Design and Performance of PVFS over InfiniBand [*]

Jiesheng Wu[*] Pete Wyckoff[†] Dhabaleswar Panda[*]

[*]Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

[†]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

I/O is quickly emerging as the main bottleneck limiting performance in modern day clusters. The need for scalable parallel I/O and file systems is becoming more and more urgent. In this paper, we examine the feasibility of leveraging InfiniBand technologies to improve I/O performance and scalability of cluster file systems. We use PVFS as a basis for exploring these features.

We design and implement a PVFS version over InfiniBand by taking advantage of InfiniBand features and resolving many challenging issues. In this paper, we design and test: a transport layer customized for the PVFS protocol by trading transparency and generality for performance, buffer management for flow control and efficient memory registration and deregistration, and communication management for reducing network congestion and achieving differentiated services.

Compared to a PVFS implementation over standard TCP/IP on the same InfiniBand network, our implementation offers three times the bandwidth if workloads are not disk-bound and 40% improvement in bandwidth in the disk-bound case. Client CPU utilization is reduced to 1.5% from 91% on TCP/IP.

1 Introduction

Cluster systems are increasingly becoming a mainstream platform for parallel computing in various application domains. Out of the latest Top 500 Supercomputers, 93 systems are clusters [14]. Cluster systems are now present at all levels of performance, due to the increasing performance of commodity processors, memory and network technologies. However, in modern day clusters, I/O is quickly emerging as the main bottleneck limiting performance. The need for scalable parallel I/O and file systems is becoming more and more urgent. As well, the use of standards in the hardware components and in the software used in the cluster is also becoming not just convenient but a necessity to ensure software reuse.

There has been a significant amount of work on parallel and cluster file systems, which has repeatedly demonstrated that a viable infrastructure consists of *commodity storage units connected with commodity network technologies*, to provide high performance and scalable I/O support in cluster systems [22, 29, 2, 28, 34, 13, 6]. The PVFS (Parallel Virtual File System) [6] is a good example of such an architecture and a leading cluster file system for parallel computing in cluster systems. It addresses the need of high performance I/O on low-cost Linux clusters. Each PVFS file is striped across multiple disks on different I/O nodes. Data is transferred between compute nodes and I/O units directly. The basic idea behind PVFS is to aggregate disk and network performance to achieve high throughput and scalable concurrent file access.

However, the performance of network storage systems is often limited by overheads in the I/O path, such as memory copying, network access costs, and protocol overhead [1, 27, 24, 19]. Emerging network architectures such as Virtual Interface (VI) Architecture [9] and InfiniBand Architecture [15] create an opportunity to address these issues without changing fundamental principles of production operating systems. Two common features shared by these networks are: *user-level networking* and *remote direct memory access* (RDMA). User-level networking allows applications to directly and safely access the network interface without going through the operating system. RDMA allows the network interface to transfer data between local and remote memory buffers without operating system and processor intervention by using DMA engines.

InfiniBand has been recently standardized by industry to design next generation high-end clusters for both data-

[*]This research is supported in part by Sandia National Laboratory's contract #30505, Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052 and #CCR-0204429.

center and high performance computing. Since it is targeted for both storage I/O and Inter-Processor Communication (IPC), InfiniBand offers additional features such as multiple transport services, atomic operations, virtual lanes, and service levels with hardware support to design high performance, highly scalable, and highly available systems. In our previous work [20], we have demonstrated that InfiniBand can offer high performance to parallel applications that use message passing.

In this paper, we examine the feasibility of leveraging InfiniBand technologies to improve I/O performance and scalability of cluster file systems. We use PVFS as a basis for exploring these features and focus on a number of challenging issues that are important for cluster file systems. First, we propose a modular architecture for designing PVFS over InfiniBand. This architecture takes full advantages of InfiniBand features. Second, we focus on improving the I/O path from the compute node to the I/O server node with multiple optimization techniques. Third, we focus on efficient memory registration and deregistration with respect to I/O intensive applications and in particular applications with noncontiguous I/O accesses [7]. Fourth, we develop schemes to provide fair and dynamic buffer sharing in I/O servers that service a large number of concurrent requests. Finally, we implement PVFS over InfiniBand by taking advantage of user-level networking and RDMA. We evaluate our implementation using PVFS and MPI-IO benchmarks and applications. We compare its performance with that of unmodified PVFS over IBNice [23], a TCP/IP implementation on InfiniBand.

This work contains several research contributions. Primarily, it takes the first step toward understanding the role of the InfiniBand architecture in next-generation cluster file systems. Our results show that:

1. The capabilities of user-level communication and RDMA can improve all performance aspects of PVFS. Compared to a PVFS implementation over IBNice, our implementation offers a factor of three improvement in throughput. Utilization decreases from 91% with IBNice to 1.5% in our native implementation.

2. A transport layer based on InfiniBand user-level programming interface requires careful design regarding aspects of flow control, buffer management, and communication strategy selection.

3. Optimizations in small data transfer, pipelined bulk data transfer, and memory management for noncontiguous I/O can achieve significant performance gains.

The rest of the paper is organized as follows. We first give a brief overview on both PVFS and InfiniBand in section 2. Section 3 presents the design of PVFS over InfiniBand. Section 4 describes our performance optimizations. Implementation is presented in section **??**. The performance results are presented in section 6. Finally we examine related work in section 7 and draw our conclusions and future work in section 8.

2 Overview of PVFS and InfiniBand

2.1 PVFS Overview

PVFS is a leading parallel file system for Linux cluster systems. It was designed to meet increasing I/O demands of parallel applications in cluster systems. As shown in Figure 1, a number of nodes in a cluster system can be configured as I/O servers and one of them is also configured to be the metadata manager. It is possible for a node to host computations while serving as an I/O node.

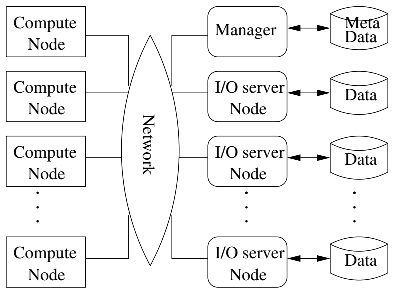


Figure 1. Typical PVFS setup.

PVFS achieves high performance by striping files across a set of I/O server nodes to achieve parallel accesses and aggregate performance. PVFS uses the native file system on the I/O servers to store individual file stripes. An I/O daemon runs on each I/O node and services requests from compute nodes, particularly read and write requests. Thus, data is transferred directly between I/O servers and compute nodes.

A manager daemon runs on a metadata manager node. It handles metadata operations involving file permissions, truncation, file stripe characteristics, and so on. Metadata is also stored in the local file system. The metadata manager provides a clusterwide consistent name space to applications. In PVFS, the metadata manager does not participate in read/write operations.

PVFS supports a set of feature-rich interfaces, including support for both contiguous and noncontiguous accesses in both memory and files [7]. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, MPI-IO [31], and an array I/O interface called the Multi-Dimensional Block Interface (MDBI). The presence of multiple popular interfaces contributes to the wide success of PVFS in both industry and university settings. Work is underway on the next major revision of PVFS which involves a complete design of all major subsystems.

2.2 InfiniBand Overview

The InfiniBand Architecture (IBA) [15] defines a System Area Network (SAN) for interconnecting both processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. InfiniBand Architecture has built-in QoS mechanisms which provide virtual lanes on each link and define service levels for individual packets.

In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). There are two kinds of these: Host Channel Adapters (HCA) and Target Channel Adapters (TCA). HCAs sit on processing nodes and their semantic interface to consumers is specified in the form of InfiniBand Verbs. TCAs connect I/O nodes to the fabric and have interfaces to consumers that are implementation specific and thus not defined in the InfiniBand specification. Channel Adapters usually have programmable DMA engines with protection features.

