pNFS and Linux:
Working Towards a Heterogeneous Future

Dean Hildebrand, Peter Honeyman, and Wm. A. (Andy) Adamson

Center for Information and Technology Integration
University of Michigan
{dhildebz, honey, andros}@umich.edu

Abstract. Heterogeneous and scalable remote data access is a critical enabling feature of widely distributed collaborations. Parallel file systems feature impressive throughput, but sacrifice heterogeneous access, seamless integration, security, and cross-site performance. Remote data access tools such as NFS and GridFTP provide secure access to parallel file systems, but either lack scalability (NFS) or seamless integration and file system semantics (GridFTP).

Anticipating terascale and petascale HPC demands, NFSv4 architects are designing pNFS, a standard extension that provides direct storage access to parallel file systems while preserving operating system and hardware platform independence. pNFS distributes I/O across the bisectional bandwidth of the storage network between clients and storage devices, removing the single server bottleneck so vexing to client/server-based systems.

Researchers at the University of Michigan are collaborating with industry to develop pNFS for the Linux operating system. Linux pNFS features a pluggable client architecture that harnesses the potential of pNFS as a universal and scalable metadata protocol by enabling dynamic support for layout format, storage protocol, and file system policies. This paper evaluates the scalability and performance of the Linux pNFS architecture with the PVFS2 and GPFS parallel file systems.

Introduction

Large research collaborations require global access to massive data stores. Parallel file systems feature impressive throughput, but sacrifice heterogeneous access, seamless integration, security, and cross-site performance. In addition, while parallel file systems excel at large data transfers, many do so at the expense of small I/O performance. While large data transfers dominate many scientific applications, numerous workload characterization studies have highlighted the prevalence of small, sequential data requests in modern scientific applications [1-3].

Many application domains demonstrate the need for high bandwidth, concurrent, and secure access to large datasets across a variety of platforms and file systems. Scientific computing that connects large computational and data facilities across the globe can generate petabytes of data. Digital movie studios that generate terabytes of data every day require access from compute clusters and Sun, Windows, SGI, and

Linux workstations [4]. This need for heterogeneous data access produces a tension between parallel file systems and application platforms.

Distributed file access protocols such as NFS [5] and CIFS [6] bridge the interoperability gap, but they are unable to deliver the superior performance of a high-end storage system. GridFTP [7], a popular remote data access tool in the Grid, offers high throughput and operating system independent access to parallel file systems, but lacks seamless integration and file system semantics.

pNFS, an integral part of NFSv4.1, overcomes these enterprise- and grand challenge-scale obstacles by enabling direct client access to storage while preserving NFS operating system and hardware platform independence. pNFS distributes I/O across the bisectional bandwidth of the storage network between clients and storage devices, removing the single server bottleneck so vexing to client/server-based systems. In combination, the elimination of the single server bottleneck and the ability for clients to access data directly from storage results in superior file access performance and scalability [8].

At the Center for Information Technology Integration at the University of Michigan, we are developing pNFS for the Linux operating system. A pluggable client architecture harnesses the potential of pNFS as a universal and scalable metadata protocol by enabling dynamic support for layout format, storage protocol, and file system policies. In conjunction with several industry partners, a prototype is under development that supports the file- [9], block- [10], object- [11], and PVFS2-based [8] storage protocols. This paper evaluates the scalability and performance of the Linux pNFS architecture with the PVFS2 [12] and GPFS [13] parallel file systems.

