Exporting Storage Systems in a Scalable Manner with pNFS

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

To meet enterprise and grand challenge-scale performance and interoperability requirements, a group of engineers—initially ad-hoc but now integrated into the IETF—is designing extensions to NFSv4 that provide parallel access to storage systems. This paper gives an overview of pNFS, an emerging NFSv4 extension that promises file access scalability plus operating system and storage system independence. pNFS bypasses the server bottleneck by enabling direct access to storage by NFSv4 clients and by providing a framework for the co-existence of NFSv4 with other file access protocols. In this paper, we describe an implementation that demonstrates and validates pNFS' potential. The I/O throughput of our prototype matches that of its exported file system and far exceeds standard NFSv4.

1. Introduction

Protocol standards enable interoperability and reduce development and management costs, but are only as useful as the number of parties that use them. Increasing performance requirements have spawned innovative protocols such as iSCSI [1], DAFS [2], OSD [3] and FCP [4], yet the emergence of so many standards threatens to reduce interoperability among storage systems.

Interoperability also depends on the operating system and hardware platform. High-performance file systems, which provide direct and parallel access to storage, are highly specialized, and often limited to a single operating system and hardware platform. However, grid computing, legacy software, and other factors are increasing the heterogeneity of clients, creating a schism between file systems and their users.

Many application domains demonstrate the need for high bandwidth, concurrent, and secure access to large datasets across a variety of platforms and file systems. DNA sequence, face and other biometrics, and artwork databases are just a few examples of files that can range up to tens of gigabytes in size and are often loaded independently by concurrent clients [5, 6]. Full database searches are often unavoidable even when using indexing [7].

The Earth Observing System Data and Information System (EOSDIS) manages data from NASA's earth science research satellites and field measurement programs, providing data archive, distribution, and information management services. In August 1999, EOSDIS data holdings were estimated at 284 TB while continuing to generate more than 850 GB per day. In 2000, EOSDIS supported more than 104,000 unique users and fulfilled more than 3.4 million product requests [8].

Digital movie studios generate terabytes of data every day and require access from Sun, Windows, SGI, and Linux workstations and compute clusters [9]. Users edit files in place or copy files between heterogeneous data stores.

High end scientific computing performs physical simulations with visualization and fault-tolerance check pointing. The Advanced Simulation and Computing program in the U.S. Department of Energy estimates that one GB/s of aggregate I/O throughput is necessary for every teraflop of computing power [10], which suggests that file systems will need to support data transfer rates of 500 GB/s by 2008.

Distributed file systems such as NFS [11] and CIFS [12] are widely used to bridge the interoperability gap, but their performance is only a fraction of the exported storage system's. To this day, they continue to have limited network, CPU, memory, and disk I/O resources due to their "single server" design, which binds one network endpoint to all files in a file system. NFSv4 [13] improves functionality by providing integrated security and locking frameworks, and migration and replication features, but retains the single server bottleneck.

Partitioning a collection of files among multiple NFS servers helps work around this limitation but increases management cost and fails to address scalable access to a single file or directory, a critical requirement of today's high performance applications[1] [7]. Some progress has been made in aggregating partitioned NFS servers into a

[1] Many programs will generate a single large file instead of many smaller ones to ease application development and data management.

single file system image [14-16], but these systems are unable to export third party file systems.

Distributed file systems are at another disadvantage when their view of storage is through a file system node. Data must always travel through the intermediary node whether it is traveling in or out of storage. This extra layer of processing prevents distributed file systems from matching the performance of the exported file system, even for a single client.

A common architectural framework should be able to encompass all storage architectures, i.e., symmetric or asymmetric[2]; in-band or out-of-band; and block-, object-, or file-based; without sacrificing performance. The NFSv4 file service, with its global namespace, high level of interoperability and portability, simple and cost-effective management, and integrated security provides an ideal base for such a framework.

This paper gives an overview of pNFS [17, 18] and describes a prototype implementation. pNFS is an extension of NFSv4 that provides file access scalability *plus* operating system, hardware platform, and storage system independence. It eliminates the performance bottlenecks of NFS by enabling the NFSv4 client for direct storage access. pNFS facilitates interoperability between standard protocols by providing a framework for the co-existence of NFSv4 and all other file access protocols. We have implemented a prototype that demonstrates and validates pNFS' potential. The I/O throughput of our prototype equals that of its exported file system (PVFS2 [19]) and is dramatically better than standard NFSv4.

The remainder of this paper is organized as follows. Section 2 describes the pNFS architecture. Sections 3 and 4 present PVFS2 and our pNFS prototype. Section 5 reports our measurements of the performance of our Linux-based prototype. Section 6 discusses related work. Section 7 discusses future work, including the impact of locking and security support on the pNFS architecture. Section 8 summarizes and concludes the paper.

