Small-File Access in Parallel File Systems

Philip Carns, Sam Lang, Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{carns,slang,rross}@mcs.anl.gov

Murali Vilayannur
VMware Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
muraliv@vmware.com

Julian Kunkel, Thomas Ludwig
Institute of Computer Science
University of Heidelberg
{Julian.Kunkel,Thomas.Ludwig}
@Informatik.uni-heidelberg.de

***Abstract*—Today's computational science demands have resulted in ever larger parallel computers, and storage systems have grown to match these demands. Parallel file systems used in this environment are increasingly specialized to extract the highest possible performance for large I/O operations, at the expense of other potential workloads. While some applications have adapted to I/O best practices and can obtain good performance on these systems, the natural I/O patterns of many applications result in generation of many small files. These applications are not well served by current parallel file systems at very large scale.**

This paper describes five techniques for optimizing small-file access in parallel file systems for very large scale systems. These five techniques are all implemented in a single parallel file system (PVFS) and then systematically assessed on two test platforms. A microbenchmark and the mdtest benchmark are used to evaluate the optimizations at an unprecedented scale. We observe as much as a 905% improvement in small-file create rates, 1,106% improvement in small-file stat rates, and 727% improvement in small-file removal rates, compared to a baseline PVFS configuration on a leadership computing platform using 16,384 cores.

I. Introduction

Today's computational science demands have resulted in ever larger parallel computers, and storage systems for these computers have likewise grown to match the rates at which applications generate data. Parallel file systems used in this environment have become increasingly specialized in an attempt to extract the best possible performance from underlying storage hardware for computational science application workloads. These specialized systems excel at large and aligned concurrent access, and some applications have recognized that performing large accesses to multi-gigabyte files is the best way to leverage parallel file systems. Other applications continue to use other I/O strategies, with varying degrees of success. Meanwhile, scientists in new domains are beginning to use high-performance computing (HPC) resources to attack problems in their areas of expertise, and these applications bring new I/O demands.

The results can be seen in recent workload studies. In practice, many HPC storage systems are used to store many small files in addition to the large ones. For example, a 2007 study of a shared parallel file system at the National Energy Research Scientific Computing Center showed that it contained over 13 million files, 99% of which were under 64 MBytes and 43% of which were under 64 KBytes [1]. A similar 2007 study at the Pacific Northwest National Laboratory showed that of the 12 million files on that system, 94% of files were under 64 MBytes and 58% were under 64 KBytes [2].

Further investigation finds that these files come from a number of sources, not just one misbehaving application. Several scientific domains such as climatology, astronomy, and biology generate data sets that are most conveniently stored and organized on a file system as independent files. The following are examples of data sets from each field (respectively):

- 450,000 Community Climate System Model files with an average size of 61 MBytes [3]
- 20 million images hosted by the Sloan Digital Sky Survey with an average size of less than 1 MByte [4]
- up to 30 million files averaging 190 KBytes generated by sequencing the human genome [5]

Accessing a large number of small files on a parallel file system shifts the I/O challenge from providing high aggregate I/O throughput to supporting highly concurrent metadata access rates. The most common technique currently used to improve metadata rates in file systems is client-side caching. The trend in HPC systems, however, is toward large numbers of multicore processors with a meager amount of local RAM per core. Applications on these systems generally use the majority of this memory, leaving little room for caching. Furthermore, traditional techniques for maintaining coherence and recovering from failures were not designed for use at this scale.

In this paper we pursue a strategy of hiding latency and reducing I/O and messaging without using additional resources on clients. We describe five techniques for improving concurrent metadata and small file I/O performance in parallel file systems: server-driven file precreaton, the readdirplus POSIX extension, file stuffing, metadata commit coalescing, and eager data movement for reads and writes. Of those five techniques, the first two have been previously demonstrated in separate parallel file system implementations. The remaining three are also known optimizations, but we apply them in a novel way to the parallel file system environment. In this paper, all five are implemented in a single file system (PVFS) and tested in a consistent environment to assess their relative value. We also extend the analysis to evaluate behavior at an unprecedented scale on an IBM Blue Gene/P system.

The paper is organized as follows. In Section II we describe the relevant aspects of PVFS. In Section III we discuss each of

the techniques used in this work to optimize small I/O access. In Section IV we evaluate these techniques using microbenchmarks and synthetic workloads in two test environments. In Section V we summarize related work, and in Section VI we summarize our findings and suggest avenues of future work.

II. The Parallel Virtual File System

The Parallel Virtual File System project is a multi-institution collaborative effort to design and implement a production parallel file system for HPC applications at extreme scale [6] [7]. The current PVFS implementation includes support for a variety of networks through a messaging abstraction [8], including TCP/IP, InfiniBand, Portals, and Myrinet MX. It is widely used as the basis for research efforts [9], [10], [11], [12], [13], and it is deployed in production at a number of sites, including the Argonne Leadership Computing Facility. This deployment is discussed in detail in Section IV.

A. PVFS Servers

A PVFS file system consists of a set of servers that provide metadata and data storage and are accessed by using a custom network protocol similar to NFSv3 [14], with modifications tailored to the needs of computational science. Each server manages its own local storage, with the production version of PVFS storing data in files in a local directory tree and metadata in a Berkeley DB database. This approach eliminates communication related to block allocation, and data is organized into typed objects similar to those stored on object storage devices.

Current versions of PVFS use a static configuration file to assign metadata server (MDS) and I/O server (IOS) roles to specific servers. It also partitions object handles over these servers, so that handles are unique in the context of a single PVFS file system. PVFS typically stripes files over all IOSes but does not store redundant data or additional replicas of metadata. For a fault-tolerant configuration, storage may be attached to multiple servers and failover enabled through packages such as the Linux-HA heartbeat package [15]. Individual directories are stored on a single MDS. Directories hold names and associated object handles for metadata objects, which may be distributed across other MDSes. This level of indirection provides a great deal of flexibility in placement that is leveraged in this work.

For I/O-intensive deployments, the PVFS file system might be configured to have one MDS and many IOSes, whereas for more general-purpose or metadata-intensive deployments all servers could act as both MDSes and IOSes. For this paper all testing was performed on PVFS file systems configured such that all servers are both MDSes and IOSes.