pNFS overview

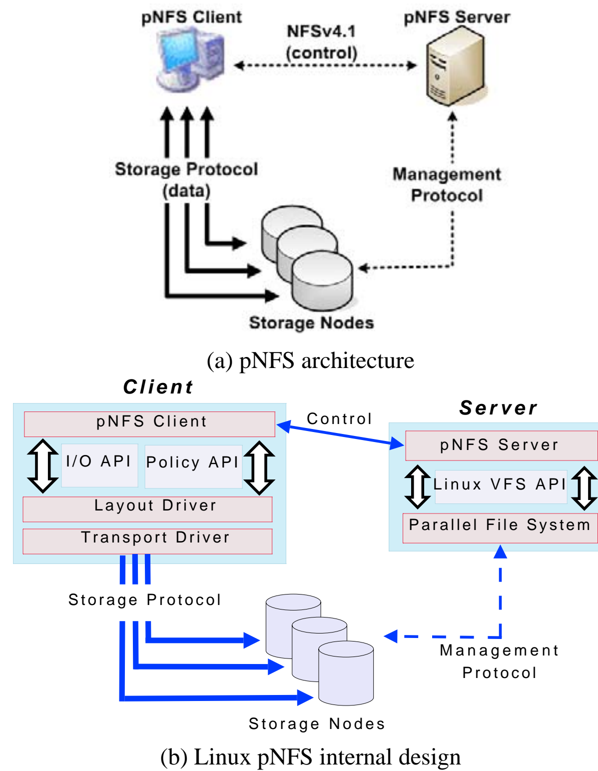
pNFS is a heterogeneous metadata protocol. pNFS clients and servers are responsible for control, file management operations, and delegate I/O functionality to a storage-specific client driver. By separating control and data flow, pNFS distributes I/O across the bisectional bandwidth of a storage network connecting clients and storage devices, removing the single server bottleneck.

Figure 1a displays the pNFS architecture. The control path contains all NFSv4.1 operations and features. While the data path can support any storage protocol, the IETF design effort focuses on file-, object-, and block-based storage protocols. Storage devices can be NFSv4.1 servers, object storage, or even block-addressable SANs. NFSv4.1 does not specify a management protocol, which may therefore be proprietary to the exported file system.

pNFS protocol extensions

This section describes the NFSv4.1 protocol extensions to support pNFS.

LAYOUTGET operation. The LAYOUTGET operation obtains file access information for a byte-range of a file, i.e., a layout, from the underlying storage



(b) Linux pNFS internal design

Fig. 1. pNFS architecture and Linux internal design. (a) pNFS splits the NFSv4.1 protocol into a control path and a data path. The NFSv4.1 protocol exists along the control path. A storage protocol along the data path provides direct and parallel data access. A management protocol binds metadata servers with storage devices. (b) The pNFS client uses I/O and policy interfaces to access storage nodes and follow underlying file system polices. The pNFS server uses Linux export operations to exchange pNFS information with the underlying file system.

system. The client issues a LAYOUTGET operation after it opens a file and before data access. Implementations determine the frequency and byte range of the request. The LAYOUTGET operation returns the requested layout as an opaque object, which allows pNFS to support arbitrary file layout types. At no time does the pNFS client attempt to interpret this object, it acts simply as a conduit between the storage system and the layout driver.

LAYOUTCOMMIT operation. The LAYOUTCOMMIT operation commits changes to the layout information. The client uses this operation to commit or discard provisionally allocated space, update the end of file, and fill in existing holes in the layout.

LAYOUTRETURN operation. The LAYOUTRETURN operation informs the NFSv4.1 server that layout information obtained earlier is no longer required. A client may return a layout voluntarily or upon receipt of a server recall request.

CB_LAYOUTRECALL operation. If layout information is exclusive to a specific client and other clients require conflicting access, the server can recall a layout from

the client using the CB_LAYOUTRECALL callback operation.[1] The client should complete any in-flight I/O operations using the recalled layout and write any buffered dirty data directly to storage before returning the layout, or write it later using normal NFSv4 write operations.

GETDEVINFO and GETDEVLIST operations. The GETDEVINFO and GETDEVLIST operations retrieve information about one or more storage nodes. Typically, the client issues the GETDEVLIST operation at mount time to retrieve the active storage nodes. The GETDEVINFO operation retrieves information about individual storage nodes.

Linux pNFS

Although parallel file systems separate control and data flows, there is tight integration of their control and data protocols. Users must adapt to different consistency and security semantics for each data repository. Using pNFS as a universal metadata protocol lets applications realize a consistent set of file system semantics across data repositories. Linux pNFS facilitates interoperability by providing a framework for the co-existence of the NFSv4.1 control protocol with all storage protocols. This is a major departure from current file systems, which can support only a single storage protocol such as OSD [14, 15].

