COMMUNICATION AND MEMORY MANAGEMENT IN NETWORKED STORAGE SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Jiesheng Wu, B.S., M.S.

* * * * *

The Ohio State University

2004

Dissertation Committee:

Professor Dhabaleswar K. Panda, Adviser

Professor Ponnuswamy Sadayappan

Professor Srinivasan Parthasarathy

Dr. Pete Wyckoff

Approved by

Adviser

Department of Computer and Information Science

ABSTRACT

Rapid advances in processors, memory, and network technologies have led to an emerging trend of using a collection of powerful, low-cost desktops and workstations, called a *cluster system*, as a cost-effective platform for various applications. Cluster systems have been increasingly becoming a mainstream platform for both scientific applications and non-scientific applications in the recent several years. These applications have been becoming more and more I/O intensive. However, I/O is quickly emerging as the main bottleneck limiting performance in today's cluster systems.

The need for scalable and high performance I/O support is becoming more and more urgent. Network storage systems, consisting of *commodity storage units connected with commodity network technologies*, provide potentials to achieve high performance, scalability, reliability, and manageability to meet the I/O demands in cluster systems. However, networked storage systems are getting larger and more complex than ever before. The performance of networked storage systems is usually limited by communication overheads. Another source of performance limitation is the lack of integration among system subsystems and the storage server applications in the general-purpose operating system. This often results in redundant data copying, multiple buffering, and other performance degradation

This dissertation investigates communication and memory management in networked storage systems over emerging network technologies to improve I/O performance. The communication and memory management includes three main aspects: (1) communication and memory management in the transport layer of networked storage systems based on emerging networking technologies such as InfiniBand; (2) integrated memory management between communication systems and file cache and file systems; and (3) cache memory management in a multi-level cache hierarchy in networked storage systems.

The communication and memory management in the transport layer of networked storage systems over InfiniBand is designed to take advantage of InfiniBand features and make the most out of RDMA operations. With this management, networked storage systems over InfiniBand has minimal communication overheads for both contiguous and non-contiguous accesses. Both performance and scalability are improved. In addition, the storage systems are more adaptive to different access patterns. The integrated memory management between communication subsystem and file cache

and file system subsystems provides a unified memory management for both networking operations and file I/O operations. This unification enables efficient interaction and cooperation between these subsystems and eliminates redundant data copying, and multiple buffering. In addition, this management provides potentials to reduce memory registration and deregistration costs for RDMA-based networks on which the networked storage systems are built. The cache memory management in a multi-level cache hierarchy is designed to aggregate cache memory resources on both client and server sides. The Demote buffering architecture achieves exclusive caching and aggregate memory resources efficiently.

We have designed and implemented our communication and memory management in an implementation of PVFS-1 over InfiniBand. We have also incorporated the integrated cache and communication buffer management in the above PVFS-1 implementation. These designs improve the performance of PVFS-1 significantly. For instance, compared to a PVFS-1 implementation over the standard TCP/IP on the same InfiniBand network, our implementation offers 200% improvement in 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. The native non-contiguous I/O access support attains a 20% improvement compared to the best result across all other approaches in the NAS BTIO benchmark which includes complex non-contiguous I/O accesses. The integrated cache and communication buffer management improves both performance and scalability. It also increases the effective cache size due to the integration of communication buffers and the cache buffers, leading to increased performance. To demonstrate the effects of exclusive caching with Demote buffering, we have designed a simulator. Simulation results of experiments with synthetic workloads demonstrate that 1.11-1.44x speedups are achieved for the Sequential workload, up to 1.13x speedups for the Random workload. Simulation results with real-life workloads validate the benefits of DEMOTE buffering by 1.08-1.15x speedups over the existing DEMOTE approach.

To my parents, Guoqiang and Meijiao
To my wife, Xueyong
To my daughter, Alyce

ACKNOWLEDGMENTS

Working on a Ph.D. is not only an academic exercise, but also a personal journey. I have been privileged to embark on this journey surrounded with the greatest people. There are so many people to thank and so much to say. These few words below cannot fully express my gratitude, which I owe to every one of them.

I would like to thank my adviser Prof. D. K. Panda for supervising and guiding me through this journey, and for his incredible insight into both academic and nonacademic matters. I am indebted for the time and effort which he spent making me into a computer science researcher. I appreciate his friendship and patience, and the wisdom which he shared with me.

To Pete Wyckoff, for encouraging me over the years, for all the pleasant and enlightening conversations, and for his humor and friendship.

I would also like to thank the other members of my dissertation committee, Prof. P. Sadayappan and Prof. S. Parthasarathy, for their valuable comments and suggestions.

I am fortunate and honored have everyone in the Network-Based Computing Laboratory (NBCL) as friends: Darius, Jiuxing, Pavan, Weikuan, Ranjit, Sayantan, Gopal, Amith, Karthik, Savitha, Sundeep, Sushmitha, Weihang, Shuang, Lei, and Wei for their friendship and discussions on various topics, as well as valuable feedback on my papers and presentations.

Finally, I thank my family: my wife, Xueyong Bai, my parents, Guoqiang Wu and Meijiao Zhu. Without their unconditional love, support and encouragement throughout all these years, I would not have been able to accomplish all that I have.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Rapid advances in processors, memory, and network technologies have led to an emerging trend of using a collection of powerful, low-cost desktops and workstations, called a *cluster system*, as a cost-effective platform for various applications. Cluster systems have been increasingly becoming a mainstream platform for both scientific applications and non-scientific applications in the recent several years.

Applications on cluster systems have been becoming more and more I/O intensive. The promise of cluster systems cannot be realized without high performance I/O support. However, the I/O system has been becoming the main bottleneck limiting performance in today's cluster systems. While this can be in part attributed to rapid advances in other system components such as processor, memory, and network technologies, it is also true that the demands required by the diverse and challenging applications are out-pacing the rate of progress in the design of I/O systems. The need for scalable and high performance I/O support is becoming more and more urgent.

Networked storage systems, based on recent advances in both storage architectures and networking, provide potentials to achieve high performance, scalability, reliability, and manageability to meet the I/O demands in cluster systems. A large number of networked storage systems [86, 30, 17, 8, 2, 36, 21, 67, 79, 27, 70, 78] have been proposed and used as a mainstream solution in various domains, such as data-centers, high performance computing systems, and corporate computing environments. The basic idea behind networked storage systems is to consolidate storage services through networks for data sharing and manageability and to aggregate capability of storage devices for performance, scalability and reliability. Figure 1.1 shows one of typical storage architectures in cluster systems. In this example, commodity I/O servers, cluster processing nodes are both connected to commodity networks. It can provide aggregated performance and data sharing to applications. It can increase scalability and reliability. It can reduce the complexity of management because the I/O service is consolidated in the networked storage system. In addition, it can take advantage of the advent in commodity components.

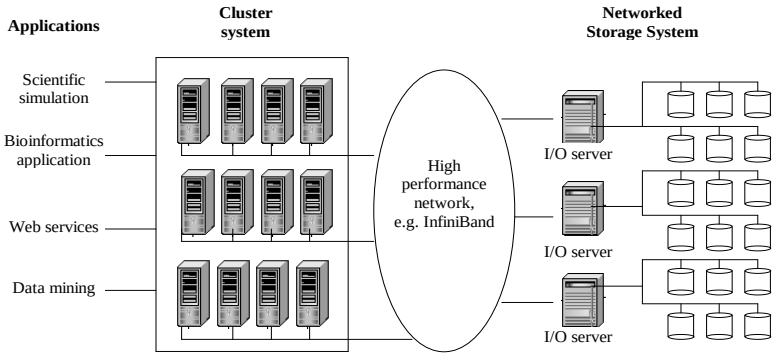


Figure 1.1: An Example of a Cluster System with a Networked Storage System.

Recent advances in both storage architectures and networking technologies have facilitated networked storage systems. Storage architectures have evolved to utilize clusters of intelligent and modular devices and commodity units in an efficient manner. Storage networking interconnects have moved to direct access to intelligent network interface cards, protocol processing offload, and Remote Direct Memory Access (RDMA) to reduce the overheads such as memory copying, network access costs, and protocol overhead in the I/O path. Often, as many developers are acutely aware of, hardware and architecture developments that purport to improve performance lack synergy with the software systems they were intended to enhance. Not surprisingly, new developments in both storage architectures and networking technologies have a profound impact on the design and implementation of networked storage software. In this dissertation, we explore the effects of these trends on the development of networked storage systems. In particular, we look at how we can take advantage of networking technologies to design cluster file systems and how these features influence communication and memory management.

The remaining parts of this chapter are organized as follows. In Section 1.1, we introduce different storage architectures. Section 1.2 presents networking technologies used in networked storage systems. Section 1.3 describes networked storage systems over InfiniBand. Problem statement is presented in Section 1.4. Section 1.5 describes approaches to the problems addressed in this dissertation. Overview of this dissertation is presented in Section 1.6.

1.1 Storage Architectures

There are four common storage architectures used currently: *Direct-Attached Storage*, *Storage-Area Networks*, *Network-Attached Storage*, and *SAN file systems*. The fifth one, a new architecture which is now under standardization, is *Object-Based storage system*. The sixth one, *cluster-based storage system*, combining both clustering

technologies and many concepts of Object-Based storage system, has been emerging as one of main solutions to the I/O support in high performance computing clusters and cluster-based data-centers.

1.1.1 Direct-Attached Storage

Direct-Attached Storage (DAS) systems are probably the most popular storage architecture in various systems such as personal computers and workstations. Block-based storage devices are connected to the I/O bus of a host machine directly via SCSI or ATA/IDE. While DAS offers minimum security concerns, it has serious limitations on connectivity, capacity and data sharing. For example, at most 16 SCSI devices/hosts can be connected to a 16-bit SCSI bus.

DAS is a common choice for applications that require high-performance, but have limited data sharing among servers. A good example is a small database or a file server.

1.1.2 Storage-Area Networks

Storage-Area networks (SANs) were introduced to overcome the limitations in the DAS architecture. SAN is a switched fabric that provides a fast and scalable interconnect to a large number of storage devices and hosts. It provides high performance and good scalability. It also enables the consolidation and sharing of storage devices.