2. pNFS architecture

In pNFS, the NFS client and server continue to perform control and file management operations and relegate the responsibility for achieving scalable I/O throughput to a storage-specific driver. By separating control and data flows, pNFS allows data to transfer in parallel from many clients to many storage endpoints. This removes the single server bottleneck by distributing I/O across the bisectional bandwidth of the storage network between the clients and storage devices.

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

Figure 1 depicts the architecture of pNFS, which adds a layout driver, an I/O driver, and a file layout retrieval interface to the standard NFSv4 architecture.

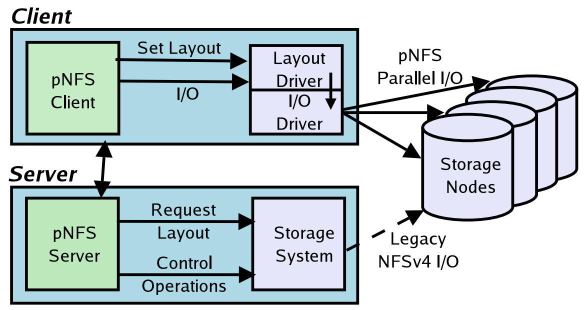

Figure 1. pNFS architecture
pNFS extends NFSv4 with the addition of a layout driver, an I/O driver, and a file layout retrieval interface. The pNFS server obtains an opaque file layout map from the storage system and transfers it to the pNFS client and subsequently to its layout driver for direct and parallel data access.

A benefit of pNFS is its ability to match the performance of the underlying storage system's native client while continuing to support all standard NFSv4 features. This support is ensured by introducing pNFS extensions into a "minor version", a standard extension mechanism of NFSv4. In addition, pNFS does not impose restrictions that might limit the underlying file system's ability to provide quality-enhancing features such as usage statistics or storage management interfaces.

2.1. Design goals

The goals of pNFS are:

- Enable implementations to match or exceed the performance of the underlying file system. Provide high per-file, per-directory and per-file system bandwidth and capacity.
- Support any storage protocol, including but not limited to block, object, and file storage protocols.
- Obey NFSv4 minor versioning rules, which state that all future versions must have legacy support.
- Operate over any NFSv4 internet infrastructure. Support existing storage protocols and infrastructures, e.g., SBC on Fibre Channel [20] and iSCSI, OSD on Fibre Channel and iSCSI, NFSv4, etc.
- Handle arbitrarily large file layout maps.

2.2. Layout and I/O driver

The *layout driver* understands the file layout of the storage system. A layout consists of all information required to access any byte range of a file. For example, a block layout may contain information about block size, offset of the first block on each storage device, and an array of tuples that contains device identifiers, block numbers, and block counts. An object layout specifies the storage devices for a file and the information necessary to translate a logical byte sequence into a collection of objects. A file layout is similar to an object layout but uses file handles instead of object identifiers. The layout driver uses the layout to translate read and write requests from the pNFS client into I/O requests understood by the storage devices. The I/O driver performs raw I/O, e.g., Myrinet GM [21], Infiniband [22], TCP/IP, to the storage nodes.

To ensure support for all I/O protocols, every pNFS implementation must include a standard interface that the layout driver implements. The layout driver can be specialized or (preferably) implement a standard protocol such as the Fibre Channel Protocol (FCP), allowing multiple file systems to all share the same layout driver. Storage systems adopting this architecture reduce development and management obligations by obviating a specialized file system client, which reduces the cost of high-end storage systems.

2.3. NFSv4 protocol extensions

2.3.1. File system attribute

A new file system attribute, LAYOUT_CLASSES, contains the supported layout drivers. A pNFS client retrieves this attribute when encountering an unknown file system identifier and uses it to select an appropriate layout driver. To prevent namespace collisions, a global registry maintainer such as IANA [23] will store the layout driver identifiers.

2.3.2. LAYOUTGET

The LAYOUTGET operation obtains file access information for a byte-range of a file, e.g., the file layout, from the underlying storage system. The client issues a LAYOUTGET operation after it opens a file and before it accesses file data. Implementations determine the frequency and byte range of the request. A new procedure is required since some systems limit attribute size.

The arguments are:

- File handle
- Offset
- Extent
- Access type
- Open owner
- Maximum count and cookie

The file handle uniquely identifies the file. The offset and extent arguments specify the requested region of the file. The access type specifies whether the requested file layout information is for reading, writing, or both. This is useful for file systems that, for example, provide read-only replicas of data. The server uses the NFSv4 "open owner" to verify that the process' file access permissions are valid and to renew lease timeouts for the client. The maximum count specifies the maximum number of bytes for the result, including XDR overhead. The client retrieves the remaining layout information using the cookie, similar to the NFSv4 READDIR operation.

