CITI Technical Report 05-02

Scaling NFSv4 with Parallel File Systems

Dean Hildebrand
dhildebz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

ABSTRACT

Large grid installations require global access to massive data stores. Parallel file systems give high throughput within a LAN, but cross-site data transfers lack seamless integration, security, and performance. The GridNFS project, aims to provide scalable, transparent, and secure data management as well as a scalable and agile name space. A key challenge in exporting a parallel file system with NFSv4 is to provide high performance without sacrificing consistency. This paper introduces extensions to the NFSv4 protocol to support parallel access. We implemented a prototype of our design and present experiments demonstrating its scalable architecture.

March 7, 2005

Center for Information Technology Integration
University of Michigan
535 W. William St., Suite 3100
Ann Arbor, MI 48103-4978

Scaling NFSv4 with Parallel File Systems

Dean Hildebrand
Center for Information Technology Integration
University of Michigan
dhildebz@eecs.umich.edu

Peter Honeyman
Center for Information Technology Integration
University of Michigan
honey@citi.umich.edu

Abstract

Large grid installations require global access to massive data stores. Parallel file systems give high throughput within a LAN, but cross-site data transfers lack seamless integration, security, and performance. The GridNFS project, aims to provide scalable, transparent, and secure data management as well as a scalable and agile name space. A key challenge in exporting a parallel file system with NFSv4 is to provide high performance without sacrificing consistency. This paper introduces extensions to the NFSv4 protocol to support parallel access. We implemented a prototype of our design and present experiments demonstrating its scalable architecture.

1. Introduction

Collaborations such as TeraGrid [1] allow global access to massive data sets in a nearly seamless environment distributed across several sites. The degree of transparency between sites can determine the success of these collaborations. Several factors affecting data access transparency are latency, bandwidth, security and software interoperability.

To improve performance and transparency within each site, the use of symmetric or asymmetric parallel file systems[1] is on the rise, allowing applications direct, concurrent and scalable access to a single file system. Parallel file systems allow storage systems to grow with storage needs and reduce management costs by aggregating all storage into a single framework.

Figure 1 shows a general model for the flow of data in this environment consisting of four primary components. The first component is storage, which can be anything from a SAN to a single directly attached disk. The second component is a set of metadata nodes that describe and control access to storage. The third component is a set of file nodes that provide a front-end for storage access. All data must flow from storage through these nodes. The fourth component is a set of application nodes that generate and analyze data. In a symmetric parallel file system, file nodes, metadata nodes, and application nodes exist on the same machine. In a conventional distributed file system, the file nodes (servers) and application nodes (clients) are distinct elements.

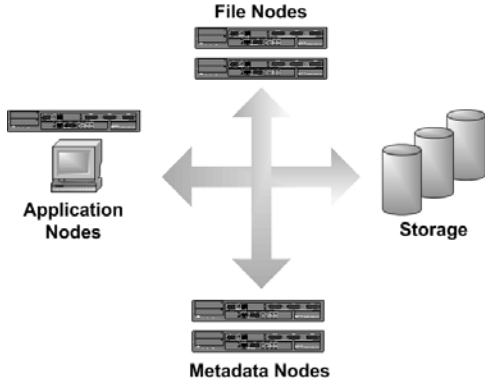

Figure 1. General data access model

This paper focuses on the third and fourth components of the model: file nodes and application nodes. The need for remote access from multiple operating systems, metadata scalability, and security and performance over the WAN often necessitates separating file nodes and application nodes.

The GridNFS project at the University of Michigan aims to facilitate the management and flow of large data sets in the grid. It aims to provide scalable, transparent, and secure data management as well as scalable and agile name space management for establishing and controlling identity in virtual organizations [2] and for specifying virtual organization data resources. To realize these two vital but missing capabilities, GridNFS extends the "best of breed" Internet technologies with established Grid architectures and protocols. The foundation for data sharing in GridNFS is NFS version 4 [3], the IETF standard for distributed file systems that is designed for security, extensibility, consistency, and high performance.

