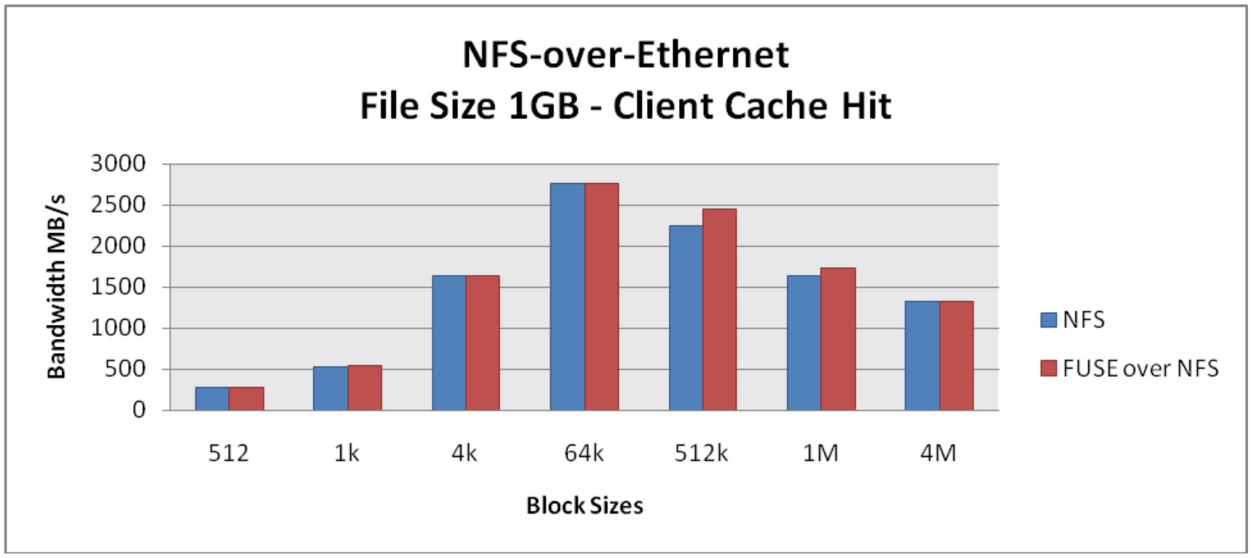
5.3.4.3 NFS-over-Ethernet

File size 1GB

The following graphs show the comparison of read performance for a 1GB file measured using FUSE mounting NFS-over-Ethernet and NFS-over-Ethernet directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



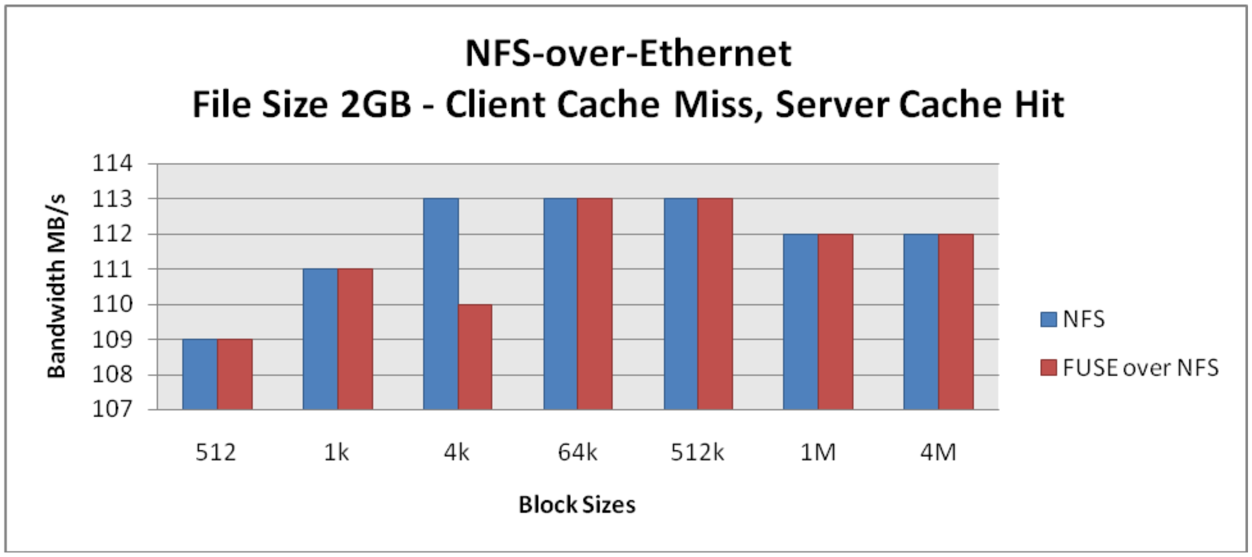
Graph 16: 1GB file read via NFS-over-Ethernet for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server