Figure 1b depicts the architecture of pNFS on Linux, which adds a *layout driver* and a *transport driver* to the standard NFSv4 architecture. The *layout driver* interprets and utilizes the opaque layout information returned from the pNFS server. A layout contains the information required to access any byte range of a file. In addition, a layout may contain file system specific access information. For example, the object-based layout driver requires the use of OSD access control capabilities [16]. To perform direct and parallel I/O, a pNFS client first requests layout information from the pNFS server. The layout driver uses the information to translate read and write requests from the pNFS client into I/O requests directed to storage devices. For example, the NFSv4.1 file-based storage protocol stripes files across NFSv4.1 data servers (storage devices); only READ, WRITE, COMMIT, and session operations are used on the data path. The pNFS server can generate layout information itself or request assistance from the underlying file system. The *transport driver* performs I/O, e.g., iSCSI [17] or ONC RPC [18], to the storage nodes.

pNFS client extensions: pluggable storage protocol

Layout drivers are pluggable, using a standard set of interfaces for all storage protocols. An *I/O interface* facilitates the management of layout information and performing I/O with storage. A *policy interface* informs the pNFS client of file system and storage system specific policies.

[1] NFSv4 already contains a callback operation infrastructure for delegation support.

I/O interface

The current Linux pNFS client prototype can access data through the Linux *page cache*, using *O_DIRECT*, or directly by bypassing the Linux page cache and all NFSv4 I/O request processing (*direct* access method).

The page cache access method uses a writeback cache to gather or split write requests into block-sized requests before being sent to storage, with requested data cached on the client. For read requests, the Linux kernel readahead algorithm sets the request size. The page cache access method is the default behavior of the Linux NFS client. The O_DIRECT access method continues to use a writeback cache, but data does not pass through the Linux page cache. O_DIRECT is useful for applications, such as databases, that perform small I/O requests but handle data caching themselves. The direct access method bypasses the Linux page cache and NFS writeback cache and sends I/O requests directly to the layout driver with the request offset and extent untouched. Many scientific applications use the direct access method as a foundation beneath higher-level tools such as MPI-IO [19, 20].

Policy interface

The functionality of a layout driver can depend on the application, the supported storage protocol, and the underlying parallel file system. For example, many HPC applications do not require a data cache (or its associated overhead) and therefore prefer to use the O_DIRECT or direct access methods. Additional policies include the file system stripe and block size, when to retrieve layout information, and the I/O request size threshold (which improves performance under certain workloads [21]).

In addition, the policy interface enables layout drivers to specify if it uses existing Linux data management services or use customized implementations. The following is a list of services available to layout drivers:

- Linux page cache
- NFSv4 writeback cache
- Linux kernel readahead

A layout driver can define custom policies or acquire specific policies from the pNFS server.

pNFS server extensions

The Linux pNFS server implementation is designed to export a pNFS capable file system. The pNFS server performs state management for all NFSv4.1 operations and extends the existing Linux NFS server interface with the underlying file system. The Linux pNFS server does not provide a management protocol between the pNFS server and the storage devices (Figure 1a); this is left to the pNFS capable file system.

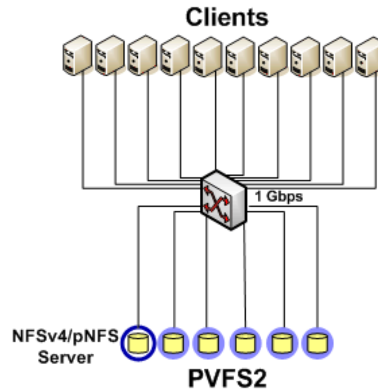


Fig. 2. Aggregate I/O throughput experimental setup. Up to ten clients access six PVFS2 storage nodes. .With standard NFSv4, all clients perform I/O through an NFSv4 server residing on a storage node. With pNFS, clients use a PVFS2 layout driver for direct and parallel access to PVFS2 storage nodes.

The pNFS server is storage protocol agnostic. The pNFS server obtains storage device and layout information from the underlying file system as opaque data. The pNFS server transfers the opaque data to the pNFS client and subsequently to a layout driver, which can interpret and use the information for direct access to storage. A callback exists to allow the underlying file system to initiate a layout recall. When a pNFS client returns a layout, the pNFS server passes the opaque layout information back to the underlying file system.