GridNFS allows researchers access to remote files and databases using the same programs and procedures that they use to access local files, as well as obviating the need to create and update local copies of a data set

[1] In symmetric file systems, nodes perform identical tasks. Asymmetric file systems assign distinct roles to nodes, e.g., metadata management, storage recovery, etc.

manually. To meet quality of service requirements across metropolitan and wide-area networks, GridNFS may need to use all available bandwidth provided by the parallel file system's file nodes. In addition, GridNFS must be able to provide parallel access to a single file from large numbers of clients, a common requirement of high-energy physics applications.

This paper discusses the challenge of achieving full utilization of a storage system's available bandwidth with NFSv4 and introduces extensions that allow NFSv4 to scale beyond a single server by distributing data access across file nodes in the remote data store. These extensions include a new server-to-server protocol and a file description and location mechanism. For the rest of this paper, we will refer to NFSv4 with these extensions as *Split-Server NFSv4*.

The remainder of this paper is organized as follows. Section 2 establishes the throughput scaling focus in this paper. Section 3 discusses scaling limitations of the NFSv4 protocol. Section 4 covers related work. Section 5 describes the NFSv4 protocol extensions in Split-Server NFSv4. Sections 6 and 7 discuss fault tolerance and security implications of these extensions. Section 8 provides performance results of our Linux-based prototype and discusses performance issues of NFS with parallel file systems. Section 9 is devoted to future directions and Section 10 concludes this paper.

2. Scaling I/O in NFS

NFS I/O consists of four major access models:

1. A single client accessing a single file.
2. A single client accessing multiple files.
3. Multiple clients accessing a single file.
4. Multiple clients accessing separate files.

To exhaust all available bandwidth when exporting a parallel file system, NFS depends on the parallel file system to receive or produce data at network speed or faster. Since storage networks generally have larger bandwidth capacity than the client network, a single client accessing a single file should receive very good performance. Gains in this area will be realized through increased disk and network bandwidths, as well as the resolution of issues discussed in Section 8.4.

This paper focuses on access models 3 and 4: increasing the aggregate throughput of multiple clients accessing a single file or separate files by balancing client load among file nodes. We assume distributed locking is provided by the underlying parallel file system, and therefore consistent file access is its responsibility. Split-Server NFSv4 depends on the performance of the parallel file system in this area.

3. NFSv4 state maintenance

NFS versions 2 and 3 [4, 5] are stateless, which simplifies crash recovery semantics and many other aspects of the protocol. A separate protocol, the *Network Lock Manager*, isolates the inherently stateful aspects of file locking.

NFSv4 departs from the stateless model to support exclusive opens called *share reservations*, mandatory locking, and file delegations. The NFSv4 server must store some information about clients, users, and files, as well as information about the outstanding share reservations, locks, and delegations.

A share reservation controls access to a file. A client issuing an OPEN operation to a server specifies both the type of access required (read, write, or both) and the types of access to deny others (deny none, deny read, deny write, or deny both). The NFSv4 server maintains access/deny state to ensure that future OPEN requests do not conflict with current share reservations. NFSv4 also supports mandatory and advisory byte-range locks.

An NFSv4 server can pass control of a file to a client in response to an OPEN request. A *delegation* grants the client exclusive responsibility for consistent access to the file. The NFSv4 server remembers all outstanding delegations on a file for revocation on conflicting requests.

The need to manage consistency of state information on multiple nodes fetters NFSv4's ability to export an object via multiple servers. This "single server" constraint becomes a bottleneck if load increases while other nodes in the parallel file system are underutilized. Partitioning the file space among multiple NFS servers can work around this limitation to an extent, but increases management cost and fails to address scalable access to a single file or directory—a critical requirement of many high performance applications [6]. Some work has been done to aggregate partitioned NFS servers into a single file system image [7, 8], but this is at the expense of interoperability with other file systems.

