CITI Technical Report 06-04

LARGE FILES, SMALL WRITES, AND pNFS

Dean Hildebrand dhildebz@umich.edu
Lee Ward lee@sandia.gov
Peter Honeyman honey@citi.umich.edu

ABSTRACT

Workload characterization studies highlight the prevalence of small and sequential data requests in scientific applications. Parallel file systems excel at large data transfers but sometimes at the expense of small I/O performance. pNFS is an NFSv4.1 high-performance enhancement that provides direct storage access to parallel file systems while preserving NFSv4 operating system and hardware platform independence. This paper demonstrates that distributed file systems can increase write throughput to parallel data stores—regardless of file size—by overcoming parallel file system inefficiencies. We also show how pNFS can improve the overall write performance of parallel file systems by using direct, parallel I/O for large write requests and a distributed file system for small write requests. We describe our pNFS prototype and present experiments demonstrating the performance improvements.

May 10, 2006

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

LARGE FILES, SMALL WRITES, AND pNFS

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

Lee Ward
Scalable Computing Systems Department
Sandia National Laboratories
lee@sandia.gov

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

ABSTRACT
Workload characterization studies highlight the prevalence of small and sequential data requests in scientific applications. Parallel file systems excel at large data transfers but sometimes at the expense of small I/O performance. pNFS is an NFSv4.1 high-performance enhancement that provides direct storage access to parallel file systems while preserving NFSv4 operating system and hardware platform independence. This paper demonstrates that distributed file systems can increase write throughput to parallel data stores—regardless of file size—by overcoming parallel file system inefficiencies. We also show how pNFS can improve the overall write performance of parallel file systems by using direct, parallel I/O for large write requests and a distributed file system for small write requests. We describe our pNFS prototype and present experiments demonstrating the performance improvements.

Categories and Subject Descriptors
D.4.8 [**Operating Systems**]: Performance – *measurements.*

General Terms
Algorithms, Performance, Design, Experimentation, Standardization

Keywords
Parallel I/O, Parallel File System, NFSv4, pNFS, Distributed File System, Small Write Performance Improvement

1. INTRODUCTION
In recent years, parallel file systems have emerged to meet enterprise and grand challenge-scale data and performance requirements [1-4]. These systems are shattering bandwidth records through large bulk data transfers between thousands of nodes and disks, but overlook the performance of small I/O requests, both to small and large files.

Parallel file systems improve the aggregate throughput of bulk data transfers by scaling disks, disk controllers, network, and servers—every aspect of the system architecture. As system size increases, the cost of locating, managing, and protecting data increases the per-request overhead. This overhead is small relative to the overall cost of large data transfers, but considerable for smaller data requests. Many parallel file systems ignore this high penalty for small I/O, focusing entirely on large data transfers.

Unfortunately, not all data comes in big packages. Numerous workload characterization studies have highlighted the prevalence of small and sequential data requests in modern scientific applications [5-11]. This trend will likely continue since many HPC applications take years to develop, have a productive lifespan of ten years or more, and are not easily re-architected for the latest file access paradigm [12]. Furthermore, many current data access libraries such as HDF5 and netCDF rely heavily on small data accesses to store individual data elements in a common (large) file [13, 14].

Distributed file systems are optimized for small data accesses [15, 16]; not surprisingly, studies demonstrate that small I/O is their middleware niche [17]. However, their "single server" design, which binds one network endpoint to a given collection of files, limits opportunities to scale with network, CPU, memory, and disk I/O resources. NFSv4 [18] improves functionality by providing integrated security and locking frameworks, and migration and replication features, but retains the single server bottleneck.

pNFS [19, 20] is an extension of NFSv4 that provides file access scalability plus operating system, hardware platform, and storage system independence. pNFS overcomes the performance bottlenecks of NFS with parallel file systems by enabling the NFSv4 client to access storage directly. Our earlier work [21] demonstrates that pNFS matches the performance of the native parallel file system client for large data transfers.

This paper investigates the performance of parallel file systems with small writes. We demonstrate that distributed file systems can increase write throughput to parallel data stores—regardless of file size—by overcoming parallel file system small write inefficiencies. We then use pNFS to improve the overall write performance of parallel file systems by using direct, parallel I/O for large write requests and a distributed file system for small write requests. The pNFS heterogeneous metadata protocol allows any parallel file system to realize these write performance improvements.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview pNFS and discusses the prototype used in this paper. Section 4 details the issues with writing small amounts of data in scientific applications. Section 5 describes how pNFS can improve these applications. Section 6 reports the results of our experiments with benchmarks and a real scientific application. Section 7 discusses future work. Section 8 summarizes and concludes the paper.

ICS06, June 28-30, Cairns, Queensland, Australia.

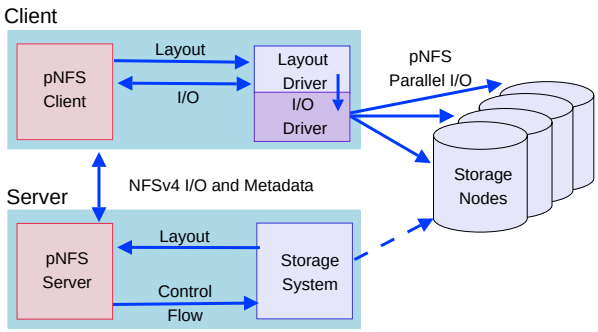


Figure 1. pNFS architecture

pNFS extends NFSv4 by adding 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.