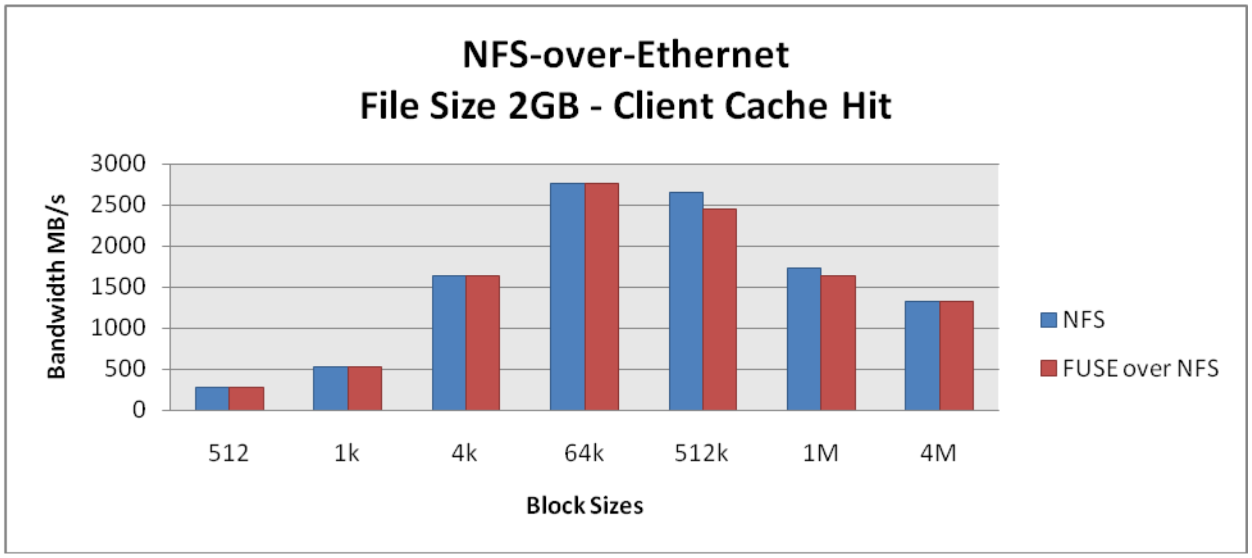
Graph 17: Consecutive 1GB file reads via NFS-over-Ethernet for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance.

File size 2GB

The following graphs show the comparison of read performance for a 2GB file measured using FUSE mounting NFS-over-Ethernet and NFS-over-Ethernet directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE.