The returned values are:

- Offset
- Extent
- Cookie
- Opaque layout

The returned offset and extent values must describe a byte-range at least as large as the requested size. By returning file layout information to the client as an opaque object, pNFS is able to support all 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. The byte range described by the returned layout may be larger than the requested size due to block alignments, layout prefetching, etc.

2.3.3. LAYOUTCOMMIT

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.

2.3.4. LAYOUTRETURN

This operation informs the server that obtained layout information is no longer required. Clients return a layout voluntarily or when they receive a server recall request.

2.3.5. CB_LAYOUTRECALL

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.[3] 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.

2.3.6. GETDEVINFO and GETDEVLIST

The GETDEVINFO and GETDEVLIST operations retrieve additional information about one or more storage nodes. The layout driver executes these operations if the

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

device information inside the file layout does not provide enough information for file access, e.g., SAN volume label information or port numbers.

3. Parallel Virtual File System Version 2

As proof of concept, we implemented a pNFS prototype that exports the PVFS2 file system. This section presents an overview of PVFS2, a user-level, open-source, scalable, asymmetric parallel file system designed as a research tool and for production environments. We chose PVFS2 because its user level design provides a streamlined architecture for rapid prototyping of new ideas, which overrides its lack of locking and security support. Figure 2 displays the PVFS2 architecture.

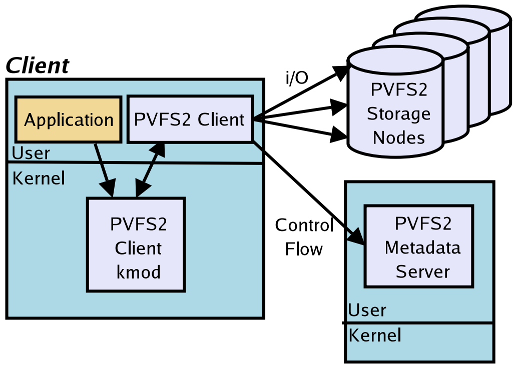

Figure 2. PVFS2 architecture
PVFS2 consists of clients, metadata servers, and storage nodes. The PVFS2 kernel module enables integration with the local file system. Data is striped across storage nodes using a user-defined algorithm.

PVFS2 consists of clients, storage nodes, and metadata servers. Metadata servers store all information about the file system in a Berkeley DB database, distributing metadata via a hash on the file name. File data is striped across storage nodes, which can be increased in number as needed.

PVFS2 uses algorithmic file layouts for distributing data among the storage nodes. The data distribution algorithm is user defined, defaulting to round-robin striping. The clients and storage nodes share the data distribution algorithm, which does not change during the lifetime of the file. A series of file handles, one for each storage node, uniquely identifies the set of file data stripes. Data is not committed with the metadata server; instead, the client ensures that all data is committed to storage by negotiating with each individual storage node.

An operating system specific kernel module exists for integration into a user's environment and access by other file systems such as NFS. It allows users to mount and access PVFS2 through a POSIX interface. Currently, an implementation of this module exists only on Linux. Data is memory mapped between the kernel module and the PVFS2 client program to avoid extra data copies.

Large parallel applications generally manage data consistency through organized and cooperative clients instead of locks. As such, PVFS2 breaks POSIX consistency semantics, which require sequential consistency of file system operations, and replaces them with *nonconflicting writes* semantics, guaranteeing that writes to non-overlapping file regions will be visible on all subsequent reads once the write completes.

4. pNFS prototype

Prototypes of new protocols are essential for their clarification and provide insight and evidence of their viability. A minimum requirement for the fitness of pNFS is its ability to provide parallel access to arbitrary storage systems. This agnosticism toward storage system particulars is vital for widespread adoption. As such, our prototype focuses on the retrieval and processing of the file layout to demonstrate that pNFS is agnostic of the underlying storage system and can match the performance of the storage system it exports. Figure 3 displays the architecture of our pNFS prototype with PVFS2 as the exported file system.