B. PVFS Clients

PVFS clients access the file system through one of a number of application programmer interfaces (APIs). A provided Linux VFS module enables access with the standard POSIX I/O APIs used by UNIX utilities and applications. A user-space library provides access through what is simply called the PVFS system interface, and higher-level libraries such as MPI-IO libraries use this interface to bypass the kernel when accessing PVFS files.

Regardless of the application interface used, eventually PVFS clients access objects by first mapping from a pathname to an object handle via a *lookup* operation. This lookup is followed by a *getattr* that acquires file statistics and the file distribution. A PVFS file distribution includes a list of objects holding data for the file and a function that maps file positions into locations in the distributed objects, similar to a layout in pNFS. The file distribution does not change once the file is created (with the exception of stuffed files, discussed below), so clients may cache this data indefinitely. Because of this property, once clients obtain this information they can directly contact IOSes for all subsequent read and write operations.

PVFS clients employ a name space cache and attribute cache for *lookup* and *getattr* operations, respectively. The primary purpose of these caches is to compensate for metadata access patterns generated by the Linux kernel VFS. It is not unusual for the VFS to perform multiple stats or path lookups of the same file in rapid succession as part of a single file access. The experiments presented in this paper were carried out with both the attribute and name space cache timeouts set to 100 ms. This setting is sufficient to hide duplicate *lookup* and *getattr* operations without risking excessive state skew across clients. It is conceptually similar to the `readdir_ttl` and `stat_ttl` timeouts employed by the Ceph file system [16].

III. Optimizing Small-File Access in PVFS

Because very little data passes between clients and servers in small-file access, performance is usually dictated by the number of individual network or I/O operations necessary to perform the desired I/O and the degree to which the latency of these operations can be hidden from the client. In this work we employ a strategy of hiding latency and reducing messages and I/O without using additional resources or imposing additional coordination requirements on clients.

This section describes five optimizations targeted at various aspects of small-file access. Some of these optimizations are already provided in PVFS production releases but are presented for the first time in this work, while others are still in the prototype phase.

The obvious question remains: What is a small file? In our case, a small file is any file for which it makes no sense to impose striping; the optimizations here that deal with creation, removal, and statistics gathering on small files are applicable to even multi-megabyte files. In the tests in this paper we used a 2 MByte strip size. A small-file access, in the context of this work, is one that can fit in the message buffer along with our control messages (16 Kbyte in current PVFS releases).

A. Precreating Objects

Files in PVFS are created through a multistep process that is driven by the client creating the file. The client first creates an object to store metadata and a collection of objects to store data by contacting the servers on which these objects will reside.

The metadata object is updated, with a separate operation, to hold the list of data objects and a distribution function describing how file positions map into regions of data objects. A directory entry is then added on the appropriate server. In the event of an error, the client is responsible for cleaning up stray objects. If the client fails during the create, objects may be orphaned, but the name space remains intact.

This approach has two limitations (beyond the possibility of orphaned objects). First, we must send n+3 messages to create a file striped over n objects. The latency of these operations limits the rate at which any one client may create files and generates a great deal of message traffic if many files are created. Second, none of the latency of these operations is hidden from clients. While clients are able to overlap creation of objects across multiple servers by sending requests simultaneously, they must wait for all these operations to complete before updating metadata, and then they must wait for the metadata update to complete before creating a directory entry.

Precreating file system objects addresses both of these concerns. In our implementation of precreation, each MDS uses a special *batch create* operation to preemptively contact IOSes and generate a collection of data objects to be used in subsequent file creations. These lists of objects are stored on disk on the MDS. An augmented create operation is used from clients to request that an MDS perform the first three steps of PVFS file creation all at once: allocate the local metadata object, associate preallocated data objects with the new file, and fill in the distribution function. The client then performs the directory entry insert as usual. When the list of preallocated objects runs low on an MDS, it uses the batch create operation to refill the list in the background.

The end result of our precreate approach is that clients send only two messages, and because the amount of local I/O and number of messages for precreating objects is small, the overhead is quite low. Additionally, the time spent creating objects is hidden from clients.

Sun Microsystem's Lustre file system [17] implements a precreation strategy similar to the algorithm presented in this work. In particular, metadata servers request that object storage targets precreate data objects in advance for use in file creation. This set of objects is replenished asynchronously.

B. "Stuffing" in Parallel File Systems

While precreating objects does substantially improve the performance of creating files, there is still room for improvement when files will remain small. First, if a file is not going to grow beyond the size of one strip, then all but the first data object will remain empty. In this case there was no reason to allocate those objects at all. Second, these objects complicate the calculation of file size. PVFS does not track file size on MDSes; instead, clients communicate with IOSes to gather partial file sizes and use this data to calculate a final file size.

Inode stuffing is a technique used in local file systems to store data for small files in the inode rather than allocating a data block. Our implementation of "stuffing" for PVFS is in this spirit, and it works for configurations where servers are working as both MDSes and IOSes. The approach takes advantage of our precreate optimization. When an MDS receives an augmented create operation, the server allocates a local metadata object and a local data object to hold the first strip of data, then fills in the distribution function, marking the file with an attribute indicating that the file is stuffed. It returns the new metadata object handle to the client, who creates the directory entry. At this point we have a "stuffed" PVFS file, which is a file with only one of its data objects allocated and with that object allocated on the same server as the metadata object.

Clients are augmented to understand this lazy allocation of data objects. Clients cache distribution information as usual; and as long as they are operating only on the first strip, no additional information is needed. If a client attempts to access beyond the first strip, it first sends an *unstuff* operation to the MDS. This forces allocation of the remaining data objects (using precreated data objects, so no communication is necessary) if they haven't already been allocated, and returns this new list of objects to the client. The client then performs I/O directly to the IOSes as usual. This capability is designed to allow us to default to this stuffed storage method, as the cost of transitioning to a striped file is very low.

Stuffing has two major impacts. First, it significantly reduces the number of data objects created (as long as files remain small), and in doing so cuts down on the number of precreate messages. Further, `stat` operations no longer need information from additional servers to obtain the file size for small files, significantly reducing communication during statistics gathering.

The Red Hat Global File System implemented inode stuffing in the context of a shared disk file system [18]. Each inode in the file system occupies a full disk block in order to minimize block sharing. If the file size is small enough, the actual file contents are stored within this same block in order to both reduce storage usage and allow retrieval of metadata and file contents in a single block access. This is similar to the file stuffing presented in this paper, although our aim was to minimize client/server messaging rather than shared disk access costs.

