PVFS over InfiniBand: Design and Performance Evaluation

Jiesheng Wu, Pete Wyckoff[†], and Dhabaleswar K. 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

Technical Report
OSU-CISRC-5/03-TR25

PVFS over InfiniBand: Design and Performance Evaluation [*]

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 technology to improve I/O performance and scalability of cluster file systems. We use Parallel Virtual File System (PVFS) as a basis for exploring these features.

In this paper, we design and implement a PVFS version on InfiniBand by taking advantage of InfiniBand features and resolving many challenging issues. We design the following: a transport layer customized for PVFS by trading transparency and generality for performance; buffer management for flow control, dynamic and fair buffer sharing, and efficient memory registration and deregistration.

We define strategies to make communication selection in the PVFS transport layer over InfiniBand. Three techniques: Inline, Fast RDMA Write, and Pipelined Bulk data transfer in the transport layer are also proposed. Our results show that these techniques achieve significant improvement on performance. We also design a two-level memory registration and deregistration architecture to provide efficient memory registration and deregistration. Our results show that this architecture works better than other schemes and provides efficient memory registration and deregistration in the I/O intensive environment.

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. To the best of our knowledge, this is the first design, implementation and evaluation of PVFS over InfiniBand. The research results demonstrate how to design high performance parallel file systems on next generation clusters with InfiniBand.

[*]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.

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 [12]. 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 systems 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 [3, 6, 28, 19, 27, 31, 32]. 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 [2, 16, 22, 26]. Emerging network architectures such as Virtual Interface (VI) Architecture [8] and InfiniBand Architecture [13, 23] 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 datacenter 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 [17], 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 technology 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, including PVFS software architecture which can take full advantage of InfiniBand features, efficient transport layer to support PVFS protocols, and buffer management. 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 [20], 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 research shows that:

1. The capabilities of InfiniBand user-level communication and RDMA can improve all performance aspects of PVFS, including bandwidth, access time, and CPU utilization.
2. A transport layer based on InfiniBand user-level programming interface requires careful design regarding aspects of communication strategy selection and various optimizations in itself and interactions with other software components.
3. Memory registration and deregistration for networks with remote DMA capabilities adds a new dimension to transport issues for I/O intensive applications. They pose challenges on cluster file systems and require careful management of buffer resources.
4. Compared to a PVFS implementation over TCP/IP on the InfiniBand network, our implementation offers a factor of three improvement in throughput. Utilization decreases from 91% with IBNice to 1.5% in our native implementation.

The rest of the paper is organized as follows. We first give a brief overview on InfiniBand in section 2. Section 3 presents the architecture of PVFS over InfiniBand. Sections 4 and 5 describe the design of the PVFS transport layer and buffer manager over InfiniBand, respectively. The performance results are presented in section 6. Finally we examine related work in section 7 and draw our conclusions and discuss future work in section 8.

2 Overview of InfiniBand

The InfiniBand Architecture (IBA) [13] 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 Proposed PVFS Architecture

In this section, we first give a brief overview of PVFS. Then we define a general software architecture of PVFS based on InfiniBand.

3.1 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.

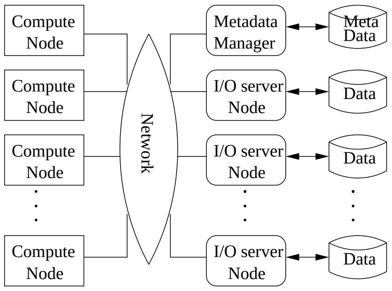


Figure 1. Typical PVFS setup.

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 [29], 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.

3.2 Proposed PVFS Software Architecture

Figure 2 shows our proposed PVFS software architecture over the InfiniBand network. 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.

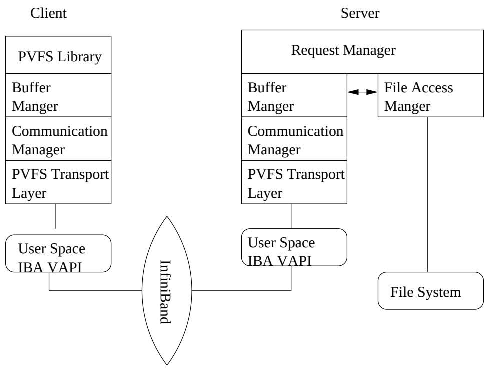


Figure 2. Proposed PVFS Software Architecture on InfiniBand Network.

