Improving the Performance of the POSIX I/O Interface to PVFS

Murali Vilayannur [†] Robert B. Ross [‡] Philip H. Carns [*]
Rajeev Thakur [‡] Anand Sivasubramaniam [†] Mahmut Kandemir [†]
[†] *Department of Computer Science and Engineering*
Pennsylvania State University, State College, PA 16803
{vilayann,anand,kandemir}@cse.psu.edu
[‡] *Mathematics and Computer Science Division*
Argonne National Laboratory, Argonne, IL 60439
{rross,thakur}@mcs.anl.gov
[*] *Parallel Architecture Research Laboratory*
Clemson University, Clemson, SC 29634
pcarns@parl.clemson.edu

Abstract

The ever-increasing gap in performance between CPU/memory technologies and the I/O subsystem (disks, I/O buses) in modern workstations has exacerbated the I/O bottlenecks inherent in applications that access large disk resident data sets. A simultaneous development in recent times has seen the maturity of Linux-based off-the-shelf clusters of PCs for low-cost, high-performance computing solutions. A common technique to alleviate the I/O bottlenecks on such platforms is the use of parallel file systems. One such parallel file system is the Parallel Virtual File System (PVFS), which is a freely available tool to achieve high-performance I/O on Linux-based clusters.

In this paper, we describe some of the key performance and scalability improvements that we have implemented for the UNIX I/O interface to PVFS. To illustrate the performance gains, we present experimental results using Bonnie++, a commonly used file system benchmark to test file system throughput; a synthetic parallel I/O application for calculating aggregate read and write bandwidths; and a synthetic benchmark which calculates the time taken to `untar` the Linux kernel source tree to measure performance of large number of small file operations. We also compare the I/O performance of these techniques when using a Myrinet-based network and when using a fast Ethernet-based network for I/O-related communications. With these techniques we achieve aggregate read and write bandwidth as high as 550 MB/s with Myrinet and 160 MB/s with fast Ethernet.

Keywords: parallel file systems, kernel space threads, aggregate file system bandwidth, POSIX I/O interface.

1 Introduction

In recent years, the disparity between I/O performance and CPU performance has led to I/O bottlenecks in many applications that use large data sets. This gap is becoming more problematic as we move to multi-processor and cluster systems, where the compute power is multiplied by the number of processing units available. A simultaneous trend in recent times has been the shift to clusters of off-the-shelf PCs and networking hardware, which are rapidly becoming the platform of choice for demanding applications primarily because of their cost effectiveness and widespread availability. An opportunity for high-performance I/O exists on these platforms in the form of I/O subsystems in each node. The multiple CPUs and their memories can provide processing and primary storage parallelism, while the multiple disks can provide secondary storage parallelism for both data access and transfer. Parallel file systems exploit this feature to hide the I/O bottlenecks from the applications. While many commercial parallel file systems have been developed for supercomputers and parallel machines, such as PFS for the Intel Paragon [9], and GPFS [17] for the IBM SP, and many academic endeavors, such as Galley [14], PIOUS [12], and PPFS [7], many of these solutions are not available or not intended for production use on Linux-based clusters.

PVFS [4] is a freely available parallel file system that is intended both as a research tool in parallel I/O, and as a stable file system that can be used on production Linux clusters. The initial goals of PVFS were to provide data striping and file partitioning in a distributed environment and to provide an interface that is reasonably close to the standard UNIX I/O interface. The first prototype implementation was developed with the main goal of achieving high-performance I/O and was intended to be run on a 4-node Alpha cluster in 1994. In recent times, clusters of a thousand machines and more have become more common. Hence, the goals of PVFS have been redefined to achieve not only high performance but also scalability. In the past, PVFS has been optimized for bandwidth-limited parallel applications [4] that access the file system through the MPI-IO [10] and native library interface. PVFS has also been optimized for parallel applications with non-contiguous [5] access patterns. The UNIX I/O interface to PVFS is a fairly recent addition, which was intended more for convenience than for achieving high-performance I/O.

In this paper, we describe the key performance and scal-

ability problems that hinder the use of standard UNIX I/O interfaces through PVFS for high-performance parallel I/O, and we present techniques that we have implemented to alleviate the following:

- File system overheads, such as the amount of buffer copying and the number of context switches between processes and client-side daemons.
- TCP/IP connection management overheads.
- Lack of aggregation in directory read operations.

We describe these problems and our proposed solutions in greater detail in subsequent sections. To illustrate the performance improvements, we present experimental results obtained from the Bonnie++ [6] file system benchmark, a synthetic parallel I/O workload, and a synthetic workload which times the `untar` of the Linux kernel source tree. The experimental results presented in this paper are from a 16-node Intel Pentium-III Linux-based cluster of workstations at Clemson University. Each node on this cluster consists of dual 1 GHz Intel Pentium-III microprocessor equipped with 1 GB RAM and two 30-GB IDE-Maxtor hard drives. Each node also has two network interfaces, an Intel 82557 (Ethernet Pro100) fast Ethernet interface, and a Myrinet [3, 13] (M3M-PCI64B-2 with 2 MB of on-board RAM operating at 1.28 Gbps) interface. All the nodes are connected through fast Ethernet and Myrinet switches (three 8-port M2M-SW8 switches).

The rest of this paper is organized as follows. Section 2 outlines the design and implementation of PVFS, and Section 3 describes the performance and scalability bottlenecks and the enhancements proposed to alleviate them. Experimental results on a variety of benchmarks are presented in Section 4. Section 5 summarizes the contributions of this paper and discusses directions for further improvements.