4. Related work

AFS [9] and NFSv3 constrain file modifications to a single server, a bottleneck for a single file or directory. AFS file system design of volumes, cells, sites, etc and its lack of native file access, impairs its integration with high performance file systems. NFSv3 has long suffered from well-known security, consistency, and performance problems that preclude its use in a WAN environment.

GridFTP [10] is used extensively in the grid to enable high throughput, operating system independent, and secure WAN access to high-performance file systems. Successful and popular, GridFTP nevertheless has some serious limitations: it copies data instead of providing shared access to a single copy, complicating its

consistency model and decreasing storage capacity; lacks a global namespace; and cannot integrate with the local file system.

GridNFS is not intended to replace GridFTP, but to work alongside it. For example, in tiered projects such as ATLAS at CERN, GridFTP remains a natural choice for long-haul scheduled transfers among the upper tiers, while the file system semantics of GridNFS offers advantages in the lower tiers. GridNFS lets domain scientists work with files directly using conventional names, which promotes effective data management. GridNFS also offers seamless support for operating system extensions such as RDMA or file replication and migration.

GPFS [11], Lustre [12], PolyServe Matrix Server [13] and GFS [14, 15] are examples of parallel file systems, architectures in which client nodes access data in parallel from storage nodes or disks. They provide the high-speed storage systems that Split-Server NFSv4 utilizes to improve I/O throughput.

5. Design

The design goals of our NFSv4 extensions are:

- Read and write performance scales linearly as parallel file system nodes are added or removed.
- Single file system image with no partitioning.
- Negligible impact to NFSv4 security model and fault tolerance semantics.
- Support for COTS NFSv4 servers and clients.
- Independent of underlying parallel file system.

5.1. NFSv4 extensions

Our design exports the file system from all available parallel file system nodes. Any increase or decrease in available throughput of the parallel file system, e.g., additional nodes, increased network bandwidth, etc., will be reflected in Split-Server NFSv4.

To export a file from multiple NFSv4 servers exporting shared storage, the servers need a common view of the global state. NFSv4 servers must share state information and must do so consistently, i.e., with single-copy semantics. Without a consistent view of the state, conflicting file and byte-range locks may cause data corruption and leave the door open to malicious clients wishing to read and write unauthorized data.

We use *state servers* to replicate the portions of state needed to serve READ, WRITE, and COMMIT requests at I/O nodes, known as *data servers*. Figure 2 displays the Split-Server NFSv4 architecture. By transforming NFSv4 into an out-of-band protocol, shown in Figure 3, we unleash the I/O scalability of the underlying parallel file system.

A system administrator can partition the file space among several state servers, ensuring that all state for a single file resides on a single state server. Control processing can be distributed by allowing data servers to handle operations that do not affect NFSv4 server state, e.g., unmodified SETATTR and GETATTR.

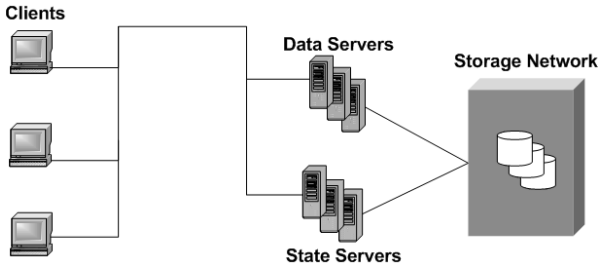

Figure 2. Split-Server NFSv4 architecture. Storage consists of a parallel file system such as GPFS. NFSv4 servers are divided into data servers, which handle all READ, WRITE, and COMMIT requests, and state servers, which handle all other requests.

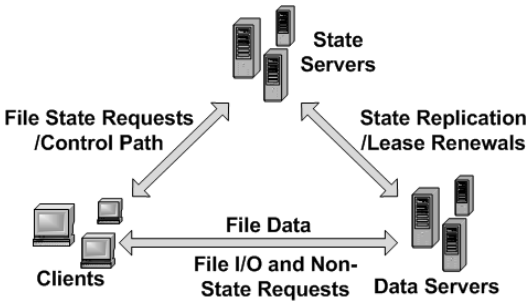

Figure 3. Process flow. Client's access data servers for I/O and non-state related requests and access state servers for file independent and file state requests. State servers replicate state on data servers.