The pNFS server manages state information for all outstanding layouts. This includes tracking granted and returned layouts as well as portions of layouts that have been recalled.

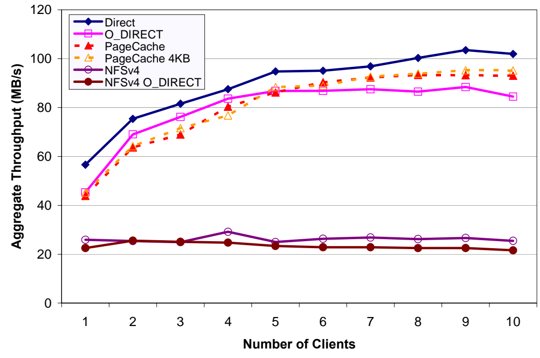
Evaluation

This section analyzes the performance of our Linux pNFS prototype using the page cache, O_DIRECT, and direct access methods with the PVFS2 and GPFS parallel file systems.

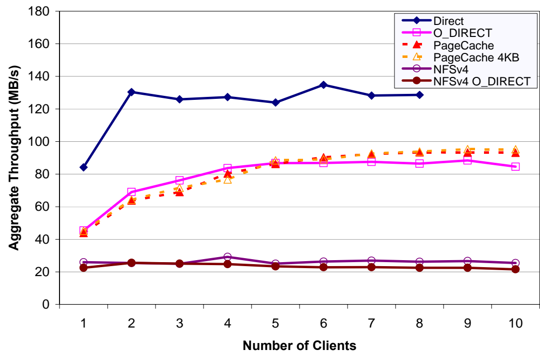
Experimental setup

The first two experiments use the `IOR` micro-benchmark [22] to measure the aggregate I/O throughput to a PVFS2 file system as we increase the number of clients. The third set of experiments evaluates the data transfer performance between GPFS and PVFS2 file systems using a single 10 Gbps client. All nodes run Linux 2.6.17.

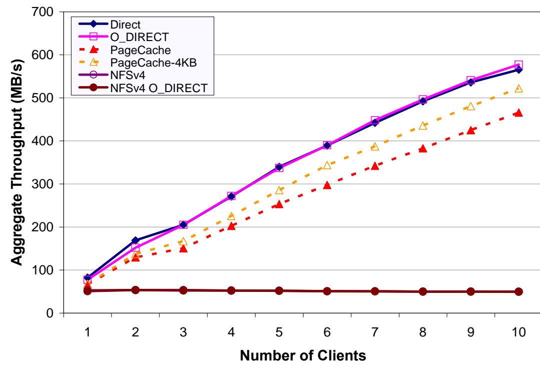
1 Gbps Clients: Clients in Figures 3 and 4 are equipped with dual 1.3 GHz P3 processors, 2 GB memory, and an Intel Pro gigabit card.



(a) Write 200 MB



(b) Write 4 GB



(c) Read

Fig. 3. pNFS and NFSv4 aggregate I/O throughput to a PVFS2 file system. Aggregate I/O throughput to six PVFS2 storage nodes using NFSv4 (with and without O_DIRECT) as well as pNFS with the direct, O_DIRECT, and page cache access methods. pNFS scales with additional clients, with the direct access method achieving the best performance. In addition, performance remains the same across I/O request sizes. NFSv4 performance remains flat in all experiments.

10 Gbps Client: The client in Figure 6 is equipped with dual 1.4 GHz Opteron processors, 4 GB memory, and one Neterion 10 Gigabit card.

PVFS2 Configuration: The PVFS2 1.5.1 file system has six storage nodes, with one storage node doubling as a metadata manager, and a 64 KB stripe size. Each node is equipped with dual 1.7 GHz P4 processors, 2 GB memory, one Seagate 80 GB 7200 RPM hard drive with Ultra ATA/100 interface and 2 MB cache, and one 3Com 3C996B-T gigabit card.

GPFS Configuration: The GPFS file system has six I/O nodes attached to a FibreChannel shared disk array. Each node is equipped with dual 1.3 GHz P3 processors, 2 GB memory, and an Intel Pro gigabit card.