2 PVFS: System Architecture

The goals of PVFS as a parallel file system are to provide high-speed access to file data for parallel applications. The main features of PVFS are that it provides a cluster-wide name space for clients to access their data files, enables users to control striping parameters when creating files, delivers scalable performance under concurrent reads/writes, and allows for legacy binaries to operate on PVFS volumes without having to be recompiled. PVFS is designed as a client-server system as shown in Figure 1 and described in greater detail below.

2.1 PVFS: Servers

PVFS uses two server components, both of which run as user-level daemons on one or more nodes of the cluster. One of these is a meta-data server (called MGR) to which

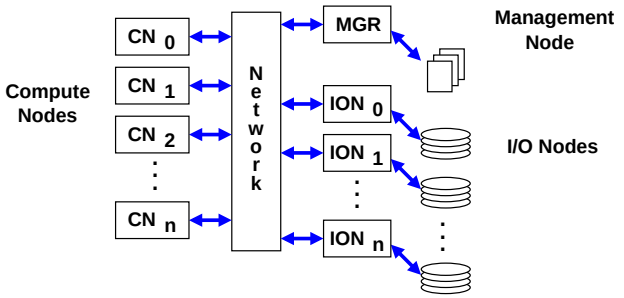

Figure 1: Overall system architecture

requests for meta-data management (access rights, directories, file attributes and physical distribution of file data) are sent. In addition, there are several instances of a data server daemon (called IOD), one on each node of the cluster whose disk is being used to store data as part of the PVFS name space. This daemon listens on a TCP socket for requests to read/write from/to its local disk using normal Linux file system calls. There are well-defined protocol structures for exchanging information between the clients and the servers. For instance, when a client wishes to open a file, it communicates with the MGR daemon which provides it the necessary meta-data information (such as the location of IOD servers for this file, or stripe information) to do subsequent operations on the file. Subsequent reads and writes to this file do not interact with the MGR daemon and are handled directly by the IOD servers. This strategy is key to achieving scalable performance under concurrent reads and write requests from many clients and has been adopted by more recent parallel file system efforts.

2.2 PVFS: Clients

PVFS supports many different client APIs, such as:

- Native libpvfs API,
- Standard UNIX/POSIX API [8], and
- MPI-IO API [10, 19].

Applications written with the native library interface must be linked with *libpvfs* to read and write files off of a PVFS file system. The native API has functions analogous to the POSIX API functions for contiguous reads and writes. In addition, it includes support for non-contiguous reads and writes with a single function call [5]. The MPI-IO interface has been implemented on top of PVFS by using the ROMIO [19] implementation of MPI-IO. ROMIO is designed to be ported easily to new file systems by implementing only a small subset of functions on the new file system [20]. PVFS also supports the standard UNIX I/O functions, such as `open`, `close`, `read`, and `write`, as well as legacy UNIX utilities such as `ls`, `cp`, and `rm`. Interfacing to the UNIX I/O functions is accomplished by

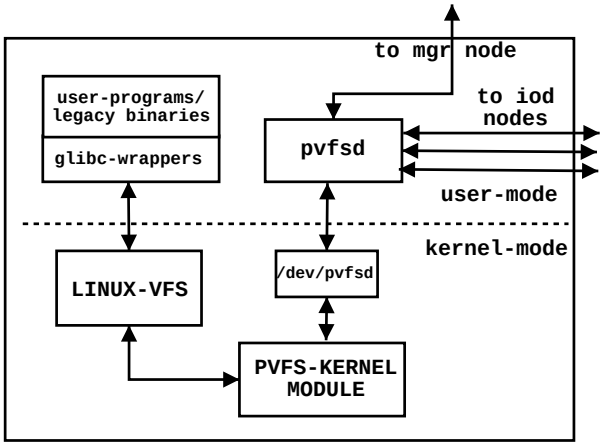

Figure 2: System architecture

loading a Linux kernel module that hooks into the appropriate place in the Linux VFS layer without having to recompile the kernel and/or reboot the machine. Once mounted, the PVFS file system can be traversed and accessed with existing binaries just as any other file system. Many network file systems like NFS have weaker consistency guarantees on file system data and meta-data, since they are primarily targeted at workloads where it is not common to have many processes accessing the same files or directories from many nodes simultaneously. PVFS, on the other hand, cannot afford to have such weaker file system semantics guarantees because it is primarily targeted at workloads that exhibit such access patterns. Therefore, PVFS (at this stage) does not cache file data and meta-data in the Linux page cache; in other words, all file system accesses have to incur a network transaction. We are, however, incorporating consistency management and client-side caching to experiment with different file system semantics issues to improve performance.

2.3 Design of the PVFS Kernel Module

When the PVFS kernel module is loaded, it registers itself with the Linux VFS layer. After the PVFS file system is mounted, subsequent file system calls on the PVFS volume are dispatched by the VFS layer to the module. The module uses a device file to communicate the request to a user level daemon (pvfsd) which satisfies the requests by talking to the MGR and/or the IOD servers. This design is conceptually similar to the Coda design [2, 15] and is shown in Figure 2. We briefly go over a simplified algorithm for the read and write control paths here.