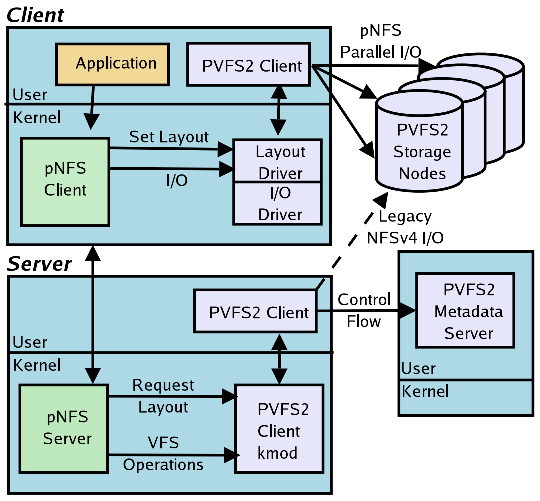

Figure 3. pNFS prototype architecture
Layout and I/O drivers communicate with the PVFS2 user mode client for data access. The pNFS server obtains the opaque file layout from the PVFS2 metadata server via the PVFS2 client, transferring it back to the pNFS client and subsequently to the PVFS2 layout driver for direct and parallel data access.

4.1. File Layout

Numerous file layout schemes exist today, and more will be invented in the future, so pNFS is intentionally absent of any knowledge of the underlying file system's file layout information. Among the possible ways to distribute data among storage nodes are:

- *Round Robin* – Blocks striped on storage nodes in round robin fashion, e.g., RAID0.
- *Replicated* – Each block exists on more than one storage node, e.g., RAID1.
- *Parity* – Round robin striping with distributed parity, e.g., RAID5.
- *Nested* – Layouts composed from simpler ones [24].

The PVFS2 file layout information consists of:

- File system id
- Set of file handles, one for each storage node
- Distribution id, uniquely defines layout algorithm
- Distribution parameters, e.g., stripe size

Since a PVFS2 layout applies to an entire file, no matter what byte range the pNFS client requests using the LAYOUTGET operation, the returned byte range is the entire file. Therefore, our prototype requests a layout only once for each open file, incurring only a single additional round trip. If the pNFS client is eager with its requests, it can even eliminate this single round trip time by including the LAYOUTGET in the same request as the OPEN operation. We will see the differences between these two designs in Section 5.

The pNFS server obtains the layout via a new ioctl operation in the PVFS2 client kernel module. Ideally, a new Linux layout retrieval VFS operation will supplant this.

4.2. Layout and I/O Drivers

The PVFS2 layout driver registers itself with the pNFS client along with a unique identifier. The pNFS client matches this identifier with the value of the LAYOUT_CLASSES attribute to select the correct layout driver for file access. If there is no matching layout driver, standard NFSv4 read and write mechanisms are used as the default.

The PVFS2 layout driver, a pared down version of a PVFS2 client, supports only three operations: read, write, and a layout injection ioctl. Our prototype layout driver registers these operations, defined by the file_operations structure, with the pNFS client.

The syntax for these functions is:

```
ssize_t read(struct file* file,char __user* buf,
             size_t count, loff_t* offset)
ssize_t write(struct file* file,const char __user*
              buf,size_t count,loff_t* offset)
int     ioctl(struct inode* ino,struct file* file,
              unsigned int cmd,unsigned long arg)
```

To inject the file layout map, the pNFS client passes the opaque array as an argument to the ioctl function. Once the layout driver has finished processing the layout, the pNFS client is free to call the driver's read and write functions. When data access is complete, the pNFS client issues a standard NFSv4 close operation to the server.

5. Evaluation

In this section, we present the results of experiments that assess the performance of our pNFS prototype. We demonstrate that pNFS can use the generic layout driver interface to scale with PVFS2, and can achieve performance vastly superior to NFSv4.

Our experiments were performed on a network of forty identical nodes partitioned into twenty-three clients, sixteen storage nodes, and one metadata server. Each node is a 2 GHz dual-processor Opteron with 2 GB of DDR RAM and four Western Digital Caviar Serial ATA disks, which have a nominal data rate of 150 MB/s and an average seek time of 8.9 ms. The disks are configured with software RAID 0. The operating system kernel is Linux 2.6.9-rc3. The version of PVFS2 is 1.0.1.

We test four configurations: two accessing PVFS2 storage nodes directly via pNFS and PVFS2 clients; and two with unmodified NFSv4 clients, one accessing an Ext3 file system, and one accessing a PVFS2 file system with the NFSv4 server, exported PVFS2 client and PVFS2 metadata server all residing on the metadata server. The metadata server runs eight pNFS or NFSv4 server threads when exporting the PVFS2 or Ext3 file systems. We verified that varying the number of pNFS or NFSv4 server threads does not affect its performance.

We compare the aggregate I/O throughput using the IOZone [25] benchmark tool as we increase the number of clients. Since we see pNFS as a possible replacement for high-performance file system clients, our goal is for pNFS to match PVFS2 performance. The first set of experiments involves two processes on each client reading and writing separate 200 MB files. The second set of experiments involves each client reading and writing disjoint 100 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 several executions of the benchmark. The write timing includes a flush of the client's cache to the server. pNFS and PVFS2 perform synchronous data and metadata updates while NFSv4 exports Ext3 and PVFS2 synchronously. All read experiments use warm storage node caches to eliminate disk access irregularities.