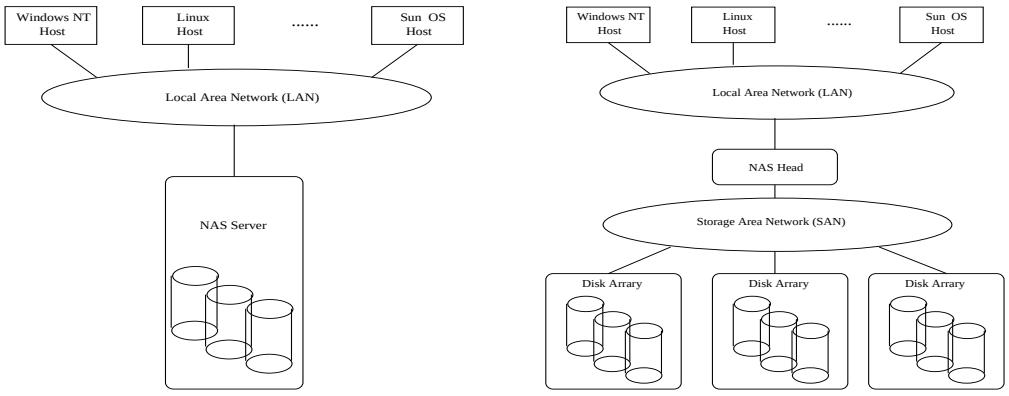
SAN provides direct block-level access to storage devices. The storage application such as file system and database maps its data structures (files, directories and tables) to blocks on the storage devices. To do this mapping, *metadata* is required to be maintained. For multiple hosts to share data blocks, they must also share metadata. Metadata consistency among the hosts must be guaranteed.

The complexity to maintain metadata consistency has resulted in block sharing only among tightly coupled storage applications such as cluster file systems and databases. In these infrastructures, mechanisms such as distributed lock management and/or global lock management are provided to achieve synchronization between hosts to share both data and metadata. For example, the Global File System (GFS) is a Shared-Disk filesystem for Fibre Channel [87]. It was implemented based on device locks (Dlock) on the storage devices. VAXcluster [52] and Frangipani [93] rely on Distributed Lock Manager (DLM) to coordinate shared access to storage devices. Lock management significantly increases the complexity of storage applications and storage devices in SANs. Thus, most other infrastructures only allow hosts to share data indirectly through files using Network-Attached Storage, as shown in Figure 1.2(b).

SAN architectures are often chosen for those applications with a need for highly scalable performance from the storage devices. A good example is a distributed database running on a cluster of workstations.

1.1.3 Network-Attached Storage

Network-Attached Storage (NAS) refers to the storage architecture which provides file access services to hosts across platforms (Operating Systems). In NAS, the metadata which describes how files are stored on devices is managed completely on the file server. This centralized metadata management enables cross-platform data sharing at the cost of directing all I/O through the single file server. NAS provides file-level access, which is a higher level abstraction than block-level access provided by SAN and DAS. Files can be stored on a SAN or with DAS. Thus, NAS may be implemented on top of a SAN or with DAS. Figure 1.2 shows two typical NAS implementation on top of DAS and a fast SAN. In Figure 1.2(a), the NAS file server accesses storage devices which are locally attached. While in Figure 1.2(b), the NAS file server is often called *NAS Header Server*, which accesses storage devices through a SAN fabric.



(a) NAS Server

(b) NAS Head Server

Figure 1.2: Typical Network-Attached Storage Systems.

With NAS, the process to provide data sharing is much simpler than that in a SAN in terms of both security and consistency concerns. However, the single file server is a potential bottleneck for both scalability and performance. Clients may rarely see the aggregate performance of the storage devices since the performance will be limited by the performance of the file server.

NAS is often chosen for applications with a need for cross-platform shared storage. Good examples are a collection of Web servers in an enterprise, accessing HTML content stored as files, or a network of workstations in a department.

1.1.4 SAN File Systems

SAN file systems have been recently introduced to address the limitations of NAS. In a SAN file system, as illustrated in Figure 1.3, clients, the file server (maybe more than one) and storage devices are all connected to a storage area network. Unlike NAS, SAN file systems allow clients to access storage devices directly. Unlike SAN, the file server can share the metadata with clients. The complexity of metadata management is reduced. Therefore, SAN file systems combine features from both SAN and NAS.

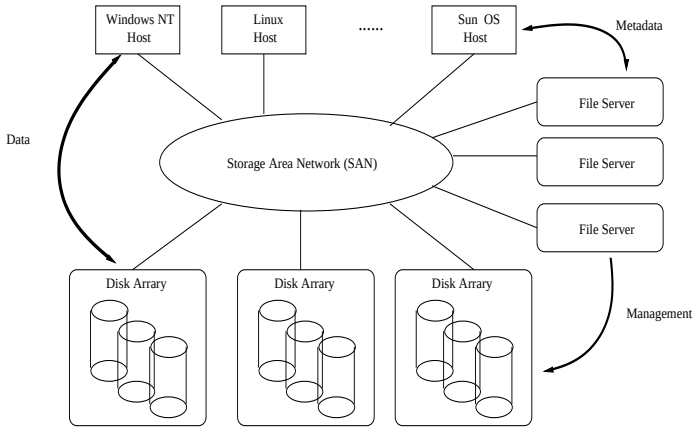


Figure 1.3: A SAN file system in which all clients, file servers, and storage devices are connected to a Storage Area Network.

However, similar to SAN, the storage devices have no mechanisms for authoring I/O access from clients. The SAN mechanism for device security only protect the entire devices, not data within the device [58]. Conventionally, this requires that clients in a SAN file system environment be trusted.

1.1.5 Object-Based Storage Systems

In summary, NAS provides a high-level abstraction that enables secure data sharing across different operating system platforms, but often at the cost of limited performance due to file server contention. SAN offers fast and scalable block-level access to shared data; but without a file server to authorize I/O and maintain the metadata, this direct access comes at the cost of limited security and data sharing. SAN file systems increase file serving performance by allowing direct client access at the cost of security. However, an ideal storage architecture would provide strong security,

high-performance, cross-platform data sharing, and high scalability. None of these architectures can meet these requirements. Recent industry and academic research have proposed an object-based storage architecture which is considered as a convergence of NAS and SANs to address the limitations in today's storage architecture.

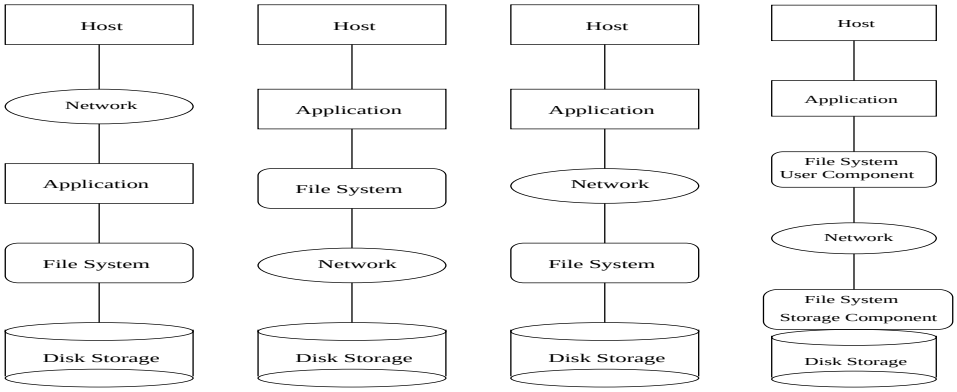
Object-based storage architecture attempts to capture both the direct access nature of SANs and the data sharing and security capability of NAS. A storage object is a logical collection of bytes on a storage device, with well-known methods for access, attributes describing characteristics of the data, and security policies that prevent unauthorized access. Storage objects are stored on *object-based storage* device (OSD). OSD can be in many forms such as disk, disk array, and tapes. The difference between an OSD and a block-based device is the interface, not the physical media.

The interface to OSDs is very similar to that of a file system. Clients can create, delete, read and write objects through this interface. This higher-level, file-like interface is exactly what enables a more cross-platform capable storage device. Yet these storage devices can still be accessed directly by the hosts, thus allowing for higher performance. OSDs treat each object individually, it is easy to enforce security policies on a per-object basis. In addition, metadata and its management are offloaded to OSDs, which removes the dependency between the metadata and storage application, making data sharing between different applications feasible. Furthermore, intelligence on the OSDs provides the potential for the storage devices to learn important characteristics of the environments in which they operate, and then adjust resource management and apply application-specific optimizations to better utilize resources.

The key change brought by object-based storage technology is a different partition of the storage system from the previous storage architectures. Figure 1.4 shows different storage system models in DAS, SANs, NAS, and OSD architectures, respectively. Compared to other architectures, a portion of storage management has been pushed down to the OSD devices in the OSD architecture.

1.1.6 Cluster-Based Storage Systems

The concept of object-based storage technology has been incorporated in many storage systems such as the next generation of IBM StorageTank [45]; Lustre cluster file system; and the second generation of Parallel Virtual File System (PVFS), with object-based storage as their basic storage interface. Since the object-based storage devices are not available yet, cluster-based storage systems such as Lustre and PVFS attempt to deploy the concept of object-based storage technology while using commodity processors, network interconnects, memory, and storage devices. These Cluster-Based Storage Systems are built upon off-the-shelf components. Figure 1.5 shows the architecture shared by both Lustre and PVFS. In this architecture, similar



(a) DAS (b) SAN (c) NAS (d) OSD

Figure 1.4: Models of Storage Systems in Different Storage Architectures.

to the OSD architecture, three entities are involved: clients, I/O servers, and managers. Also similar to the OSD architecture, the storage system has been partitioned into two main parts: the user component on the client side and the server component on the I/O server side. The managers focus on maintaining the metadata that describes how a storage system maps to the I/O server storage devices. Clients of the storage system can consult the storage manager to learn of this metadata and access control. Provided with this metadata and access capabilities, clients can directly access the I/O server storage devices for data.

The I/O servers can perform storage management on its controlled storage devices and authorize client access using their processing power. Compared to SAN file systems, the cluster-based storage systems can provide better security. Compared to OSD, the cluster-based storage systems share many same concepts. However, the cluster-based storage systems have less industry standardization and mainly focus on cluster computing and cluster-based data-center environments.