- read
  1. Enqueue a read request to the character device queue.
  2. Wake up the daemon (pvfsd) if necessary.
  3. Wait until interrupted by a signal or when operation is complete.
  4. If operation is complete, copy the data from the kernel buffers to the user land virtual addresses; else return appropriate error code.
- write
  1. Enqueue a write request to the character device queue. This step also involves copying the data from the program's user land virtual addresses to the kernel's temporary buffers.
  2. Wake up the daemon (pvfsd) if necessary.
  3. Wait until interrupted by a signal or until the daemon writes back the virtual address to which the data to be written needs to be copied.
  4. Copy the data from the kernel buffers to the daemon's virtual address.
  5. Enqueue another write request to the character device queue. This step of the write process involves copying the data from the kernel's temporary buffers to the address space of the daemon. Wait until the operation is complete or for a signal to terminate the operation.
  6. Return appropriate error codes or operation success
- user space daemon
  1. Block on the device queue waiting for a request to appear.
  2. Read the request from the character device queue. Ascertain the operation type.
  3. If it is a write request that was enqueued in Step 1, allocate a virtual address region large enough to accommodate the write, and enqueue an acknowledgment into the device queue.
  4. If it is a write request that was enqueued in Step 5 or any other operation, stage the appropriate operation, and enqueue the results of the operation as an acknowledgment into the device queue, waking up the blocked process.

It is conceivable to use the process context (i.e. the context of the process when it is inside the kernel) to communicate with the IOD and MGR servers to service file system requests (thus avoiding the need for any daemons). However, we did not wish to add new file descriptors on behalf of the process without its knowledge, as such an action could have unpleasant side effects. Moreover, a user space daemon-based approach lends well to better code reuse, easier debugging, and simplicity. On the other hand, it does not perform well for many read/write-intensive workloads because of context switching and buffer copying overheads as

elaborated upon in Section 3. Therefore, we propose an alternative kernel space approach that largely alleviates many of the performance limitations.

3 Improving Performance

The chief performance bottlenecks that we address in the subsequent sections are

- increased buffer copies and context switching overheads,
- resource/socket utilization overheads, and
- lack of aggregation in directory read operations.

3.1 Reducing Buffer Copy and Context Switching Overheads

As illustrated earlier, a typical read/write operation incurs at least two buffer copies before the data gets into/from the applications address space (1 copy incurred when transferring from/to the daemon's address space to/from temporary kernel buffers, 1 copy incurred when copying from/to kernel buffers to/from the user program's address space, not including the daemon's read/write from/to the kernel socket buffers when communicating with the IOD and MGR servers). Additionally, the write operation incurs an extra latency because of the extra step involved in waiting for a suitably large virtual address from the daemon's address space. Moreover, each of these operations involves at least two context switches before the file data is written out to the file system or read into the application's address space. A context switch forces the TLBs to be flushed, since the address space is changed. As a result, the memory references that follow the switch are slower than they would be had it not been for the switch. Moreover, the cost of context switching increases with faster processors because of the increased mismatch between processor and memory speeds. For all these reasons, the above design is not appropriate for a high-performance parallel file system.

Beginning with Linux kernel version 2.4, two APIs, *kernel_thread* (which was essentially a front end for the *clone* system call) and *daemonize* are exported to kernel modules to create kernel threads with no user space context associated with them. This approach allows modules to create active entities that execute completely inside the kernel. Since they have no associated user space component, context switching overheads are less. In addition, since they run inside the kernel address space, they can use the temporary kernel buffers directly to stage reads or writes without any extra copying. Moreover, in recent times there have been trends to incorporate many traditional user space services into the kernel, most notably Web servers such as khttpd [21] and TUX [11], for the sake of performance. Therefore, this mechanism can be used to avoid buffer copying, system call, and context switching overheads. Another consequence of executing in kernel space is the fact that we can remove the character device interface altogether and have the daemon operate on the queues directly. With the new mechanism, the read/write control path is briefly described below:

- read
  1. Enqueue a read request to the queue.
  2. Wake up the kernel space daemon (kpvfsd) if necessary.
  3. Wait until interrupted by a signal or when operation is complete.
  4. If operation is complete, copy the data from the kernel buffers to the user land virtual addresses; else return appropriate error code.
- write
  1. Enqueue a write request to the queue. This step also involves copying the data from the program's user land virtual addresses to the kernel's temporary buffers.
  2. Wake up the kernel space daemon (kpvfsd) if necessary.
  3. Wait until interrupted by a signal or when operation is complete.
  4. Return appropriate error codes or operation success.
- kernel space daemon
  1. Block on the queue waiting for a request to appear.
  2. Read the request directly from queue. Ascertain the operation type.
  3. Stage the read/write operation directly to/from the kernel buffers.
  4. Enqueue the results of the operation as an acknowledgment into the queue, waking up the blocked process.

Note that in the above algorithm, we have removed the additional step for writes that was needed in the user space daemon case, and thus we expect to see higher benefits for writes than for reads. This method is shown in Figure 3. Since the kernel space daemon talks to the IOD and the MGR servers in order to satisfy the file system requests, it needs to use kernel space system calls to create and manipulate TCP sockets. The performance benefits of invoking kernel space system calls are mainly due to avoidance of costly user space/kernel space transitions and the associated data passing. In Section 4, we present experimental results from Bonnie++ and a read/write-intensive synthetic parallel workload to illustrate the benefits of the kernel space approach.

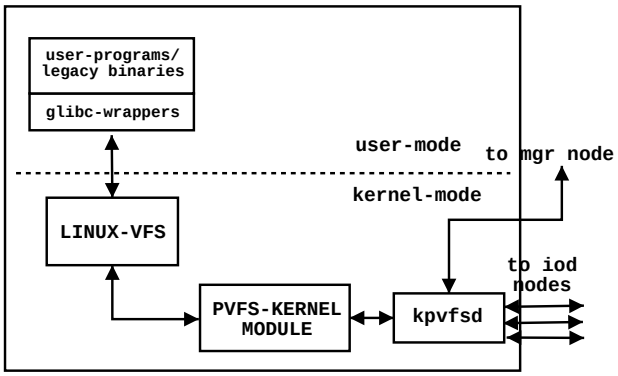

Figure 3: Modified system architecture

3.2 Resource Utilization Management

As stated earlier, the client-side daemon uses TCP sockets to connect and exchange information with the IOD servers. The previous implementation managed connections to the IOD servers at the granularity of a file i.e., it kept as many socket connections per file as there are number of IOD servers over which the file is striped. While such an approach is simple, it suffers from serious scalability and performance issues when there are many simultaneous open files and IOD servers. It is a scalability issue for two reasons, namely, the limited number of open file descriptors that is usually permitted for a process, and the poor scalability of the `select` system call with a large number of file descriptors [1] on the servers. It is a performance issue because of the overhead involved in setting up and tearing down connections to the servers for every new file that is accessed. The overhead is even greater for clients that access the file system through the PVFS library, since the number of file descriptors is potentially increased by a factor proportional to the number of processes that are executing on the node.