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.

InfiniBand network offers much more flexible design space for PVFS compared to other networks. Communication manager 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.

In the original PVFS architecture, file access manager and request manager are currently undergoing redesign for the second version of PVFS. Since these components are independent of network characteristics, they are not discussed further here. We refer readers to [5] and [25] for details.

In this paper, we focus on the transport layer and buffer manager, which become more complicated when designing PVFS over InfiniBand as compared to the original design of PVFS over TCP/IP. Communication manager is also unique over InfiniBand, however, due to the space limitation, we do not cover it in details in this paper.

4 Designing 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. Second, we make appropriate communication strategy selection for them, including communication choices, message transfer mechanisms and event handling. Then we propose optimized small data transfers and pipelined bulk data transfers to further optimize the PVFS transport layer.

4.1 Messages and Buffers in PVFS

Messages in PVFS can be categorized as follows:

1. **Request messages:** A request message is sent by the compute node to the server (I/O server node or the metadata manager server) to direct it to initiate operations such as read, write, and lookup. The manager node also uses a request message to inform the I/O server node 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 and remain active for a long period of time. 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.

4.2 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 Send/Recv operations and both read and write RDMA operations. For each operation, the initiator can choose whether to generate a completion event or not. Send/Recv operations and RDMA Write with Immediate data operations consume receive descriptors and result in Solicited or Unsolicited completion on the receive side [13]. 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 characteristics and suitable communication choices.

Both Send/Recv and RDMA Write with Immediate Data can be used to transfer reply messages, the choice can be made according to performance attained by the two operations on a given hardware platform. In our testbed, no significant performance difference was detected. Thus, we choose send/recv since its design complexity is somewhat less than RDMA Write with Immediate Data.

For data messages, the decision pertaining whether to use RDMA Write or Read is also critical and discussed in section 4.3. For small data messages, a tradeoff can be made between the use of zero-copy RDMA data transfers and non zero-copy transfers. We discuss the details of this choice in section 4.5.

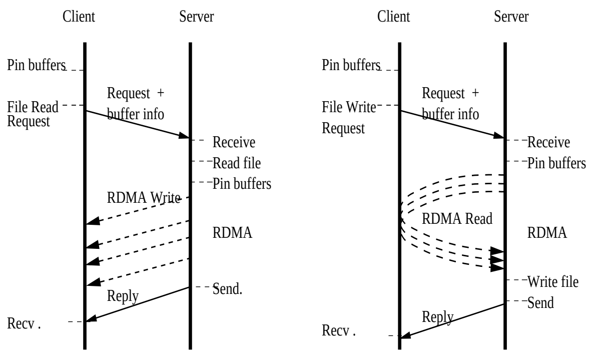
The completion of Send, RDMA Write and Read operations on the initiator side is somewhat complicated by the need to drive the message progress engine. It can be expected that better performance can be achieved by avoiding an explicit completion notification; however, this notification provides an easy 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 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.

4.3 Message Transfer Mechanisms

As discussed in previous subsection, 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*, *server-based RDMA*, *client-based RDMA*, and *hybrid RDMA*. We elaborate these mechanisms below and show how to map PVFS operations to them.

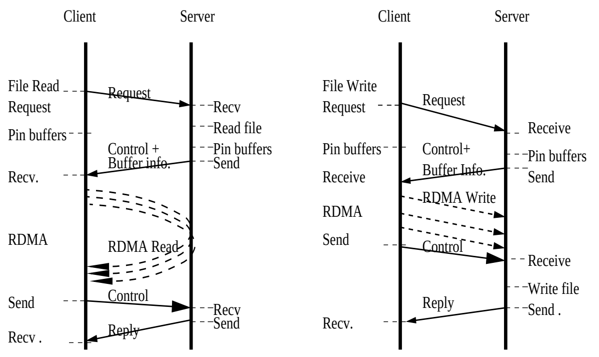
Table 1. Communication Choices

Message	Characteristics				Choices		
	Expected?	Size	In-place processing?	Immediate Attention?	Operation	Recv Completion	Send Completion
Request	No	Short	Yes	Yes	Send/Recv	Solicited	Yes
Reply	Yes	Short	Yes	Yes	Send/Recv, RDMA Write	Solicited	Yes
Control	No	Short	Yes	Yes	Send/Recv	Solicited	Yes
Data	Yes	Variable sizes	Zero-copy	No	RDMA Read, RDMA Write	No	Selectable