5.2. Configuration and setup

The mechanics of a client connection to a server are the same as NFSv4 with the client mounting the state server managing the file space of interest.

Data servers register with state servers at start-up or any time thereafter and are immediately available to Split-Server NFSv4 clients, allowing easy incremental growth.

5.3. Distribution of state information

On receiving an OPEN request, a state server determines which data server will service the data request. Our implementation currently uses a round-robin algorithm across the data servers. The state server then replicates the appropriate state for the request on the selected data server.

The following items constitute a unique identifier for share reservation state:

- Client Name
- Client IP Address
- Access Bits
- File handle
- Client Verifier
- File Open Owner
- Deny Bits
- State Server ID

On receiving a CLOSE request from a client, the state server reclaims the state from the data server. Once reclamation is complete, the standard NFSv4 close procedure proceeds.

Beyond share reservations, lock support does not require maintaining any additional state. NFSv4 uses POSIX locks and relies on the locking subsystem of the underlying parallel file system. Delegations also require no additional state on the data servers as state servers manage conflicting access requests for a delegated file.

5.4. Redirection of clients

Client redirection uses a new attribute called FILE_LOCATION, which extends the recommended FS_LOCATIONS attribute to enable Split-Server NFSv4 to provide access to a single file via multiple nodes.

The FILE_LOCATION attribute specifies:

- Data server location information
- Root pathname
- Read-only flag

Clients use this information to direct READ, WRITE and COMMIT requests to the named server. The root pathname allows each data server to have its own namespace. The read-only flag declares whether the data server will accept WRITE commands. This flag can limit the number of nodes that can issue updates, possibly reducing data consistency overhead.

6. Fault tolerance

Our failure model follows that of NFSv4 with the following modifications:

1. A failed state server can recover its runtime state by retrieving each part of the state from the data servers.
2. The failure of a data server is not critical to system operation.

6.1. Client failure and recovery

An NFSv4 server places a lease on all share reservations, locks, and delegations. Clients must send RENEW operations, i.e., heartbeat messages, to the server to retain their leases. If a server does not receive a RENEW operation from the client within the lease period, the server is allowed to reap all state associated with the given client. In NFSv4, implicit RENEW operations occur on all operations that require the client to send its identifier, saving network bandwidth and server CPU cycles.

Since our out-of-band extensions redirect READ, WRITE, and COMMIT operations to the data servers, the renewal implicit in these operations no longer occurs on the state server. In Split-Server NFSv4, RENEW operations are sent to a client's mounted state server either by the client or by the data server that is actively fulfilling client requests. Enabling data servers to send RENEW messages on behalf of a client may improve scalability by limiting the maximum number of renewal messages received by a state server to the number of data server nodes, potentially much smaller than the number of Split-Server NFSv4 clients.

6.2. State server failure and recovery

A recovering state server stops servicing requests while querying the data servers and using its State Server ID to identify and rebuild its state.

6.3. Data server failure and recovery

A failed data server is discovered by the state server as it tries to replicate state or by clients as they issue requests. Clients obtain a new data server by re-requesting the FILE_LOCATION attribute from the appropriate state server. A partitioned data server immediately stops fulfilling client requests, preventing a state server from granting conflicting file access requests.

7. Security

The addition of data servers to the NFSv4 protocol does not require extra security mechanisms. The client uses the security protocol negotiated with a state server for all nodes. Servers communicate over RPCSEC_GSS, the secure RPC mandated for NFSv4.

8. Evaluation

In this section, we present the results of our scalability experiments with unmodified NFSv4 vs. Split-Server NFSv4 as they export a GPFS file system. The test environment is shown in Figure 4. All nodes are connected via an IntraCore 35160 gigabit switch with 1500 byte Ethernet frames.