Scalability and performance

Our initial experiments, shown in Figure 3, evaluate different ways of accessing storage. We use the policy interface to perform a fine-grained performance analysis of the Linux page cache and NFSv4 I/O subsystem while using the page cache, O_DIRECT, and direct access methods. To provide a baseline, we also access storage using NFSv4 with and without O_DIRECT. We use our pNFS prototype with a PVFS2 file system and a PVFS2 layout driver. The PVFS2 blocksize is 4 MB and the NFSv4 `wsize` and `rsize` are 64 KB.

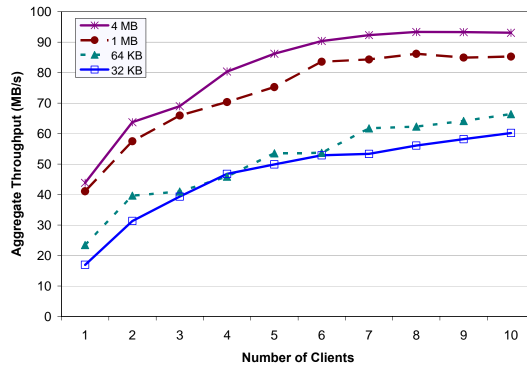
In Figure 3a, clients write separate 200 MB files in 4 MB chunks. With six data servers and a 200 MB file, the amount of data written to each disk is 200 MB / 6 = 33 MB. With a 33 MB file, the individual disk bandwidth is 22 MB/s, which gives the PVFS2 file system a theoretical maximum write bandwidth of 132 MB/s.

NFSv4 write performance is flat, obtaining an aggregate throughput of 26 MB/s. NFSv4 with O_DIRECT is also flat with a slight reduction in performance. The direct access method has the greatest aggregate write throughput, obtaining over 100 MB/s with eight clients. The aggregate write throughput of pNFS clients using the page cache is consistently 10 MB/s lower than pNFS clients using the direct method. *PageCache-4KB*, which writes data in 4 KB chunks, has the same performance as *PageCache*, which demonstrates the Linux pNFS client's ability to gather small write requests into larger and more efficient requests. O_DIRECT obtains an aggregate write throughput between the direct and page cache access methods, but flattens out as we increase the number of clients. The relative performance of the O_DIRECT and the page cache access methods is consistent with the relative performance of NFSv4 with and without O_DIRECT.

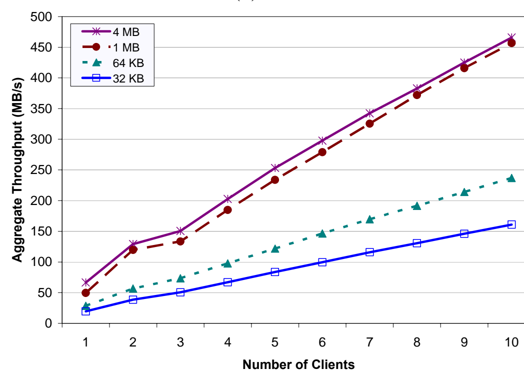
In Figure 3b, clients write separate 4 GB files in 4 MB chunks. With a 4 GB file, the amount written to each disk is approximately 715 MB. With a 715 MB file, the individual disk bandwidth is 32 MB/s, which gives the PVFS2 file system a theoretical maximum write bandwidth of 192 MB/s.

All results remain unchanged except for the direct access method, which experiences a major performance increase. The direct access method single client performance increases by over 20 MB/s, which rises even higher with two clients to 130 MB/s.

Why is the direct access method the only method to benefit from the increase in individual disk bandwidth? The answer is a more effective use of disk bandwidth.



(a) Write



(b) Read

Fig. 4. pNFS and I/O blocksize. pNFS aggregate I/O throughput to six PVFS2 storage nodes using the pNFS page cache access method with four block sizes. The results are independent of the application write and read request size since the requests are gathered together until they match the block size.

NFSv4 fault tolerance semantics mandate that written data exist on stable storage before it is evicted from the client cache. This policy results in numerous disk sync operations while writing a large file. The direct access method sidesteps this requirement and performs a single disk sync operation after all data has been sent to the storage nodes. Many scientific applications can re-create lost data, allowing the direct access method to provide a valuable performance boost.