The InfiniBand communication stack consists of different layers. The interface presented by Channel Adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. The completion of requests is reported through Completion Queues (CQs). Applications can check the completion queue to see if any request has been finished.

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. A receiver must explicitly post a descriptor to receive messages in advance. In memory semantics, RDMA write and RDMA read operations are used. RDMA operations enable the initiator to write data into or read data from memory buffers of the peer side without intervention of the peer side.

3 Design of PVFS over InfiniBand

In this section, we describe the design of PVFS over InfiniBand. First, we define a general software architecture of PVFS based on InfiniBand, then we show the design of each component. We mainly focus on the PVFS transport layer, buffer manager, and communication manager. Other components are currently undergoing redesign for the second version of PVFS, but are not specific to the network and are not discussed further here, including the file access manager [5] and request manager [26].

3.1 PVFS Architecture

Figure 2 shows the PVFS software architecture. Since the metadata server is a simpler case of the I/O server, we only show the architecture of the client and the I/O server here.

There are six modules in the PVFS architecture. A buffer manager, a communication manager, and a PVFS transport layer reside on both the client and server sides. The PVFS library is used by the client to generate requests. A request manager and a file access manager exist on the server side to process client requests.

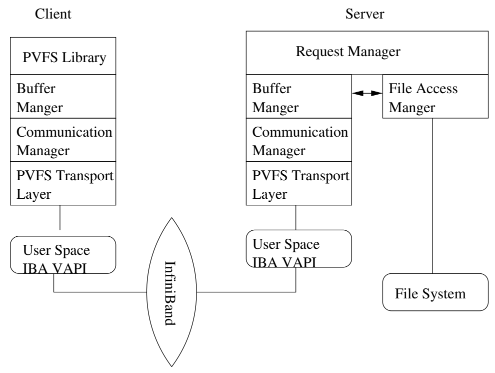


Figure 2. PVFS Software Architecture.

The transport layer transfers data using user-level InfiniBand primitives. The buffer manager supplies the transport layer buffers and also supplies buffers to the file access manager for file accesses. The request manager receives requests and decides in what order to service requests, using information supplied by the file access manager. The communication manager chooses communication mechanisms and schedules data transfers. In this paper, we focus on the transport layer, buffer manager, and communication manager, which become more complicated when designing PVFS over InfiniBand as compared to the original design of PVFS over TCP/IP.

3.2 PVFS Transport Layer

The PVFS transport layer provides data, metadata, and control channels between PVFS compute nodes, I/O server nodes, and the metadata manager. In this section, we first analyze the characteristics of various types of messages in PVFS, then we choose appropriate communication schemes for them, respectively.

3.2.1 Messages and Buffers in PVFS

Messages in PVFS can be categorized as follows:

1. **Request messages:** A request message is sent by the compute nodes to the servers (both I/O server nodes and the metadata manager server) to direct them to initiate operations such as read, write, and lookup. The manager node also uses a request message to inform the I/O server nodes of metadata management operations if needed.
2. **Reply messages:** A reply message is sent by a server to inform the request initiator of completion of a request. It usually contains the status of operations and related information such as the number of bytes read or written.
3. **Data messages:** Data messages are used to transfer payload for file reads and writes.

4. **Control messages:** Control messages are internal messages in the PVFS system. Compute nodes, I/O server nodes and the metadata server all use control messages to exchange information such as flow control to maintain PVFS protocols. Some control information may be exchanged implicitly using request and reply messages.

There are two types of buffers:

1. **Internal buffers:** Internal buffers are allocated by the PVFS system. They are pinned when a connection is established, remain active for a long period of time, and on the servers they can be used to service multiple clients.
2. **RDMA buffers:** RDMA buffers are used to achieve zero-copy data transfer between the compute nodes and the I/O server nodes. On the client side, RDMA buffers are provided by the application when it initiates read and write operations. On the I/O server side, RDMA buffers are allocated to stage data in memory before it moves to the disk or to the network.

3.2.2 Communication Choices

InfiniBand provides both reliable and unreliable connection and datagram services. Since PVFS requires a reliable transport layer, we focus only on the reliable connection service.

In reliable connection service, InfiniBand offers both Send/Recv operations and both read and write RDMA operations. The initiator can choose for each operation whether to generate a completion event. Send/Recv operations and RDMA Write with Immediate operations consume receive descriptors and result in Solicited and Unsolicited completion on the receive side [15]. These features provide a flexible design space and the opportunity to optimize performance. However, the obvious question which arises is how to choose efficient communication operations and completion schemes for each of the message types in PVFS.

Generally speaking, each message type can use either send/recv or RDMA operation; however, a better fit can be obtained for particular message types according to how well they align with the characteristics of the corresponding communication operations. Table 1 lists message chracteristics and suitable communication choices.

The completion of Send, RDMA Write and Read operations on the initiator side is somewhat complicated by the need to drive the message progress engine. It can be expected that better performance can be achieved by avoiding an explicit completion notification; however, this notification provides an easy way to manage resources and quickly check the status of communication. For example, considering a PVFS file write, if the I/O server uses RDMA Read operations to bring data from the compute node buffer, the

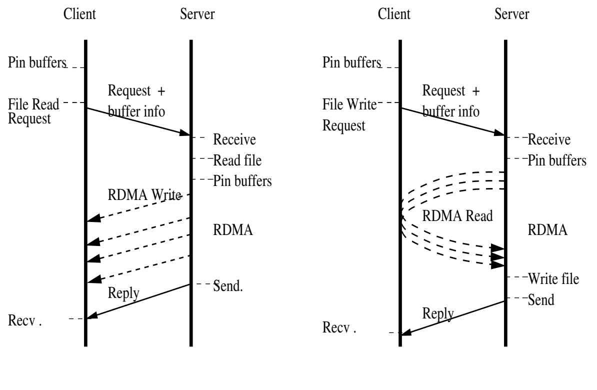


Figure 3. Server-based RDMA Mechanism.

server would like to know when the RDMA Reads are complete so that it can initiate file write operation to move the data to disk, but it is not necessary for every RDMA Read operation to generate completion notification. Therefore, in our design, every send generates a completion and the last RDMA operation in a functional message also generates a completion.

3.2.3 Message Transfer Mechanisms

As discussed in 3.2.2, appropriate communication operations must be chosen for each message type. In this subsection, we show how to use them to transfer messages. There are four basic message transfer mechanisms: *Send/Recv mechanism*, *server-based RDMA mechanism*, *client-based RDMA mechanism*, and *hybrid RDMA mechanism*. We elaborate these mechanisms and how to map PVFS operations to them.

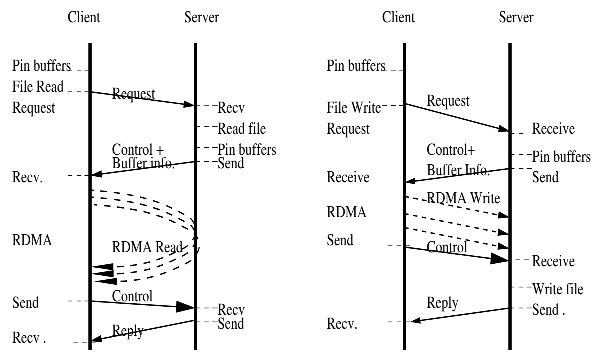
In Send/Recv mechanism, messages are sent from send internal buffers to receive internal buffers. Request and control messages are sent by this mechanism. Data messages also can be sent using this mechanism, at the cost of some memory copies. Send/Recv message transfer is flow controlled as described in section 3.3.1.

In server-based RDMA mechanism, RDMA operations are initiated only by the I/O servers. The clients are responsible for providing RDMA buffer information. Figures 3(a) and 3(b) show the operations involved in read and write transfers, respectively. Since client RDMA buffer information can be provided along with the request messages, the I/O servers can initiate RDMA operations asynchronously according to when they can be scheduled.