Server System: The five server nodes are equipped with 850 MHz Pentium 4 processors with a 256 KB cache, 2 GB of RAM, a Seagate 80GB, 7200 RPM hard drive with an Ultra ATA/100 interface and a 2 MB cache, an IBM/Hitachi 32GB, 7200 RPM hard drive with an Ultra ATA/100 interface and a 2 MB cache, and two 3Com 3C996B-T gigabit cards. They run a modified Linux 2.4.18 kernel with Red Hat 9.

Client System: Client nodes one through three are equipped with dual 1.7 GHz Pentium 4 processors with a 256 KB cache, 2 GB of RAM, a Seagate 80 GB, 7200 RPM hard drive with an Ultra ATA/100 interface and a 2 MB cache, and a 3Com 3C996B-T gigabit card. Client

node four is equipped with 1.4 GHz Intel Xeon processors with a 256 KB cache, 1 GB RAM, an Adaptec 40 GB, 10K RPM SCSI hard drive using Ultra 160 host adapter, and a AceNIC gigabit Ethernet card. All clients run the Linux 2.6.1 kernel with a Red Hat 9 distribution.

Netapp FAS960 Filer: The storage device has two processors, 6 GB of RAM, and a quad gigabit card. It is connected to eight disks running RAID4.

The five servers are running the parallel file system GPFS v1.3 with a 40 GB file system and a 16 KB block size on the Netapp Filer. GPFS maintains a 32 MB file and metadata cache known as the *pagepool*. All NFS experiments use forty server threads except the Split-Server NFSv4 write experiments, which uses a single server thread since we are seeking the best possible performance (discussed in Section 8.4).

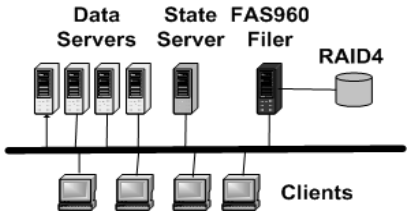

Figure 4. Experimental setup. The system has four Split-Server NFSv4 clients and five GPFS servers exporting a common file system. The GPFS servers are exported by Split-Server NFSv4, consisting of a single state server and at most four data servers.

8.1. Scalability experiments

To evaluate the scalability of our design, we compare the aggregate I/O throughput as we increase the number of clients accessing GPFS, NFSv4, and Split-Server NFSv4. Since both standard NFSv4 and Split-Server NFSv4 export a GPFS file system, the GPFS configuration provides the theoretical ceiling on NFSv4 and Split-Server NFSv4 I/O throughput. The extra hop between the GPFS server and the NFS client prevents the performance of NFSv4 and Split-Server NFSv4 from equaling GPFS performance. The goal is for Split-Server NFSv4 to scale linearly with GPFS. The GPFS configuration consists of a four node GPFS file system directly connected to the filer. The NFSv4 configuration consists of a single NFSv4 server running on a GPFS node and four clients. The Split-Server NFSv4 configuration consists of a state server, four data servers (each running on a GPFS file system node), and four clients. At most one client accesses each data server during an experiment.

To measure the aggregate throughput, we used the IOZone [16] benchmark tool. The first set of experiments involves each client reading/writing a separate 500 MB file. The second set of experiments involves each client reading/writing disjoint 500 MB portions of a single pre-existing file. The aggregate throughput is calculated when the last client completes its task. The presented value is the average over ten executions of the benchmark. The write timing includes a flush of the client's cache to the server. Clients and servers purge their caches before each read experiment. All read experiments use a warm filer cache to eliminate disk access irregularities.

The experimental goal is to demonstrate that Split-Server NFSv4 scales linearly with additional resources. We engineered a server bottleneck in the system by using a small GPFS pagepool and block size, and by cutting the number of server clock cycles in half. By ensuring that each server is fully utilized, we are confident that our results are applicable to any system that needs to scale with additional servers.