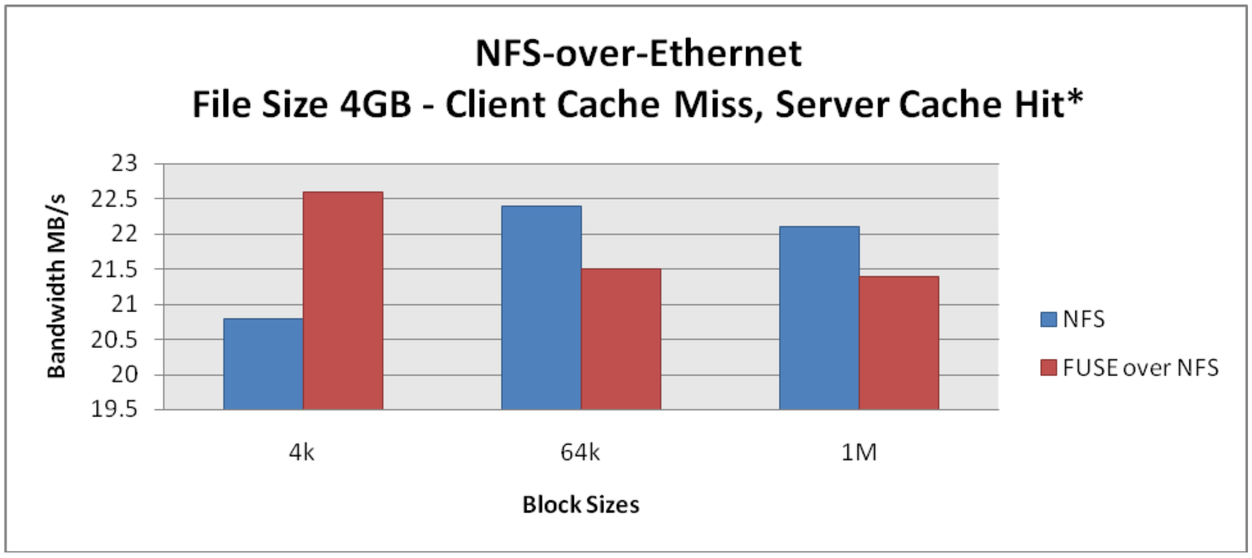
Graph 18: 2GB file read via NFS-over-Ethernet for different block sizes from the test client. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server



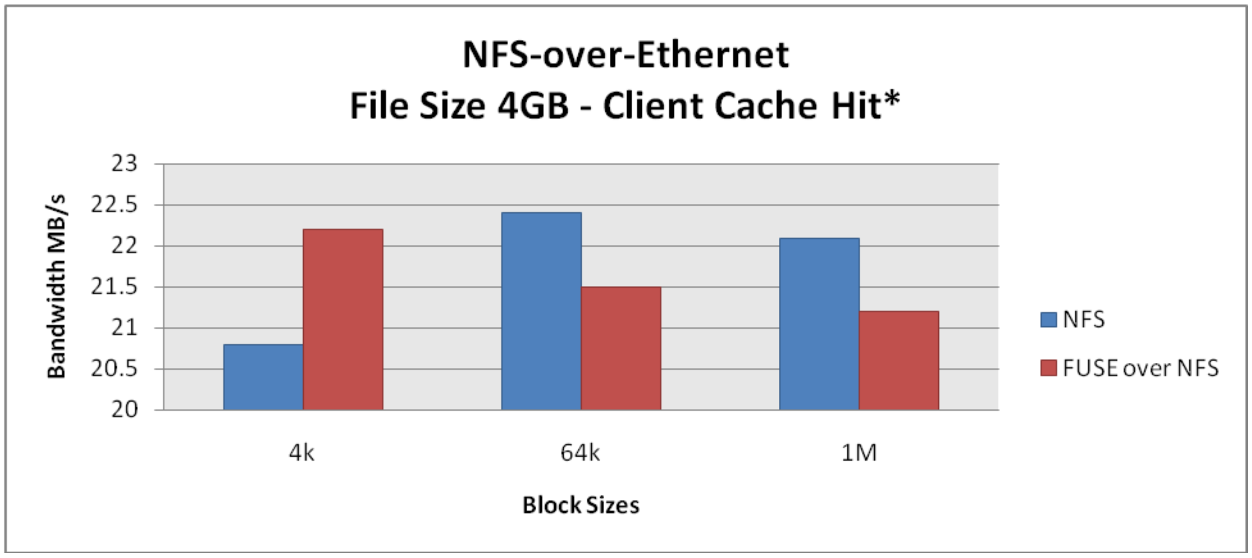
Graph 19: Consecutive 2GB file reads via NFS-over-Ethernet for different block sizes from the test client. The read request is being serviced from the clients cache completely and hence the increase in performance.