The Panasas file system implements a related mechanism as well. Small files are mirrored across exactly two devices, while larger files are striped more widely [19]. This provides different latency, bandwidth, and space efficiency characteristics for each class of file.

C. Coalescing Metadata Commits

PVFS ensures metadata consistency on disk by requiring operations that modify metadata be committed to storage before the clients are notified of completion. In order to provide this consistency, PVFS performs a flush of Berkeley DB's dirty pages (a call to `DB→sync()`) to disk for each metadata write, effectively serializing metadata writes.

In I/O-intensive workloads, which perform only intermittent operations to metadata, this behavior is desirable, as the

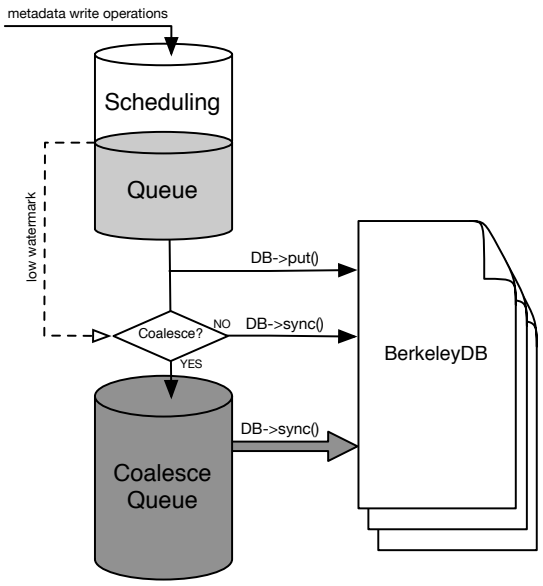


Fig. 1. Metadata coalescing control flow

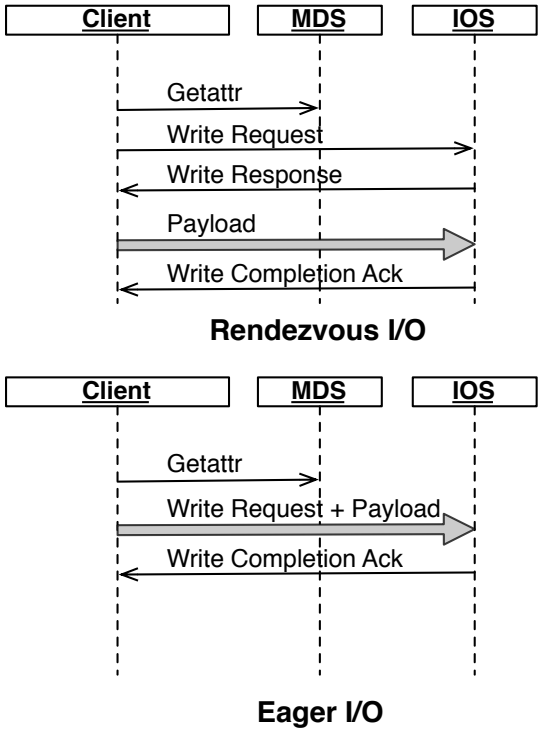


Fig. 2. Eager and rendezvous write examples

objective of an MDS is to minimize the latency of individual operations. In metadata-intensive workloads, which can perform thousands of modifying operations to an MDS at once, the objective of an MDS becomes maximizing throughput of the operations it receives.

To achieve both goals under varying workloads, PVFS MDSes perform a per-operation flush to disk under low server load, and dynamically coalesce flush operations as the number of concurrent modifying operations increase, reducing flush frequency and improving server throughput under highly intensive metadata workloads.

Metadata commit coalescing in PVFS is built around its event-driven programming model, which allows for dynamic scheduling of incoming operations based on system information. The MDS places incoming operations that require a flush to disk in a scheduling queue. The size of this queue acts as a simple measure of the server's ability to keep up with incoming operations. When an operation is removed from the queue and serviced, a decision is made to perform or delay the flush of dirty pages based on the size of the queue. If the size of the queue is below a predefined low watermark value, the flush is performed, and the operation is marked complete. If the scheduling queue size is above the low watermark, then the flush is delayed, and the operation is placed in a separate coalescing queue for future completion. Once the size of the coalescing queue grows past a high watermark value, the flush is performed, and all the delayed operations in the coalescing queue are marked complete. This approach delays the response for some operations but generates significantly fewer local I/O operations. When the number of messages in the scheduling queue again falls below the low watermark, the coalescing queue is immediately flushed, returning the system to low latency mode. Figure 1 shows the control flow of the metadata coalescing logic.

The PVFS metadata commit coalescing approach is similar to group commit timers in high-volume database systems [20], where uncommitted operations are grouped before a commit is made. The timer approach requires a knowledge of system parameters to adjust for varying workloads; it will be less prone to oscillate between low-latency and high-throughput modes under certain workloads but also less responsive to workload changes. Our approach targets HPC environments, where high-volume workloads tend to arrive in bursts, resulting in dramatic workload changes in short periods of time.

D. Eager I/O

MPI implementations have long provided different mechanisms for moving messages of different sizes. One example can be found in the original MPICH implementation [21]. Usually at least two modes are provided: a *rendezvous* mode for large messages and an *eager* mode for small messages. In rendezvous mode, the receiver must respond with an acknowledgment before data is sent, ensuring that space is available on the receiver for the incoming data. In eager mode, the sender immediately pushes data to the receiver, minimizing latency.

PVFS is designed primarily for large data access and therefore uses a combination of a rendezvous-like mode and packetization that enables pipelining of I/O on servers. For small I/O, PVFS switches to a mode similar to the eager mode in MPI. This is illustrated in Figure 2. In both cases the client begins by retrieving attributes (such as datafile layout)

with a *getattr* request. In the rendezvous case, the client and server then handshake before transmitting the payload. The server completes the rendezvous write by sending an acknowledgment. In the eager case, however, the data is optimistically included in the write request in order to eliminate a round-trip message exchange. Eager read operations work in much the same fashion, except that data is included in the acknowledgment rather than the request.

PVFS places an upper bound on the maximum size of unexpected messages (new incoming requests) to servers. This dictates the transition point between rendezvous and eager mode by bounding the amount of data that can be packed into a write request. The same size limit is used for read acknowledgments as well.