Figures 4(a) and 4(b) show the operations involved to perform reads and writes when initiated using RDMA operations from the client. Generally speaking, the client-based RDMA mechanisms require the server to send a control message containing its RDMA buffer information before data transfer can begin. It also requires that the client no-

Table 1. Communication Choices

Message	Characteritics				Choices		Comments
	Unexpected	Size	In-place processing	Immediate Attention	Operation	Completion	
Request	Unexpected	Short	Yes	Yes	Send/Recv	Solicited	
Reply	Expected	Short	Yes	Yes	Send/Recv or RDMA Write	Solicited	Send/Recv is simpler than RDMA Write with Immediate data.
Control	Unexpected	Short	Yes	Yes	Send/Recv	Solicited	
Data	Expected	Variable sizes	Zero-copy expected	No	RDMA Read or Write	No	Tradeoff between zero-copy and non zero-copy, discussed in 4.1.



(a) PVFS read (b) PVFS write

Figure 4. Client-based RDMA Mechanism.

tify the servers when RDMA operations are finished. In the PVFS Read case, this needs a separate control message. In the PVFS Write case, this notification may be carried out using RDMA Write with Immediate data on the last RDMA Write operation, if the 64-bit immediate data is sufficient to carry the necessary state information. It can be seen that more control messages are usually needed in the client-based RDMA mechanism, compared to the server-based RDMA mechanism.

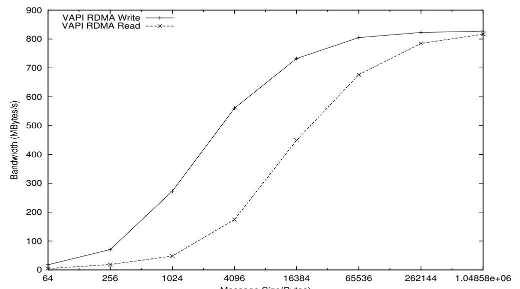


Figure 5. RDMA Read and Write Throughput.

RDMA read is a round-trip operation and its performance is usually lower than that of RDMA Write, as shown in Figure 5. Therefore, one can consider a hybrid RDMA mechanism, wherein only RDMA Write operations are used. In the hybrid mechanism, a PVFS read is designed with server-based RDMA Write as shown in Figure 3(a) and a PVFS write is designed with client-based RDMA Write as shown in Figure 4(b). This mechanism is a common method to design file and storage systems on networks that do not support the RDMA Read operation [22, 34]. Using client-based RDMA Write to design PVFS write can take advantage of higher performance of InfiniBand RDMA Write operations; however, there are several disadvantages. First, the server needs to dedicate sufficient buffer space to each client. If the client exits abnormally, the server cannot reuse these buffers. Second, extra control messages may be required to avoid the loss of scalability and resource consumption associated with this mechanism. For example, a certain number of buffers can be registered and assigned to each client and the registration information can be cached on the client. However, larger message transfers will require more space than had been preallocated forcing the use of more control messages to synchronize use of the finite buffer space. Third, as mentioned in [21], using server-based RDMA Read to design a PVFS write permits a natural flow control algorithm between the network and disks as the server will fetch new data from the network no faster than it can write it to disk. This is not possible in the hybrid mechanism.

In our design, we use the following combinations of the above mechanisms. Send/Recv is used to transfer request, control and reply messages. Server-based RDMA Write is used for PVFS read operations. Server-based RDMA Read is used to implement large PVFS write operations, but client-based RDMA Write is used for small PVFS writes, as discussed in section 4.1.

3.2.4 Polling or Interrupt on Events

InfiniBand provides an aggregated event notification mechanism for scalable event notification and delivery. A single structure, Completion Queues, is used to notify and deliver events for a large number of connections. Events such as arrival of a client request, or completion of a data transfer, can be efficiently detected by entries in one or more Completion Queues. There are two basic methods to catch an event. One is that applications explicitly poll related Completion Queues to retrieve interested events. Another one is to invoke pre-registered event handlers to notify applications of events by interrupts. In this method, applications can sleep and relinquish CPU when waiting for an event. Polling is usually CPU intensive; however, it offers better response latency. While servicing an interrupt always increases the latency, it does consume fewer CPU cycles, particularly if it is necessary to poll for a long time before the event arrives.

Important goals when designing PVFS over InfiniBand are to minimize CPU overhead on the client side, minimize response latency for short transfers, and maximize throughput for large transfers. In our design, notification of completion of sending request messages on the client side is done using polling and notification of completion of incoming reply messages with interrupts. On the server side, all event notification is done with polling, as is appropriate for a dedicated machine.

3.3 Buffer Management

A buffer manager provides buffers to the PVFS transport layers. Buffers are either internal buffers or RDMA buffers. There are three main tasks in a buffer manager. First, flow control on internal buffers is to ensure that every message sent by a Send operation has a receive buffer posted on the receiver side. Second, it should provide efficient memory registration and deregistration operations for RDMA buffers. Third, a buffer manager should provide fair and dynamic sharing to buffer consumers. This task is particularly important in the I/O server.

3.3.1 Flow Control on Internal Buffers

Internal buffer management is a well-discussed issue in the literature [16, 18]. A small set of internal buffers are allocated and pinned on both sides of a connection. Each connection has a separate pool of internal receive buffers. To ensure that an incoming message can be put in an internal receive buffer, a credit-based flow control mechanism is deployed on a per-connection basis. At the beginning, some number of receive descriptors, each associated with an internal receive buffer, are posted for each connection. Then, the number of currently posted receive buffers is advertised by flow control updates, which can be piggybacked on other messages or sent as a control message. This information can also be exchanged implicitly in the flow of matched request and response message pairs.

3.3.2 Server RDMA Buffer Management

Server RDMA buffers are used to receive data from clients and to read data from files. These buffers are effectively used to bridge the performance gap between network and disk. Due to highly concurrent requests and possible large request sizes, a significant portion of the total memory must be allocated as RDMA buffers on a dedicated server. Clearly, the server can reuse these buffers for different requests. Thus, all these regions can be pre-registered at startup. The I/O server then keeps using them to service client requests. A slightly more complicated solution is that the I/O server may dynamically register or deregister some regions. For example, if the working set of client requests is not large enough, the I/O server can deregister some regions which are seldom used. This may improve performance since the system I/O cache competes for memory. The fewer buffers that are registered, the more buffers that can be used for I/O cache and other purposes. Even with this dynamics, it can be expected that the frequency of memory registration and deregistration is low in the I/O server side. Thus, efficient memory registration and deregistration is not a huge issue.

The more important function for a server buffer manager is to provide a fair and dynamic buffer sharing among all clients. This task is not difficult in PVFS over TCP/IP. First, TCP/IP provides a stream communication, the server can receive and send a large data multiple times using a smaller buffer. Second, the client side can stop sending data if there is no space left in the socket receive buffer of the server side. Third, *select* provides a mechanism to notify the server of data arrival before data placement. In the PVFS transport layer based on InfiniBand, all data is transferred as whole messages, not as bytes in a stream. Buffers are also supplied explicitly. Message transfers are thus atomic, and data placement and data arrival are not separated as they are in TCP/IP. Therefore, explicit buffer assignment is needed in PVFS over InfiniBand.

Another issue is that transfer sizes for requests are different. This variability requires that the buffer manager be able to supply different sizes of virtually contiguous buffers. Avoiding fragmentation is important in this scenario.