8.2. Read performance

First, we measure the read performance as the number of clients increases from one to four. Figure 5a presents the results with separate files and Figure 5b presents the results with a single file. GPFS sets the ceiling on performance with an aggregate read throughput of 23 MB/s with a single server and with four servers reaching 94.1 MB/s and 91.9 MB/s in multiple and single file experiments respectively. The slight decrease in performance for the single file experiment is because all servers must access a single metadata server. With Split-Server NFSv4, as we increase the number of clients and data servers the aggregate I/O throughput increases linearly, reaching 65.7 MB/s with multiple files and 59.4 MB/s for the single file experiment. NFSv4 aggregate throughput remains flat at approximately 16 MB/s in both experiments, very explicitly demonstrating the single server bottleneck.

8.3. Write performance

The second experiment measures the aggregate write throughput as we increase the number of clients from one to four. We first measure the performance of all clients writing to separate files, as shown in Figure 6a.

GPFS sets the limit with an aggregate write throughput of 16.7 MB/s with a single server and a maximum of 61.4 MB/s with four servers. The fourth server overloads the filer's CPU. NFSv4 and Split-Server NFSv4 initially have an aggregate throughput of approximately 8 MB/s. The aggregate throughput of Split-Server NFSv4 increases linearly, reaching a maximum of 32 MB/s. As in the read experiments, the aggregate throughput of NFSv4 remains flat as the number of clients is increased.

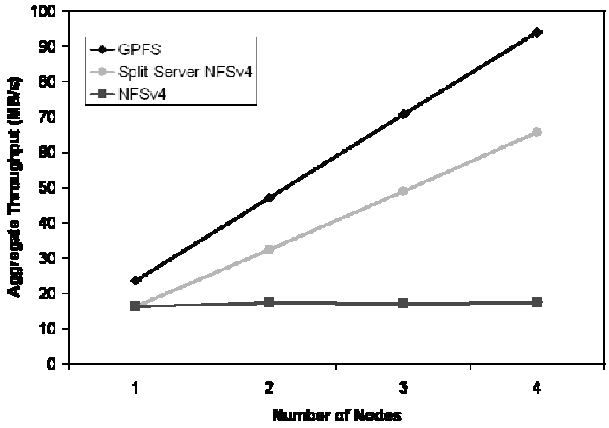

Figure 5a. Separate files

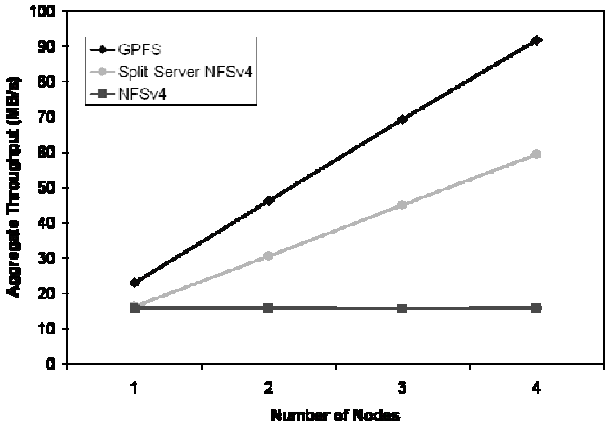

Figure 5b. Single file

Aggregate read throughput - GPFS consists of up to four file system nodes. NFSv4 is up to four clients accessing a single GPFS server. Split-Server NFSv4 consists of up to four clients accessing up to four data servers and a state server. Split-Server NFSv4 scales linearly as we increase the number of GPFS nodes but NFSv4 performance remains flat.