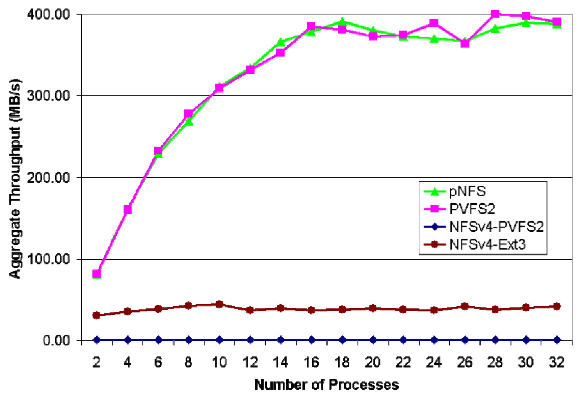

Figure 4. Aggregate write throughput with sixteen clients and separate files. Each client spawns two write processes. pNFS and PVFS2 use sixteen storage nodes. pNFS scales with PVFS2 while NFSv4 performance remains flat.

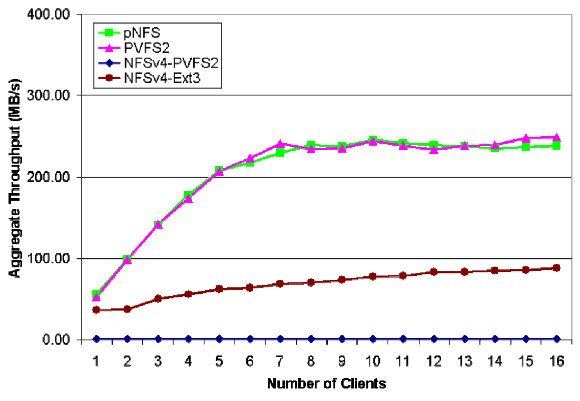

Figure 5. Aggregate write throughput with sixteen clients and a single file. pNFS and PVFS2 use sixteen storage nodes. pNFS scales with PVFS2 while NFSv4 performance remains flat with PVFS2 and approaches the maximum single link bandwidth with Ext3.

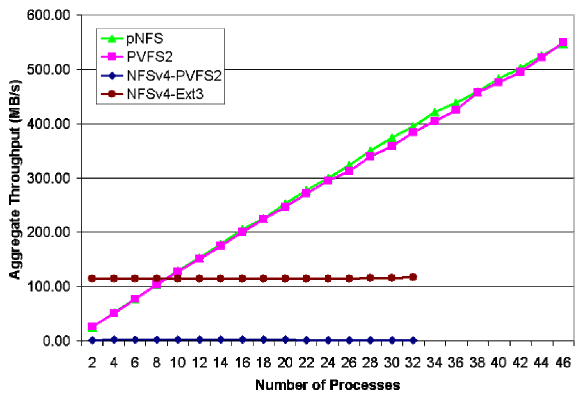

Figure 6. Aggregate read throughput with twenty-three clients and separate files. Each client spawns two read processes. pNFS and PVFS2 use sixteen storage nodes. pNFS and PVFS2 scale linearly while NFSv4 performance remains flat.

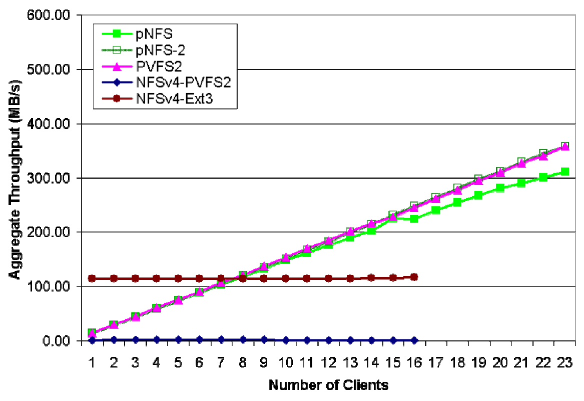

Figure 7. Aggregate read throughput with twenty-three clients and a single file. pNFS and PVFS2 use sixteen storage nodes. pNFS and PVFS2 scale linearly while NFSv4 performance remains flat. pNFS performance is slightly below PVFS2 due to increasing layout retrieval congestion. pNFS-2, which removes the extra round trip time of LAYOUTGET, matches PVFS2's performance.

Our first experiment investigates the overhead of the LAYOUTGET operation in pNFS with a single client. Unlike NFS versions 2 and 3, NFSv4 is a stateful protocol in which a client must call a server to open a file; the first time the client opens the file, it must also send a call to the server to coordinate sequence numbers. With PVFS2, only a single LAYOUTGET is required after the client opens the file, so that the overhead is a single roundtrip or none at all, if it is included together with the open request.