The new implementation manages connections at a node granularity; in other words, we keep as many connections as there are IOD servers, and reuse the connections for all files. This approach has not only improved performance for workloads that manipulate a large number of files, but has also improved the scalability when there are many simultaneously open files and IOD servers. In Section 4, we present experimental results from a synthetic benchmark that times the `untar` of the Linux kernel source tree to illustrate the benefits of the IOD connection management.

3.3 Aggregation of Directory Read Operations

A final improvement in performance is in the aggregation of directory read operations on a PVFS file system. As stated earlier, since PVFS is primarily targeted at workloads where it is normal to have many clients accessing the same files and directories concurrently, it does not cache directory entries on the client machine's dentry cache. While such a design greatly simplifies consistency semantics issues, it comes at the cost of performance. Aggregation in directory read operations is hindered by the fact that there is no API which returns the number of entries under a directory. The problem is also compounded by the fact that PVFS reports the sizes of directories as 0. Hence, `glibc` wrappers for the `getdents` system call make estimates based on the size reported by a `stat` on the directory, and repeatedly make the system call until all the entries have been exhausted or there is an error. Therefore, the previous implementation makes a network call for obtaining every directory entry that was extremely slow. Since the memory for copying the directory entry is allocated by `glibc` and the sizes of directory entries as reported by the kernel is dependent on what `struct dirent` definition is being used by the system, the VFS layer passes an opaque (opaque to the file system) pointer to the directory entry and an opaque call back function (`filldir_t`) to the underlying file system which returns an error when space provided by `glibc` is exhausted. Thus, the underlying file system must repeatedly call the opaque function until it returns an error. The new implementation makes a request to the MGR server to fetch a fixed number of directory entries at a time (currently 64) to amortize the network transfer costs, and repeatedly invokes the `filldir_t` function until it returns an error. In Section 4, we illustrate the performance benefits of this approach by timing a recursive listing of a PVFS file system directory containing the Linux kernel source tree.

4 Experimental Results

All these experiments were conducted on a 16-node Intel Pentium-III based cluster running Linux kernel 2.4.18 using both the fast Ethernet-based network and a Myrinet-based network. The platform was configured to run the MGR server on one of the nodes (on port 3000) and two instances of the IOD servers on all the nodes (one each for the Myrinet and the Ethernet interfaces on ports 7000 and 7001). Only one IOD server was used per node for each test run. In effect, we created two separate 16 IOD server PVFS file systems on the platform.

4.1 Aggregate Bandwidth Tests

Our first test program is a parallel MPI program that determines the aggregate read/write bandwidths for varying block sizes, iteration count, and number of clients (*pvfs_test.c* from the PVFS distribution). Each process opens a new PVFS file that is common to all processes, concurrently writes data blocks to disjoint regions of the file, closes the file, reopens the file, reads the same data blocks back from the file, and then closes the file. The tasks of the parallel application synchronize before and af-