In Figure 3c, clients read separate 200 MB files in 4 MB chunks. NFSv4 read performance is flat, obtaining an aggregate throughput of 52 MB/s. The aggregate read throughput of pNFS clients using the two methods that avoid the page cache is the same as we increase the number of clients, nearly exhausting the available network bandwidth of with ten clients. Working through the page cache reduces the aggregate read throughput by up to 110 MB/s. Increasing the file size does not affect performance since the disks do not need to perform sync operations after read requests.

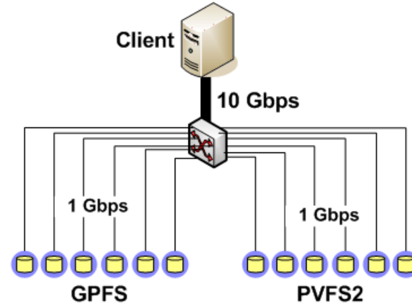


Fig. 5. Inter-file system data transfer experimental setup. A 10 Gbps client transfers data between PVFS2 and GPFS file systems. Each file system has six storage nodes connected via gigabit Ethernet.

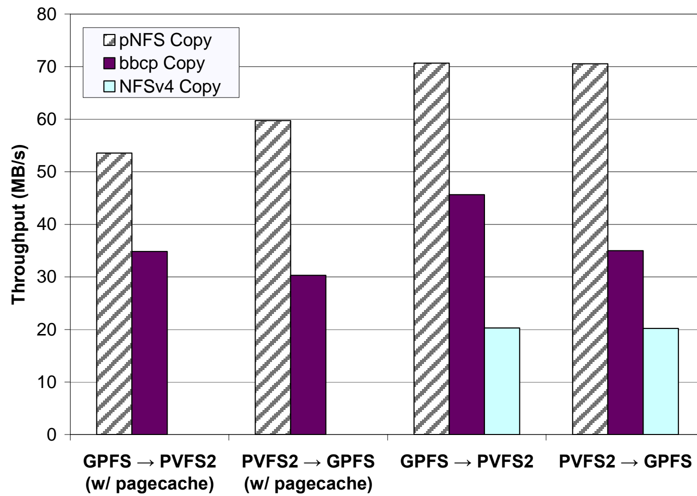


Fig. 6. Inter-file system data transfer performance using pNFS, NFSv4, and `bbcp`. pNFS outperforms NFSv4 and `bbcp` by using direct and parallel I/O with both GPFS and PVFS2. NFSv4 performance is limited by its single server design. `bbcp` performance suffers because it accesses a single GPFS I/O node.

Block size sensitivity

Our second set of experiments sets out to verify the performance sensitivity to layout driver block size (I/O transfer size) when working through the Linux page cache. We use our pNFS prototype with a PVFS2 file system and a PVFS2 layout driver.

As shown in Figure 4, increasing the block size from 32 KB to 4 MB improves aggregate I/O throughput, although this boost eventually hits a performance ceiling. The request size is 4 MB, but results are independent of application request size since the pNFS writeback cache gathers requests into the block size.

Inter-file system data transfer performance

Our last set of experiments evaluate cross-file system data transfer performance using different types of layout drivers on a single client. As shown in Figure 5, we use a single client with a 10 Gbps network card to transfer data between PVFS2 and GPFS file systems, each with six storage nodes attached to a 1 Gbps network. We compare the I/O throughput of using pNFS, NFSv4, and `bbcp` [23] to copy a 1 GB file between PVFS2 and GPFS. We analyze the PVFS2 layout driver with both the page cache and direct access methods. `bbcp` is a peer-to-peer data transfer program similar to `scp` or `rcp`.

The pNFS experiment first mounts the PVFS2 file system using the PVFS2 layout driver and mounts the GPFS file system using the NFSv4.1 file-based layout driver. The `dd` command then copies data between the two mount points. Similarly, NFSv4 uses the `dd` command to copy data between the two NFSv4 mounted file systems. The `bbcp` experiments do not use the file-based layout driver since `bbcp` can access GPFS I/O nodes directly. Therefore, `bbcp` copies data from a single GPFS storage node to the pNFS/PVFS2 mount point on the client. The `dd` command serializes the transfer of data between file systems by buffering data on the client before writing it to the target file system.