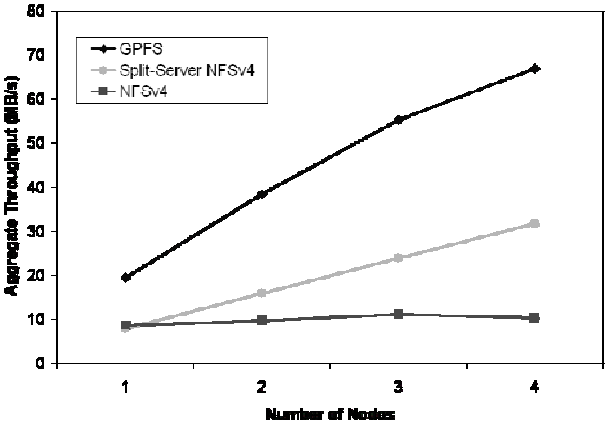

Figure 6a. Separate files

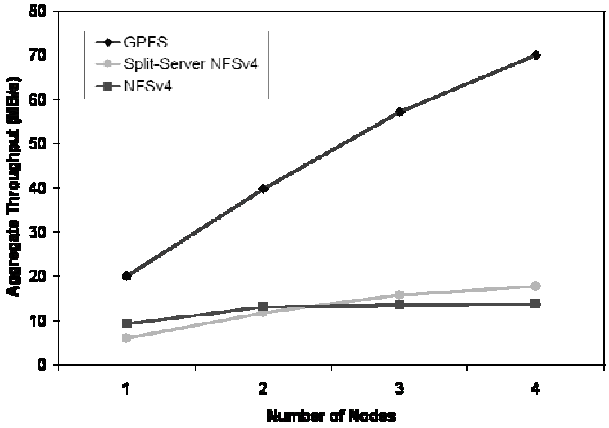

Figure 6b. Single file

Aggregate write throughput - GPFS consists of up to four file system nodes. NFSv4 is up to four clients accessing a single GPFS server. Split-Server NFSv4 consists of up to four clients accessing up to four data servers and a state server. With separate files, Split-Server NFSv4 scales linearly as we increase the number of GPFS nodes but NFSv4 performance remains flat. With a single file, Split-Server NFSv4 experiences reduced performance due to mtime synchronization.

Figure 6b shows the experimental results of each client writing to different regions of a single file. The write performance of GPFS and NFSv4 is similar to the separate file experiments. The major difference occurs with Split-Server NFSv4, achieving an initial aggregate throughput of 6.1 MB/s and increasing to 18.7 MB/s. The initial poor performance and lack of scalability is the result of modification time (*mtime)* synchronization between GPFS servers. This is disabled when accessing GPFS directly, but is mandatory with NFS to ensure client cache consistency. GPFS selects the first node that accesses a file as its metadata server, thereby causing the GPFS server that the state server exports to be among servers that synchronize the mtime attribute, further reducing performance.

8.4. Discussion

GPFS synchronizes the **`mtime`** attribute to comply with the NFS protocol. As demonstrated in the previous section, this comes at a price of severely hindering its scalability for access to a single file. Client cache synchronization requires this overhead, but environments exist where it is unnecessary. Some programs cache data themselves and use the OPEN option O_DIRECT to disable client caching for a file. Other programs require only non-conflicting write consistency, handling data consistency without relying on locks or cache consistency

mechanisms. PVFS2 [17] is designed for such programs. To succeed in these environments, the NFS protocol must relax its client cache consistency semantics.

Traditionally, NFS block sizes have been very small. Block sizes were 4 KB in NFSv2, grew to 8 KB in NFSv3, and most recent implementations now support 32 KB or 64 KB. Synchronous writes along with hardware and kernel limitations are some of the original reasons for small block sizes. Another is UDP, which divides each block into multiple requests over the wire so that the loss of a single request means the loss of the entire block. With the introduction of TCP to the Linux implementation of NFS in 2002, jumbo frames, and larger buffer space allow for larger block sizes, but the current Linux kernel has a 32 KB limit. This creates a disparity with many parallel file systems, which use a stripe size of greater than 64 KB. To avoid this data request, inefficiency on the file node, NFS implementations need to catch up to parallel file systems like GPFS that support block sizes of greater than 1 MB to transfer data between storage, file and application nodes.