(a) PVFS read (b) PVFS write

Figure 3. Server-based RDMA Mechanism.



(a) PVFS read (b) PVFS write

Figure 4. Client-based RDMA Mechanism.

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. Flow control issues related to Send/Recv message transfer are described in section 5.1.

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 notify the servers when RDMA operations are finished. It can be seen that more control messages are usually needed in the client-based RDMA mechanism, compared to the server-based RDMA mechanism.

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 [19, 32]. 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 [18], 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.

4.4 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 the associated 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 and control messages with interrupts. On the server side, all event notification is done with polling, as is appropriate for a dedicated machine.

4.5 Transport Layer Optimizations

We consider two schemes to optimize small data transfers: Inline and Fast RDMA Write. For bulk data transfers, pipelining communication and I/O is also considered.

4.5.1 Inline Data Transfer

Zero-copy data transfers require that application buffers be registered before data transfer and may be deregistered after data transfer. For small data messages, the performance benefit of zero-copy transfer may not offset the cost of memory registration and deregistration. In *Inline data transfer* scheme, 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 server side, a copy is not usually necessary because it can process messages in place. This technique has been used elsewhere [9].

4.5.2 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* is mainly used to optimize PVFS write operations. However, it is also used to optimize PVFS read operations by avoiding application buffer registration and deregistration.

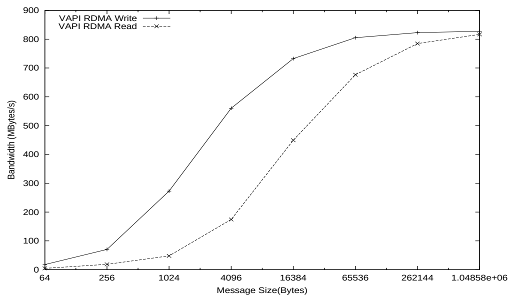


Figure 5. RDMA Read and Write Throughput.

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.

Fast RDMA Write is carried out between application buffers and the server Fast RDMA buffers directly if application buffers are already registered. Otherwise, it is carried out between the Fast RDMA buffers of both sides with cost of one memory copy on the client side. Thus, this scheme takes advantage of both higher RDMA write performance and the tradeoff between memory cost and buffer registration and deregistration costs. Fast RDMA Write operations are initiated by the client for small writes; while they are initiated by the server for small reads.

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.

4.5.3 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 in the above example), not one buffer for the entire size (100 MB).

4.6 Summary of the PVFS Transport Layer Design and Optimizations

In our design, Send/Recv is used to transfer request, control and reply messages. Server-based RDMA mechanism is used for large PVFS read and write operations, respectively. Pipelined bulk data transfer is applied to large data transfers in Server-based RDMA mechanism. Inline and Fast RDMA Write are used for small read and write operations. Communicate manager has thresholds to make appropriate choices for data message transfers.

Send operations always solicit completion on the receiver side. They also generate completion on the sender side. The last RDMA operation in a functional message generates a completion. Notification of completion of sending request messages on the client side is done using polling and notification of completion of incoming reply and control messages with interrupts. On the server side, all event notification is done with polling.

5 Designing Buffer Manager

A buffer manager provides buffers to the PVFS transport layer and the file access manager. 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. We focus on these issues below.

5.1 Flow Control on Internal Buffers

Internal buffer management is a well-discussed issue in the literature. 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 control messages. This information can also be exchanged implicitly in the flow of matched request and response message pairs.