1.2 Networking Technologies

A common feature shared by NAS, SANs, SAN file system, Object-based Storage system, and cluster-based storage system, is to transfer data through networks. In this dissertation, we call them networked storage systems for simplicity. The key enabling technology for networked storage systems is storage networking technologies, because a basic requirement in these systems is to provide high performance and reliable access to large volumes of data. In this section, several important networking

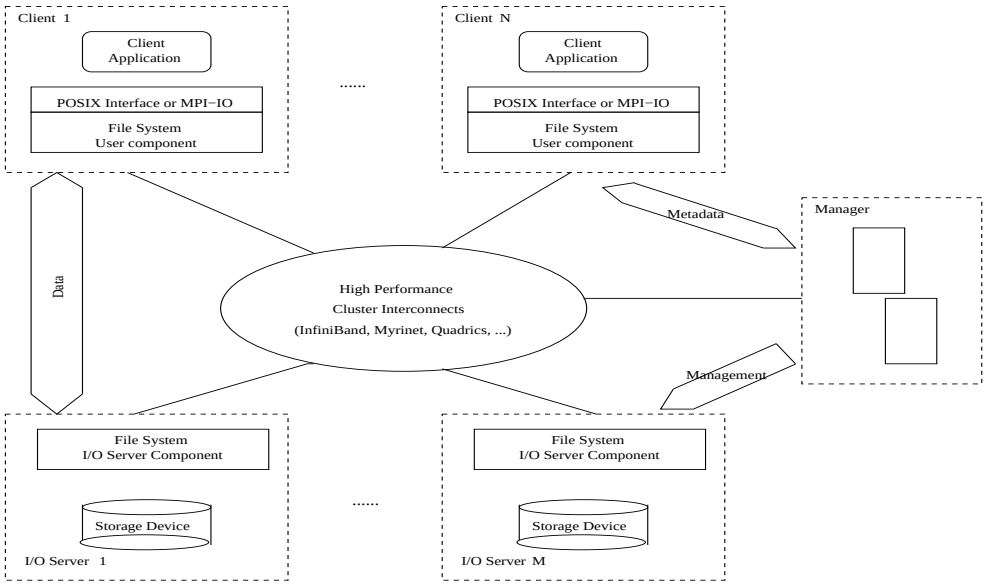


Figure 1.5: Cluster-Based Storage Systems. Clients, I/O servers, and managers are connected to the same high performance cluster interconnects.

technologies are introduced, including Gigabit Ethernet, Fibre Channel, 10-Gigabit Ethernet, InfiniBand, and RNIC.

1.2.1 Gigabit Ethernet

Gigabit Ethernet (GigE) [81] is the third generation of the IEEE 802.3 Ethernet standard after Ethernet (10Mbps) and Fast Ethernet (100Mbps). This standard, IEEE 802.3ab for CAT5 interconnect and 802.3z for optical fibre, increases the transmission rate from 100Mbps of Fast Ethernet to 1000Mbps. The networking community would like to use GigE as an option for storage networking solutions because GigE has a large install base, which reduces cost in both hardware and training. Gigabit Ethernet is purely a networking technology. It uses IP routing as its base and relies on upper-level protocols such as TCP for flow control and error detection/correction.

Both NAS and SANs based on Gigabit Ethernet technology have been widely used. The most representatives are NFS and iSCSI over Gigabit Ethernet. GigE-based networked storage systems can take advantage of the widely installed IP infrastructure. They offer much lower prices and management cost compared to other technologies such as Fibre Channel. The downside to utilizing TCP/IP on GigE networks for storage traffic is that TCP/IP was not designed for the purpose of networked storage systems. It does not define classes of services. It relies heavily on software layers to provide reliable access and interface with storage protocols such as SCSI. The

overhead of protocol processing and other related software components limits the performance.

1.2.2 Fibre Channel

Fibre Channel (FC) [114, 49] is a serial interconnection technology which provides switched point-to-point connections between two communicating devices. FC is a full-duplex, block-oriented serial networking protocol with most of the protocol handling done in hardware. Fibre Channel was designed as a storage networking technology. The Fibre Channel standard defines protocol layers all the way up to the application interface. It is a combination of a networking technology and a protocol definition. The protocol is implemented in hardware and has very little dependence on software processing. This keeps the data processing and turn-around overhead to a minimum. The hardware support includes error detection/correction, clock recovery, flow control and the addition of control frames. The SCSI protocol is supported directly as an upper-level protocol in Fibre Channel.

The majority of all Fibre Channel devices installed in the market today operate at either 1Gbps or 2Gbps or 4Gbps. 10Gbps is arriving on the market now. Fibre Channel has dominated the storage area network market for several years. However, with the advent of Gigabit Ethernet technology, Fibre Channel cost and complexity are finally becoming apparent. Furthermore, compared to the emerging networking technologies such as InfiniBand, the flexibility, the support of Quality of Service, Reliability, Availability, and Serviceability (RAS), and the performance in Fibre Channel lag behind the requirements of today's storage systems.

1.2.3 10 Gigabit Ethernet

10 Gigabit Ethernet [37] is the next development of Ethernet technologies after Ethernet (10Mbps), Fast Ethernet (100Mbps), and Gigabit Ethernet (1Gbps). 10 Gigabit Ethernet is a full-duplex only and fiber-only technology, it does not need the carrier-sensing multiple-access with collision detection (CSMA/CD) protocol that defines slower, half-duplex Ethernet technologies. In every other respect, 10 Gigabit Ethernet remains true to the original Ethernet model. 10 Gigabit Ethernet ensures interoperability not only with previous versions of Ethernet but also with other networking technologies such as SONET (synchronous optical network) OC-192c or SDH (synchronous digital hierarchy) VC-4-64c infrastructures. Therefore, it can be used in Local Area Networks (LANs), Metropolitan Area Networks (MAN), and Wide Area Networks (WAN).

10 Gigabit Ethernet will provide infrastructure for both network-attached storage (NAS) and storage area networks (SAN). There are numerous applications for Gigabit Ethernet in storage networks today, which will seamlessly extend to 10 Gigabit Ethernet as it becomes available. The versatility preserved by 10 Gigabit Ethernet is

a big advantage over other networking technologies. However, the performance of 10 Gigabit Ethernet might not be as good as other networking technologies which are designed for storage and high performance computing environments. For example, the first 10 Gigabit Ethernet adapter, Intel PRO/10Gbe LR server adapter can deliver 2.5 – 7.2Gbps under different configurations [44, 101]. This adapter focuses on offloading certain tasks from hosts such as TCP and IP checksums and TCP segmentation to reduce the host CPU overhead rather than on *Remote Direct Memory Access (RDMA)* and source routing used in high performance cluster interconnects such as Myrinet [64] and Quadrics [68]. This design choice represents the design philosophy along Ethernet technologies that values versatility over performance. However, as mentioned in [101], Feng *et al* believe that an OS-bypass protocol, like RDMA over IP [73], implemented over 10GbE would result in better throughput and latency and lower CPU overhead. The OS-bypass protocols require an on-board network processor on the adapter.

1.2.4 InfiniBand

The InfiniBand Architecture (IBA) [46] 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.

A queue-based transport layer is provided in IBA. A Queue Pair (QP) consists of two queues: a send queue and a receive queue. The completion of requests is reported through Completion Queues (CQs). Both channel and memory semantics are supported in the IBA transport layer. 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.

The InfiniBand Architecture is designed to be a transport service independent of protocol. It implements the layer 4-transport service in hardware. Its ability to use a single communication technology to move raw data for different domains such as storage, inter-processor communication, audio/video is superior to Ethernet technologies and Fibre Channel. Specifically, InfiniBand provides more flexibility than Fibre Channel. It defines mechanisms and management to support Quality of Service and RAS required by many storage systems and data-centers. It incorporates user-level networking technologies and RDMA to provide high bandwidth and low latency communication with minimal CPU overhead. For example, the currently available InfiniBand adapter can provide bandwidth of 890 MBytes/s and latency of 4 μs with a CPU overhead approaching 2%.

1.2.5 RDMA enabled NIC (RNIC)

Demand for networking bandwidth and increase in network speeds are growing faster than the processing power and memory bandwidth of the compute nodes that ultimately must process the networking traffic. This is especially true as the industry begins migrating to 10Gigabit Ethernet infrastructures. Therefore, reducing protocol overhead and eliminating intermediate memory buffering and copying become more important. RDMA over TCP/IP [71] addresses these issues in two very important ways: first, much of the overhead of protocol processing can be moved to the Ethernet adapter and second, each incoming network packet has enough information to allow its data payload to be placed directly into the correct destination memory location, even when packets arrive out of order. The direct data placement property of RDMA eliminates intermediate memory buffering and copying and the associated demands on the memory and processor resources of the compute nodes, without requiring the addition of expensive buffer memory on the Ethernet adapter. Additionally, RDMA over TCP/IP uses the existing IP/Ethernet based network infrastructure.

To support RDMA over TCP/IP, RNIC – RDMA enabled NIC (Network Interface Controller) – was introduced. The RNIC provides support for the RDMA over TCP protocol suite and can include a combination of TCP offload and RDMA functions in the same network adapter.

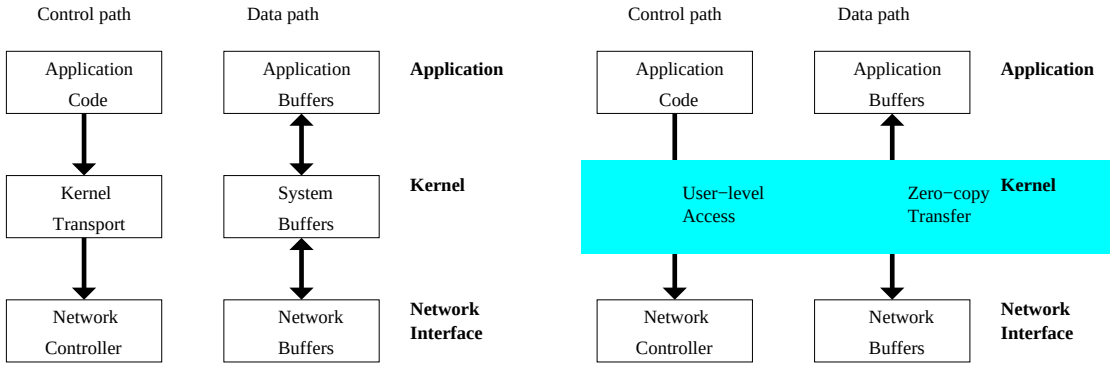
1.2.6 Existing state-of-the-art Approaches

Three important approaches have been emerged for high performance communication. They are *user-level networking*, *zero-copy transfer*, and *Remote Direct Memory Access (RDMA)*. The first two approaches have been widely used and recognized as essential for reducing communication overheads. Both of these approaches move the network interface closer to the application by removing the host processor and operating system from the communication critical path. The third approach, RDMA, has becoming more and more attractive and has been already incorporated in several popular interconnects such as Fibre Channel [78], InfiniBand [46], and RNIC. Other interconnects, such as Myrinet [13] and Quadrics [68] support RDMA as well, though they mainly aim for high performance inter-processor communication in cluster systems.