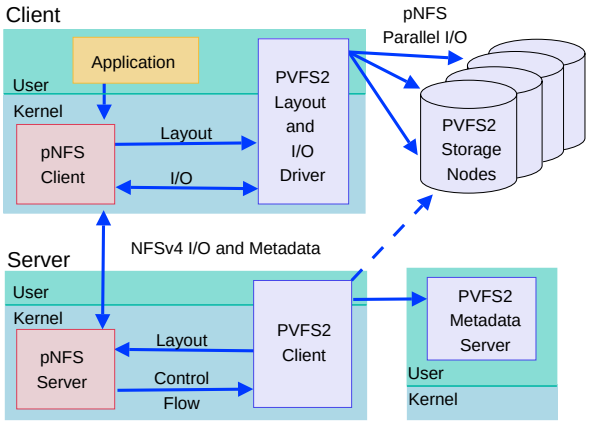


Figure 2. pNFS prototype architecture

The pNFS client uses the PVFS2 layout driver for all I/O. The pNFS server obtains the file layout from the PVFS2 metadata server via the PVFS2 client, transfers it back to the pNFS client, then to the PVFS2 layout driver for direct and parallel data access.

2. RELATED WORK

Log-structured file systems [22] increase the size of writes by appending small I/O requests to a log and then flushing the log to disk. Zebra [23] extends this to distributed environments. Side effects include large file layouts and erratic block sizes.

The Vesta parallel file system [24] improves I/O performance by optimizing data layout on storage through application provided workload characteristic information. Providing this information can be difficult for applications that lack regular I/O patterns or whose I/O access patterns change over time.

Both EMC's Celerra HighRoad file system [25] and the RAID-II network file server [26] transfer small files over the LAN to preserve SAN bandwidth for large file requests, but differentiating small and large files does not help with small requests to large files. This re-direction benefits only large requests, and may reduce the performance of small requests.

GPFS [3] forwards data between I/O nodes for requests smaller than the block size. This reduces the number of messages with the lock manager and possibly reduces the number of read-modify-write sequences.

Both the Lustre [1] and the Panasas ActiveScale [4] file systems use a write-behind cache to perform buffered writes. In addition, Lustre allows clients to place small files on a single storage node to reduce access overhead.

All of these parallel file systems focus primarily on large data transfers, with any small data transfer enhancements an afterthought. pNFS provides an operating system and platform independent architecture with a rich set of existing features that allows parallel file systems to focus on their core strengths.

The MPI-2 standard [27] introduces MPI-IO, a parallel I/O interface that allows applications and their file format libraries, e.g., HDF5, parallel NetCDF, to provide the storage layer with a more precise and global view of application I/O. Implementations of MPI-IO such as ROMIO [28] use application hints and file access patterns to improve single and parallel I/O request performance.

We see our work as beneficial and complementary to MPI-IO and its implementations. MPI-IO benefits applications that use its API and have regular I/O access patterns, e.g., strided I/O. In addition, MPI-IO small write performance continues to be limited by the deficiencies of the underlying parallel file system. Our pNFS enhancements are beneficial for existing and unmodified applications. They are also beneficial at the file system layer of MPI-IO implementations, to improve the performance of the underlying parallel file system.

3. SCALABLE I/O WITH PNFS

This section summarizes the pNFS architecture, described in more detail in an earlier paper [21].

3.1. pNFS Overview

pNFS is a heterogeneous metadata protocol. The NFS client and server perform control and file management operations and delegate the responsibility for I/O 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. Distributing I/O across the bisectional bandwidth of the storage network between clients and storage devices removes the single server bottleneck.

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.

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. 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 I/O—e.g., iSCSI [29], Portals [30], SunRPC [31]—to the storage nodes.

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," an extension mechanism that is part of the NFSv4 standard. In addition, pNFS

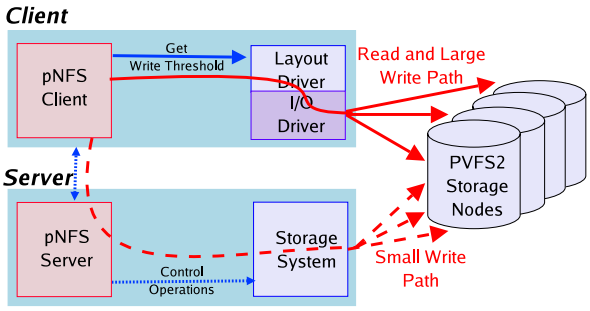


Figure 3. pNFS data paths
pNFS utilizes NFSv4 I/O along the small write path when the write request size is less than the *write threshold*.