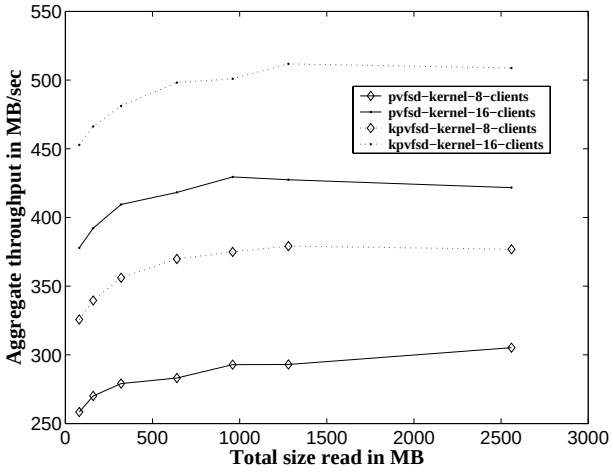

Figure 4: Myrinet: aggregate read bandwidth

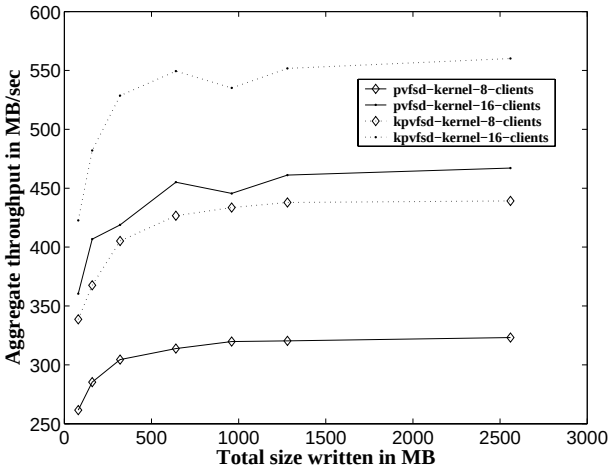

Figure 5: Myrinet: aggregate write bandwidth

ter each I/O operation. Times for the read/write operations on each node are recorded over five trial runs and the maximum averaged time over all the tasks is used to compute the bandwidth achieved. In the first experiment, we varied the block sizes and ran the experiment by using the POSIX I/O interface using both the user space and kernel space daemons for a fixed number of clients. The graphs in Figures 4 and 5 plot the file system read and write bandwidth as a function of the block size for the POSIX interface when using Myrinet for 8 and 16 clients. Figures 6 and 7 plot the same for fast Ethernet. While there is a clear benefit of using a kernel-space approach for both the networks, the benefits are more apparent for a faster network like Myrinet. In the case of Myrinet, we achieve around 20% improvement in read bandwidth for 8 clients; the improvement falls off slightly when using 16 clients. Writes with Myrinet give nearly 40% improvement, which again drops with increased number of clients. The performance improvements for a smaller number of clients can be attributed to the removal of an additional step from the write process. For a larger number of clients, the difference gets smaller because the dominating bottlenecks are possibly due to the servers. We achieved nearly 500 MB/sec read bandwidth and 550 MB/sec write bandwidth using Myrinet. The benefits when using a fast Ethernet-based network are not significant, however, indicating that the network transfer speeds are a bottleneck. The benefits accrued as a result of reduced context switches and copy overheads start to show up only at intermediate ranges of block sizes. In this case, we achieve anywhere from 2 to 10% improvement in bandwidth, and larger improvements are observed for writes only (for the same reason as above). We achieve nearly 150 MB/sec read and 160 MB/sec write bandwidth using fast Ethernet.

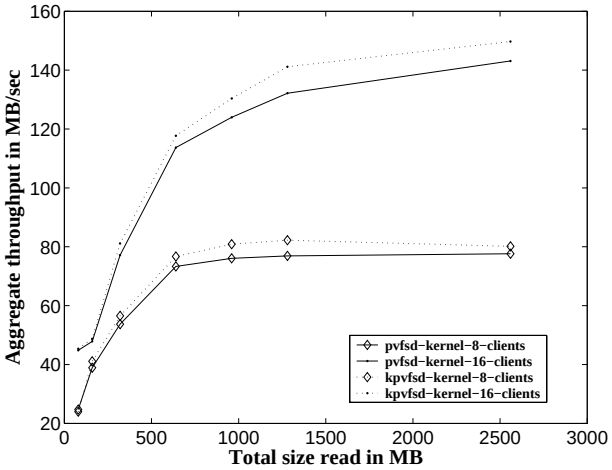

Figure 6: Fast Ethernet: aggregate read bandwidth

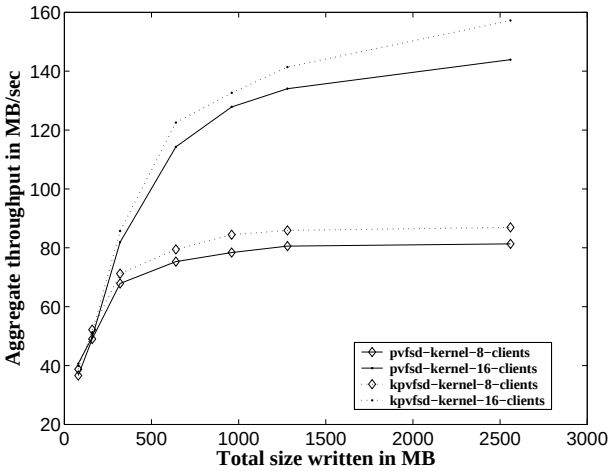

Figure 7: Fast Ethernet: aggregate write bandwidth

4.2 Bonnie++

Our second benchmark is the Bonnie++ benchmark, which performs a series of tests on the file system and reports a number of metrics. The first six tests simulate file system activity that has been observed to be a bottleneck in I/O-intensive applications. Bonnie++ performs a series of tests on a file of a known size and reports the number of kilobytes processed per second and the percentage of CPU use. The second set of six tests simulate operations such as `create`, `stat`, and `unlink`, which are observed to be common bottlenecks on proxy Web cache servers such as Squid [18], news servers such as INN [16] and email servers. The reader is referred to [6] for more information on Bonnie++.

4.2.1 File I/O Tests

The file I/O tests that are reported by Bonnie++ are of three kinds, namely, sequential writes, sequential reads, and random seeks. The sequential writes are done either per character (using `putc`) or per block (using `write`) or are rewritten (using `read`, overwrite, `write`). The sequential reads are done either per character (using `getc`) or per block (using `read`). The random seek test forks a certain number of processes, each of which do a total of 8,000 `lseek`'s to random locations in the file. After the seek, the block is read, dirtied, and written back in 10% of the cases. In all these experiments, Bonnie++ was configured to use a chunk size of 16 KB, and we varied the file size from 32 MB to 512 MB. Since PVFS does not cache file data on the client machine, none of these tests —with the exception of the `putc` test— should be affected significantly by any buffering/caching effects, since data is always transferred over the network. Hence, performance remains unaffected not only with file size, but also with memory size which is required as a parameter to Bonnie++ (unless it is set to such an extremely low value that Bonnie++ will start thrashing). Hence, we show representative results for selected file sizes. Throughput metrics for the read/write tests in Bonnie++ are in KB/sec, while seek throughputs are measured as the rate per second. These experiments were conducted for both the kernel space and user space daemon implementation on both a Myrinet-based and fast Ethernet-based network. We show the percentage performance improvements obtained with fast Ethernet in Figure 8 and Myrinet in Figure 9. Absolute throughput numbers are shown in Figures 13 and 14 in Section 7. We expected that the percentage improvement with a kernel-space approach would be felt only when the reads and writes were done using the blocked I/O method and not when using a per character-based method, since the main savings of this approach are the reduction in copying and context switching overheads, which are not too dominant with a per character-based approach. We see from the graphs, that while rewrites and blocked reads performed exceptionally well with a kernel-space approach,