These advances are outcomes of a proliferate research work and industry practices in the last decade, including Active Message [100], Fast Message [66], VMMC [12], Unet [99, 103], Virtual Interface (VI) Architecture [28], and Memory Channel [39].

User-level Networking and Zero-copy Transfer

Figure 1.6 compares the traditional kernel-based transport stack and the emerging transport stack based on user-level networking and zero-copy transfer. In the traditional approach, an application calls a system call to invoke the operating system kernel for sending data. The kernel transfers the application data into the network interface through multiple data copies. It first copies the data from the application buffer into the system buffer and then copies data from a system buffer into the network interface. Finally, the kernel initiates the network controller to send out the message onto the network. On the receiver side, the network interface interrupts the kernel to transfer the data from the network interface into a system buffer when a message arrives from the network. The kernel determines the location of the data and transfers it to the final destination in application memory. It then schedules the blocked application that was waiting for the data to arrive. In both send and receive sides, the kernel may need to copy the data several times among the system buffers as the data goes through different network software layers.



(a) Traditional Transport

(b) User-level Networking and Zero-copy Transfer

Figure 1.6: RDMA operations.

User-level networking and zero-copy data transfer bypass the kernel in the data path. They also bypass the kernel in the control path in many cases.

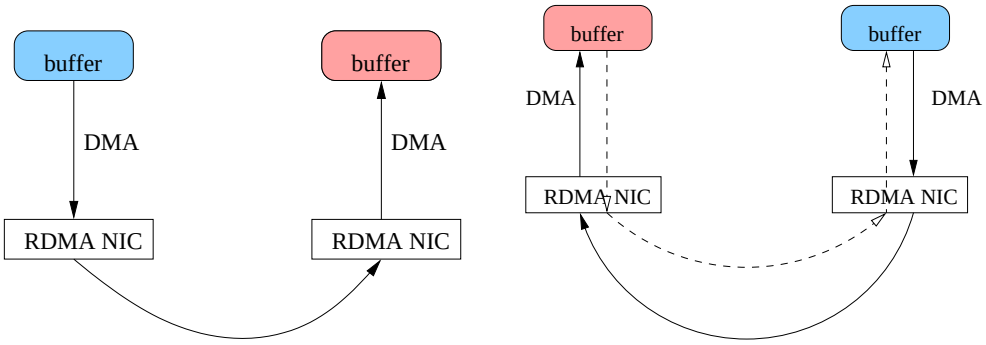
User-level networking allows an application to bypass the operating system and directly communicate with the network interface controller. Appropriate protection is provided to achieve safe user-level access. Protected user-level access improves

latency, especially for small messages. User-level networking is thus essential to minimize communication latency.

Zero-copy transfer allows a network interface to transfer data directly between the application buffers and the network interface using *Direct Memory Access (DMA)*. This technique has three advantages. First, DMA transfers are very efficient and can deliver the bandwidth close to the maximum of the connecting I/O bus. Second, the CPU is free of data transfers. Third, memory copying between the application buffer and the system buffers, or between the system buffers and network interfaces is eliminated. Copy operations are expensive, they pollute the system cache, they significantly increase the memory bus bandwidth by handling the data several times, and they are far less efficient than DMA transfers. Zero-copy transfer is thus essential to maximize communication throughput.

Remote Direct Memory Access (RDMA)

RDMA provides a new communication model – memory semantics model. In this model, an application can directly write to, and read from, the memory of a remote application, by specifying both local and remote addresses in a RDMA operation as shown in Figure 1.7. The data transfers are transparent to the remote host CPU; the data is simply copied into, or out of, the remote host memory through zero-copy DMA transfers.



(a) RDMA Write (b) RDMA Read

Figure 1.7: RDMA operations.

Compared to the send/receive communication model in traditional network systems, the RDMA model has three key advantages. The first advantage is that RDMA provides a separation between data and control flow. RDMA allows an application

to transfer data without transferring control. This separation can substantially improve the performance of network operations by allowing an application to transfer data without interrupting or involving the remote host processor [92]. In contrast, the send/receive semantics typically require that the remote receiver post a blocked receive operation. The arrival of the message interrupts the remote host to match the message with the receive operation and schedule the blocked application. This process is applied to both data and control messages. The second advantage of RDMA is that it reduces the complexity of flow control because the initiator of RDMA operations always specifies the destination address in which the data should be placed. The network interface can always deliver every incoming message to its predetermined location. Therefore, there is no need to buffer data which results in complicated flow control as demonstrated in the transport layers such as TCP/IP based on the send/receive semantics.

1.3 Networked Storage Systems over InfiniBand Networks

Access to networked storage systems is a key requirement in many domains such as data-centers and high performance computing environment to achieve scalability, reliability and low total cost of ownership. However, realizing these benefits requires efficient and manageable storage connectivity: High performance with minimal host CPU overhead; Reliable and redundant access from end to end; Easy and modular scalability; Security and data protection; and Comprehensive management and virtualization. The specification of InfiniBand was defined with all these requirements in mind since the beginning. It can be expected that networked storage systems over InfiniBand networks will be an important design choice in storage systems.

InfiniBand can be used in different storage architectures. NAS systems such as NFS and DAFS can be designed and implemented over InfiniBand networks. SAN systems such as iSCSI and SRP can also run over InfiniBand networks. Cluster-based storage systems can use InfiniBand as well.

There are significant differences in these networked storage system architectures with respect to protocols, interconnects, and storage devices. For example, the interconnect can be Fibre Channel [78], Ethernet or InfiniBand [46]. The wire protocol can be FCP [78], TCP/IP or DAPL [32]. The access method can be block-level or file/record-level. In reality, a combination of the appropriate choices may be the best response. Figure 1.1 shows one typical setup with such combination. In this setup, a storage area network is used to achieve high performance data transfers between clients and storage servers, while the file-level access protocol is exposed to clients for simplified data access, management and sharing. A simplified yet representative software architecture is illustrated in Figure 1.8.

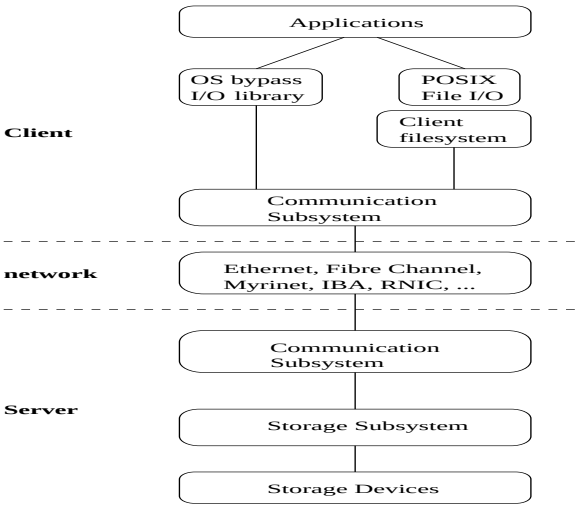


Figure 1.8: Software Architecture of Networked Storage Systems.

1.4 Problem Statement

Driven by rapidly increasing requirements, networked storage systems are getting larger and more complex than ever before. We identify the following issues as our main focuses in this dissertation: *Designing the communication subsystem for high speed, low overhead data movement over RDMA networks*; *Efficient support for variable I/O access patterns*; *Mechanisms to provide efficient integration and cooperation among different components*; and *Efficient exclusive caching*.

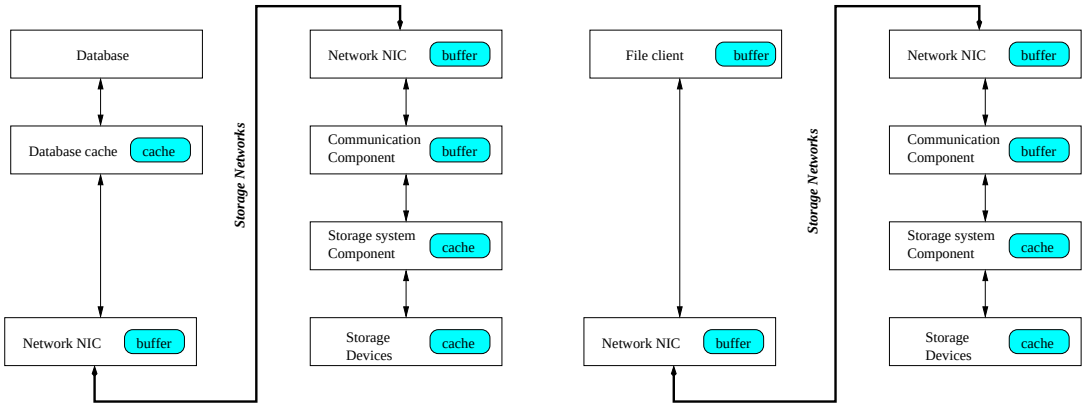
1. **High speed, low overhead data movement with RDMA** — As shown in Figure 1.8, the communication subsystem is a key component in networked storage systems. Performance of networked storage systems is often limited by the low performance of the network subsystem. This is mainly due to costs of memory copying, network access, interrupt, and protocol processing in the network subsystem [3, 74]. The advent of networking technologies and high performance transport protocols facilitates the service of storage over networks. Network architectures such as *Virtual Interface (VI)* Architecture [28], *InfiniBand* Architecture [46](IBA), *Myrinet* [64], *Quadrics* [68], and *RDMA enabled NIC (RNIC)* [71] provide several key features, namely *user-level networking*, *remote direct memory access* (RDMA), and protocol offloading to offer low latency, high throughput, and low host processing costs of network I/O. Some of these networks have been used widely in scientific applications and have been also taking into data centers and cluster systems for block storage, file system access, and transaction processing.

These enabling technologies eliminate or reduce costs of memory copy, network access, interrupt, and protocol processing in the network subsystem. However, there are a number of challenges to be addressed [115, 54, 33]. First, networks such as InfiniBand provides a rich set of features such as RDMA Write and Read, Gather/Scatter, and various completion mechanisms. An interesting question is how we can take full advantage of these features to transfer different messages such as small data, bulk data, and control messages efficiently. Second, there are some new issues associated with RDMA and user-level networking. One of the most significant issues is memory registration and deregistration costs. Reducing these costs is a challenging issue.