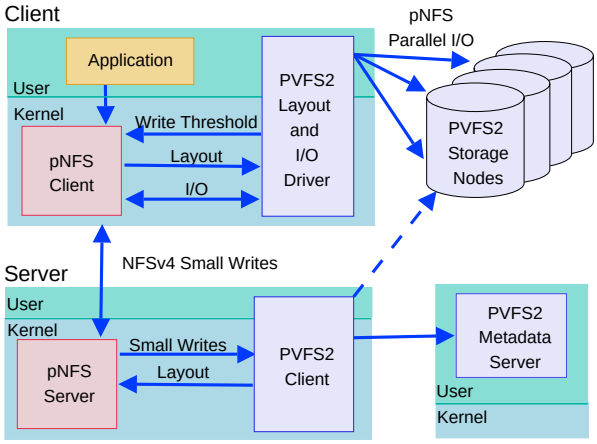


Figure 4. Updated pNFS prototype architecture with write threshold
pNFS retrieves the *write threshold* from PVFS2 layout driver to determine the correct data path for a write request.

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.

3.2. pNFS Prototype

In an earlier paper [21], we describe a pNFS prototype for Linux, depicted in Figure 2, that implements the major features of the pNFS protocol and introduces a general framework for pluggable layout drivers. The I/O throughput of that prototype equals that of its exported file system (PVFS2) and is dramatically better than standard NFSv4.

PVFS2 is a user-level, open-source, scalable, asymmetric parallel file system designed as a research tool and for production environments. Although PVFS2 runs in user-space, an operating system specific kernel module allows integration into a user's environment and access by other file systems such as NFS. This lets users mount and access PVFS2 through a POSIX interface. The PVFS2 client uses memory mapping to avoid copying pages into the kernel.

PVFS2 is designed for the large data needs of scientific applications. These applications access very large files and are generally "write once, read never"—re-reading output is rare. As such, PVFS2 uses large transfer buffers, supports limited request parallelization, incurs a per-request overhead, and does not use a client data or write back cache.

4. SMALL I/O REQUESTS

Several scientific workload characterization studies demonstrate the need to improve performance of small I/O requests to small and large files.

The CHARISMA study [5-7] finds that file sizes in scientific workloads are much larger than those typically found in UNIX workstation environments and that most applications access only a few files. Approximately 90% of file accesses are small—less than 4 KB—and represent a considerable portion of application execution time, even though approximately 90% of the data is transferred in large accesses. In addition, most files are read-only or write-only and are accessed sequentially, but some read-write files are accessed randomly.

The Scalable I/O study [8-10] had similar findings, but remarked that most requests are small writes into GB sized files, consuming 98% of the execution time of one application. Furthermore, it is common for a single node to handle the majority of reads and writes, gathering the data from, or broadcasting the data to the other nodes as necessary. This indicates that single node performance still requires attention from parallel file systems. The study also notes that a lack of portability prevents applications from using enhanced parallel file system interfaces.

A more recent study in 2004 of two physics applications [11] amplifies the earlier findings.

NetCDF (Network Common Data Form) provides a portable and efficient mechanism for sharing data between scientists and applications [13]. NetCDF defines a file format and an API for the storage and retrieval of a file's contents. It is the predominant file format standard within many scientific communities [32]. NetCDF stores data in a single array-oriented file, which contains dimensions, variables, and attributes. Applications individually define and write thousands of data elements, creating many sequential and small write requests.

HDF5 is another popular portable file format and programming interface for storing scientific data in a single file. It provides a richer data model, with emphasis on efficiency of access, parallel I/O, and support for high-performance computing, but continues to define and store each data element separately, creating many small write requests.

This paper demonstrates how pNFS can improve small write performance with parallel file systems for small and large files, regardless of whether an application or file format library generates the write requests.

5. SMALL WRITES AND pNFS

pNFS improves file access scalability by providing the NFSv4 client with support for direct storage access. We now turn to an investigation of the relative costs of the direct I/O path and the NFSv4 path.

5.1. File System I/O Features

A single large I/O request can saturate a client's network endpoint. Engineering a parallel file system for large requests entails the use of large transfer buffers, limited number of asynchronous requests, many storage nodes, and a write-through cache (if a cache even exists).

NFS implementations have several features that allow them to compete with the direct write path:

- **Asynchronous client requests.** Many parallel file systems incur a per-request overhead that is non-negligible for small requests. NFSv4 clients can hand small requests to the NFSv4 server, allowing the server to absorb this overhead without delaying the client application or consuming client CPU cycles. In addition, asynchrony allows request pipelining on the NFSv4 server, reducing aggregate latency to the storage nodes.
- **One server per request.** Data written to a byte-range that spans multiple storage nodes (e.g., multiple stripes) requires two separate requests, further increasing the per-request overhead. The NFSv4 single server design can reduce client request overhead for small requests in these instances.
- **SunRPC.** NFSv4 uses SunRPC, a low-overhead and low-latency network protocol, well suited for small data transfers.
- **Client writeback cache.** NFSv4 increases the efficiency of small write requests by gathering sequential writes requests into a single request.
- **Server write gathering.** The NFSv4 server combines sequential write requests into a single request to the exported parallel file system. This can be useful, e.g., for applications performing strided access into a single file.