Figure 6 displays the file transfer experiments. The client/server design of NFSv4 limits access to a single server on each back end, achieving an inter-file system transfer rate of 20 MB/s. `bbcp` reads and writes from/to a single GPFS node, but accesses data in parallel from/to the PVFS2 file system. This partial use of parallel I/O increases performance in all four experiments, although it is more efficient when transferring data from GPFS to PVFS2. In addition, data transfer throughput decreases slightly when using the page cache access method with PVFS2. pNFS uses parallel I/O with both file systems, which doubles the I/O throughput when transferring data from PVFS2 to GPFS and increases the I/O throughput from GPFS to PVFS2 by 55%.

Related work

Unlike much recent work that focuses on improving the performance and scalability of a *single* file system, e.g., Lustre [24], PVFS2 [12], GPFS-WAN [25, 26], Google file system [27], Gfarm [28], and FARSITE [29], the goal of pNFS is to enhance a single file access protocol to scale I/O throughput to a *diversity* of parallel file systems.

GridFTP [7] 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 is difficult to integrate with the local file system.

The Storage Resource Broker (SRB) [30] aggregates storage resources, e.g., a file system, an archival system, or a database, into a single data catalogue but does not

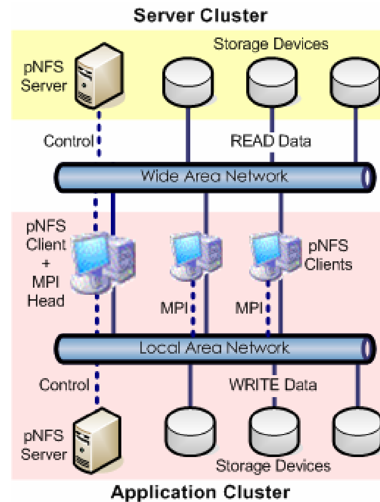


Fig. 7. pNFS and inter-cluster data transfers across the WAN. A pNFS cluster retrieves data from a remote storage system, processes the data, and writes to its local storage system. The MPI head node distributes layout information to pNFS clients.

support parallel I/O to multiple storage endpoints and has difficulty integrating with the local file system.

Several Linux pNFS layout drivers exist. A PVFS2 layout driver has existed since 2004 [8] and NFSv4.1 file-based layout drivers have been demonstrated with GPFS [13], Lustre [24], and PVFS2 [12] Sun is developing file and object layout implementations. Panasas object and EMC block drivers are also under development.

Network Appliance is using the Linux file-based layout driver to bind disparate filers (NFS servers) into a single file system image. This continues previous work that aggregates partitioned NFS servers into a single file system image [31-33].

Future work

Petascale computing requires inter-site data transfers involving clusters that may have different operating systems and hardware platforms, incompatible or proprietary file systems, or different storage and performance parameters that require differing layouts. The Linux pNFS architecture with "pluggable" storage protocols offers a solution.

Figure 7 shows two clusters separated by a long range, high-speed WAN. Each cluster has the architecture described in Figure 1a and uses a storage protocol supported by pNFS. (The management protocol is not shown.)

The *application cluster* is running an MPI application that wants to read a large amount of data from the server cluster and perhaps write to its backend. The MPI head node obtains the data location from the server cluster and distributes portions of the data location information (via MPI) to other application cluster nodes, enabling direct access to server cluster storage devices. The MPI application then reads data in

parallel from the server cluster across the WAN, processes the data, and directs output to the application cluster backend.

A natural use case for this architecture is a visualization application processing the results of a scientific MPI code run on the server cluster. Another use case is an MPI application making a local copy of data from the server cluster on the application cluster.

Conclusions

This paper describes and evaluates Linux pNFS, an integral part of NFSv4.1 that enables direct client access to heterogeneous parallel file systems. Linux pNFS features a pluggable client architecture that harnesses the potential of pNFS as a universal and scalable metadata protocol by enabling dynamic support for layout format, storage protocol, and file system policies.