2. **Variable I/O access patterns with Non-Contiguous Access** — Many research studies on I/O characteristics [9, 97, 51, 85] of applications show that applications exercise file/storage systems in very different ways, sometimes even in opposite ways. Therefore, they have different I/O requirements. Subsystems and service components in a networked storage system should cater for different I/O requirements efficiently. For example, many applications [25, 75] perform non-contiguous I/O access. Conventional I/O systems usually perform multiple simple accesses to achieve a non-contiguous access. This leads to performance degradation [75, 110]. In this example, both the communication subsystem and the storage subsystem are required to adjust themselves and offer equally high performance of simple and non-contiguous access.

3. **Efficient integration and cooperation among subsystems** — The complexity of networked storage systems mainly comes from the involvement of several important and complex subsystems. This can be observed from Figure 1.8. Intensive interaction and cooperation occur between the file system cache, the communication subsystem, the storage subsystem, and the system disk subsystem. These interactions and cooperation can be further reflected by the data flow in an I/O path. Figure 1.9 shows two typical I/O paths in the networked storage systems, one for a database storage server and another for a direct access file server.

   The *lack of integration and cooperation among system components and the storage server applications in the general-purpose operating system* also limits performance in networked storage systems [35, 65, 6]. This often results in redundant data copying, multiple buffering, and other performance degradation [65]. Redundant memory copying leads to high CPU overhead and limited server throughput. In today's networks which provides comparable performance to the memory system, memory copy becomes bottleneck. In addition, multiple buffering of data wastes memory. Consequently, the effective size of cache space is reduced, increasing cache miss rates and disk accesses. The narrow



(a) A Database Server Example

(b) A Direct Access File Server Example

Figure 1.9: I/O Path In Networked Storage Systems.

interface [35, 6, 5, 38, 83] between these subsystems has been identified as one of reasons for this problem.

4. **Efficient exclusive caching** — The gap between processors and disks and the gap between memory access and disk access have been widened recently. With the decreasing memory price, modern file and storage servers in networked storage systems typically have large caches up to several or even tens of gigabytes to speed up I/O accesses [107]. In addition, the clients also devote a large amount of memory for caching [115, 4, 61, 62]. Multiple clients may share file and storage resources through various storage networks. Therefore, a multi-level cache architecture is formed. Caching is designed to shorten access paths for frequently referenced items, and so improve the performance of the overall file and storage systems. However, most cache placement and replacement policies used in multi-level cache systems maintain the *inclusion property*: any block in an upper level buffer cache is also in a lower level cache. The inclusive cache architecture is desirable when the upper level buffer cache is significantly smaller than the lower level one. However, as discussed earlier, the cache size of each level in a modern storage and cluster file system becomes quite close. To still maintain the inclusion property will prevent us from the benefits of the aggregate cache size of the multi-level cache hierarchy and in some cases even hurt performance.

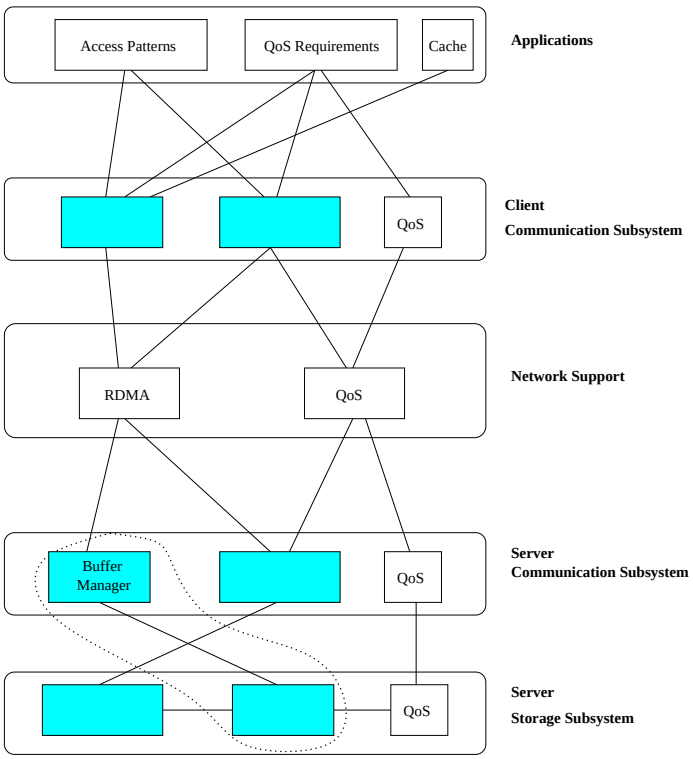


Figure 1.10: Our Proposed Software Architecture.

To aggregate the cache size of the multi-level cache hierarchy and to achieve *exclusive caching*, exclusive cache placement and replacement policies are expected.

We describe our approach to these problems in the next section.

1.5 Research Approach

To solve the aforementioned four issues, the following software architecture has been proposed, as shown in Figure 1.10. In this architecture, we focus on efficient communication and memory management in network, storage, and device subsystems. This management not only enables each subsystem to perform efficiently in an individual manner, but also provides better integration and cooperation among them. The related service components which we have focused on are highlighted in Figure 1.10, including the *buffer manager* and the *communication manager* in the communication subsystem, the *cache manager* and the *request scheduler* in the server storage subsystem. We target the following research directions:

1. **Efficient contiguous data movement over RDMA networks:** We explore various mechanisms for contiguous data transmission using RDMA operations. There are two main dimensions. One is whether RDMA operations are performed by the client side or the server side. Another dimension is which mechanism is more efficient according to the message characteristics. Our work focuses on managing possible communication mechanisms and choosing an appropriate one intelligently and automatically given a contiguous I/O access in the *communication manager* component on both sides.

2. **Efficient non-contiguous data movement over RDMA networks:** Non-contiguous data movement in networked storage systems comes from either applications or the internal implementations of the I/O system. Figure 1.11 shows these two main sources. Non-contiguous data movement poses challenges on the communication subsystem. Our work focuses on using RDMA Gather/Scatter support to provide efficient non-contiguous data movement. In addition, we design the *communication manager* to manage possible communication mechanisms and choose an appropriate one intelligently and automatically given a non-contiguous I/O access.

3. **Efficient memory registration and deregistration:** Memory registration and deregistration for networks with RDMA capabilities adds a new dimension to data movement for I/O intensive applications in networked storage systems. We propose approaches to achieve efficient memory registration and deregistration on both contiguous and noncontiguous memory regions. We also work on approaches to enable efficient memory registration and deregistration sharing among multiple networks. With these approaches and schemes, networked storage systems can take full advantage of RDMA-based data movement mechanisms. In our proposed software architecture, the *buffer manager* is responsible for efficient memory registration and deregistration on a single buffer and a list of buffers.

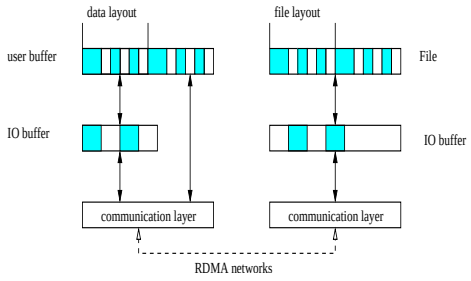


Figure 1.11: Non-contiguous Access in Networked Storage Systems.

4. **Unified cache management and buffer management:** As shown in Figures 1.8 and 1.10, the cache manager and the buffer manager are separated and commonly designed in two different subsystems: the communication subsystem and the storage subsystem. This design provides simple interaction and high level abstraction between these two subsystems. Thus, it has been accepted by many conventional networked storage systems. However, this design results in redundant data copying, multiple buffering, and other performance degradation [65]. Memory copying and multiple buffering can be observed in Figure 1.9. The communication subsystem uses its own communication buffer to read data from or write data into the storage cache. We propose a unified cache management and communication buffer management, as shown by the dotted circle in Figure 1.10. This unification provides better integration and cooperation between the communication subsystem and the storage subsystem. Our main goal is to offer a single copy data sharing among these two subsystems safely and concurrently. Thus, zero-copy I/O serving is achieved on the server side.

5. **Efficient exclusive caching with DEMOTE buffering:** Exclusive caching with DEMOTE buffering [107, 60] is another direction to achieve better cooperation between different components. The exclusive caching system is designed to aggregate the client cache and the server cache. A DEMOTE operation [107] is introduced to send the evicted blocks from the client cache to the server cache. With appropriate cache management policies, an aggregate cache between the client cache and the server cache can be achieved. We proposed a buffering scheme to mask the overhead of DEMOTE effectively.

6. **Efficient interaction and integration among storage server application components and system components:** As shown in Figure 1.10, communication, storage, and device subsystems attempt to manage their resources such as cache space, network bandwidth and disk arm to maximize the system performance. Better interaction and cooperation between them are expected to make better use of these limited resources. We have proposed *InfoCache*, a cache-centric architecture, to manage memory resources in a more aggregated and cooperative manner.

   We carefully design expressive interfaces between these components. Therefore, the internal state information of a component can be exposed to its related components. For each component, there are two types of information, one for the upper layers, and another for the lower layers. These information can be used to reduce memory copying and to implement efficient scheduling, prefetching, and other semantic policies.

1.6 Dissertation Overview

In this dissertation, we describe how we improve performance of networked storage systems through efficient communication and memory management. We present the basic design, implementation, and evaluation of a cluster file system, Parallel Virtual File System (PVFS), over InfiniBand in Chapter 2. Chapter 3 presents how to support efficient non-contiguous I/O access in PVFS over InfiniBand. In Chapter 4, several techniques are presented to reduce memory registration and deregistration overheads in I/O path for PVFS and Direct Access File System (DAFS). Chapter 5 presents an integrated communication buffer and cache memory management. Chapter 6 describes efficient exclusive caching through Demote Buffering.

In chapter 2, we describe our design, implementation, and performance evaluation of PVFS over InfiniBand. Our research focuses on taking advantage of InfiniBand features to achieve high performance contiguous data movement between I/O servers and clients with minimal CPU overhead. In particular, we have designed a transport layer customized for PVFS by trading transparency and generality for performance; we have provided communication management to choose appropriate transfer mechanisms for optimal performance; we also have designed schemes of flow control, dynamic and fair buffer sharing to better utilize memory resources.