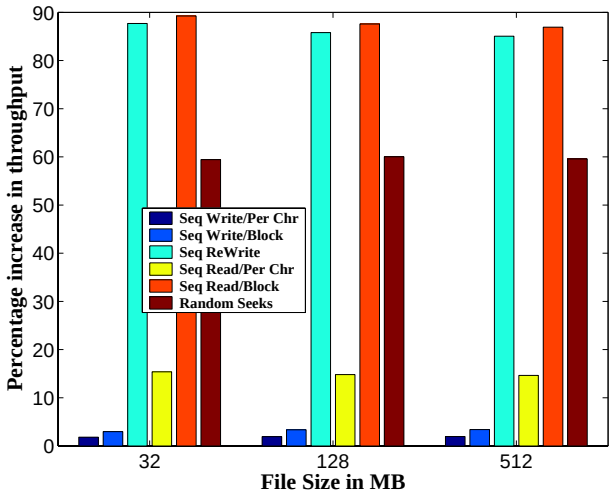

Figure 8: Fast Ethernet: Percentage increase in throughput for Bonnie++

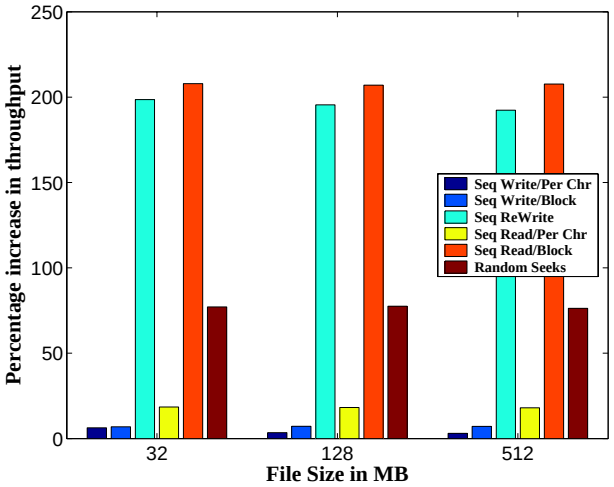

Figure 9: Myrinet: Percentage increase in throughput for Bonnie++

where we achieve nearly 90% improvement using fast Ethernet and around 200% improvement using Myrinet, we do not see a significant improvement in the performance of blocked writes, an anomalous behavior that we cannot yet explain. We also see a significant improvement in the random seek test of nearly 60% in both the networks. (As expected, the per character-based techniques do not achieve any significant benefits with this approach). The increase in performance with this approach does come with a price. Figures 10 and 11 show the absolute percentage CPU utilization for the same tests. We see that in almost all the cases, we incur a slightly higher CPU utilization overhead, which we surmise is due to the fact that in most cases the transfer is being performed over less time.