In the worst-case, a LAYOUTGET request is required on every read or write. In our test environment, the time for a LAYOUTGET request is 0.85 ms. On a 1MB transfer, this reduces throughput by only 3-4 percent; with a 10MB transfer, the relative cost is less than 0.5 percent; and is negligible as transfer size increases. The worst-case scenario should be a rare occurrence as, similar to read-ahead algorithms for reading data, clients can be optimistic in the ranges they request using the LAYOUTGET operation.

In all experiments, the performance of NFSv4 exporting PVFS2 achieves an aggregate read and write throughput of 1.9 MB/s and 0.9 MB/s respectively. We discuss the causes for this poor performance in Section 5.1.

Figures 4 and 5 present the write performance with each client writing to separate files and a single file. NFSv4 with Ext3 achieves an average aggregate throughput of 38 MB/s and 68 MB/s for the separate and single file experiments. pNFS scales equivalently to PVFS2, reaching a maximum aggregate throughput of 384 MB/s with sixteen processes for separate files and 240 MB/s with seven clients for a single file. With separate files, the bottleneck is the number of storage nodes while metadata processing limits the performance with a single file.

Figure 6 shows the read performance with two processes on each client writing to separate files. NFSv4 with Ext3 achieves its maximum network bandwidth of 115 MB/s. pNFS again achieves the same performance as PVFS2. Initially, the extra overhead required to write to sixteen storage nodes reduces throughput for two processes to 27 MB/s, but it scales almost linearly, reaching an aggregate throughput of 550 MB/s with 46 processes.

Figure 7 shows the read performance with each client writing to disjoint portions of the same pre-existing file. NFSv4 with Ext3 again achieves its maximum network bandwidth of 115 MB/s. PVFS2 scales linearly, starting with an aggregate throughput of 15 MB/s with a single client and increasing to 360 MB/s with twenty-three clients. Our pNFS prototype, which incurs a single round trip time for the LAYOUTGET, suffers slightly as the PVFS2 layout retrieval function takes longer with increasing numbers of clients, reaching an aggregate throughput of 311 MB/s. A modified prototype combines the LAYOUTGET and OPEN operations into a single call. When the (non-scalable) LAYOUTGET operation is excluded from the measurements, the prototype labelled pNFS-2, matches the performance of PVFS2.

5.1. Discussion

As Figure 7 demonstrates, pNFS scalability can be adversely affected if the LAYOUTGET operation is costly or does not scale with the number of clients. One way a pNFS client can reduce the number of layout retrievals is to request the entire file layout when it opens the file. If the file system fulfills the request, the client avoids an extra round trip. If the file system cannot fulfill such a request, e.g., large block layouts, it returns nothing and the pNFS client requests the layout when it knows the range of the file it will access. The layout driver may also be a useful guide to the pNFS client regarding layout retrieval.

One reason for the poor performance of NFSv4 with PVFS2 is a difference in block sizes. Per-read and per-write processing overhead is small in NFSv4, which justifies a small block size—32KB on Linux. PVFS2 has a much larger per-read and per-write overhead due to the larger number of servers it must contact, and therefore uses a minimum block size of 4MB. In addition, PVFS2 does not perform write gathering on the client, assuming each data request to be a multiple of the block size. To make matters worse, the Linux kernel breaks down the NFSv4 client's request on the NFSv4 server into 4KB chunks before it issues the requests to the PVFS2 client. Data transfer overhead, e.g., creating connections to the storage nodes, determining stripe locations, etc., dominates with 4KB requests, with a devastating impact on performance.

The lack of a commit operation in the PVFS2 kernel module also reduces the write performance of NFSv4 with PVFS2. To prevent data loss, PVFS2 commits every write operation, ignoring the NFSv4 COMMIT operation. Write gathering [26] on the server combined with a commit from the PVFS2 client would comply with NFSv4 fault tolerance semantics and improve PVFS2's interaction with the disk.

6. Related work

AFS [27] 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 problems, which precludes its use in a WAN environment.

GridFTP [28] 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.

The Storage Resource Broker (SRB) [29] aggregates storage resources, e.g., a file system, an archival system, or a database, into a single data catalogue. SRB also has some serious limitations: it does not enable parallel I/O to multiple storage endpoints, and cannot integrate with the local file system.

Many high-performance file systems exhibit a lack of interoperability and portability and can benefit from the open standards, reduced development costs, and storage service agnosticism of pNFS. One type is limited to storage area networks (SANs), a network that utilizes the fixed-sized block SCSI storage device command set and its Fibre Channel SCSI transport. Examples in this class include IBM's TotalStorage SAN FS [30], GPFS [31], Red Hat's GFS [32] and Veritas' SANPoint Direct [33]. The Fibre Channel network limitation may disappear with the emergence of iSCSI.