In chapter 3, we present the design, implementation and performance evaluation of non-contiguous I/O support in PVFS over InfiniBand. Two main performance issues have been addressed: non-contiguous data movement and non-contiguous disk access. We have designed several schemes to provide efficient non-contiguous data movement. In addition, communication management is provided to choose an appropriate scheme according to the characteristics of non-contiguous data access. We have also proposed an *Active Data Sieving* idea to achieve efficient non-contiguous disk access.

Chapter 4 presents solutions to achieve efficient memory registration and deregistration. Memory registration and deregistration overheads are costly. Without efficient memory registration and deregistration, the benefits of RDMA operations are elusive. We have designed a *two-level memory registration and deregistration* architecture to reduce the costs of memory registration and deregistration on a contiguous buffer. We have developed a scheme, *Optimistic Group Registration*, to provide efficient memory registration and deregistration on a list of buffers. These two solutions are complementary to our work in Chapters 2 and 3 and work together to make the most out of RDMA operations over InfiniBand networks. In the context of DAFS, we have extended the DAFS interface to enable applications to avoid the memory registration and deregistration in some cases.

Chapter 5 describes an integrated cache and communication buffer management. We have designed a general cache component for networked storage applications. The expressive interface of this cache component enables efficient interaction with other components. In particular, we have used this component to achieve unified cache and

communication buffer management which enables the communication subsystem to use the cache buffer directly in a safe and concurrent manner. Consequently, zero-copy I/O serving is achieved by eliminating redundant data copying and multiple buffering in the I/O path.

We describe a DEMOTE buffering architecture to achieve efficient exclusive caching in Chapter 6. DEMOTE buffering is designed to mask the cost of DEMOTE operation. It also provides more flexibility for optimizations, such as non-blocking operations, aggregate of control messages, gather/scatter network operations, and speculating demotions.

In Chapter 7, we present our conclusions and describe topics for further research.

CHAPTER 2

DATA MOVEMENT MECHANISMS USING RDMA

High processing overhead associated with conventional network I/O becomes a bottleneck to the networked storage systems. This is due to two main reasons. First, a large volume of data is transferred between a storage server and its possible clients. Second, checksumming, memory copying, data movement, and interrupt in conventional network I/O incur high host processing overhead [22, 26, 74]. The costs of checksumming, memory copying and data movement are usually proportional to the size of data and are often called *per-byte overhead*. Costs of interrupt and other protocol processing on a packet basis are often called *per-packet overhead*.

In today's network interface cards (NICs), the checksum is commonly offloaded into the computational power units in the NICs. This offload removes the other source of the per-byte overhead [74]. The interrupt coalesce provided by these modern NICs also reduces the per-packet costs. Thus, when the other overheads are eliminated or reduced, the memory-copying overhead accounts for a larger part in network I/O.

On the other hand, an increasing "processor-memory performance gap" has been observed in a number of studies. Hennessy *et al.* [42] show that the CPU performance grew from 1980-1998 at 60% per year, while the access time to DRAM improved at 10% per year. Recently, with the advent of networking technologies, network bandwidth has been improved at 40% per year in the last decade. Memory bandwidth becomes behind the network bandwidth.

As mentioned in Section 1.2.6 in Chapter 1, RDMA provides zero-copy data transfer between application buffers of two sides. This feature is critical to achieve high bandwidth data transfer for large volume of data in networked storage systems. In addition, RDMA operations transfer data without interrupting or involving the remote host processor. This reduces CPU overheads of the remote host processor. This leaves more CPU cycles for other useful computation. Consequently, better performance and scalability can be achieved.

In this chapter, we focus on how to take advantage of RDMA benefits in networked storage systems. Our work is based on InfiniBand which is capable of RDMA operations and designed for both I/O and inter-processor communication. The rest

of this chapter is organized as follows. First, we introduce RDMA operations in InfiniBand. Second, we discuss and compare data movement mechanisms using RDMA for I/O operations in networked storage systems. Lastly, we design and implement PVFS over InfiniBand. We use this design and implementation to explore the design issues of networked storage systems over RDMA and the impact of RDMA on the performance of networked storage systems.

2.1 RDMA Operations over InfiniBand

The InfiniBand Architecture (IBA) [46] 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 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.

The most popular InfiniBand hardware available is 4x, which provides a full-duplex 1 GBytes/second. The end-to-end bandwidth experienced by applications is up to 825 MBytes/second. The end-to-end latency is as low as 6.0 microseconds. Figure 2.1 shows the throughput of RDMA Read and Write operations in one of InfiniBand networks from Mellanox [55].

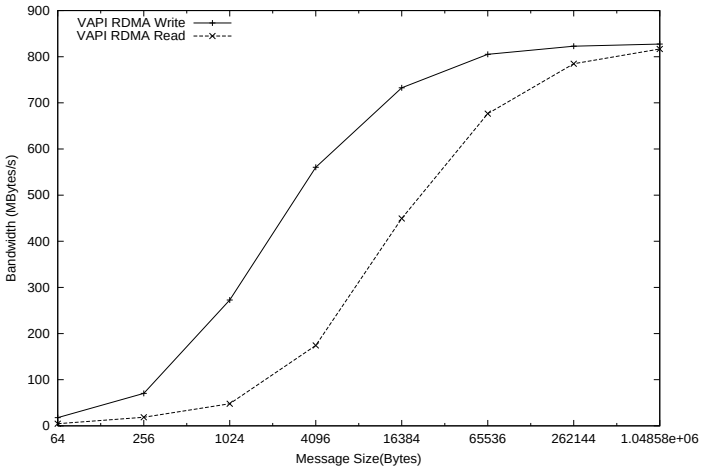


Figure 2.1: RDMA Read and Write Throughput on an InfiniBand network.

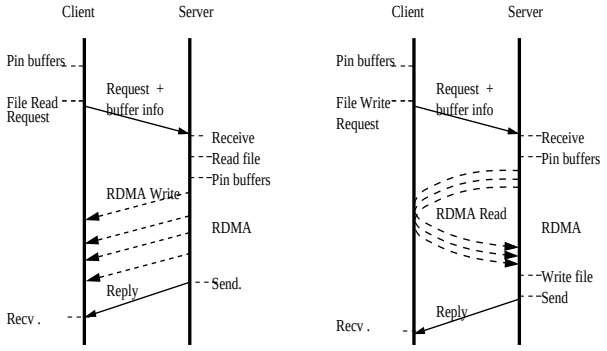
2.2 Contiguous Data Movement using RDMA

In networked storage systems, there are two basic operations read and write. In read operation, data is moved from the storage server to the client. In write operation, data is moved from the client to the server. These data movements can be mapped to RDMA Read and Write operations in different manners. In this subsection, we discuss three possible mechanisms.

Data may come from a contiguous buffer or several non-contiguous buffers. Data may be also placed in a contiguous buffer or several non-contiguous buffers. In this section, we focus on the case in which data comes from a single buffer and is placed in another single buffer. We call this data movement contiguous data movement, otherwise noncontiguous data movement. In the next chapter, we present the mechanisms of non-contiguous data movement.

2.2.1 Server-Based RDMA Mechanism

In server-based RDMA mechanism, both read and write operations are mapped to RDMA operations initiated only by the I/O servers. The clients are responsible for providing RDMA buffer information. Figures 2.2(a) and 2.2(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.



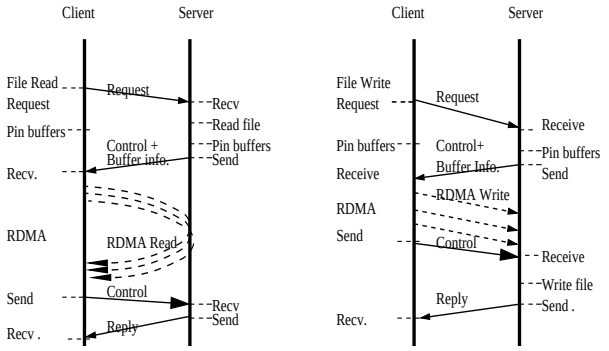
(a) read (b) write

Figure 2.2: Server-based RDMA Mechanism.

2.2.2 Client-Based RDMA Mechanism

In client-based RDMA mechanism, both read and write operations are mapped to RDMA operations initiated only by the client. Figures 2.3(a) and 2.3(b) show the operations involved to perform reads and writes, respectively.

Generally speaking, the client-based RDMA mechanism requires the server to send a control message containing its RDMA buffer information before data transfer can begin. It also requires that the client notifies the servers when RDMA operations are finished. This usually incurs more control traffic, compared to the server-based RDMA mechanism.



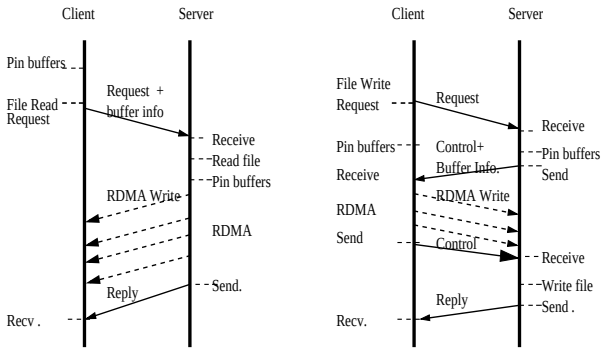
(a) read (b) write

Figure 2.3: Client-based RDMA Mechanism.

2.2.3 Hybrid RDMA Mechanism

Both Server-based and Client-based RDMA mechanisms use RDMA read operation for either write or read. Since RDMA read is a round-trip operation, its performance is constantly lower than that of RDMA Write. For example, Figure 2.1 compares their performance in an InfiniBand network. As we can see, there is a big performance gap between these two operations when the message size is small and medium. Thus, one can consider a hybrid RDMA mechanism, wherein only RDMA Write operations are used.

Figures 2.4(a) and 2.4(b) show the hybrid RDMA mechanism. In the hybrid mechanism, a read is designed with server-based RDMA Write as shown in Figure 2.4(a) and a write is designed with client-based RDMA Write as shown in Figure 2.4(b).



(a) read (b) write

Figure 2.4: Hybrid RDMA Mechanism.

The hybrid mechanism is a common method to design networked storage systems on networks that do not support the RDMA Read operation [54, 115].