E. POSIX Extensions for Directory Access

In some cases client interfaces limit performance by restricting the amount of work that a client can express in one operation. Directory access is a good example: in most cases a directory read is followed by `stat` operations on each of the items in the directory, but the POSIX interface forces the directory read to be separated from the statistics gathering, and forces each `stat` operation to be performed separately.

The HECEWG working group has proposed POSIX I/O API extensions designed to improve performance in HPC environments [22], [23]. One new call, `readdirplus`, allows an application programmer to combine the directory read request with a request for statistics on each of the objects in the directory. Given OS support, this provides the underlying file system with the option of combining these operations for better performance.

PVFS provides an interface for readdirplus in the client library. It is implemented by first performing a *readdir* request to gather a list of objects on which statistics must be gathered. Next, *listattr* requests are sent to MDSes holding relevant objects (one request per server). These obtain all metadata for directories and stuffed files, as well as relevant data objects for striped files. Finally, a second round of *listattr* requests is sent to IOSes with relevant objects (one request per server) to obtain object sizes that are used to calculate file sizes.

This optimization is obviously relevant only to applications performing directory listings on large directories, and it is useful only if applications have access to the appropriate call. While we wait for these proposed extensions to be adopted by the wider community, PVFS provides a special `pvfs2-lsplus` utility that takes advantage of this interface.

The Chirp file system has implemented a `getlongdir` operation [24] to gather readdir results and stat information in a single call that is similar to the proposed `readdirplus` POSIX extension.

IV. Evaluation

In this section we present the results of testing on two platforms. On the first, a Linux cluster, we study the performance of our optimizations at relatively small scale while varying number of clients. On the second platform, an IBM Blue Gene/P system, we examine performance at large scale with a varying number of servers.

A. Linux Cluster

All experiments in this section were conducted using 22 identically configured Linux nodes running kernel version 2.6.20. Each node contains two dual-core Opteron 2220 processors, 4 GBytes of RAM, and four 80 GByte SATA hard drives. The hard drives are accessed via the XFS file system on top of a software RAID 0. The nodes are interconnected via a 10 GByte/s Myrinet network. PVFS includes native support for the Myrinet MX protocol, but at the time of this writing the implementation was not yet compatible with server-to-server messaging as required by the precreate strategy outlined in Section III-A. All tests were therefore conducted with the TCP/IP protocol. Eight of the nodes were configured to act as PVFS file servers, and the remaining 14 were configured to act as PVFS clients.

A microbenchmark was used to isolate the impact of each optimization strategy on specific file system operations. The microbenchmark uses MPI to orchestrate file access from multiple application processes. The file system operations can be carried out by using MPI-IO, POSIX, or PVFS library interface functions. In this study we used the POSIX API, because it is the most prevalent interface for uncoordinated access to small files.

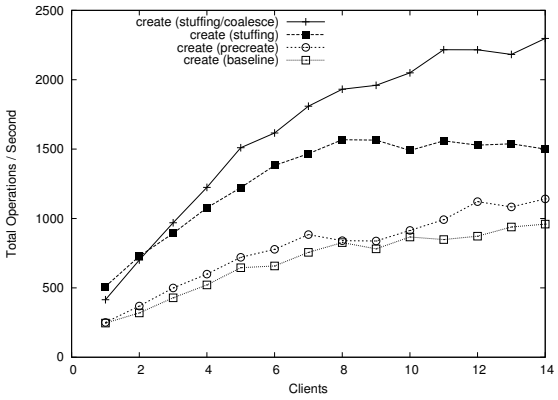
Each application process in the microbenchmark executes the following test phases: (1) create a unique subdirectory, (2) create N files, (3) read subdirectory and stat each file, (4) write M bytes to each file, (5) read M bytes from each file, (6) read subdirectory and stat each file, (7) close each file, (8) remove each file, and (9) remove subdirectory.

Processes are synchronized around each test phase in order to calculate an aggregate rate for each type of operation. All tests reported in this section were carried out with the number of files per process (N) set to 12,000 and the number of bytes written and read (M) set to 8 KBytes.

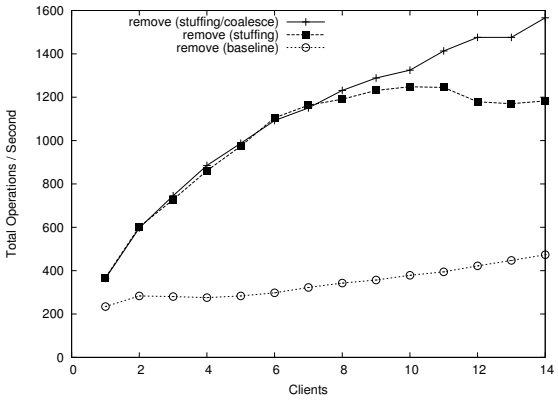
1) Creating and Removing Files: Figure 3 shows create rates and remove rates with our optimizations. When we compare the baseline create result to those with precreate, we see as much as a 19% improvement in performance. However, we would expect the impact of this optimization to increase with increasing numbers of servers, and we will observe this in the following section. This optimization also makes our stuffing implementation possible.

The stuffing optimization provides a further increase in performance over the precreate optimization. However, it appears that we have hit a peak rate per server of approximately 188 creates per second. This is likely due to the serialization of Berkeley DB synchronization operations. Also note that at least one server must host multiple subdirectories once the number of clients exceeds the number of servers. This contributes to the scalability limit that is reached at eight clients.

When we enable the coalescing optimization on top of these other optimizations, we see a further jump in performance as