EMC's HighRoad [34] uses the NFS or CIFS protocol for its control operations and stores data in an aggregated LAN and SAN environment. Its use of file semantics facilitates data sharing in SAN environments, but is limited to the EMC Symmetrix storage system.

Another type utilizes SCSI's newly emerging command set, Object Storage Device (OSD), which transmits variable length storage objects over SCSI transports. Examples in this class include Panasas' ActiveScale [35], Lustre [36] and an object based version of IBM's TotalStorage SAN FS [37].

Disk striping [38] is not a new concept and was first utilized by by the I/O subsystems of early super computers [39]. To our knowledge, the Swift file system [40] was the first to stripe data across multiple servers in a distributed environment.

7. Future work

7.1. Locking

Mandatory locking requires an additional piece of shared state between the NFSv4 client and server, a unique identifier of the locking process. This state is in addition to the client identifier that identifies the client machine. Locking support mandates that the client sends the locking identifier along with every read and write operation.

How pNFS clients will utilize their locks on the storage nodes is not standardized as of the writing of this paper. Several possibilities exist: enable the storage nodes to interpret NFSv4 lock identifiers, bundle a new pNFS operation to retrieve file system specific lock information with the NFSv4 LOCK operation, or include lock information in existing opaque file layout.

7.2. Security considerations

Separate control and data paths in pNFS introduce new security concerns to NFSv4. Although RPCSEC_GSS will continue to secure the NFSv4 control path, securing the data path requires additional effort. The current pNFS operations Internet Draft [41] does not define the new security architecture, instead describing the general mechanisms that will be required.

It is expected that file storage protocols will use the same security mechanisms between the client and storage nodes as it does between the pNFS client and server.

Object storage employs revocable cryptographic capabilities for file system objects that the metadata server passes to clients. For data access, the layout driver requires the correct capability in order to access the storage nodes. It is expected that the capability will be passed to the layout driver within the opaque layout object.

Block storage access protocols rely on SAN-based security, trusting clients to access only their allotted blocks. LUN masking/unmapping and zone-based security schemes can also fence in clients to specific data blocks. Some systems also employ IPsec to secure the data stream. Placing more trust in the client for SAN file systems is a change to the NFS trust model.

7.3. Caching

PVFS2 does not currently have a client data cache, and therefore neither does our pNFS prototype. Investigation is required to determine if standard NFSv4 consistency mechanisms are sufficient for use with high-performance file systems.

File layout caching will become critical depending on the access pattern. Our prototype caches file layout information while a file is open, but does not include support for modifying the layout or its invalidation by the pNFS server. The implementation of the remaining pNFS operations, LAYOUTRETURN, LAYOUTCOMMIT, and CB_LAYOUTRECALL will clarify file layout issues.

7.4. Additional issues and features

The following themes are also under investigation:

- Investigation is required into the impact that large file layouts.
- MPI-IO support.
- Strided LAYOUTGET requests.
- Group LAYOUTGET requests.
- Symmetric pNFS servers.

8. Conclusions

This paper describes an implementation of pNFS, an NFSv4 extension that bypasses the server bottleneck, enabling direct and parallel storage access. pNFS clients can interact with multiple storage systems on multiple hardware platforms, or access a single file via multiple I/O protocols. In time, open-source pNFS clients may obviate specialized client support for access to high-performance file systems.

Our pNFS prototype demonstrates that it is possible to achieve high throughput access to a high-performance file system while retaining the file system independence of the NFSv4 protocol. Experiments demonstrate that the prototype achieves aggregate throughput equal to that of its exported file system and far exceeds standard NFSv4 performance. As the high performance community continues to manage larger and larger files in ever more diverse environments, we see pNFS as an indispensable tool for data access.

9. Acknowledgements

This work supported by Sandia National Labs under contract number 272684. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. We thank Jim Schutt, Ruth Klundt, Lee Ward, Gary Grider and James Nunez for their valuable insights and system support and Rob Ross, Neill Miller and Robert Latham for their help with PVFS2.

References

[1] J. Satran, D. Smith, K. Meth, O. Biran, J. Hafner, C. Sapuntzakis, M. Bakke, M. Wakeley, L. Dalle Ore, P. Von Stamwitz, R. Haagens, M. Chadalapaka, E. Zeidner, and Y. Klein, "iSCSI," *IPS Internet Draft*, *www.ietf.org/internetdrafts/draft-ietf-ips-iscsi-08.txt*, 2001.