The server buffer manager in our design works as follows. First, all RDMA buffers are allocated and organized in zones, where each zone has buffers of the same size. There is a list of RDMA buffers with size of 64 Kbytes, a list of RDMA buffers with size of 128 Kbytes, up to a list of RDMA buffers with size of 2 Mbytes, which is the biggest zone size. Given a particular transfer size, we first look at the corresponding zone list to try to get a contiguous buffer. If there is no buffer available, the buffer may be chosen from a bigger zone list. If there is no bigger buffer available, the transfer will be chopped into small transfers using smaller RDMA buffers. By this way, there is no dynamic fragmentation and it is usually possible to transfer data with a given transfer size. Second, when a request is scheduled by the request manager, a transfer size is chosen to take advantage of potential overlap of communication and I/O. Buffers are allocated from one or more zone lists and assigned to the request. Then as parts of or the request complete, their assigned buffers are released back to the zone lists. This policy works well with PVFS, because the server is a single thread and all file operations are blocking. If, in the future, the server is multi-threaded and/or uses non-blocking file operations, this buffer manager policy will change accordingly.

3.3.3 Client RDMA Buffer Management

RDMA buffers in the client side are provided by PVFS applications. The client buffer manager is primarily responsible for efficient registration and deregistration of these memory regions. PVFS I/O applications require a large number of I/O buffers which may be allocated no earlier than when the request is issued. To reduce the cost of dynamic registration and deregistration, a pin-down cache [12] is incorporated in the buffer manager. Pin-down cache delays deregistration of registered buffers and caches their registration information. When these buffers are reused, their registration information can be retrieved from pin-down cache. This technique is quite effective when the amount of buffer reuse is high. Further optimizations performed by the client buffer manager are described in section 5.

3.4 Communication Management

This component is responsible for choosing an appropriate communication mechanism for each message. It also schedules data communication to reduce network congestion and avoid delaying other traffic in the network. It is capable of applying a service level to each message which marks its priority as it moves through the network. The motivation for this module comes from three aspects.

First, as mentioned earlier, there are a large number of options in InfiniBand data transport mechanisms for each possible PVFS message type. One role of the communication manager is to make this decision according to parameters in the message, such as destination, message length, and availability of remote buffer credits. Second, I/O traffic and communication traffic share the same InfiniBand network. A single I/O node must service requests from multiple clients and easily becomes a communication hot-spot. This localized communication pattern can lead to a severe form of congestion, which can seriously degrade the overall performance of the interconnect [11]. The situation becomes worse when the IO node performs RDMA operations to transfer data for both read and write operations since all data originates from the IO node. Careful arrangement of communication operations can reduce congestion and effect on other traffic. Third, InfiniBand provides virtual lanes and service levels to offer different service. The communication manager can assign service levels to messages according to the related request requirement.

4 Optimizations in the PVFS Transport Layer

In general, our experience shows that native InfiniBand primitives can be instrumental in reducing overheads and increasing bandwidth in the I/O path. We have also encountered a number of challenging issues in designing PVFS over InfiniBand, including small data transfers and bulk data transfers. To deal with these issues, we explore various optimizations here and quantify their impact on performance in Section 6.

4.1 Small Data Message Transfer

Recall that data messages are transferred with either server-initiated or client-initiated RDMA operations. Either way, application buffers must be registered before data transfer. These buffers may also need to be deregistered after data transfer due to a limitation on total size of registered buffers. For small data messages, the performance benefit of zero-copy transfer may not offset the cost of memory registration and deregistration. Two optimizations can be applied to improve this situation in the case of small data message transfers.

4.1.1 Inline Data Transfer

Data is first copied into internal buffers which are pre-registered and then transferred by Send/Recv mechanism. For PVFS write data, if they can fit in an internal buffer with the request message, data and request are sent in one message. Otherwise, following the request message, the remaining data are sent separately. For PVFS read data, the server acts similarly. Data is sent either together with the reply message or as a separate message. One copy on the client side is then required to place the data. On the the server a copy is not usually necessary because it can process the request message in place. This technique has been used elsewhere [10].

4.1.2 Fast RDMA Write

Figure 5 shows there is a significant performance difference between RDMA Read and RDMA Write when the transfer size is not large. This implies that using RDMA Write for small data transfers is preferable if the benefit can offset the overhead of doing so. *Fast RDMA Write* optimizes PVFS write and read operations as follows.

To optimize small writes, the client does RDMA Write to transfer data to the I/O server. However, as shown in Figure 4(b), two additional control messages are needed. To avoid the first control message, a small set of RDMA buffers (called Fast RDMA buffers) are allocated and registered when a connection is established. The buffer information is cached on the peer side. Thus, the client can RDMA write data directly into the Fast RDMA buffers on the server. We use RDMA Write with Immediate data to avoid the second control message. Furthermore, we add another interface, *check_if_registered*, into our pin-down cache implementation. This call returns registration information if a buffer happens to be cached. Otherwise, it returns NULL. Before data transfer, the client calls *check_if_registered* to see if the user buffer is registered. If so, Fast RDMA Write is carried out between the registered user buffer and the server Fast RDMA buffers. If not, the client first copies data into *its* Fast RDMA buffers, and Fast RDMA Write is carried out between the Fast RDMA buffers of both sides. In the latter case, the request message can be combined with the data message in one operation using one data segment. In the former case, one RDMA Wirte with Immediate data can be used, however, two data segments must be specified in RDMA Write gather list, one for request and one for data.

To optimize small reads, the client again calls *check_if_registered* first to find out whether the user buffer is registered. If so, it sends to the server information on the registered user buffer in the request message. Otherwise, it supplies to the server a pointer to a Fast RDMA buffer. Similarly, the reply message can be combined with the data message in this case. Then, the server performs RDMA Write as instructed. The client must copy data out of the Fast RDMA buffer if the user buffer was not registered.

The number of Fast RDMA buffers per connection needed on the server side is variable according to resource availability. However, this number and the Fast RDMA buffer size can become a hindrance to scalability in a large system. In PVFS, since there is only one outstanding read or write from each client, one Fast RDMA buffer for each connection works well. Thus, scalability is not an issue. If more than one outstanding request is allowed, as expected in the next-generation design of PVFS, more Fast RDMA buffers can offer better performance. However, flow control must be applied to ensure that future requests do not overwrite earlier ones. The optimal Fast RDMA buffer size should be decided by comparing the cost of memory registration and deregistration to the cost of copying. A similar technique has been used on other networks without RDMA Read support [21, 17].

4.2 Pipelined Bulk Data Transfer

Scientific applications frequently write large amounts of data (100 MB to 10 GB), such as to perform checkpoints or to output results. There are two major phases in each I/O path: communication phase, where data is transfered between client buffers and server buffers, and I/O phase, where data is moved from server buffers to disk. Overlap between these two phases is necessary for high performance in the case of large write (or read) requests. One way to achieve communication and I/O overlap is to split large transfers into multiple smaller transfers. For example, when a client wants to read 100 MB from a server, the server can read 1 MB, then start a 1 MB RDMA write operation to the client, then repeat these two operations another 99 times.

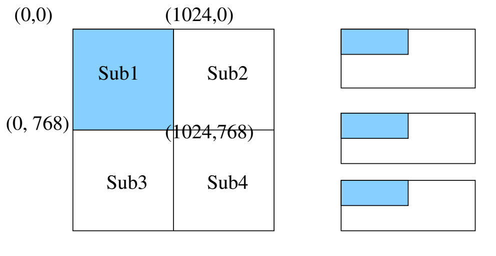
In PVFS, the transfer size is usually the same as the stripe size of the file due to the contiguity of client buffers and server files. In cases where larger transfer units are possible, the transfer size should be no more than half of the total size to achieve good pipelining.

Pipelining communication and I/O also reduces memory pressure in I/O servers. The I/O server can use double buffering to service concurrent requests. Thus, each request only needs buffer space for two transfer sizes (2 MB), not one buffer for the entire size (100 MB).

5 Efficient Support for List I/O

Another challenging issue we faced is to provide efficient list I/O operations in PVFS over InfiniBand. The list I/O interface provided by PVFS allows a set of buffers to be used as read or write as destinations in memory on the client and a set of offsets in the file on the server. This interface offers efficient noncontiguous accesses in both memory and file space [7]. For example, MPI-IO [31] uses this interface to implement noncontiguous accesses using the MPI DataType representation. Supporting list I/O efficiently in PVFS is critical to user applications. In our design, we applied the small data transfer and pipelined bulk data transfer optimizations discussed above to list I/O operations as appropriate. However, there is still a performance problem related to the registration of memory regions in list I/O.