(a) Creation



(b) Removal

Fig. 3. Linux cluster: File creation and removal rates

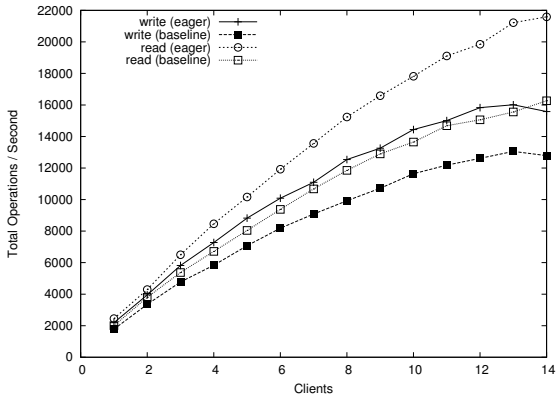


Fig. 4. Linux cluster: Eager I/O

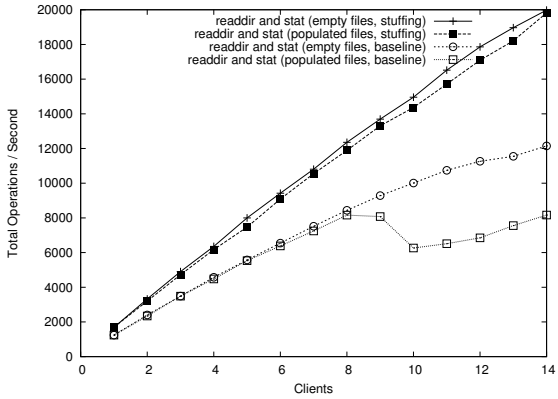


Fig. 5. Linux cluster: Readdir and stat with stuffing

Berkeley DB synchronization operations are combined. We chose a high watermark of 8 and a low watermark of 1. Preliminary testing indicated these to be optimal values for this configuration. With this optimization it is apparent that we have not hit a peak create rate with just 14 clients. Overall, the set of optimizations provides as high as a 139% performance improvement over the baseline configuration.

At this time we have not implemented any sort of bulk object removal, so there is no equivalent to precreating for file removal. We see the largest gain at this scale when stuffing is enabled, because clients need to remove only one data object per file (the server does not do this automatically) rather than n data objects. Again we see that without coalescing we appear to have hit a maximum rate per server, in this case 150 file removes per second; and similar to the create case we see that coalescing enables the servers to improve their rate beyond this point for increasing numbers of clients.

To better understand the impact of Berkeley DB on our performance, we also ran create tests with servers using a tmpfs file system for underlying storage. Assuming a zero cost for tmpfs writes, we found that Berkeley DB synchronization accounts for approximately 70% of the remaining time after our optimizations were applied. Eliminating this synchronization cost allowed us to reach a rate of 7,400 creates per second at 14 clients. This indicates that Berkeley DB, or our current use of it, is the major remaining performance bottleneck for creates.

As noted in Section III-B, the stuffed files would have to be "unstuffed" if a client attempted to write data beyond the first strip of the file. Although this paper does not deal with larger file sizes, we did instrument the unstuff operation on larger write workloads to measure this overhead. We found the unstuff to impose a one-time cost of approximately 4.1 ms.

The microbenchmark results in Figure 3 focus on measuring aggregate metadata rate rather than single-operation latency. However, we include a single-client configuration as the first data point, which is analogous to a sequential single-client workload. From this measurement we can see that precreation, stuffing, and metadata coalescing improve sequential operation latency as well as aggregate operation rates.

2) Eager I/O Optimizations: Figure 4 shows the effect of our eager I/O optimizations on small reads and writes. We achieve better scalability in the eager message case for two reasons: fewer messages are passed over the wire, and server-side processing overhead is less. The performance of both reads and writes tails off slightly as the number of clients is increased, but at this scale the servers are not yet fully saturated. At 14 clients this results in a 22% improvement in write

TABLE I
LINUX CLUSTER: LS TIMES FOR 12,000 FILES

Utility	Baseline, s	Stuffing, s
/bin/ls -al	9.65	8.53
pvfs2-ls -al	6.19	4.85
pvfs2-lsplus -al	2.72	2.65

performance and a 33% improvement in read performance.

3) Large Directory Listings: Figure 5 readdir and stat performance when clients use the VFS interface. The results show rates for both empty files and populated 8 KByte files. File stuffing has a significant impact on `stat` operations because the VFS client is able to obtain file size in the same message used to obtain other statistics. Of note is the difference in performance between empty files and very small files. In Section III-A we described the mechanism by which PVFS allocates data objects as one step of file creation. Servers perform this data object allocation by inserting an appropriate entry into its underlying metadata database. Flat files are used to store file contents, but these are not allocated until data is first written to the file. If a client requests the size of an empty data object, the server responds by detecting that the underlying file does not yet exist. If the data object is populated, however, the server responds by caching a reference to the underlying file and performing an `fstat` to retrieve its size. By measuring the time required on an XFS file system to attempt to open 50,000 nonexistent files and then measuring the time required to successfully open and fstat the same number of files, we see that these two operations require an average of 0.187 seconds and 0.660 seconds respectively. This discrepancy in cost transfers directly to the cost of PVFS stat operations.

The impact of the readdirplus optimization is evident in interactive access to the file system through the `ls` utility. The standard GNU `ls` does not include `readdirplus` support, but PVFS provides file-system-specific utilities called `pvfs2-ls` and `pvfs2-lsplus` for listing PVFS directories. The former is equivalent to `ls` but uses the PVFS system interface. The latter goes one step further by also making use of the `readdirplus` API.

Table I shows the execution time of all three directory listing utilities for a directory containing 12,000 files. The `pvfs2-ls` achieves a 36% speedup simply by utilizing the native PVFS library to bypass the Linux kernel. The `pvfs2-lsplus` utility achieves an additional 128% speedup. The second column indicates that all three also benefit from the reduction in messaging provided by stuffing of small files.

B. IBM Blue Gene/P

The experiments in this section were conducted on Intrepid, an IBM Blue Gene/P (BG/P) system at the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory. In its final configuration, the ALCF BG/P will consist of 40,960 compute nodes in 40 racks providing 80 TBytes of RAM and a peak performance of 556 Teraflops.

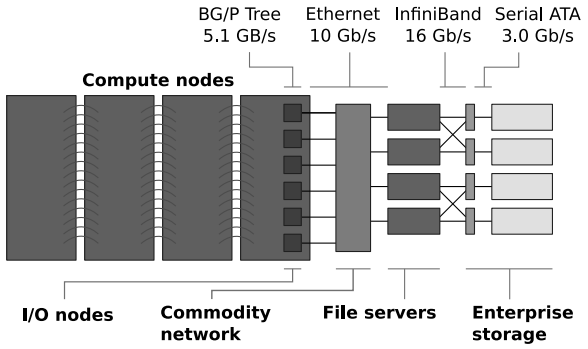


Fig. 6. IBM Blue Gene/P I/O system

Access to 17 DataDirect Networks S2A9900 SANs will be provided by 136 file servers, with a total of 4.3 PBytes of storage and a peak I/O rate of approximately 78 GBytes/s.