5.2. Small Write Performance Example: Postmark Benchmark

To observe a parallel file system's performance loss in a real-world environment, we ran the Postmark benchmark with our pNFS prototype, standard NFSv4, and Ext3. Postmark simulates metadata and small I/O intensive applications such as electronic mail, netnews, and web based services [33]. Postmark creates and performs transactions on a large number of small randomly sized files (between 1 KB and 500 KB). Each transaction first deletes, creates, or opens a file, and then appends 1 KB. Data is sent to stable storage before the file is closed. Postmark performs 2,000 transactions on 100 files. The experiments use eight 1.7 GHz dual P4 processors with gigabit Ethernet. PVFS2 has six storage nodes and one metadata server.

Table 1 shows the Postmark results for Ext3, NFSv4, and pNFS. Ext3 outperforms remote clients, achieving a write throughput of 5.02 MB/s. NFSv4 achieves a write throughput of 4.03 MB/s. pNFS exporting the PVFS2 parallel file system performs poorly, achieving a write throughput of only 0.65 MB/s. This is due to the inability of PVFS2 to parallelize requests effectively and its use of a write-through cache. Using the features discussed in Section 5.1, NFSv4 raises the write throughput to PVFS2 up to 2.4 MB/s. This demonstrates that the parallel, direct I/O path is not always the best choice and the indirect path is not always the worst choice.

Table 1: Postmark write throughput with 1 KB block size. NFSv4 outperforms direct, parallel I/O for small writes.

File System	Write Throughput (MB/s)
Ext3	5.02
NFSv4/Ext3	4.03
pNFS/PVFS2	0.65
NFSv4/PVFS2	2.44

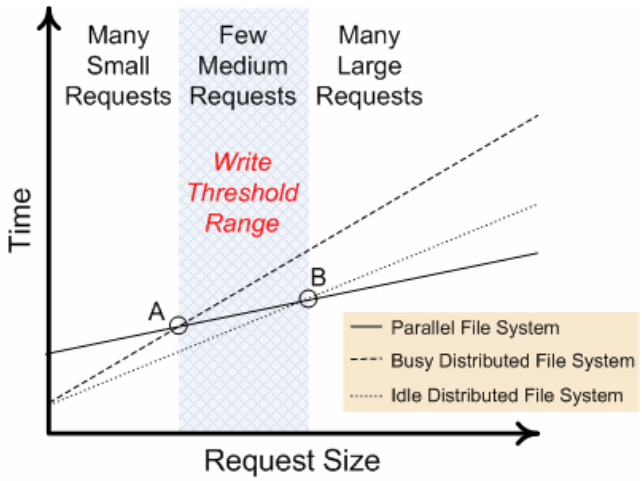


Figure 5. Determining the write threshold value
Write execution time increases with larger request sizes. Application write requests are either small or large, with few requests in the middle. The write threshold can be any value in this middle region.

5.3. pNFS Write Threshold

To use the indirect I/O path for small writes, we modified our pNFS client prototype to allow it to use the NFSv4 I/O protocol as well as the I/O protocol of the underlying file system. To switch between them, we added a *write threshold* to the layout driver. Write requests smaller than the threshold follow the slower NFSv4 data path. Write requests larger than the threshold follow the faster layout driver data path. Figures 3 and 4 illustrate the implementation of the write threshold in both the general pNFS architecture and in our prototype.

pNFS features a heterogeneous metadata protocol that enables it to benefit from the strengths of disparate I/O protocols. A write threshold improves overall write performance for pNFS by hitting the sweet spot of both the NFSv4 and underlying file system I/O protocols.

Just as any improvement to NFSv4 improves access to the file system it exports, our improvements to pNFS are portable and benefit all parallel file systems equally. We therefore see our improvements as allowing pNFS (and its exported parallel file systems) to concentrate on large data requirements, while native NFSv4 efficiently processes small I/O.

5.4. Setting the Write Threshold

The big advantage of a write threshold is that applications that mix small and large write requests get the "best" I/O path automatically.

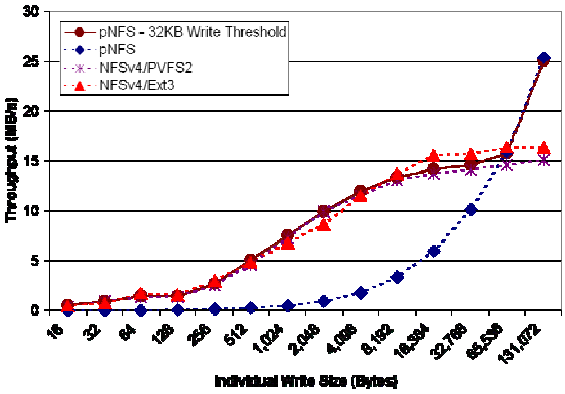


Figure 6. Single client consecutive write throughput
Write throughput of a single client issuing consecutive small write requests. NFSv4 exporting PVFS2 outperforms pNFS until a write size of 64 KB. pNFS with a 32 KB write threshold achieves the best overall performance. Data points are a power of two; lines are for readability.