This complication in list I/O may result in a large number of buffer registration and deregistration events, even when using a registration cache. Considering the following example as illustrated in Figure 6(a), assume a process uses the upper corner 1024×768 subarray (*sub1*) in a two-dimensional 2048×1536 array of characters as the destination for a file read. There are 768 buffers, one for each row, which are not contiguous with each other. If these 768 buffers are registered individually through the pin-down cache, it may result in serious performance problems, for many reasons. First, it results in high registration overhead, since it is likely that some of the individual rows are not already pinned. Second, cache misses increase because a large number of pin-down entries are needed for each list I/O, perhaps causing some cached buffers to deregister. This not only increases the registration time for the list I/O buffers for this operation, but also may require another registration to repin frequently used buffers. Registration cache thrashing may occur. Thus, reducing the number of buffers needed to be registered as much as possible is critical to alleviate these problems.



(a) One Big Buffer (b) Three Big Buffers

Figure 6. Examples of the List I/O Extended Interface

Another observation we have on VI-networks [3] and InfiniBand is that to register and deregister a large buffer once is much more efficient than to register and deregister the same size buffer in multiple small chunks. For example, it takes 530 μs to register and deregister a 400 kB buffer, while it takes 12 400 μs to register and deregister 100 buffers of size 4 kB each on our InfiniBand testbed.

Based on these observations, we suggest a procedure to reduce registration overheads in the case of list I/O operations. The first step is to concatenate buffers which are actually parts of the same allocation in process virtual memory space. This can effectively reduce the number of registrations. The trivial case is when all buffers in a list I/O operation are contiguous, with no holes, and can then obviously be treated as one region. The more likely case issue is the one illustrated in Figure 6(a). Holes between these buffers may or may not have been allocated in the memory of the process, and there is no guaranteed way to tell. The *mincore* system call in some operating systems could be used to query the status of individual pages, but the result is not guaranteed accurate. More information is needed from the application itself.

The second step in our optimization is to decide how much of the buffer should be registered. Given that we know that (some of) the individual components of the list I/O are parts of a single contiguous virtual memory allocation, is it better to register the entire large allocation or just a subset which covers the areas to be accessed in a list I/O? It is likely true that we should prefer to perform a single registration of the entire allocation, given the observation on how much faster that will be, and the chance that some part of the registered area will be reused in later calls. However, there is a limitation on the total size of registered buffers as well as the number of registered buffers. Again, it is up to the PVFS applications to specify the expected registered length of a buffer by considering the access patterns of the region.

To convey this information from the application, we propose an extension to the PVFS list I/O interface, which allows PVFS applications to pass additional buffer information for efficient memory registration:

```
pvfs_read_list(int fd,
    int      mem_list_count,
    void *   mem_offsets[],
    int      mem_lengths[],
    void *   allocation_offsets[],
    int      allocation_lengths[],
    int      file_list_count,
    int64_t  file_offsets[],
    int32_t  file_lengths[])
```

Two arrays are added as input parameters, shown in bold above, to provide information on the actual allocations underlying the memory regions passed in the call. For a buffer listed in *mem_offsets*, the address of the largest buffer to which it belongs (the “parent buffer”) is specified in *allocation_offsets*. The length of the part of the parent buffer which is expected to be registered is specified in *allocation_lengths*. Clearly, this length must not be less than the related length specified in *mem_lengths*. For example, in Figure 6(a) the parent buffer address is the initial address of the whole two-dimensional array, and the allocation length is the length of the array which covers all list buffers, i.e. $2048 \times 767 + 768$ bytes. The allocation lengths can also be the whole length of the array if the application knows it is beneficial to pin the entire array. These two additional parameters provide important information to the buffer manager to achieve efficient memory registration and deregistration. They also provide flexibility to PVFS applications. Figure 6(b) demonstrates the case where list I/O is used for buffers which come from different allocations in the process memory space. If we had proposed that *pvfs_read_list* be extended to *assume* a single contiguous allocation, this usage model would no longer be available to applications. Instead, PVFS applications can use these two new parameters to notify the buffer manager that list I/O buffers are subsets of three different parent buffers, in this case.

This extended list I/O interface works well with MPI-IO which uses MPI derived datatypes to achieve noncontiguous accesses. If an access is contiguous in memory, only one registration is needed. If an access is noncontiguous in memory, for example, a process uses the subarray as shown in Figure 6(a) as a derived Datatype to access a file, before calling a PVFS list I/O routine, the MPI-IO layer such as ROMIO [31] can parse the corresponding datatype and fill out appropriate values in the input parameters. This results in requiring only one registration in the common case where a single initial buffer address is used for each list I/O request.

Since the native PVFS interfaces are usually used in other portable layers such as ROMIO, this extension will not disturb end-user applications.

6 Performance Results

We have implemented PVFS on our InfiniBand testbed with designs described in Sections 3 and 4. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [23], which is a user-level programming interface developed by Mellanox and compatible with the InfiniBand Verbs specification. This section presents performance results from a range of benchmarks on our implementation of PVFS over InfiniBand. First, we quantify that PVFS can take full advantages of InfiniBand features to achieve high throughput, low CPU utilization, and high scalability by comparing performance of our implementation with that of PVFS over IBNice [23], a TCP/IP implementation for InfiniBand. We use both PVFS and MPI-IO micro-benchmarks as well as applications to carry out the comparison. Then we examine the impact of system optimizations on PVFS performance. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

6.1 Experimental setup

Our experimental testbed consists of a cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.0.6-rc1-build-002. The adapter firmware version is fw-23108-1.16.0000_5-build-001. Each node has a Seagate ST340016A, ATA 100 40 GB disk. We used the Linux RedHat 7.2 operating system.

6.2 Network and File System Performance

Table 2 shows the raw 4-byte one-way latency and bandwidth of VAPI and IBNice. IBNice is a TCP/IP stack over InfiniBand offered by Mellanox. The benchmark we used for this purpose is *ttcp*, version 1.12-2, with a large socket buffer size of 256 kB to improve IBNice performance. The VAPI Send/Recv and RDMA Write performance is measured using the Mellanox *perf_main* benchmark. The VAPI RDMA Read performance is measured using our own program which is constructed similarly to *perf_main*.

Table 3 compares the read and write bandwidth to an *ext3fs* file system on the local 40 GB disk against bandwidth achieved to memory, using *ramfs*, a RAM file system. The *bonnie* file-system benchmark is used.

It can be seen that there is a large difference in bandwidth realizable over the network compared to that which can be obtained to a disk-based file system. However, applications can still benefit from fast networks for many reasons

Table 2. Network performance

	Latency (μs)	Bandwidth (MB/s)
IBNice	40.1	185
VAPI Send/Recv	9.2	825
VAPI RDMA Write	6.0	827
VAPI RDMA Read	12.4	816

Table 3. File system performance

	Write (MB/s)	Read (MB/s)
ext3fs	25	20
ramfs	556	1057

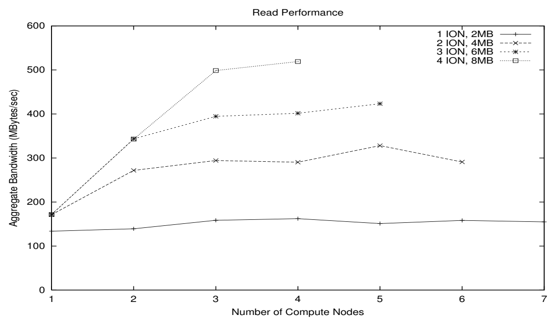
in spite of this disparity. Data is frequently already in server memory due to file caching and read-ahead when a request arrives. Also, in large disk array systems, the aggregate performance of many disks can approach network speeds. Caches on disk arrays and on individual disks also serve to speed up transfers. Therefore, the following experiments are designed to stress the network data transfer independent of any disk activities. We mainly focus on experiments on a memory-resident file system. Results on ramfs are representative of workloads with sequential I/O on large disk arrays or random-access loads on servers which are capable of delivering data at network speeds. We also show some results on ext3fs to quantify the impact of CPU utilization.