Experiments with the Linux pNFS architecture demonstrate that using the page cache inflicts an I/O performance penalty and that I/O performance is highly subject to I/O transfer size. In addition, Linux pNFS can use bi-directional parallel I/O to raise data transfer throughput between parallel file systems.

Acknowledgments

This material is based upon work supported by the Department of Energy under Award Numbers DE-FG02-06ER25766 and B548853, Sandia National Labs under contract B523296, and by grants from Network Appliance and IBM. We thank Lee Ward, Gary Grider, James Nunez, Marc Eshel, Garth Goodson, Benny Halvey, and the PVFS2 development team for their valuable insights and system support.

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Schlatter Ellis, and M. Best, "File-Access Characteristics of Parallel Scientific Workloads," *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075-1089, 1996.

[2] P.E. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed, "Input/Output Characteristics of Scalable Parallel Applications," in *Proceedings of Supercomputing '95*, San Diego, CA, 1995.

[3] F. Wang, Q. Xin, B. Hong, S.A. Brandt, E.L. Miller, D.D.E Long, and T.T. McLarty, "File System Workload Analysis For Large Scale Scientific Computing Applications," in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, 2004.

[4] D. Strauss, "Linux Helps Bring Titanic to Life," *Linux Journal*, 46, 1998.

[5] B. Callaghan, B. Pawlowski, and P. Staubach, NFS Version 3 Protocol Specification. RFC 1813, 1995.

[6] Common Internet File System File Access Protocol (CIFS), `msdn.microsoft.com/library/en-us/cifs/protocol/cifs.asp`.

[7] 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*, 28(5):749-771, 2002.

[8] D. Hildebrand and P. Honeyman, "Exporting Storage Systems in a Scalable Manner with pNFS," in *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Monterey, CA, 2005.

[9] S. Shepler, M. Eisler, and D. Noveck, NFSv4 Minor Version 1. Internet Draft, 2006.

[10] D.L. Black and S. Fridella, pNFS Block/Volume Layout. Internet Draft, 2006.

[11] B. Halevy, B. Welch, J. Zelenka, and T. Pisek, Object-based pNFS Operations. Internet Draft, 2006.

[12] Parallel Virtual File System - Version 2, `www.pvfs.org`.

[13] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2002.

[14] Panasas Inc., "Panasas ActiveScale File System," `www.panasas.com`.

[15] EMC Celerra HighRoad Whitepaper, `www.emc.com`, 2001.

[16] R.O. Weber, SCSI Object-Based Storage Device Commands (OSD). Storage Networking Industry Association. ANSI/INCITS 400-2004, `www.t10.org`, 2004.

[17] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, Internet Small Computer Systems Interface (iSCSI). RFC 3720, 2001.

[18] R. Srinivasan, RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, 1995.

[19] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference, volume 2--The MPI-2 Extensions.* Cambridge, MA: MIT Press, 1998.

[20] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[21] D. Hildebrand, L. Ward, and P. Honeyman, "Large Files, Small Writes, and pNFS," in *Proceedings of the 20th ACM International Conference on Supercomputing*, Cairns, Australia, 2006.

[22] IOR Benchmark, `www.llnl.gov/asci/purple/benchmarks/limited/ior`.

[23] bbcp. `www.slac.stanford.edu/~abh/bbcp`.

[24] Cluster File Systems Inc., Lustre: A Scalable, High-Performance File System. `www.lustre.org`, 2002.

[25] P. Andrews, C. Jordan, and W. Pfeiffer, "Marching Towards Nirvana: Configurations for Very High Performance Parallel File Systems," in *Proceedings of the HiperIO Workshop*, Barcelona, Spain, 2006.

[26] P. Andrews, C. Jordan, and H. Lederer, "Design, Implementation, and Production Experiences of a Global Storage Grid," in *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, 2006.

[27] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003.

[28] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi, "Grid Datafarm Architecture for Petascale Data Intensive Computing," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002.

[29] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[30] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Canada, 1998.

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

[32] 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*, 17(1):21-37, 2003.

[33] P. Lombard and Y. Denneulin, "nfsp: A Distributed NFS Server for Clusters of Workstations," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, 2002.