5.2 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 according to the working set of concurrent client requests. 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* and *poll* provide mechanisms to notify the server of data arrival before data placement. In the PVFS transport layer based on InfiniBand, 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 could be different. This variability can offer better performance, while it 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 on RDMA buffers 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, as discussed in section 4.5.3. Buffers are allocated from one or more zone lists and assigned to the request. Then as parts of or the entire request is completed, 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 becomes multi-threaded and/or uses non-blocking file operations, this buffer manager policy needs to be changed accordingly.

5.3 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. Memory registration and deregistration are expensive operations. Thus, they impact performance significantly when they are performed dynamically. On the other hand, PVFS I/O applications require a large number of I/O buffers which may be allocated no earlier than when the request is issued, it is not possible to pre-register all I/O buffers. Therefore, dynamic registration and deregistration are not easily avoided.

To reduce the cost of dynamic registration and deregistration, a pin-down cache [10] 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.

However, I/O intensive applications which PVFS mainly targets use a large number of different I/O buffers. The buffer reuse ratio may be low. This poses a challenge on approaches such as pin-down cache which work well only in the case where applications keep using a moderate number of buffers. In the next section, we propose a *two-level architecture* to support efficient memory registration and deregistration for I/O intensive applications.

5.4 Fast Memory Registration and Deregistration

Dynamic buffer registration is not avoided if applications keep using different buffers. To reduce its cost, InfiniBand software and adapters are expected to provide efficient registration operation. There are some optimization on buffer deregistration in the literature. Zhou *et al.* [32] demonstrated *batched deregistration* is an efficient way to reduce the average cost of deregistering memory for database applications.

We propose a two-level architecture: *pin-down cache plus Fast Memory Registration component (termed as FMR) and Deregistration component (termed as FMD)*. We refer to this two-level architecture as Fast Memory Registration and Deregistration (*FMRD*) scheme in the rest of this paper. This architecture offers advantages from both pin-down cache and batched deregistration.

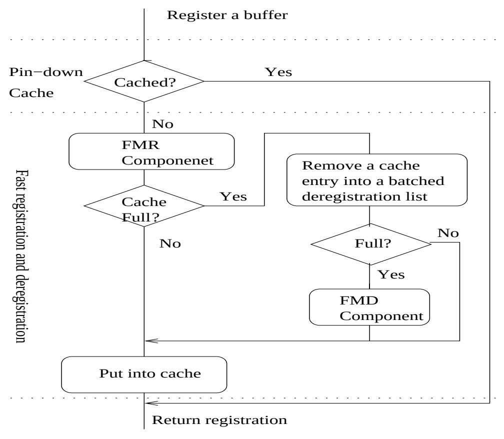


Figure 6. Fast Memory Registration and Deregistration (FMRD).

As shown in Figure 6, when a buffer is to be registered, first, it checks if its registration is cached; if yes, information is returned immediately. Otherwise, FMR is invoked to register the user buffer. The registration information is inserted into the cache. If there is no space left in the cache, one entry is evicted from the cache and put into a deregistration list. FMD is invoked to deregister all buffers in the deregistration list when the number of entries in the list reaches a threshold.

When a buffer is to be unregistered, only some information such as reference count of the buffer is modified in the cache. Real deregistration is delayed. Deregistration occurs later in a batched fashion during registration.

The fast memory registration component also takes advantage of Mellanox fast memory region registration extension in VAPI [21]. In this extension, a buffer registration is divided into two steps: 1) allocation of a fast memory region resource; 2) mapping a buffer to a fast memory region resource. Fast memory region resources can be allocated before any I/O operations. Thus the first step can be kept out of the critical path. Since multiple buffers can be mapped to the same fast memory region resource, only a moderate number of fast memory region resources are needed. The second step is in the critical path, however in VAPI it is much more efficient than the regular memory registration operations.

In FMRD, FMR component only performs the second step when registering a buffer. FMD component performs batched deregistration. In addition, they both interact with the pin-down cache. We tested FMR and FMD components in our testbed. We found that FMR component can save around 55 μs for each buffer registration. FMD component can save around 60 μs for each buffer deregistration in average. Their impact on PVFS performance is quantified in details in section 6.7.