6.3 PVFS Concurrent Read/Write Bandwidth

The test program used for concurrent read and write performance is *pvfs-test*, which is included in the PVFS release package. We followed the same test method as described in [6]. In all tests, each compute node writes and reads a single contiguous region of size $2N$ MB, where N is the number of I/O nodes in use.

Figure 7 shows the read and write performance with IBNice on the InfiniBand networks. For reads, the bandwidth increases at a rate of around 120 MB/s with each additional compute node. Similar performance can be seen for writes with IBNice. The bandwidth here increases at a rate of approximately 160 MB/s with each additional compute node when there are sufficient I/O nodes to carry the load.

Figure 8 shows the read and write performance of our implementation PVFS over native VAPI. The same physical networks are used yet significant performance improvements by designing and implementing PVFS on native VAPI layers is achieved. Since data transfers are mostly performed using RDMA initiated by the I/O nodes, the aggregate capacity of all the I/O nodes can be delivered to compute nodes. The bandwidth increase from adding another I/O node is roughly 400 MB/s for simultaneous reads from many compute nodes. For writes, the bandwidth increases at approximately the same rate, though slightly less



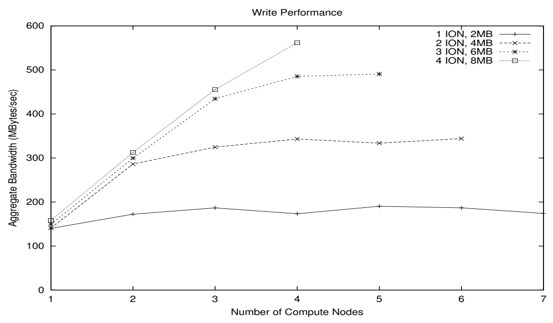


Figure 7. PVFS performance with IBNice (TCP/IP over InfiniBand).

due to the lower performance of RDMA Read compared to RDMA Write.

6.4 MPI-IO Micro-Benchmark Performance

The same test as in the previous section was modified to use MPI-IO calls rather than native PVFS calls. The number of I/O nodes is fixed at four, and the number of compute nodes was varied from one to four. Figure 9 shows the performance of MPI-IO over PVFS on VAPI and IBNice, for both memory and disk file systems. On the RAM file system, Figure 9, shows that PVFS native over VAPI offers about three times better performance than PVFS over IBNice. Even on a disk file system, ext3fs, it can be seen that although each I/O server is disk-bound, a significant performance improvement, 15–42%, is still achieved. This is because the lower overhead of PVFS-VAPI leaves more CPU cycles free for I/O servers to process concurrent requests. With four compute nodes, MPI-IO over PVFS-VAPI can achieve 95 MB/s aggregate write bandwidth, which is almost four times the peak write bandwidth of the disks we used for the tests. This shows that PVFS-VAPI offers almost perfect performance aggregation of multiple I/O servers.

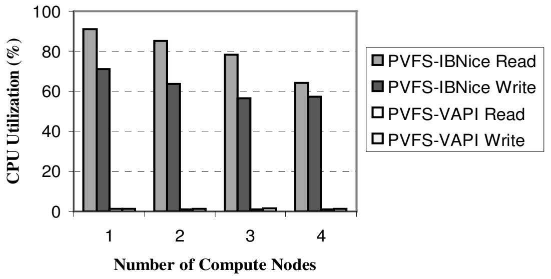


Figure 10. CPU Utilization of MPI-IO

Figure 10 shows CPU utilization on the compute nodes when the same program runs with four I/O servers on ramfs. It can be seen that the CPU overhead of compute nodes is as high as 91% in PVFS-IBNice. This is because that the overhead of PVFS over IBNice is dominated by the data transfer, mostly because of copying overhead, context switches and system calls in IBNice. CPU utilization drops off with increasing number of compute nodes, because the waiting time increases in each request when the server has more concurrent requests to service. However, the CPU utilization is still considerably high. In contrast, the overhead of PVFS over VAPI is dominated by request initialization and response handling costs in the PVFS client code, since the NIC handles data transport using RDMA and there is no kernel involvement in the I/O path. The CPU overhead is as low as 1.5% which will enable greater scalability to a large number of compute node clients.

6.5 Impact of Small Data Transfer Optimizations

To evaluate the impact of various small data transfer optimizations, we measured the access time of small PVFS read and write requests for different design schemes. The access size varies from 128 B to 64 kB. Figure 11 shows that these optimizations result in significant improvements on write performance. It also shows these optimization schemes differ. As mentioned in section 3, server-based data transfer is the basic scheme used for PVFS writes. When user buffer registrations are all cached, the server uses RDMA read to move data directly from user buffers, noted as *Server-based, 100% hit* in the plot. If user buffers are not cached, user buffers must first be registered. The worst case where all buffers are not cached is labeled *Server-based, 0% hit* in Figure 11.

In both cases, the Fast RDMA scheme offers the best performance. When buffers are all cached, 100% of accesses hit in the pin-down cache. Since one copy is needed in the Inline scheme, the Fast RDMA scheme outperforms the Inline scheme, especially for large messages. Both schemes offer better performance than the Server-based scheme, as both Send/Recv and RDMA Write in the native VAPI layer perform significantly better than RDMA Read on small data transfers. When no buffers are cached, 0% of accesses hit in the pin-down cache. One copy is needed in both the Inline and Fast RDMA schemes. Since RDMA Write performs slightly better than Send/Recv in our testbed, the Fast RDMA scheme offers the best performance. As shown in the left graph of Figure 11, there is a significant performance drop in the Server-based scheme which requires reg-