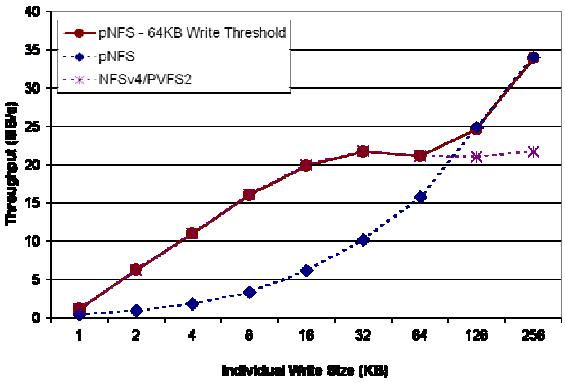


Figure 8. Single client random write throughput
Write throughput of a single client issuing random small write requests. NFSv4 exporting PVFS2 outperforms pNFS until a write size of 128 KB. pNFS with a 64 KB write threshold achieves the best overall performance.

The *optimal* write threshold value depends on several factors, including server capacity, network performance and capability, and the utilized distributed and parallel file systems. One way to choose a good threshold value is to compare execution times for distributed and parallel file systems with various write sizes and see where the curves cross. However, the optimal threshold is sensitive to system load.

Figure 5 displays write request execution time with increasing request size for a parallel file system and for an idle and busy distributed file system. When the distributed file system is lightly loaded, the transfer size at which the parallel file system outperforms the distributed file system, labeled B, is the optimal write threshold. When the distributed file system is heavily loaded, each request takes longer to complete, so the slope increases and intersects the parallel file system at the smaller threshold size, labeled A. (If the distributed file system is thoroughly overloaded, the threshold value tends to zero, i.e., never use a distributed file system so heavily loaded.)

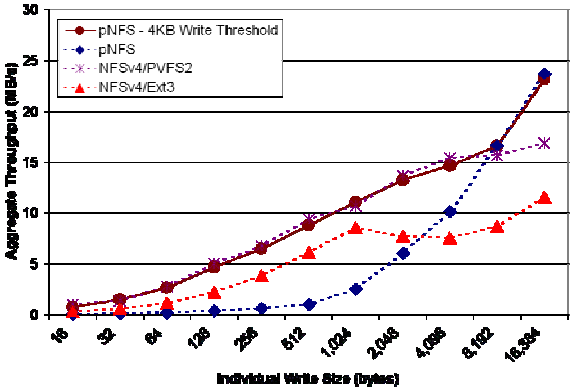


Figure 7. Multiple client consecutive write throughput
Aggregate write throughput of a ten clients issuing consecutive small write requests to a single file. NFSv4 exporting PVFS2 outperforms pNFS until a write size of 8 KB. pNFS with a 4 KB write threshold achieves the best overall performance.

The workload characterization studies mentioned in Section 4 state that scientific applications usually have a large gap between small and large write request sizes, with very few requests in the middle. Our experiments reveal that small requests are smaller than the "busy" write threshold value and the large requests are larger than the "idle" write threshold values, i.e., applications will reap large gains for any write threshold value between A and B. For example, the ATLAS digitization application (Section 6.3) achieves the same performance with any write threshold between 32 KB and 274 KB. In addition, 87 percent of the write requests are smaller than 4 KB, which suggests that we could make the threshold even smaller without hurting performance.

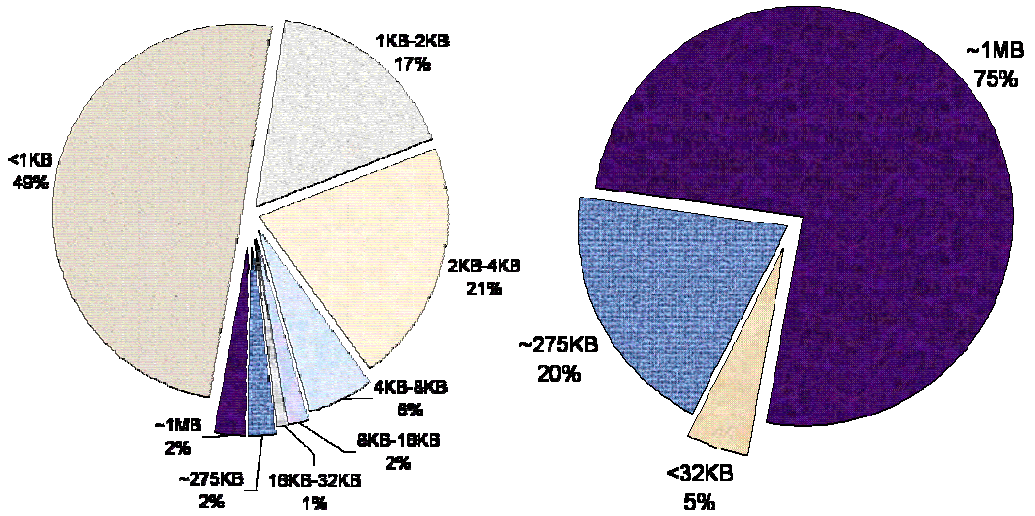
The write threshold can be set at any time, including compile time, when a module loads, and run time. For example, system administrators can determine the write threshold as part of a file system and network installation and optimization. A natural value for the write threshold is the write gather size of the distributed file system.