Figure 6 illustrates the architecture of the I/O system on the ALCF BG/P. I/O forwarding is used to minimize the number of compute processes presented to the file system. Each set of 64 compute nodes (CNs) forwards system calls to an intermediate I/O node (ION) via a custom tree network. A daemon on the ION known as the CIOD (Control and I/O Daemon) is responsible for accepting system call requests and invoking them on behalf of the compute node kernel. This is the same approach as used in the previous Blue Gene/L systems [25]. Each ION has the same specifications as a CN, except that it runs a Linux kernel and possesses an additional network connection. A commodity switched 10 Gb/s Myrinet network connects all IONs to all file servers, while a point-to-point InfiniBand network connects 8 file servers to each enterprise SAN. Each SAN contains 480 1 TByte disk drives.

At the time of writing, 4,096 compute nodes, 32 file servers, and 4 S2A9900 Data Direct Networks storage systems were available for our testing. Each CN contained a quad core PowerPC 450 processor and 4 GBytes of RAM running a custom IBM operating system, while each file server contained two dual core Opteron 2216 processors and 8 GBytes of RAM running Linux kernel version 2.6.16. A separate XFS file system was used on each SAN LUN. PVFS server daemons were run on each file server, and PVFS client software was installed on each I/O node.

Application processes were executed on all four cores of each of the 4K CNs available for our experiments. Operations from these application processes were forwarded to 64 IONs, which serve as the PVFS clients in the system. We believe that many applications will run in this mode, and this gives us the best picture of the impact of our optimizations on large scale applications.

1) Small File Metadata Operations: Figure 7 shows the create and remove operation rates of 16K application processes on BG/P, with and without our optimizations. We held the number of application processes constant while varying the number of PVFS servers. In the unoptimized cases, IONs must send n+3 messages to create and n+2 messages to remove files, where n is the number of servers. This means

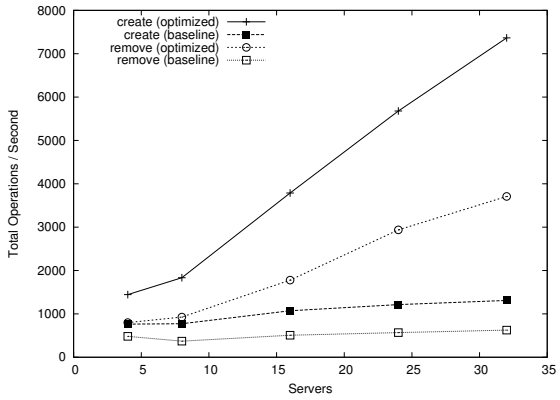


Fig. 7. 16,384 processes on BG/P: create and remove

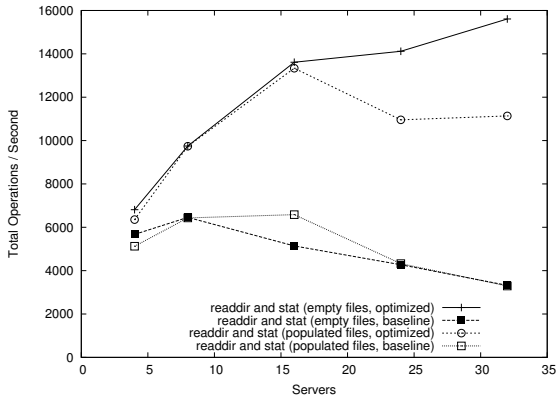


Fig. 8. 16,384 processes on BG/P: readdir and stat

that as we increase the number of servers, each server is still receiving approximately the same number of messages. In the baseline configuration the servers are also serializing synchronization of modifications to the Berkeley DB. These two factors together mean that our rates for create and remove are low for a single server and do not improve significantly as we increase the number of servers.

For the optimized cases, we see that adding servers does substantially improve our overall rate, and we see no indication of hitting a peak rate with 16K application processes. Because we are generating only a single data object per file, adding additional servers means less work per server in terms of both messaging and metadata access. The number of messages is reduced in both the create and remove case, but the change is more significant for create operations. The reason is that the stuffing optimization cuts the number of messages from IONs to two: one to create the metadata and data objects on a single server, and one to add the new directory entry to the parent directory. In the optimized remove case the ION must still perform three separate operations: removing the metadata object, removing the data object, and removing the directory entry. Both create and remove benefit from sync coalescing in order to optimize metadata access for throughput rather than latency.

TABLE II
16,384 PROCESSES ON BG/P: MDTEST MEAN OPERATIONS/SECOND

Process	Baseline	Optimized	Percent Improvement
Directory creation	12163.831	40799.785	235
Directory stat	50402.179	60543.205	20
Directory removal	9778.694	16329.199	67
File creation	1823.450	18324.970	905
File stat	4489.135	54148.693	1106
File removal	1288.583	10656.798	727

Figure 8 shows results of `readdir` and `stat` operations for both empty files and populated 8 KByte files on BG/P. Without optimizations, clients must send n+1 operations per `stat` operation in order to retrieve the metadata for the file and then calculate the file size. We see that the overall operation rate goes down as we increase the number of servers and thus increase the number of messages that must be sent and received.

With optimizations, clients need send only 1 operation per `stat` to obtain all file statistics, as long as files remain stuffed as in this test. The impact of this change is apparent in the graph, with operations on small files improving by as much as a factor of 2 (at 16 servers) and performance generally improving as we increase the number of servers. The property discussed in Section IV-A3, related to `stat` operations on empty files taking significantly less time on PVFS file systems than on files with data in them, appears to be a factor in the optimized runs. However, this property alone does not explain the dropoff in performance after 16 servers; we would expect this additional I/O cost to appear uniformly over the range of servers tested. We intend to explore this behavior more fully. Also note that the CN operating system on Blue Gene does not have access to an API to allow use of the readdirplus extension, which would help to mitigate stat overhead for this case.

2) The mdtest Benchmark: The mdtest benchmark is a commonly used synthetic test of metadata operations. Table II shows the mean operation rates reported by mdtest version 1.7.4 with 16,384 client processes and 32 servers on the BG/P system. The experiment was performed with 10 files per process and unique subdirectories for each process (similar to the microbenchmark parameters). At this scale we see a significant improvement in directory operations due to the metadata coalescing optimization. The file operations benefited from both metadata coalescing and file stuffing to achieve an even greater increase in operation rate.