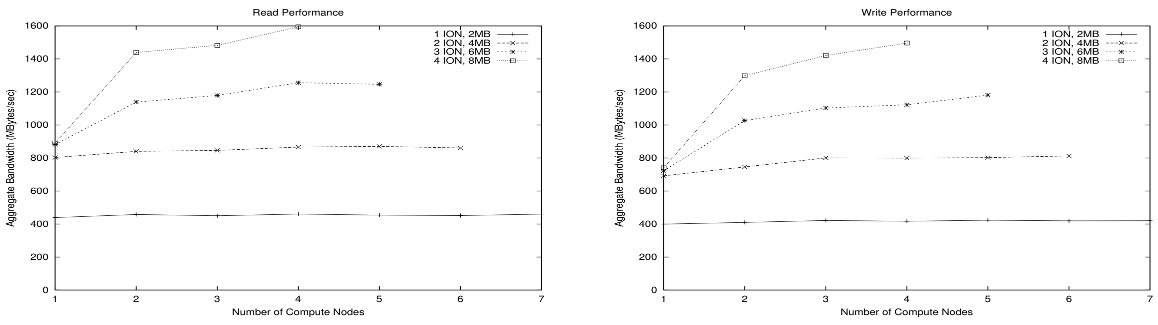


Figure 8. PVFS performance with InfiniBand VAPI

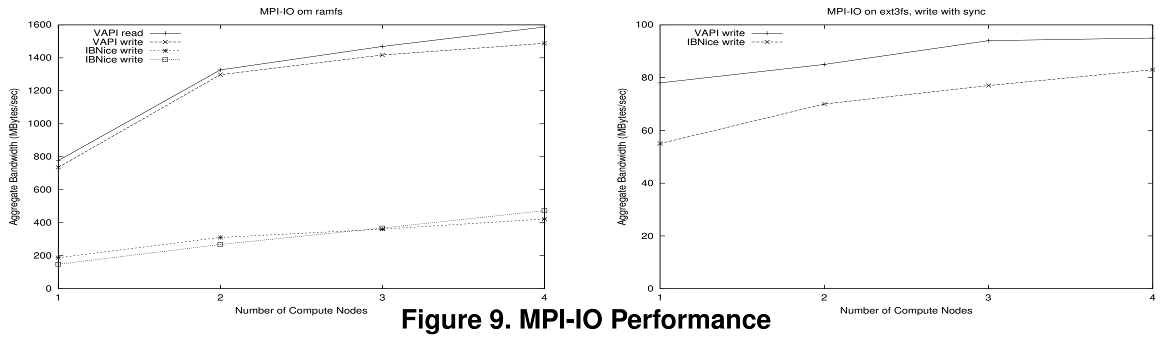


Figure 9. MPI-IO Performance

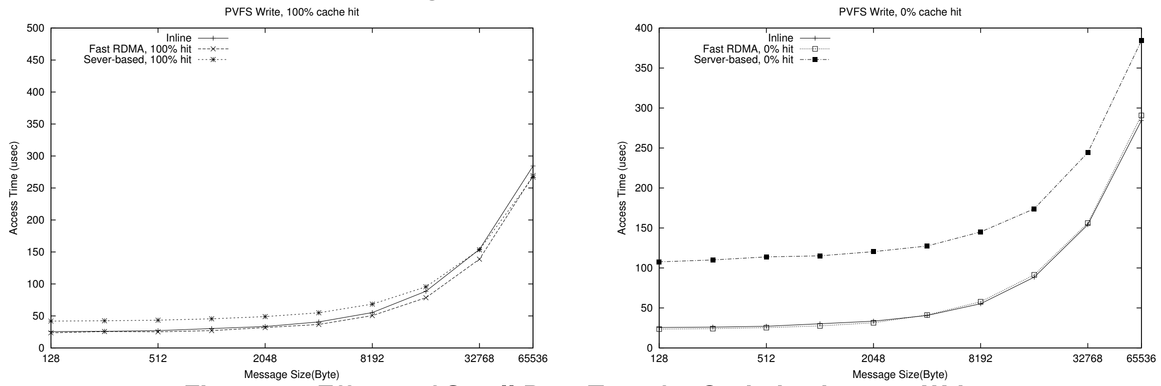


Figure 11. Effects of Small Data Transfer Optimizations on Write

istering user buffers performs worst due to the prohibitively costly memory registration. in the current release of Inifni-Band software.

Similar results are achieved for read performance, but not shown here. These results were used to decide the scheme used by the communication manager. Since there is no material difference between the Inline and Fast RDMA schemes, for simpler design complexity, the Inline scheme is used to transfer messages less than 4 kB, Fast RDMA for messages up to 64 kB, and Server-based to transfer data larger than 64 kB.

6.6 Impact of Pipelined Bulk Data Transfer

This experiment was designed to show the effect of pipelined bulk data transfers in PVFS over InfiniBand. In this test, a PVFS client transfers 32 MB to or from an I/O server using the RAM file system. This test represents workloads in which large amounts of data are moved to or from a single large buffer on the client, such as for a check-point snapshot.

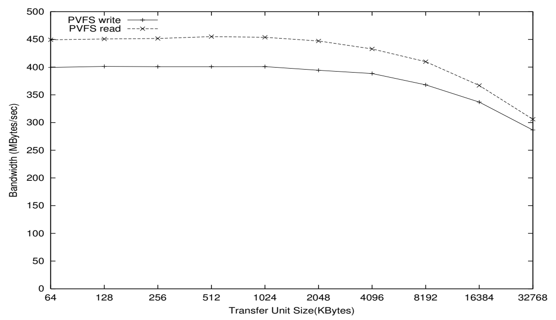


Figure 12. Effects of Pipelined Bulk Data Transfer, 32 MBytes

Figure 12 shows the impact of transfer unit size on PVFS performance, from a single 32 MB on the right-hand side of the graph to 512 small transfers on the left. The results show that a transfer size smaller than about 2 MB is sufficient to allow complete overlap between I/O access and communication. There is a slight degradation when the transfer size is very small due to the effect of communication startup overheads.

6.7 Impact of List I/O with File Discontiguity

The test application *mpi-tile-io* [25] implements tiled access to a two dimensional dense dataset. This type of workload is seen in visualization applications and in some numerical applications. For our tests, we used four compute nodes and four I/O server nodes. Each compute node renders to one of a 2×2 array of displays, each with 1024×768 pixels as illustrated in Figure 6(a). The size of each element is 24 bytes, leading to a file size of 72 MB.

The access pattern in this test is noncontiguous in file space but contiguous in memory. This is a good candidate to exercise PVFS list I/O. We test two versions of *mpi-tile-io*: one is to use multiple contiguous I/O operations to achieve noncontiguous file accesses ("Without list I/O"), the other uses PVFS list I/O to make a single noncontiguous access.

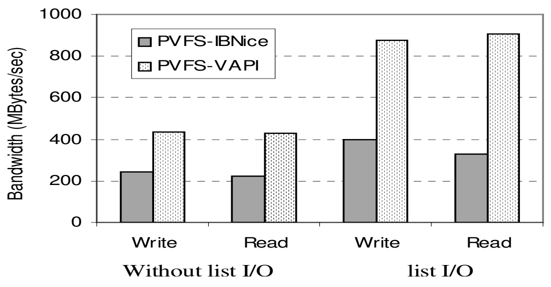


Figure 13. Performance of tiled I/O.

Figure 13 shows the results for both PVFS-VAPI and PVFS-IBNice. It can be seen that list I/O improves performance significantly. This is because 768 individual requests are required to access a tile when not using PVFS list I/O. The overhead of request and reply messages becomes dominant in this case. With list I/O, however, only 6 requests are required since one list I/O request is large enough to contain 128 file accesses specifications. Compared to the performance of PVFS-VAPI and PVFS-IBNice, with list I/O, PVFS-VAPI offers 2.7 and 2.2 times the bandwidth on read and write, respectively. Without list I/O, the improvement is 79% and 93%. This difference is because the access size is larger with list I/O and can yield more improvement from the VAPI layer.

6.8 Impact of List I/O Memory Registration

We modified *mpi-tile-io* to show the effect of list I/O memory registration and deregistration on its performance. The modification is to use noncontiguous accesses in memory as well. We use the same parameters as in the previous section, except that the size of an element is now 64 bytes. This change forces the test not to use Fast RDMA Write to transfer data so that the client will be forced to register memory to carry out the transfers. Also each client allocates a full 2048×1536 array so that its individual subarray will be stored as noncontiguous stripes in memory. Thus, noncontiguous accesses in both memory and file space are necessary. We also changed the ROMIO [31] source code to use the extended interfaces for *pvfs_read_list* and *pvfs_write_list* that we proposed in Section 5.

The number of memory registration and deregistration events is interesting as it will indicate transfer overhead. With the unmodified PVFS interface, each process initiates 6 list I/O requests; however, the requests reference memory regions which are not adjacent thus 768 memory registration operations are needed. With the extended interface, though, PVFS is informed that these 768 memory regions are actually all from the same allocation. In fact, the ADIO layer [30] knows this information when it parses the derived Datatype and composes parameters of PVFS list calls; the application is unchanged. Thus, only 6 memory registration operations are needed, 5 of which are cached.