6. EVALUATION

In this section, we evaluate the performance of our pNFS prototype with the write threshold heuristic.

6.1. Experimental Setup

Our IOR and random write IOZone experiments use a pair of sixteen node clusters connected with Myrinet. One cluster consists of 1.1 GHz dual-processor PIII Xeon nodes while the other consists of 1 GHz dual-processor PIII Xeon nodes. Each node has 1 GB of memory. The PVFS2 1.1.0 file system has eight storage nodes and one metadata server. Each storage node has an Ultra160 SCSI disk controller and one Seagate Cheetah 18 GB, 10,033 RPM drive, which has an average seek time of 5.2 ms. The NFSv4 server, PVFS2 client its exports, and the PVFS2 metadata server are installed on a single node. All nodes run Linux 2.6.12-rc4.



a. Percentage of total number of requests b. Percentage of total amount of data output

Figure 9. ATLAS digitization write request size distribution with 500 events

Our ATLAS experiments use an eight node cluster of 1.7 GHz dual P4 processors, 2 GB of memory, 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. The PVFS2 1.1.0 file system has six storage nodes and one metadata server. The NFSv4 server, PVFS2 client it exports, and the PVFS2 metadata server are installed on a single node. All nodes run Linux 2.6.12-rc4.

6.2. IOR and IOZone Benchmarks

6.2.1. Experimental Design

The first experiment consists of a single client issuing one thousand sequential write requests to a file, using the IOR benchmark [34]. A test completes when data is committed to disk. We repeat this experiment with ten clients writing to disjoint portions of a single file. The second experiment consists of a single client randomly writing a 32 MB file using IOZone [35].

For each experiment, we first compare the aggregate write throughput of pNFS and NFSv4 with a range of individual request sizes. We then set the write threshold to be the request size at which pNFS and NFSv4 have the same performance, and re-execute the benchmark.

6.2.2. Experimental Evaluation

Our first experiment, shown in Figure 6, examines single client performance. NFSv4 writes to PVFS2 or Ext3 perform comparably because the NFSv4 write size of 32 KB is less than the PVFS2 stripe size of 64 KB, so writes are restricted to a single disk.

The performance of a single pNFS client writing through the NFSv4 server to PVFS2 outperforms writing directly to PVFS2 until the request size reaches 64 KB. For 16-byte writes, NFSv4 has sixty-seven times the throughput, with the ratio decreasing to one at 64 KB. The maximum throughput difference of 10.2 MB/s occurs at 4 KB. Write performance through the NFSv4 server reaches its peak at 32 KB, the NFSv4 client request size. At 64 KB, direct storage access begins to outperform indirect access. pNFS with a write threshold of 32 KB offers the performance benefits of both I/O protocols by using NFSv4 I/O until 32 KB, then switching to direct storage access with the PVFS2 I/O protocol.

Figure 7 shows the results of ten nodes writing to disjoint segments of the same file. Ext3 performance is limited by random requests from the NFSv4 server daemons. Using NFSv4 I/O to access PVFS2 does not incur as many random accesses since the writes are spread over eight disks.

PVFS2 exhibits linear scaling as it spreads its requests across all eight storage nodes. The aggregate performance of NFSv4 is the same as with a single client, with the write performance crossover point between pNFS and NFSv4 occurring at 4 KB. With 16-byte writes, NFSv4 has twenty times the bandwidth, with the ratio decreasing to one at just below 8 KB. The maximum bandwidth difference of 9 MB/s occurs at 1 KB. At 8 KB, direct storage access begins to outperform indirect access. pNFS with a write threshold of 4 KB offers the performance benefits of both I/O protocols.

Figure 8 shows the performance of randomly writing a 32 MB file with a range of request sizes. NFSv4 outperforms pNFS until the individual write size reaches 128 KB, with a maximum difference of 13 MB/s occurring at 16 KB. pNFS using a write threshold of 64 KB again experiences the performance benefits of both I/O protocols.

6.3. ATLAS Applications

Not every application generates the small write behavior discussed in Section 4. For example, large writes dominate the FLASH I/O benchmark workload [36], with 99.7 percent of requests greater than 163 KB (with default input parameters). However, in addition to the workload characterization studies, there is increasing anecdotal evidence to suggest that small write behavior is quite common.