Both the mdtest results in Table II and the microbenchmark results in Figures 7 and 8 include file creation, removal, and stat rates. The mdtest benchmark reports a much higher rate than our microbenchmark code in all three cases. We believe that this discrepancy on Blue Gene is due primarily to a subtle difference in timing methodology between the two benchmark programs. Algorithm 1 shows the method used in the microbenchmark, which synchronizes using a barrier and records the time to complete operations independently on each

Algorithm 1 microbenchmark

```
1: MPI_Barrier()
2: t1 = MPI_Wtime()
3: for all files do
4:     create file
5: t2 = MPI_Wtime()
6: MPI_Allreduce((t2-t1), time, MPI_MAX)
```

Algorithm 2 mdtest

```
1: MPI_Barrier()
2: t1 = MPI_Wtime()
3: for all files do
4:         create file
5: MPI_Barrier()
6: t2 = MPI_Wtime()
7: if rank == 0 then
8:         time = t2 -t1
```

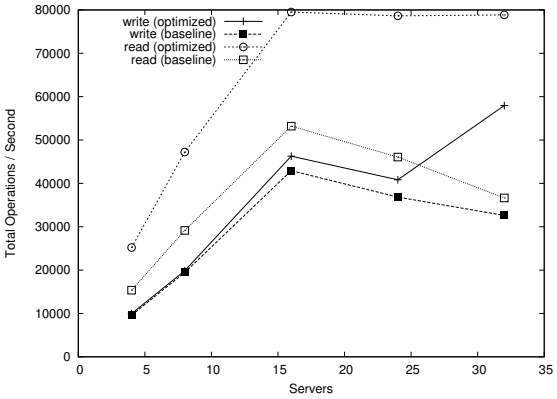


Fig. 9. 16,384 processes on BG/P: I/O

process. The largest of these values is used as the total elapsed time.

The mdtest code uses Algorithm 2, which synchronizes with a barrier as well but performs timing only on rank 0. With tens of thousands of processes, there is potential for variance in the amount of time needed for an individual process to exit a barrier. If rank 0 is late leaving the first barrier, for example, then Algorithm 2 will report a smaller elapsed time because it utilizes timing information only from that process. Given the relatively short run time of our tests, even a small delay could have a measurable impact on resulting rates. We expect that the results of mdtest and the microbenchmark would converge if executed with a sufficiently large file set to amortize the effects of MPI synchronization.

3) Small-File I/O: Figure 9 shows the impact of our optimizations on small-file I/O on the BG/P. These are the highest operation rates seen in our study, reaching nearly 80K writes/sec for eager read operations. The optimizations yielded as much as a 77% improvement in write performance and a 115% improvement in read performance in the largest configuration.

To help understand these results, we contacted colleagues working on the I/O forwarding system for BG/P. In the process of developing a successor to the ZOID I/O forwarding system [26] for BG/P, Iskra has been studying the performance of IONs and the BG/P tree network. By testing POSIX `read` and `write` operations to `/dev/zero` and `/dev/null`, he found that 64 CNs could drive 8 Kbyte I/O operations through the tree and CIOD at approximately 12K to 14K operations per second. Since we are seeing approximately 1.2K ops/sec moving through our IONs, the tree and CIOD are not the limit on our operation rate.

To better understand the performance seen in this experiment, we performed additional testing using 256 processes on a single ION with 8 PVFS servers for the optimized case. With this combination the server is unlikely to be overloaded, as the ION has only one 10Gbit/s link. Running our microbenchmark to perform I/O to 100 files per process, the ION attained a rate of approximately 1,130 operations per second for both reads and writes, with optimizations. This matches closely with the maximum rate seen in the larger experiments, leading us to believe that we are hitting the maximum rate at which an ION can generate requests with the current software in the optimized read case. Further study is necessary to determine why optimized writes in the large-scale test did not match the results of the small-scale run.

V. Related Work

Devulapalli and Wyckoff focused specifically on strategies for speeding PVFS file creation [27]. The three principal strategies examined were compound operations, leased handles, and datafile precreation. Datafile precreation was performed by clients, in contrast to our strategy in which precreation was performed by metadata servers to minimize client messaging and state.

Kuhn, Kunkel, and Ludwig observed that not all applications require complete metadata for small-file workloads [28]. They therefore implemented a directory-level hint in PVFS to eliminate metadata objects and reference data files directly from directory entries. This was shown to speed file creation, stat, and remove operations.

In earlier research, we explored the use of intelligent servers and collective communication to improve metadata latency in PVFS [29] [30]. One goal of that work was to offload work from clients by allowing the servers to perform complex file system operations on their behalf. The servers also used collective communication algorithms similar to those found in message-passing libraries in order to structure communication more efficiently. Both of these techniques are complementary to work presented in this paper. The experimental results examined metadata performance on file systems with hundreds of servers but only a limited number of clients, however.

Several recent parallel I/O studies have included some aspects of concurrent metadata performance. Welch et al. measured the file creation, utime, and stat rates for a Panasas

file system with one metadata server and up to 15 client processes [19]. Cope et al. compared the file creation rates for PVFS, GPFS, Lustre, and TerraFS using a testbed with two data servers and up to 12 client processes [31]. Yu, Vetter, and Oral measured the time necessary to create both shared and independent files on Lustre with up to 512 client processes on the Jaguar system at Oak Ridge National Laboratory [32].

VI. Conclusions

As new science teams adopt computational science as part of their discovery process, it becomes ever more important that parallel file systems cater to less "ideal" access patterns. In this work we have pursued a strategy of enabling a greater degree of latency hiding and reducing number of messages and I/O operations as a way of improving the performance of a parallel file system under a load with many small and independent I/O operations. We have described specific techniques that implement this strategy with roots in a variety of fields, including message passing libraries, databases, local file systems, and parallel file systems.

We studied the performance of these techniques in a consistent environment by implementing them all in the PVFS parallel file system and evaluating PVFS in a modest cluster system and on a large-scale leadership platform. Using a custom microbenchmark and the mdtest benchmark, we found that our optimizations result in as much as a 905% improvement in smalls-file create rates, 1,106% improvement in small-file stat rates, and 727% improvement in small-file removal rates, as compared to a baseline PVFS configuration on a leadership computing platform. Further, the stuffed file approach used here can transparently move to a striped distribution, allowing this optimization to be applied by default without significant performance degradation. Our testing confirms the viability of these techniques in the context of large-scale computing systems. Further, we believe this work to be the first that examines small I/O performance for HPC systems at this scale.