6 Performance Results

We have implemented PVFS on our InfiniBand testbed with designs described in Sections 4 and 5. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [21], 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 [20], 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 quantify the impact of system optimizations in the transport layer and different buffer management schemes on 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. 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 of an *ext3fs* file system on the local 40 GB disk against bandwidth achieved on a memory-resident file system, using *ramfs*. The *bonnie* [11] file-system benchmark is used.

Table 2. Network performance

	Latency (μs)	Bandwidth (MB/s)
IBNice	40.1	185
VAPI Send/Recv	8.1	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

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 in spite of this disparity. Data is frequently 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 on the scalability of I/O server.

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 network. 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 of PVFS over InfiniBand VAPI. The same physical network is used, yet significant performance improvement by designing and implementing PVFS on native VAPI layer 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 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 was 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-based and disk-based file systems. On *ramfs* 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 shows CPU utilization on the compute nodes when the same program runs with four I/O servers on *ramfs*.

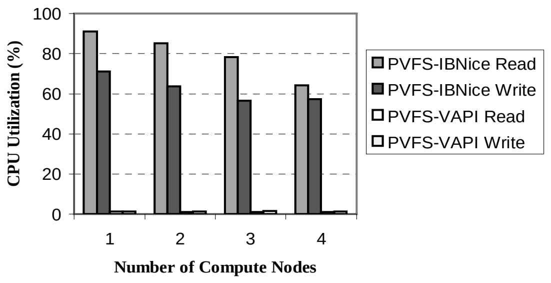


Figure 10. CPU Utilization of MPI-IO

It can be seen that the CPU overhead of compute nodes is as high as 91% in PVFS-IBNice. This is because 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 HCA 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%. This demonstrates potential for 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 4, 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* if FMRD is not used and *Server-based, 0% hit + FMRD* if FRMD is used in Figure 11.

When buffers are all cached, 100% of accesses hit in the pin-down cache. The Fast RDMA scheme offers the best performance. Since one copy is needed in the Inline scheme, the Fast RDMA scheme outperforms the Inline scheme, especially for large messages. Inline scheme performs better than the Server-based scheme for messages up to 32 kB. For messages larger than 32 kB, copying cost offsets the performance gap between Send/Recv and RDMA Read. When no buffers are cached, one copy is needed in both the Inline and Fast RDMA schemes. Since RDMA Write performs slightly better than Send/Recv in

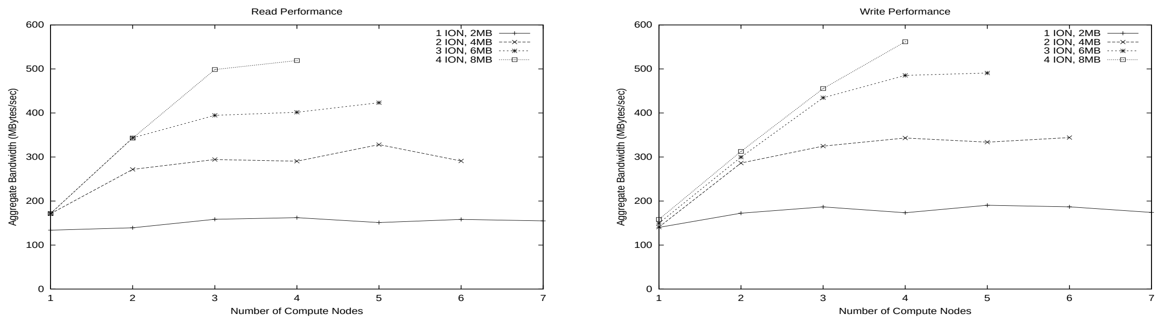


Figure 7. PVFS performance with IBNice (TCP/IP over InfiniBand).