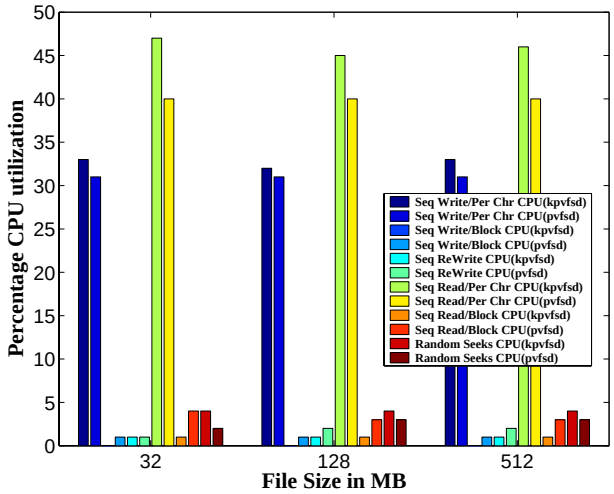

Figure 10: Fast Ethernet: CPU utilization for Bonnie++

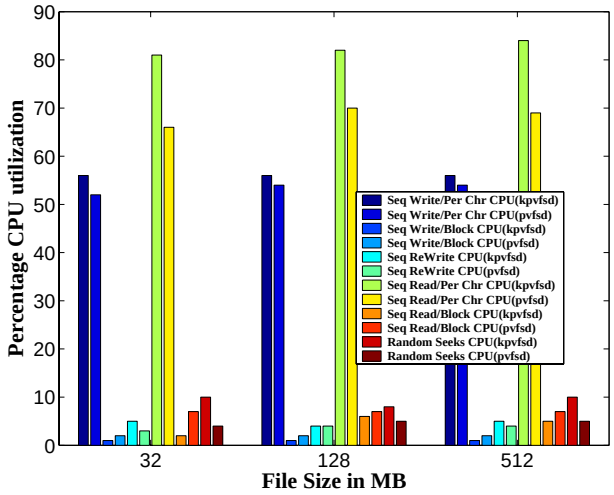

Figure 11: Myrinet: CPU utilization for Bonnie++

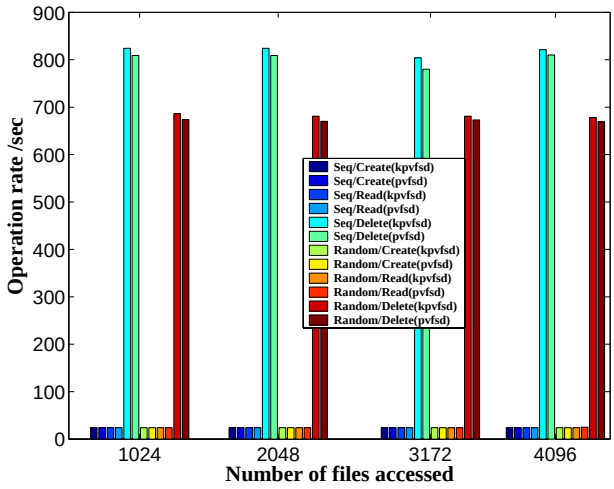

Figure 12: Myrinet: small file operation throughput

4.2.2 File Creation Tests

We used two file creation tests from Bonnie++: sequential test, where a random number of alpha-numeric characters follow a seven-digit number to construct the file name, and a random test where the random characters precede the seven-digit number. The sequential tests `create` the files in increasing numeric order, `stat` them in the `readdir` order, and `unlink` them in the same order. The random tests `create` the files in random order (appears random because of the alpha-numeric characters in front of the seven-digit number in the filename), `stat` them in random order, and `unlink` all the files in random order. In these tests, it is also possible to specify the maximum size and the minimum size of the files that are created, in which case the files are written to at creation and read back after the `stat`. We report this experiments run for the Myrinet network only, since the results are not significantly different for the fast Ethernet-based network. The output from this test is the operation rate measured per second and is parameterized by the number of files, which we varied from 1024 to 4096. Since these tests don't really transfer large amounts of data, the benefits of using a kernel space approach is not expected to be significant, and Figure 12 reinforces the belief. Further, we see particularly low transaction rates for create operations and small file operations; this will be further investigated in the next section. There is no significant change in the CPU utilization for this workload.

4.3 IOD Connection Management

While the scalability benefits of managing connections to the IOD server at a node granularity is apparent, we wanted to quantify the performance benefits of this approach. Table 1 lists the time taken with and without the connection management for untarring the Linux 2.5.39 kernel source tree on a PVFS volume for both the user space and ker-

Table 1: Linux kernel untar

	Fast Ethernet	Myrinet
pvfsd no-mgmt.	1067.50sec	959.40sec
pvfsd mgmt.	1017.96sec	903.41sec
kpvfsd no-mgmt.	1045.55sec	933.64sec
kpvfsd mgmt.	1008.50sec	885.90sec

Table 2: Myrinet: Average time spent waiting in queue (in μsec)

Upcall type	count	pvfsd no-single	pvfsd single	kpvfsd no-single	kpvfsd single
create	15695	41931	38605	41411	38493
setmeta	17764	3052	3025	2975	2970
write	30640	2745	2717	2674	2637
read	1140	2277	3180	2124	2185
mkdir	2070	1693	1681	1652	1666
getmeta	152607	978	912	944	878
lookup	35531	374	367	335	330
total	255447	3778	3532	3705	3482

nel space daemon-based approaches. From Table 1, we see that this strategy did improve performance but not significantly (nearly 3% for fast Ethernet and 5% for Myrinet). In order to investigate the reasons for the insignificant performance improvements for this workload, we instrumented the kernel module to count the number of requests of each type and compute the average time spent by the requesting process for its request to be serviced. These statistics are made accessible to user-space through the *proc* file system. Table 2 lists the time spent (in μseconds) by a process, while waiting for its request to be serviced by the daemon (Myrinet-based network) for this workload. This general purpose file system workload is characterized by meta-data intensive operations and small file data transfers. As the table illustrates, the meta-data operations, and in particular, the create operation dominates over the rest. A create operation is directed to the single MGR server, which then checks permissions, and fans out requests to the appropriate IOD servers directing them to open this file. Once it gets acknowledgements from all the IOD servers, it sends back an acknowledgement to the client, which includes amongst other things, the locations of the IOD servers and physical distribution of file data. Thus, the poor performance of meta-data operations can be attributed to the following reasons; serialization imposed by the single MGR server, communication overhead with the IOD servers, and the lack of a client-side meta-data cache. Moreover, PVFS was designed with a goal of providing high-performance parallel I/O for large data transfers, and it has not been well-tuned for small file operations (The 2.5.39 kernel source tree has an average file size of around 10 KB) or for meta-data intensive operations.

Table 3: Recursive directory listing

	Fast Ethernet	Myrinet
pvfsd-unpatched	881.32sec	878.69sec
kpvfsd-unpatched	879.78sec	877.19sec
pvfsd-patched	177.18sec	158.77sec
kpvfsd-patched	173.22sec	153.81sec

4.4 Aggregation of Directory Read Operations

The performance improvements obtained by aggregating directory read operations are shown in Table 3. It lists the time taken to recursively list a PVFS directory, which contains the Linux 2.5.39 kernel source code for both a fast Ethernet-based network and a Myrinet-based network. The Linux kernel source tree has around 15,000 files spread over 2,070 directories. The command we used to time this experiment was, `time ls -aR /mnt/pvfs/linux-2.5.39/ > /dev/null`. As expected, directory aggregation increases performance by almost a factor of 5. The kernel space approach, however, does not significantly improve the performance (as compared to a user space approach) in both the networks, since the volume of data transferred is not very large. We note here that, this is still much slower than for a network file system with client-side caching of directory entries. For instance, the time taken to recursively list the same kernel source tree on NFS on the same platform is nearly 10 seconds. Hence, a lot of performance tuning still remains to be done.

5 Concluding Remarks and Future Work

PVFS is an actively supported, high-performance, robust, and usable parallel file system for Linux-based commodity clusters. It supports a number of different APIs, most notably the UNIX/POSIX I/O API and the MPI-IO API, which allow legacy binaries as well as high-performance scientific applications to access PVFS file systems. In this paper, we have addressed several performance and scalability problems of the PVFS kernel module that allows applications to access PVFS file system with the standard UNIX I/O interfaces.