File size 4GB

The following graphs show the comparison of read performance for a 4GB file measured using FUSE mounting NFS-over-Ethernet and NFS-over-Ethernet directly. It can be observed that the results show that there is a negligible overhead incurred due to the extra hop introduced by FUSE. A hit in the read performance when compared with the 1GB and 2GB case may be because the file did not fit into the server's cache and hence servicing the entire file from disk.



Graph 20: 4GB file read via NFS-over-Ethernet for different block sizes. The file is cached at the server by performing a read from a non-test client and the test client issues the read to the server



Graph 21: Consecutive 4GB file reads via NFS-over- Ethernet for different block sizes. Since the file size is greater than the clients cache size, the client needs to contact the server for every read. Hence, unlike the previous two scenarios a boost in the performance for a client cache hit is not seen for the case where the file size is greater than the cache size.

6 Conclusions

Scaling metadata performance is more complex than scaling raw I/O performance, and with distributed metadata the complexity increases further. Distributed parallel file systems like PVFS strive to provide significant I/O parallelism. However they still fall short when it comes to scaling meta-data performance.

By adopting the GIGA+ concept we have successfully demonstrated that scaling meta-data performance is possible and if used in conjunction with a parallel file system can serve to significantly alleviate the problem of scalable directories.

6.1 Current Status

As the projects stands, the GIGA+ user space file system has been implemented to work on ext3 file systems. The current implementation supports scaling of directories through splitting. We support an arbitrary number of directory levels.

We also support creation of a million files and possibly much more. The current implementation completely implements the GIGA+ indexing scheme as in [2]. Key file system API's including directory and file creation have been implemented. The existing system can easily be extended to incorporate essential file system API's such as readdir. While current experimental results are already persuasive, the true potential of user space GIGA+ cannot be realized until a fully functional file system is implemented.

6.2 Motivation for Future Work

Experimental results are compelling enough to warrant extension to the existing system. Empirical results show that the user level GIGA+ implementation has potential to provide the necessary directory scaling service that GIGA+ promises.

Some key features that should be added to the user space GIGA+ file system

- Implementation of readdir
- Other File System API's
- Optimizations with data sent across the RPC's (i.e. the Bitmap)
- Deployment on a Parallel file system

7 References

[1] Private Communications with Garth Gibson, Milo Polte and Swapnil Patil

[2] Swapnil Patil, Garth Gibson, Sam Lang, Milo Polte: GIGA+ Scalable Directories for Shared File Systems (PDSW'07) http://www.pdl.cmu.edu/PDL-FTP/HECStorage/sc07-patil.pdf

[3] FUSE Sourceforge Page: http://fuse.sourceforge.net

[4] FUSE on Wikipedia http://en.wikipedia.org/wiki/Filesystem_in_Userspace

[5] AVFS: http://www.inf.bme.hu/~mszeredi/avfs/

[6] File Systems using FUSE: http://fuse.sourceforge.net/wiki/index.php/FileSystems

[7] ONC RPC: http://www.ietf.org/rfc/rfc1831.txt

[8] XDR: http://www.faqs.org/rfcs/rfc1014.html

[9] Amina Saify, Garima Kochher, Jenwei Hsieh, Onur Celebioglu: Enhancing High-Performance Computing Clusters with Parallel File Systems

[10] Ananth Devulapalli, Pete Wyckoff: File Creation Strategies in a Distributed Metadata File System. http://www.osc.edu/~pw/papers/devulapalli-create-ipdps07.pdf

[11] R. Fagin, J. Nievergelti, N. Pippenger, and H. R.Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. ACM Transactions on Database Systems, 4(3), Sept. 1979.

[12] PVFS2. Parallel Virtual File System, Version 2 http://www.pvfs2.org

[13] Swapnil Patil: Specification for the GIGA+ indexing technique March' 08

[14] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West. : Scale and Performance in a Distributed File System; Appears in the ACM Transactions on Computer Systems, Vol. 6, No. 1, Pages 51-81. February 1988