2.2.4 Comparison between Three Mechanisms

The server-based RDMA mechanism has the simplest control path. The client buffer information can be associated with the client request. So the server has full control to initiate RDMA operations. In addition, the server knows when the corresponding disk write/read operations are finished. It can send the reply back to the client at an appropriate time. Therefore, the server has a natural flow control algorithm between the network and disks. One drawback is that RDMA read operation is used in the storage write call, which delivers relatively low performance with small and medium data transfers.

The client-based RDMA mechanism normally needs more control messages. An extra control message may be needed for the server buffer information. Notification is also needed to tell the server when the data has been read from the server buffer for a read operation and when the data arrives the server buffer for a write operation. Read operation also suffers from the lower performance of network RDMA Read. A possible advantage of this mechanism is that less CPU may be required to serve each request on the server side since the clients initiate all RDMA operations. This mechanism may achieve better scalability considering the large number of the clients.

The hybrid scheme takes advantage of higher performance of RDMA write operation for both read and write. For write, it has the same drawbacks as the client-based RDMA mechanism. However, if the extra control message and notification can be

done in an efficient manner, this scheme may achieve the best performance. In Section 2.5.3, we show our solutions to reduce the extra costs and quantitatively compare the hybrid scheme with the server-based scheme.

2.3 Related Issues

The aforementioned mechanisms can be applied to implement data movement in all storage read and write calls in general. There are several related issues to be addressed: *buffer registration and deregistration, request/reply/control message transfers, small and bulk read/write processing, and completion mechanisms.*

2.3.1 Buffer registration and deregistration

RDMA operation requires that both source buffers and destination buffers be registered before communication. After the communication, these buffers may be deregistered. Since both registration and deregistration operations are relatively expensive, efficient registration and deregistration are expected. We discuss buffer registration and deregistration in Chapter 4.

2.3.2 Request/Reply/Control Message Transfers

A client/sever model is often used in the networked storage systems. For each request, a pair of request and reply messages are presented, besides data movement. In addition, there are also some other messages used to maintain the wire protocols and exchange necessary information between two sides. A good system should provide efficient processing of these messages as well. Typically these messages are short. The request and control messages are unexpected, while the reply message is expected. All these messages need to get immediate attention to make progress. For example, a request message is expected to be picked up by the server when it arrives. Therefore, an appropriate communication choice should be chosen by matching these characteristics with the communication services. In the Subsection 2.5.2, we show how we make such choices in PVFS over InfiniBand.

2.3.3 Small Read/Write Request

For small requests, the overhead of the request and reply messages dominates. The data movement overhead may not account too much. An important optimization is to put data along with the request or the reply message. This optimization eliminates the date movement message. It also avoids buffer registration and deregistration at the cost of copying data into the request/reply message.

2.3.4 Bulk Read/Write Request

There are two major phases in the I/O path of networked storage systems: data movement phase, where data is transfered between client buffers and server buffers; and I/O phase, where data is moved between server buffers and disks. Overlap between these two phases is necessary for high performance in the case of large requests. One way to achieve communication and I/O overlap is to split the whole processing into multiple smaller processing. 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. Then the total cost can be approximated as follows:

$$T = max(t_{network}, t_{disk}) + n \times t_{overhead},$$

where $t_{network}$ is the network transfer time; t_{disk} is the disk access time; n is the number of transfers; and $t_{overhead}$ is the communication startup overhead. In the RDMA networks, the communication startup overhead is negligible compared to both the network time and the disk access time.

2.3.5 Completion Mechanisms

Completions of network operations including both send/receive operations and RDMA operations are important to drive the progress. Either the server or the client should be capable of detecting these completions efficiently.

There are two approaches, namely polling and interrupt. 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. We should make tradeoff between them according to the characteristics of the storage server application and the client application. For example, if the storage server application is granted to a dedicated machine, and the disk I/O operations are blocking, polling may be a better choice. However, if the disk I/O operations are non-blocking, especially which are achieved by multi-threading as seen in the current POSIX AIO library, probably the interrupt may be a better choice. In the latter case, the CPU cycles can be used to process I/O requests when there is no completion.

2.4 Designing PVFS over InfiniBand

In this section, we first give a brief overview of PVFS. Then we define a general software architecture of PVFS based on InfiniBand. We use PVFS as the foundation to show how we choose appropriate mechanisms for data movement and request/reply/control messages. We also evaluate the impact of RDMA operations on the performance of PVFS.

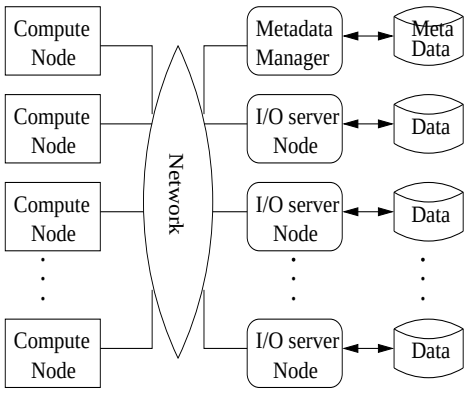


Figure 2.5: Typical PVFS setup.

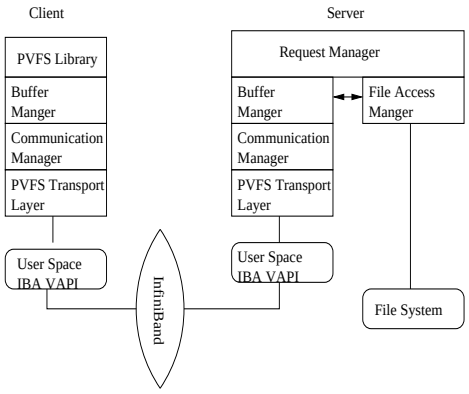


Figure 2.6: Proposed PVFS Software Architecture on InfiniBand Network.

2.4.1 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 2.5, 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.

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 cluster-wide 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 [24]. PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API, MPI-IO [90], 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.

2.4.2 Proposed PVFS Software Architecture

Figure 2.6 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.

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.

We focus on the transport layer. We also discuss the communication choices, which is a part of functions provided by the communication manager.

2.5 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 completion handling. Then we propose optimized small data transfers and pipelined bulk data transfers to further optimize the PVFS transport layer.

2.5.1 Messages and Buffers in PVFS

Messages in PVFS can be categorized as follows: *Request messages, Reply messages, Data Messages, and Control messages.* These messages are different in many ways and therefore expect different communication choices.

There are two types of 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. *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.

2.5.2 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 [46]. 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 2.1 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 2.5.3. 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 2.5.3.

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.

Table 2.1: Communication Choices

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

2.5.3 Choosing Data Movement Mechanisms

We choose the server-based RDMA mechanism for data movement. However, we perform the following optimizations.

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 [30].

2. *Fast RDMA:* Figure 2.1 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.

To optimize small writes, the client does RDMA Write to transfer data to the I/O server. However, as shown in Figure 2.3(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.

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. We divide large transfers into multiple smaller transfers. 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.

2.5.4 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.

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.

2.6 Performance Results of PVFS over InfiniBand

We have implemented PVFS on our InfiniBand testbed with designs described in Sections 2.4 and 2.5. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [57], 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 advantage 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 [56], 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. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

2.6.1 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.

2.6.2 Network and File System Performance

Table 2.2 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 2.3 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* [43] file-system benchmark is used.

Table 2.2: 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 2.3: File system performance

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

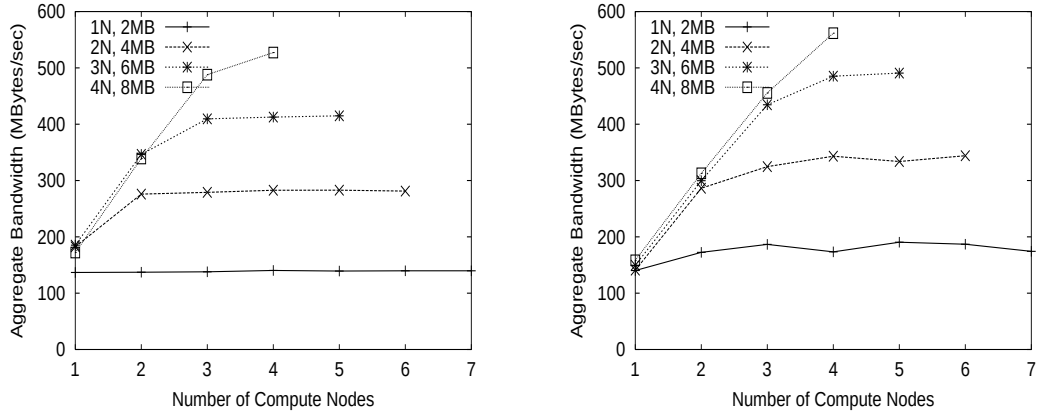
It can be seen that there is a large difference in bandwidth realizable over the network compared to that which can be obtained to a disk-based file system. However, applications can still benefit from fast networks for many reasons in spite of this disparity. Data is frequently in server memory due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

2.6.3 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 [19]. 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 2.7 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.

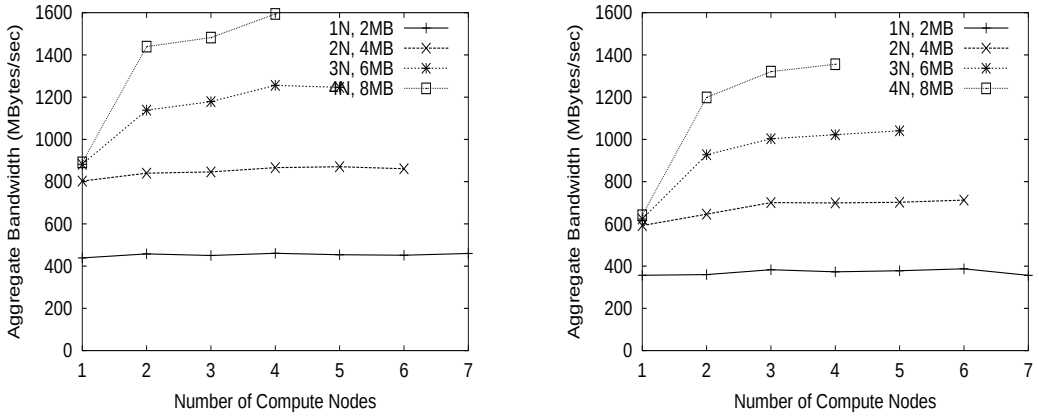


(a) PVFS Read

(b) PVFS Write

Figure 2.7: PVFS performance with IBNice (TCP/IP over InfiniBand).