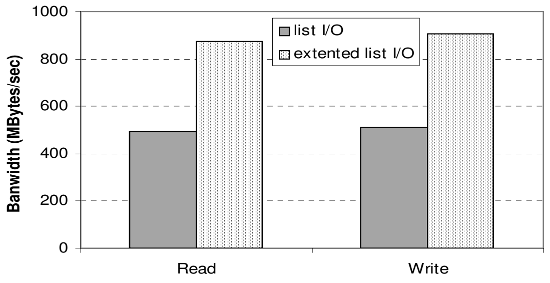


Figure 14. Effects of the Extended List I/O Interface

Figure 14 shows there is a 85% bandwidth improvement for reads, and 71% for writes, owing to the list I/O buffer management optimization. Note that the deregistration time is not taken into account in read and write operations in this test, since the size of the cache is sufficient to contain all the entries required for either case. If this were not true, the performance gap would be even larger.

7 Related Work

Various user-level communication protocols have been used for network storage in the past. Zhou *et al.* [34] present their experiences with VIA networks for database storage. They implemented a block-level storage architecture that takes advantage of features found in VI communication systems. They found that VIA can improve I/O performance between the database system and the storage back-end. Magoutis *et al.* [22] explore DAFS performance characteristics, also on VIA. Our work is based on the InfiniBand architecture which provides more features and services than VIA such as RDMA Read and service levels, yielding a more flexible design space and different design goals and techniques.

A set of transport layers based on user-level communication networks have been discussed and targetted for different domains. Zahir [33] describes a storage-networking transport layer for the Lustre file system based on VI-like networks. Carns [4] designs a Buffer Message Interface (BMI) as a transport layer for the next generation PVFS. His prototype implementation works on both TCP/IP and Myrinet/GM. Liu *et al.* [17] propose a client/server communication middleware over system area networks. It provides a communcation abstraction to upper layers by hiding the discrepancy of various system area networks in the middleware. We share similarities with these efforts in designing a transport layer on InfiniBand to support PVFS, although our work differs in significant ways. First, the design of this transport layer is customized for high performance by taking advantage of PVFS protocol characteristics. Second, our transport layer is capable of cooperating with buffer management and communication management to deal with particular issues in I/O intensive applications.

Memory registration and deregistration are a common issue in modern networks which provide RDMA capabilities. Basu *et al.* [32] show how the NIC and host-level software can collaborate to manage large amounts of host memory. Tezuka *et al.* [12] propose a pin-down cache to reduce memory registration and deregistration overhead for zero-copy communication. Zhou *et al.* [34] propose a *batched deregistration* scheme to deregister all buffers in a region in one operation. Significant changes in both host-level software and the NIC have been made in their approach. In our work, we deploy a pin-down cache in the PVFS layer; however, we focus on optimization on reducing calls to the cache.

Coll *et al.* [8] show the importance of the placement of I/O nodes in a cluster system. The main purpose of such placement is to reduce I/O traffic congestion and the effect on other traffic from the physical level. Our work focuses on the software level. Their work can be combined with our communication management to achieve the best performance.

8 Conclusions and Future Work

In this paper, we study how to leverage InfiniBand technologies to improve I/O performance and scalability of cluster file systems. We design and implement a version of PVFS that takes advantage of InfiniBand features. Our work shows that the InfiniBand network and its user-level communication and RDMA features can improve all aspects of PVFS, including throughput, access time, and CPU utilization. However, InfiniBand networks also pose a number of challenging issues to I/O intensive applications such as PVFS. In particular, we address the issues in this paper with: a transport layer customized for the PVFS protocol by trading transparency and generality for performance, buffer management for flow control and efficient memory registration and deregistration, and communication management for reducing network congestion and achieving differentiated services.

Compared to a PVFS implementation over standard TCP/IP on the same InfiniBand network, our implementation offers three times the bandwidth if workloads are not disk-bound and 40% improvement in bandwidth if disk-bound. The client CPU utilization is reduced to 1.5% from 91% on TCP/IP.

As of this writing, a major rewrite of PVFS is in active development. Our work is directly applicable to this next generation PVFS over networks with user-level access and RDMA capabilities. We are eagerly working with the PVFS team to incorporate our design into the next PVFS and to implement PVFS on InfiniBand networks.

Acknowledgments

We would like to thank PVFS team for giving us the latest version of PVFS implementation and Rob Ross for providing us with insights into their implementation. We are also thankful to Jiuxing Liu and Sushmitha P. Kini for many discussions with us.

References

[1] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley. Cheating the i/o bottleneck: Network storage with trapeze/myrinet. In *Proceedings of the Usenix Technical Conference. New Orleans, LA.*, 1998.

[2] M. Bancroft, N. Bear, J. Finlayson, R. Hill, , R. Isicoff, and H. Thompson. Functionality and performance evaluation of file systems forstorage area networks (san). In *the Eighth NASA Goddard Conference on Mass Storage Systems and Technologies*, 2000.

[3] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishna, P. Sadayappan, H. Sah, and D. K. Panda. VIBe: A Micro-benchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations. In *IPDPS*, April 2001.

[4] P. Carns. Design and analysis of a network transfer layer for parallel file systems. Master thesis. http://parlweb.parl.clemson.edu/techreports/.

[5] P. Carns. Parallel Virtual File System Version 2. http://parlweb.parl.clemson.edu/pvfs2/.

[6] 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, Atlanta, GA, 2000. USENIX Association.

[7] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[8] S. Coll, F. Petrini, E. Frachtenberg, and A. Hoisie. Performance Evaluation of I/O Traffic and Placement of I/O Nodes on a High Performance Network. In *Workshop on Communication Architecture for Clusters 2002 (CAC '02)*, April 2002.

[9] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[10] DAFS Collaborative. Direct Access File System Protocol, V1.0, August 2001.

[11] G. F. Pfister and V. A. Norton. Hot-spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers, C-34 (10)*, pages 943–048, 1985.

[12] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In Proceedings of 12th IPDPS.

[13] http://www.lustre.org/. Lustre: Scalable clustered object storage, June 2002.

[14] http://www.top500.org/. TOP500 List for November 2002, Nov. 2002.

[15] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[16] InfiniBand Trade Association. Socket Direct Protocol Specification V1.1, 2002.

[17] J. Liu, M. Banikazemi, B. Abali, and D. K. Panda. A Portable Client/Server Communication Middleware over SANs: Design and Performance Evaluation with InfiniBand. In *SAN-02 Workshop (in conjunction with HPCA)*, Feb. 2003.

[18] Lawrence Livermore National Laboratory. MVICH: MPI for Virtual Interface Architecture, August 2001.

[19] C. Lever and P. Honeyman. Linux nfs client write performance. In *Proceedings of the Usenix Technical Conference, FREENIX track, Monterey*, June 2001.

[20] J. Liu, J. Wu, S. P. Kinis, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. K. Panda. MPI over InfiniBand: Early Experiences. Technical Report, OSU-CISRC-10/02-TR25, Computer and Information Science department, the Ohio State University, January 2003.

[21] K. Magoutis. Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD. In *Proceedings of USENIX BSDCon 2002 Conference*, February 2002.

[22] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[23] Mellanox Technologies. Mellanox InfiniBand InfiniHost Adapters, July 2002.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[25] R. Ross. Parallel I/O Benchmarking Consortium. http://www-unix.mcs.anl.gov/ rross/pio-benchmark/html/.

[26] R. B. Ross. Reactive scheduling for parallel i/o systems. PhD dissertation. http://parlweb.parl.clemson.edu/techreports/.

[27] M. W. Sachs and A. Varma. Fibre Channel. *IEEE Communications*, pages 40–49, Aug 1996.

[28] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *fast02*, pages 231–244. USENIX, Jan. 2002.

[29] Storage Networking Industry Association. Shared storage model. www.snia.org/tech_activities/shared_storage_model.

[30] R. Thakur, W. Gropp, and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Oct. 27–31, 1996.

[31] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[32] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proc. Hot Interconnects V*, August 1997.

[33] R. Zahir. Lustre Storage Networking Transport Layer. http://www.lustre.org/docs.html.

[34] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 257–268. IEEE Computer Society, 2002.