All the testing performed here relied upon per-process subdirectories to avoid contention of directories, which are stored on single servers in PVFS. With Patil et al. we are investigating distributed directory support in PVFS to address this potential bottleneck [33]. The testing performed as part of this work generated new questions about how these optimizations impact the performance of PVFS, particularly in conjunction with I/O forwarding and at very large scale. Understanding the behavior of complex I/O systems is becoming increasingly difficult as we build systems with ever larger component counts and additional layers of system software. We are investigating novel techniques to capture information on storage system behavior and extract knowledge on system behavior from this data to enable more effective performance understanding and debugging for storage systems at scale.

Acknowledgments

We thank the PVFS community for their efforts in making PVFS a successful part of HPC storage systems. W. Allcock, K. Harms, A. Cherry, S. Coghlan, and P. Beckman of the Argonne Leadership Computing Facility were instrumental in enabling our testing on the Blue Gene/P system. We are also thankful to Kamil Iskra of Argonne National Laboratory for his help in understanding the performance characteristics of the Blue Gene/P I/O nodes and tree network. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

[1] Petascale Data Storage Institute, "NERSC file system statistics," World Wide Web electronic publication, 2007. [Online]. Available: http://pdsi.nersc.gov/filesystem.htm

[2] E. Felix, "Environmental Molecular Sciences Laboratory file system statistics," 2007. [Online]. Available: http://pdsi-scidac.org/fsstats/index.html

[3] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller, and D. Bernholdt, "Monitoring the Earth System Grid with MDS4," in *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. Washington: IEEE Computer Society, 2006, p. 69.

[4] E. H. Neilsen Jr., "The Sloan Digital Sky Survey data archive server," *Computing in Science and Engineering*, vol. 10, no. 1, pp. 13–17, 2008.

[5] J. K. Bonfield and R. Staden, "ZTR: A new format for DNA sequence trace data," *Bioinformatics*, vol. 18, no. 1, pp. 3–10, 2002.

[6] "The Parallel Virtual File System," http://www.pvfs.org.

[7] R. Latham, N. Miller, R. B. Ross, and P. H. Carns, "A next-generation parallel file system for Linux clusters," *LinuxWorld Magazine*, January 2004. [Online]. Available: http://www.pvfs.org/documentation/papers/linuxworld-JAN2004-PVFS2.pdf

[8] P. H. Carns, W. B. L. III, R. Ross, and P. Wyckoff, "BMI: A network abstraction layer for parallel I/O," in *Proceedings of IPDPS '05*, April 2005.

[9] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating parallel file systems with object-based storage devices," in *Proceedings of Supercomputing*, November 2007. [Online]. Available: http://www.osc.edu/ pw/papers/devulapalli-pvfs-osd-sc07.pdf

[10] A. Ching, R. Ross, W. keng Liao, L. Ward, and A. Choudhary, "Noncontiguous locking techniques for parallel file systems," in *Proceedings of Supercomputing*, November 2007.

[11] Y. Zhu and H. Jiang, "Ceft: A cost-effective, fault-tolerant parallel virtual file system," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 291–306, February 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1140805

[12] W. Yu, S. Liang, and D. K. Panda, "High performance support of Parallel Virtual File System (PVFS2) over Quadrics," in *International Conference on Supercomputing (ICS '05)*, 2005, pp. 323–331. [Online]. Available: http://www.cse.ohio-state.edu/ liangs/paper/yuw-ics05.pdf

[13] D. Hildebrand and P. Honeyman, "Exporting storage systems in a scalable manner with pNFS," in *13th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2005.

[14] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," in *USENIX Summer*, 1994, pp. 137–152. [Online]. Available: citeseer.ist.psu.edu/pawlowski94nfs.html

[15] A. Robertson, "Linux-HA heartbeat design," in *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[17] "Lustre file system," http://www.sun.com/software/products/lustre/.

[18] S. R. Soltis, T. M. Ruwart, G. M. Erickson, K. W. Preslan, and M. T. O'Keefe, "The global file system," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, H. Jin, T. Cortes, and R. Buyya, Eds. New York: IEEE Computer Society Press and Wiley, 2001, ch. 23, pp. 344–363.

[19] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA: USENIX Association, 2008, pp. 1–17.

[20] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group commit timers and high volume transaction systems," in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, 1987, pp. 301–329.

[21] E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, 1996.

[22] "POSIX HEC Extensions Project," http://www.pdl.cmu.edu/posix/.

[23] G. Grider, L. Ward, R. Ross, and G. Gibson, "A business case for extensions to the POSIX I/O API for high end, clustered, and highly concurrent computing," http://www.opengroup.org/platform/hecewg/uploads/40/10891/POSIX-IO-API-Business-case-HEC-ggrider.pdf, 2006.

[24] D. Thain and C. Moretti, "Efficient access to many small files in a filesystem for grid computing," in *2007 8th IEEE/ACM International Conference on Grid Computing*, 2007, pp. 243–250.

[25] J. J. Ritsko, I. Ames, S. I. Raider, and J. H. Robinson, "Blue Gene," *IBM Journal of Research and Development*, vol. 49, March/May 2005.

[26] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008, pp. 153–162.

[27] A. Devulapalli and P. Wyckoff, "File creation strategies in a distributed metadata file system," in *Proceedings of IPDPS'07*, March 2007.

[28] M. Kuhn, J. M. Kunkel, and T. Ludwig, "Directory-based metadata optimizations for small files in PVFS," in *Euro-Par*, 2008, pp. 90–99.

[29] P. H. Carns, "Achieving scalability in parallel file systems," Ph.D. dissertation, Clemson University, May 2005.

[30] P. H. Carns, B. W. Settlemyer, and I. Walter B. Ligon, "Using server-to-server communication in parallel file systems to simplify consistency and improve performance," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ: IEEE Press, 2008, pp. 1–8.

[31] J. Cope, M. Oberg, H. Tufo, and M. Woitaszek, "Shared parallel file systems in heterogeneous Linux multi-cluster environments," in *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*.

[32] W. Yu, J. S. Vetter, and S. Oral, "Performance characterization and optimization of parallel I/O on the Cray XT," in *IPDPS*, 2008, pp. 1–11.

[33] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte, "GIGA+: Scalable directories for shared file systems," in *Petascale Data Storage Workshop (in Conjunction with SC07)*, Nov. 2007.