Multiple NFS server threads can also reduce I/O throughput. Even with a single NFS client, the parallel file system assumes all requests are from different sources and performs locking between threads. In addition, server threads can process read and write requests out of order, hampering the parallel file system's ability to improve its interaction with the physical disk.

In NFSv3, the lack of OPEN and CLOSE commands leads to an implicit open and close of a file in the underlying file system on every request. This does not degrade performance with local file systems such as Ext3, but the extra communication required to contact a metadata server in parallel file systems severely restricts NFSv3 throughput.

9. Future work

We are currently clarifying the FILE_LOCATION attribute with the Network Working Group and working toward its adoption in a minor version extension of the NFS protocol.

Coordinated use of the parallel file system's cache may be important under certain I/O access patterns. Server load balancing algorithms other than round robin may be better suited to these environments.

Even though the state servers do not handle I/O requests, they may prove to be a bottleneck for a single file. The Google file system [18] uses read-only replication of metadata, but this is be insufficient for our design as each state server requires write access. Automatic partitioning of the file space among state servers and automatic failover between state servers are also areas for future research.

10. Conclusion

This paper discusses the performance issues involved in exporting a parallel file system via NFS and introduces extensions to the NFSv4 protocol to improve the aggregate bandwidth and transparency between remote sites in a data grid. Using on-demand replication and a new FILE_LOCATION attribute, Split-Server NFSv4 provides parallel and scalable access to a parallel file system. We implemented a prototype of our design and demonstrated that Split-Server NFSv4 scales linearly with the number of parallel file system nodes.

11. Acknowledgements

We thank Gary Grider and Lee Ward for valuable insights and Brian Dixon for help with GPFS.

This research is partially supported by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories under contract B523296.

12. References

[1] "IBM GPFS Scores Top Marks in Bandwidth Challenge at SC2003," in *GRIDtoday*, vol. 2, Dec. 1, 2003.

[2] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal on High-Performance Computing Applications*, vol. 15, pp. 200-222, 2001.

[3] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network File System Version 4 Protocol Specification," *RFC 3530*, April 2003.

[4] Sun Microsystems Inc., "NFS: Network File System Protocol Specification," *RFC 1094*, March 1989.

[5] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3 Design and Implementation," In Proceedings of the Summer USENIX Conference, June 1994.

[6] S. Berchtold, C. Boehm, D.A. Keim, and H. Kriegel, "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space," *ACM PODS*, May 1997.

[7] G.H. Kim, R.G. Minnich, and L. McVoy, "Bigfoot-NFS: A Parallel File-Striping NFS Server (Extended Abstract)," www.bitmover.com/lm, 1994.

[8] F. Garcia-Carballeira, A. Calderon, J. Carretero, J. Fernandez, and J.M. Perez, "The Design of the Expand File System," *International Journal of High Performance Computing Applications*, vol. 17, pp. 21-37, 2003.

[9] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, 1988.

[10] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke., "Data Management and Transfer in High Performance Computational Grid Environments," *Parallel Computing Journal*, vol. 28, pp. 749-771, May 2002.

[11] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *USENIX Conference on File and Storage Technologies*, 2002.
[12] Cluster File Systems Inc., "Lustre: A Scalable, High-Performance File System," `www.lustre.org`, 2002.
[13] Polyserve Inc., "Polyserve Matrix Server Architecture," *White Paper*, `www.polyserve.com`, 2003.
[14] Red Hat Software Inc., "Red Hat Global File System," `www.redhat.com`.
[15] S.R. Soltis, T.M. Ruwart, and M.T. O'Keefe, "The Global File System," In Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems, 1996.
[16] W.D. Norcott and D. Capps, "IOZone Filesystem Benchmark," `www.iozone.org`, 2003.
[17] PVFS2 Development Team, "Parallel Virtual File System, Version 2," `www.pvfs.org/pvfs2`, September 2003.
[18] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," *In Proceedings of the19th ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003.