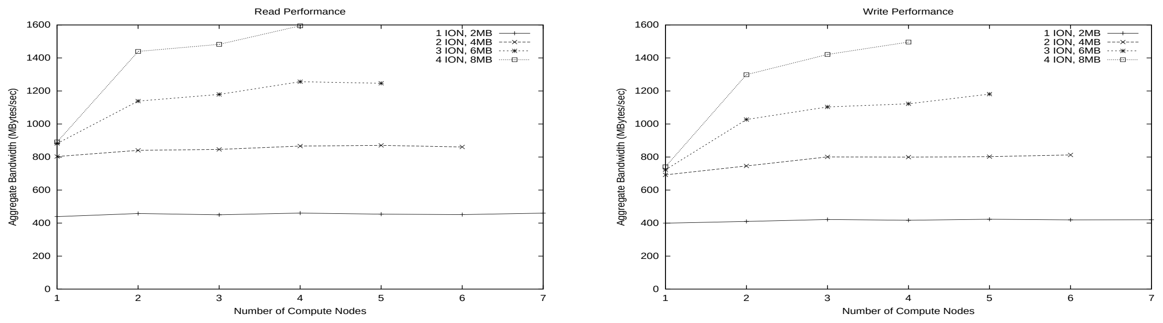


Figure 8. PVFS performance with InfiniBand VAPI

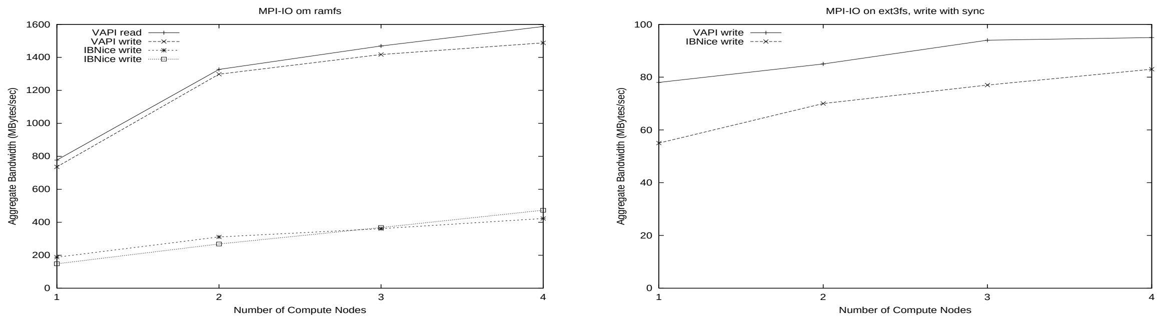


Figure 9. MPI-IO Performance

our testbed, the Fast RDMA scheme offers the best performance. As shown in the right graph of Figure 11, there is a significant performance drop in the Server-based scheme which registers user buffers without FMRD. This is due to the prohibitively costly memory registration through normal registration interface. With FMRD, the Server-based scheme performs better, but still worse than the Inline and Fast RDMA schemes. However, the gap decreases with increasing message size. This is because memory copy cost in the Inline and Fast RDMA schemes increases faster with increasing message sizes than FMRD registration cost increases. The Server-based scheme is expected to be used for large messages.

Similar results were observed for read performance, but not shown here. These results were used to decide the scheme used for messages by the communication manager. Since there is no significant performance difference between the Inline and Fast RDMA schemes for messages up to 4 kB, for simpler design complexity, the Inline scheme was used to transfer messages less than 4 kB, Fast RDMA was used 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 *ramfs*. 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 checkpoint snapshot.