One application that exhibits small write behavior is the ATLAS simulator. ATLAS [37] is a particle physics experiment that seeks new discoveries in head on collisions of high-energy protons using the Large Hadron Collider accelerator [38]. Beginning in 2007, ATLAS will generate approximately a

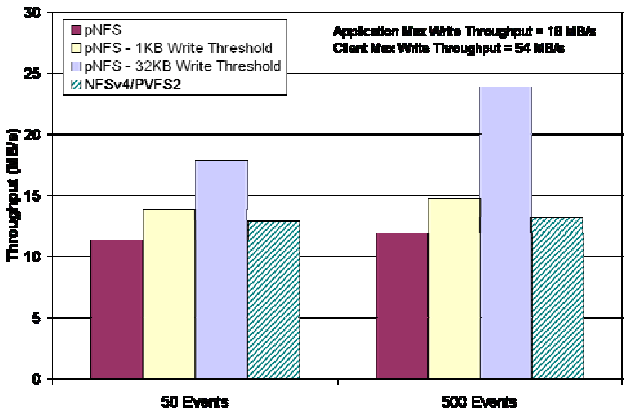


Figure 10. ATLAS digitization write throughput for 50 and 500 events
pNFS with a 32 KB write threshold achieves the best overall performance by directing small requests through the NFSv4 server and the 275 KB and 1MB requests to the PVFS2 storage nodes.

petabyte of data each year. This data will be distributed for analysis to a multi-tiered collection of decentralized sites.

Currently, ATLAS is performing large-scale simulation of the events that will occur within its detector. These simulation efforts influence detector design and the development of real-time event filtering algorithms for reducing the amount of data. The ATLAS detector can detect one billion events with a combined data volume of forty terabytes *each second*. After filtering, data from fewer than one hundred events per second are stored for offline analysis.

The ATLAS simulation event data model consists of four stages. The `Event Generation` stage produces pseudo-random events drawn from a statistical distribution of previous experiments. The `Simulation` stage then simulates the passage of particles (events) through the detectors. The `Digitization` stage combines hit information with estimates of internal noise, subjecting the hits to a parameterization of the known response of the detectors to produce simulated digital output (digits). The `Reconstruction` stage performs pattern recognition and track reconstruction algorithms on the digits, converting raw digital data into meaningful physics quantities.

6.3.1. Experimental Design

This paper focuses on the `Digitization` stage, which is the only stage that generates a large amount of data. With 500 events, `Digitization` produces approximately 650 MB of output data to a single file. Data is written randomly with write request size distributions shown in Figure 9. Figure 9a shows that only 4 percent of write request sizes are 275 KB or greater, with the rest below 32 KB. Figure 9b shows that 96 percent of write requests only write 5 percent of the data, with 95 percent of data written in requests greater than 275 KB. This distribution of write request size and total amount of data output closely matches the workload characterization studies discussed in Section 4. Analysis of the `Digitization` write request distribution with varying numbers of events indicates that the distribution in Figure 9 is a representative sample.

An analysis of the `Digitization` trace data found a large number of fsync system calls. For example, executing `Digitization` with 50 events produced more than 900 synchronous fsync calls. Synchronously committing data to storage reduces request parallelism and the effectiveness of write gathering.

ATLAS developers inform us that the overwhelming use of fsync is an implementation issue rather than an application necessity. Therefore, to evaluate `Digitization` write throughput we used IOZone to replay the write trace data without the fsync calls for 50 and 500 events.

6.3.2. Experimental Evaluation

To evaluate pNFS with the ATLAS simulator, we analyzed the `Digitization` write throughput with several write threshold values.

We initially used the IOZone benchmark to determine the maximum PVFS2 write throughput. The maximum write throughput for a single-threaded application and an entire client is 18 MB/s and 54 MB/s respectively. The single threaded application maximum performance value sets the upper limit for ATLAS write throughput. Increasing the number of threads simultaneously writing to storage increases the maximum write throughput three-fold. Since ATLAS `Digitization` is a single threaded application generating output for serialized events, it cannot directly take advantage of this extra performance.

As shown in Figure 10, pNFS achieves a write throughput of 11.3 MB/s and 11.9 MB/s with 50 and 500 events respectively. The small write requests reduce the application's optimal write throughput by approximately 6 MB/s.

With a write threshold of 1 KB, 49 percent of requests are re-directed to the NFSv4 server, increasing performance by 23 percent. With a write threshold of 32 KB, 96 percent of write requests use the NFSv4 I/O path. With 50 events, the increase in write performance is 57 percent, for a write throughput of 17.8 MB/s. With 500 events, the increase in write performance is 100 percent, for a write throughput of 23.8 MB/s.

It is interesting to note that 32 KB write threshold performance exceeds the single-threaded application maximum write throughput. Since the NFSv4 server is multi-threaded, it can process multiple simultaneous write requests and outperform a single-threaded application. This is yet another benefit of the increased parallelism available in distributed file systems.

When pNFS funnels all `Digitization` output through the NFSv4 server, the performance drops dramatically, but is still slightly better than the performance of pNFS with direct I/O. In this experiment, the improved write performance of the smaller requests overshadows the reduced performance of sending large write requests through the NFSv4 server.

The 50 and 500 event experiments have slightly different write request size and offset distributions. In addition, the 500 event simulation has ten times the number of write requests. We believe the difference between the pNFS write threshold performance improvements in the 50 and 500 event experiments is due to a difference in behavior of the NFSv4 writeback cache with these different write workloads.

6.4. Discussion

Our experiments show that writing to the direct data path is not always the best choice. Write request size plays an important role in determining the preferred data path.

The Linux NFSv4 client gathers small writes into 32 KB requests. With very small requests, the overhead of gathering requests diminishes its potential, but it is still beneficial. As the size of each write request grows, the benefit is considerable.