[2] DAFS Collaborative, "DAFS: Direct Access File System Protocol," www.dafscollaborative.org, September 2001.

[3] T10 Committee, "Draft OSD Standard," *Storage Networking Industry Association (SNIA)*, www.t10.org/ftp/t10/drafts/osd/osd-r10.pdf, July 2004.

[4] T10 Committee, "Draft Fibre Channel Protocol - 3 (FCP-3) Standard," www.t10.org/ftp/t10/drafts/fcp3/fcp3r03d.pdf, January 2005.

[5] S. Chacko and S. Fellini, "Easy Large-Scale Bioinformatics on the NIH Biowulf Supercluster," biowulf.nih.gov/present/easy.html, 2003.

[6] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D Steele, and P. Yanker, "Query by Image and Video Content: the QBIC System," *IEEE Computer*, September 1995.

[7] 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.

[8] "Earth Observing System Data and Information System," spsosun.gsfc.nasa.gov/eosinfo/EOSDIS_Site/index.html.

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

[10] "SGS File System RFP," *DOE NNCA and DOD NSA*, April 25, 2001.

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

[12] Microsoft Corporation, "CIFS Protocol," msdn.microsoft.com/library/default.asp?url=/library/en-us/cifs/protocol/cifs.asp.

[13] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network File System Version 4 Protocol Specirfication," www.ietf.org/rfc/rfc3530.txt, April 2003.

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

[15] 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.

[16] P. Lombard and Y. Denneulin, "nfsp: A Distributed NFS Server for Clusters of Workstations," *IPDPS 2002*, 2002.

[17] G. Gibson and P. Corbett, "pNFS Problem Statement," *Internet Draft*, www.ietf.org/internet-drafts/draft-gibson-pnfs-problem-statement-01.txt, July 2004.

[18] G. Gibson, B. Welch, G. Goodson, and P. Corbett, "Parallel NFS Requirements and Design Considerations," *Internet Draft*, www.ietf.org/internet-drafts/draft-gibson-pnfs-reqs-00.txt, October 2004.

[19] PVFS2 Development Team, "Parallel Virtual File System, Version 2," www.pvfs.org/pvfs2, September 2003.

[20] T.M. Anderson and R.S. Cornelius, "High Performance Switching with Fibre Channel," *Digest of Papers Compcon*, pp. 261-268, 1992.

[21] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, and C.L. Seitz, "Myrinet A Gigabit-per-Second Local-Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.

[22] "Infiniband. Arch. Spec. Vol 1 & 2. Rel. 1.0," www.infinibandta.org/download_spec10.html, 2000.

[23] "Internet Assigned Numbers Authority," www.iana.org.

[24] F. Isaila and W.F. Tichy, "Clusterfile: A Flexible Physical Layout Parallel File System," *3rd IEEE International Conference on Cluster Computing*, October 2001.

[25] W.D. Norcott and D. Capps, "IOZone Filesystem Benchmark," `www.iozone.org`, 2003.

[26] C. Juszczak, "Improving the Write Performance of an NFS Server," *In Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 247--259, 1994.

[27] 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.

[28] 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.

[29] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," *In Proceedings of CASCON'98*, 1998.

[30] "IBM TotalStorage: Introducing the SAN File System," `www.redbooks.ibm.com/redbooks/pdfs/sg247057.pdf`, November 2003.

[31] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *USENIX Conference on File and Storage Technologies*, 2002.

[32] Red Hat Software Inc., "Red Hat Global File System," `www.redhat.com/software/rha/gfs`.

[33] "VERITAS SANPoint Direct™ File Access," `www.veritas.com`, April 2002.

[34] "EMC Celerra HighRoad Whitepaper," `www.emc.com`, December 2001.

[35] Panasas Inc., "Panasas ActiveScale File System Datasheet," `www.panasas.com`, 2003.

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

[37] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an Object Store," *IBM Storage Systems Technology Workshop*, November 2002.

[38] Kenneth Salem and Hector Garcia-Molina, "Disk Striping," *In Proceedings of the 2nd International Conference on Data Engineering*, pp. 336-342, 1986.

[39] O. G. Johnson, "Three-dimensional wave equation computations on vector computers," *In Proceedings of the IEEE*, vol. 72, January 1984.

[40] L. Cabrera and D.D.E. Long, "SWIFT: Using Distributed Disk Striping To Provide High I/O Data Rates," *Computing Systems*, vol. 4, pp. 405-436, 1991.

[41] B. Welch, B. Halevy, D. Black, A. Adamson, and D. Noveck, "pNFS Operations Summary," *Internet Draft*, `www.ietf.org/internet-drafts/draft-welch-pnfs-ops-00.txt`, October 2004.