Considerable room for improvement and tuning remains. While PVFS has been optimized for applications that demand high file system bandwidths, it is still largely untuned for small file operations and meta-data intensive operations, which are important for many general purpose applications like mail and news servers. Performance could be increased further by having a pool of kernel threads (possibly limited by the number of CPUs, and having each thread bound to a CPU) that services requests from per CPU request queues or by employing an event driven non-blocking approach to servicing requests out of order from the request queue. We are also exploring the incorporation of client-side caching

and distributed consistency schemes to ensure good performance without compromising on file system data and meta-data consistency semantics.

6 Availability

Source code, documentation, and mailing-list information for PVFS are available from the following Web site: *http://www.parl.clemson.edu/pvfs*.

7 Appendix

We plot the absolute throughput numbers obtained with Bonnie++ on a fast Ethernet-based network and a Myrinet-based network in Figures 13 and 14, respectively.

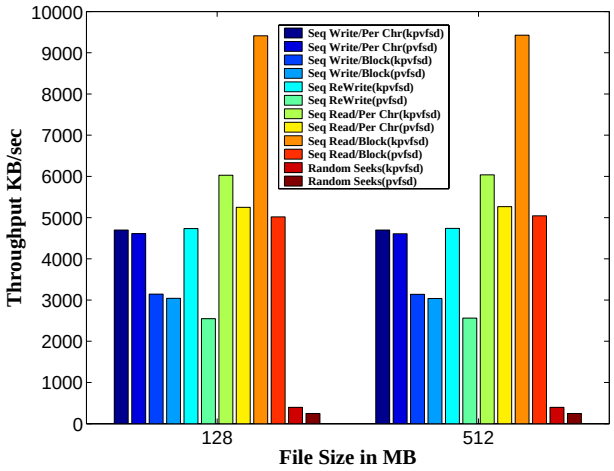

Figure 13: Fast Ethernet: throughput for Bonnie++

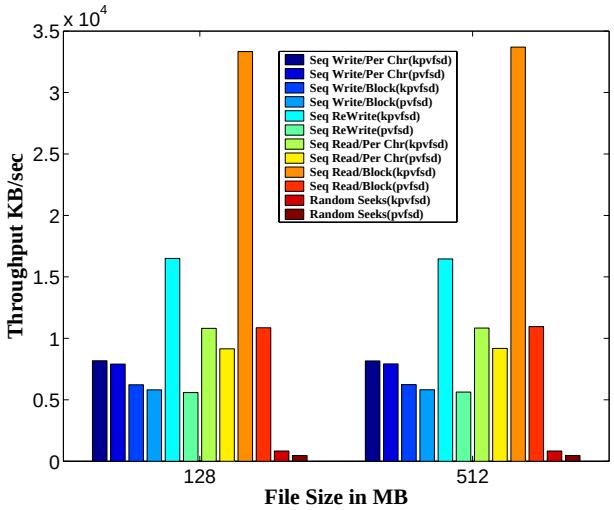

Figure 14: Myrinet: throughput for Bonnie++

8 Acknowledgements

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX, in *Proceedings of the USENIX Annual Technical Conference*, pages 253–265, June 1999.

[2] P. J. Braam. The Coda distributed file system. *Linux Journal*, 50, June 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, et.al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1): pages 29–36, February 1995.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters, in *Proceedings of the 4th Annual Linux Showcase and Conference,*, pages 317–327, October 2000.

[5] A. Ching, A. Choudhary, W. Liao, R. B. Ross, and W. Gropp, Noncontiguous I/O through PVFS, in *Proceedings of 2002 IEEE International Conference on Cluster Computing*, September 2002.

[6] R. Coker. Bonnie++ file-system benchmark. On the WWW at *http://www.coker.com.au/bonnie++/*.

[7] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system, in *Proceedings of the 9th ACM International Conference of Supercomputing*, pages 385–394, Barcelona, ACM Press, July 1995.

[8] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)-part 1: System application program interface (API) [C language]. 1996 edition.

[9] Intel Scalable Systems Division. Paragon system user's guide. Order number 312489-004, May 1995.

[10] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. July 1997. On the WWW at *http://www.mpi-forum.org/docs/docs/html*.

[11] I. Molnar. TUX Web server 1.0, On the WWW at *http://people.redhat.com/mingo/TUX-PATCHES*.

[12] S. A. Moyer and V. S. Sunderam. PIOUS: A Scalable parallel I/O system for distributed computing

environments, in *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[13] Myrinet software and documentation. On the WWW at *http://www.myri.com/scs*.

[14] N. Nieuwejaar and D. Kotz. The Galley parallel file system, in *Parallel Computing*, 23(4), pages 447–476, June 1997.

[15] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment, in *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

[16] R. Salz. InterNetNews: Usenet transport for Internet sites, in *Proceedings of the Summer 1992 USENIX Conference*, San Antonio, TX, pages 93–98, June 8-12, 1992.

[17] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters, in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, January 28-30, 2002.

[18] Squid Web Proxy Cache. On the WWW at *http://www.squid-cache.org/*.

[19] R. Thakur, E. Lusk, and W. Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*, Technical Memorandum ANL/MCS–TM–234, Mathematics and Computer Science Division, Argonne National Laboratory, IL, October 1997.

[20] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel I/O interfaces, in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, IEEE Computer Society Press, October 1996.

[21] A. van de Ven. khttpd: Linux HTTP accelerator. On the WWW at *http://www.fenrus.demon.nl*.