Performing an increased number of parallel asynchronous write requests also improves performance. This is seen in both Figures 6 and 8, as the performance of writing 32 KB requests exceeds that of writing directly to storage.

The Linux NFSv4 server does not perform write gathering. Our experiments clearly show the benefit of increasing the write request size. The ability for the NFSv4 server to combine small requests from multiple clients into a single large request should also win big.

7. FUTURE WORK

We are investigating a number of potential improvements:

- Implement strided read and write interfaces in NFSv4.
- Implement Linux NFSv4 server request gathering.
- Implement symmetric pNFS servers.

8. CONCLUSIONS

Diverse file access patterns and computing environments in the high performance community make pNFS an indispensable tool for scalable data access. This paper demonstrates that pNFS can increase write throughput to parallel data stores—regardless of file size—by overcoming parallel file system small write inefficiencies. pNFS improves the overall write performance of parallel file systems by using direct, parallel I/O for large write requests and a distributed file system for small write requests. Our evaluation results using a real scientific application and several benchmarks demonstrate the benefits of this design. The pNFS heterogeneous metadata protocol allows any parallel file system to realize these write performance improvements.

9. ACKNOWLEDGEMENTS

This work is partially supported by Sandia National Labs. 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 B523296. We thank Jim Schutt, Ruth Klundt, Gary Grider, and James Nunez for their valuable insights and system support.

10. REFERENCES

[1] Cluster File Systems Inc., "Lustre: A Scalable, High-Performance File System," 2002.

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

[3] 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*, 2002.

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

[5] D. Kotz and N. Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload," in *Proceedings of Supercomputing '94*, 1994.

[6] A. Purakayastha, C. Schlatter Ellis, D. Kotz, N. Nieuwejaar, and M. Best, "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor," in *Proceedings of the Ninth International Parallel Processing Symposium*, 1995.

[7] 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, pp. 1075-1089, 1996.

[8] E. Smirni and D.A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications," in *Proceedings of the Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, 1997.

[9] E. Smirni, R.A. Aydt, A.A. Chien, and D.A. Reed, "I/O Requirements of Scientific Applications: An Evolutionary View," in *Proceedings of the Fifth IEEE Conference on High Performance Distributed Computing*, 1996.

[10] 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*, 1995.

[11] F. Wang, Q. Xin, B. Hong, S.A. Brandt, E.L. Miller, and D.D.E Long, "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*, 2004.

[12] ASCI Purple RFP, `www.llnl.gov/asci/platforms/purple/rfp`.

[13] R. Rew and G. Davis, "The Unidata netCDF: Software for Scientific Data Access," in *Proceedings of the Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, Anaheim, CA, 1990.

[14] NCSA, "HDF5 ", `hdf.ncsa.uiuc.edu/HDF5`.

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

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

[17] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," in *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, 1991.

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

[19] B. Welch, B. Halevy, D. Black, A. Adamson, and D. Noveck, "pNFS Operations Summary," Internet Draft, `draft-welch-pnfs-ops-00.txt`, 2004.

[20] G. Gibson, B. Welch, G. Goodson, and P. Corbett, "Parallel NFS Requirements and Design Considerations," Internet Draft, `draft-gibson-pnfs-reqs-00.txt`, 2004.

[21] 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.

[22] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, (**10**)1, pp. 26-52, 1992.

[23] J.H. Hartman and J.K. Ousterhout, "The Zebra Striped Network File System," *ACM Transactions on Computer Systems*, (**13**)3, 1995.
[24] P.F. Corbett and D.G. Feitelson, "The Vesta Parallel File System," *ACM Transactions on Computer Systems*, (**14**)3, pp. 225-264, 1996.
[25] EMC Celerra HighRoad Whitepaper, `www.emc.com`, 2001.
[26] A.L. Drapeau, K. Shirriff, E.K. Lee, J.H. Hartman, E.L. Miller, S. Seshan, R.H. Katz, K. Lutz, D.A. Patterson, P.H. Chen, and G.A. Gibson, "RAID-II: A High-Bandwidth Network File Server," in *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.
[27] 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, 1998.
[28] 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.
[29] 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," Internet Draft, `draft-ietf-ips-iscsi-08.txt`, 2001.
[30] R. Brightwell, A.B. Maccabe, R. Riesen, and T. Hudson, "The Portals 3.3 Message Passing Interface," 2003.
[31] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," *RFC 1831*, 1995.
[32] Unidata Program Center, "Where is NetCDF Used?," `www.unidata.ucar.edu/software/netcdf/usage.html`.
[33] J. Katcher, "PostMark: A New File System Benchmark," Technical Report TR3022, Network Appliance, 1997.
[34] IOR Benchmark, `www.llnl.gov/asci/purple/benchmarks/limited/ior`.
[35] W.D. Norcott and D. Capps, "IOZone Filesystem Benchmark," 2003.
[36] FLASH I/O Benchmark, `flash.uchicago.edu/~jbgallag/io_bench`.
[37] ATLAS, `atlasinfo.cern.ch`.
[38] The Large Hadron Collider, `lhc.web.cern.ch`.