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 total communication

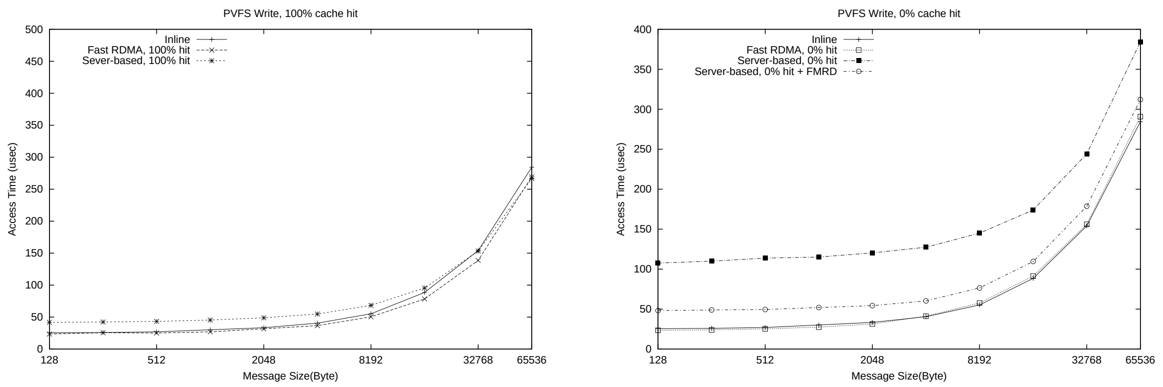


Figure 11. Small Data Transfer Optimizations on Write

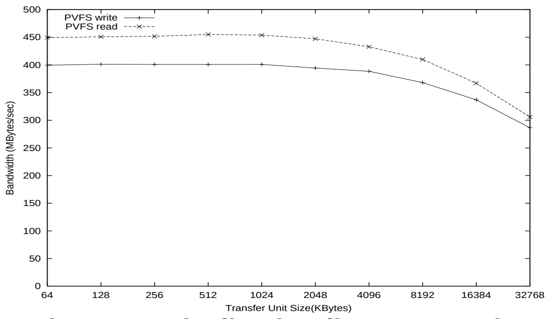


Figure 12. Pipelined Bulk Data Transfer

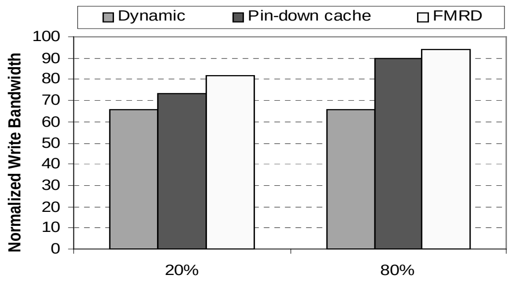


Figure 13. Effects of Memory Registration and Deregistration

startup overheads from a large number of communication operations.

6.7 Fast Memory Registration and Deregistration

We run pvfs-test program again with three different memory registration and deregistration schemes. Results are presented in Figure 13. The first one is to dynamically register and deregister I/O buffers per each I/O operation, noted as *Dynamic* in the plot. The second one is to use pin-down cache only, noted as *Pin-down cache*. The third one is to use FMRD, noted as *FMRD*. The test program performs 1000 I/O operations, in which I/O buffers are from a buffer pool with 1000 different buffers. We control pin-down cache hit ratio explicitly. We choose 20% and 80% cache hit as representatives of low buffer reuse and high buffer reuse cases, respectively. The cache size is 100, which allows us to take deregistration into account.

Figure 13 shows PVFS write bandwidth with different schemes. Note that these results are normalized to the results of the case where there is not any buffer registration and deregistration. We make three observations. First, memory registration and deregistration have a significant impact on performance. Up to 35% decrease is seen in the dynamic scheme. Second, significant improvement on performance with pin-down cache and FMRD is achieved. Particularly, if the buffer reuse ratio is 80%, pin-down cache increases bandwidth by about 24%, while FMRD increases bandwidth by about 28%. Third, FMRD works much better than pin-down cache in cases where buffer reuse ratio is low. There is about 9% improvemnt compared to pin-down cache when buffer reuse ratio is 20%.

6.8 Performance of Discontiguous File I/O

The test application *mpi-tile-io* [24] 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. 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 [7]. 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 14 shows the results for both PVFS-VAPI and PVFS-IBNice. 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%, respectively. The improvement difference between using list I/O and not using it is because the access size is larger for each pair of request and reply messages with list I/O and

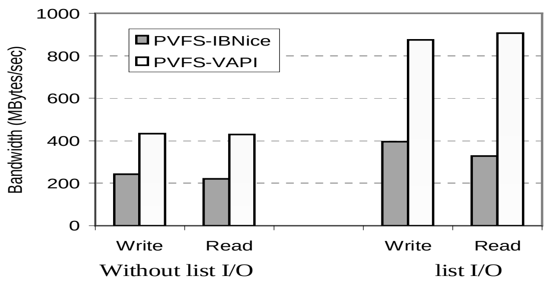


Figure 14. Performance of tiled I/O.

can yield more improvement from the VAPI layer.

7 Related Work

Various user-level communication protocols have been used for network storage in the past. Zhou *et al.* [32] 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.* [19] 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 targeted for different domains. Zahir [31] described a storage-networking transport layer for the Lustre file system based on VI-like networks. Carns [4] proposed a Buffer Message Interface (BMI) as a transport layer for the next generation PVFS. This prototype implementation works on both TCP/IP and Myrinet/GM. Liu *et al.* [14] proposed a client/server communication middleware over system area networks. It provides a communication 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 important issues in modern networks which provide RDMA capabilities. Basu *et al.* [30] show how the NIC and host-level software can collaborate to manage large amounts of host memory. Tezuka *et al.* [10] propose a pin-down cache to reduce memory registration and deregistration overhead for zero-copy communication. Zhou *et al.* [32] 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, the two-level architecture is indeed a combination of the pin-down cache and batched deregistration. In addition, we facilitate the registration procedure and reduce registration calls from applications.

Work in [1, 15] explores InfiniBand features to support efficient communication subsystems and mainly centers around Quality of Service. Our work focuses on issues in the transport layer and buffer management for I/O intensive applications.

8 Conclusions and Future Work

In this paper, we study how to leverage the emerging InfiniBand technology to improve I/O performance and scalability of cluster file systems. We designed and implemented 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 network also poses a number of challenging issues to I/O intensive applications which PVFS targets. We addressed these issues in this paper by designing: a transport layer customized for the PVFS protocol by trading transparency and generality for performance, buffer management for flow control, dynamic and fair buffer sharing, and efficient memory registration and deregistration. Inline, Fast RDMA Write, and Pipelined Bulk data transfers were designed and implemented in the transport layer. Our results show that these techniques bring significant performance gains. We also demonstrated that our proposed two-level memory registration and deregistration architecture works better than other schemes and offers efficient memory registration and deregistration in the I/O intensive environment.

Compared to a PVFS implementation over the 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.

Work is underway on designing efficient noncontiguous access in PVFS on InfiniBand and optimizing MPI-IO over PVFS. 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 working with the PVFS team to incorporate our design into the next generation PVFS and to implement it on InfiniBand.

Acknowledgments

We would like to thank the PVFS team at Argonne National Laboratory and Clemson University for giving us the access to the latest version of PVFS implementation and for providing us with crucial insights into the implementation. We are also thankful to Jiuxing Liu and Sushmitha P. Kini for many discussions with us.

References

[1] F. J. Alfaro, J. L. Sanchez, J. Duato, and C. R. Das. A Strategy to Compute the Infiniband Arbitration Tables. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[2] 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.

[3] 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.

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

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

[10] 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 *12th Int. Parallel Processing Symposium*, March 1998.

[11] http://www.textuality.com/bonnie/. Bonnie: A File System Benchmark.

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

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

[14] 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.

[15] E. J. Kim, K. H. Yum, C. Das, M. Yousif, and J. Duato. Performance enhancement techniques for infiniband architecture. In *International Symposium on High Performance Computer Architecture*, Feb. 2003.

[16] C. Lever and P. Honeyman. Linux NFS Client Write Performance. In *Proceedings of the Usenix Technical Conference, FREENIX track, Monterey*, June 2001.

[17] 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.

[18] 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.

[19] 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.

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

[21] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 0.95, March 2003.

[22] 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.

[23] T. M. Pinkston, A. F. Benner, J. Duato, M. Kruase, I. Robinson, and M. Talluri. InfiniBand: The "De Facto" Future Standard for System and Local Area Networks or Just a Scalable Replacement for PCI Buses? *Cluster Computing 6*, pages 95–103, 2003.

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

[25] R. B. Ross. Reactive Scheduling for Parallel I/O Systems. PhD dissertation. http://parlweb.parl.clemson.edu/techreports/.

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

[27] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *First USENIX Conference on File and Storage Technologies*, pages 231–244. USENIX, Jan. 2002.

[28] Storage Networking Industry Association. Shared Storage Model. www.snia.org/tech_activities/shared_storage_model.

[29] 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.

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

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

[32] 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.