Figure 2.8 shows the read and write performance of our implementation of PVFS over InfiniBand VAPI. The same physical network is used, yet significant performance



(a) PVFS Read

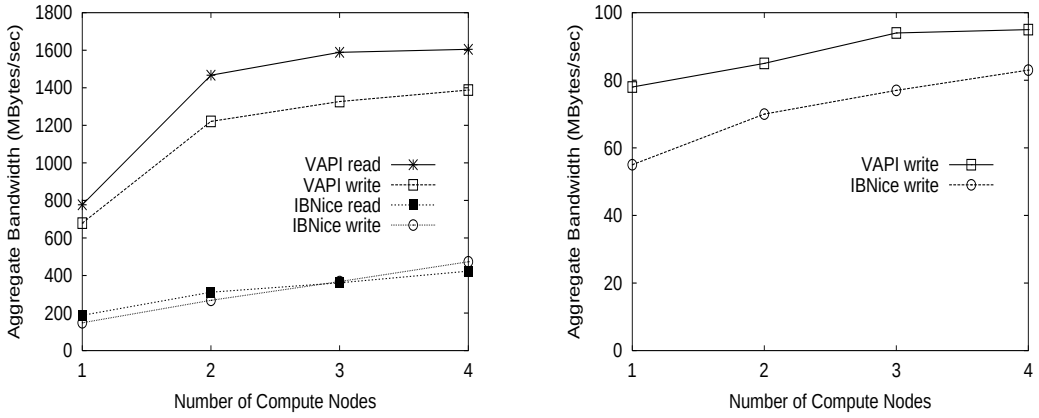
(b) PVFS Write

Figure 2.8: PVFS performance with InfiniBand VAPI

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.

2.6.4 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 2.9 shows the performance of MPI-IO over PVFS on VAPI and IBNice, for both memory-based and disk-based file systems. On the RAM file system, Figure 2.9 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.



(a) MPI-IO on ramfs

(b) MPI-IO on ext3fs, write with sync

Figure 2.9: MPI-IO Performance on PVFS over InfiniBand

Figure 2.10 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 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.

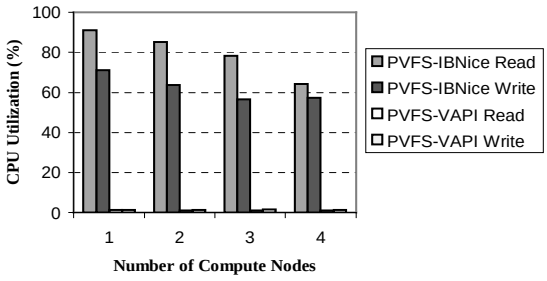


Figure 2.10: CPU Utilization of MPI-IO

2.6.5 Impact of Small Data Transfer Optimizations

To evaluate the impact of various small data transfer optimizations, we measured the access time of small PVFS write requests for different design schemes. Figure 2.11 shows the results using *Inline*, *Fast RDMA*, and *Server-Based RDMA*. The access size varies from 128 B to 64 kB.

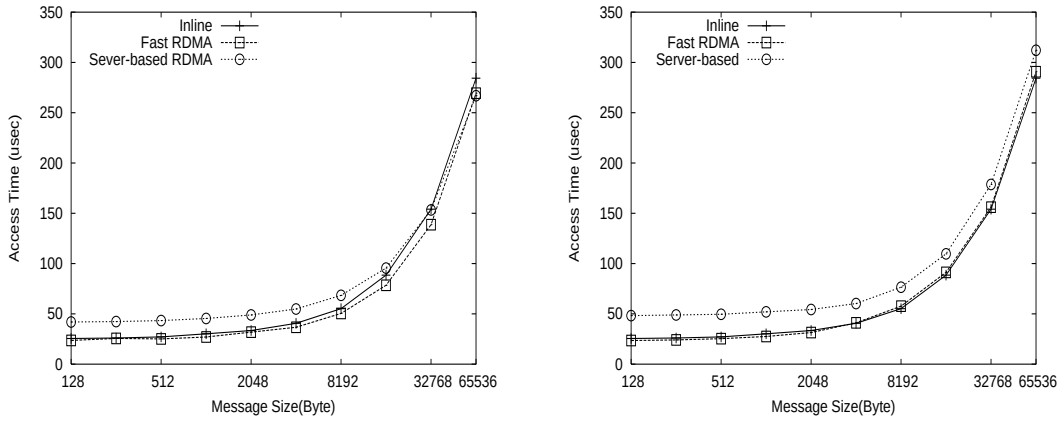
As mentioned in section 2.5, user buffers must be registered before communication. This happens in both Fast RDMA and Server-Based RDMA schemes. Applications can have different buffer usage patterns. We consider two extreme cases. As shown in Figure 2.11(a), the same buffer is used repeatedly. Therefore, registration is required only at the first time. Figure 2.11(b) shows another case in which different buffers are used. Thus, every PVFS write needs buffer registration on the client side in the Fast RDMA and Server-Based RDMA schemes. Note that the user buffer usage does not affect the performance of Inline scheme.

Figure 2.11 shows that significant improvement can be achieved using Inline and Fast RDMA schemes. It also shows these optimization schemes differ. Without the cost of buffer registration as shown in Figure 2.11(a), 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 relatively 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. With the cost of buffer registration as shown in Figure 2.11(b), 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. Both Inline and Fast RDMA schemes outperform Server-Based RDMA because of the costly registration. 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 the registration cost increases. The Server-based scheme is expected to be used for large messages.

In our implementation, 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.

2.6.6 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.



(a) PVFS Write, Same Buffer

(b) PVFS Write, Different Buffers

Figure 2.11: Small Data Transfer Optimizations on Write

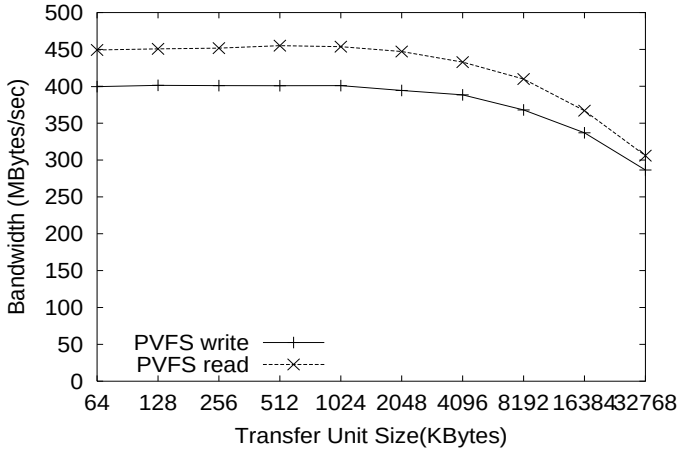


Figure 2.12: Pipelined Bulk Data Transfer

Figure 2.12 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 startup overheads from a large number of communication operations.

2.7 Summary of the Design of PVFS over InfiniBand

With the design of PVFS over InfiniBand, 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. In this section, we focus on management of communication choices and mechanisms for both data and other messages.

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.

CHAPTER 3

EFFICIENT NONCONTIGUOUS I/O ACCESS

The access patterns generated by many of scientific and non-scientific applications sometimes tend to be many small accesses scattered widely across a striped file, a model which has to date not been well supported. Previous research shows that only about a tenth or less of the peak I/O performance can be realized by many applications [91, 53]. One of the main reasons is that the I/O interfaces available to applications and the I/O methods supported by file systems do not match well to applications' access characteristics. Most file systems are optimized for large contiguous file accesses, while in many applications, each process tends to access a large number of relatively small regions that are not located sequentially in the file [9, 63, 84]. Noncontiguity can exist in both the file itself and in the memory of the client.

Traditionally, noncontiguous access is achieved with a set of contiguous calls, each of which accesses only a single contiguous piece. Several techniques [89, 34, 80, 50, 47] were proposed to optimize noncontiguous accesses in situations where only contiguous I/O access support is available. Thakur *et al.* [90] noted that native noncontiguous access support in file systems themselves is important. They proposed an interface that describes noncontiguity in both memory and the file in a simple manner. This interface not only can be used to implement noncontiguous I/O access functions in the upper programming interfaces such as MPI-IO [59] efficiently, but also allows the file systems themselves to make further optimization on the noncontiguous accesses. Ching *et al.* [1] implemented this interface in PVFS. Their implementation is called *list I/O*.

There are two important issues in providing efficient noncontiguous accesses in cluster file systems wherein the compute nodes and the I/O nodes are connected by high performance networks.

- High-performance noncontiguous data transmission between the compute node and the I/O node.
- Efficient processing of these small requests on the I/O nodes is crucial to application performance.

These two issues result in more serious performance problems when the network is not the bottleneck in a cluster file system. The issue of noncontiguous data transmission is often ignored in conventional networks. The performance differences between different ways to handle noncontiguous data transmission might not have much impact on the performance of noncontiguous I/O accesses because of the high overhead and low bandwidth in these networks. We observe that noncontiguous data transmission becomes an important factor affecting the performance of noncontiguous I/O accesses in high performance networks such as InfiniBand [46]. Similarly, the inefficiency of processing noncontiguous small requests on the I/O node might not be realized by applications on conventional networks since network performance is the main bottleneck. However, as high performance networks become a popular means to connect the compute nodes and the I/O nodes in cluster file systems, this inefficiency has direct and significant impact on the performance of applications.

In this chapter, we address two issues involved in noncontiguous I/O accesses in cluster file systems over high performance networks: noncontiguous data transmission and noncontiguous disk accesses. For noncontiguous data transmission, we propose a novel approach, *RDMA Gather/Scatter*, to transfer noncontiguous data between the clients and the I/O servers. For noncontiguous disk accesses in the I/O server nodes, we have implemented a new scheme termed as *Active Data Sieving* to reduce disk access costs for a large number of small and noncontiguous accesses. Unlike other data sieving implementations, a cost model is used by the I/O nodes to actively and intelligently decide whether it is beneficial to perform data sieving or not. We have designed and incorporated these approaches in a version of PVFS over InfiniBand.

3.1 Noncontiguous Access in PVFS

In this section, we first describe the current design and implementation of PVFS list I/O. We then show two different ways in which noncontiguous accesses arise, both of which pose challenges on efficient noncontiguous I/O access in PVFS. As illustrated in the example in Figure 3.1, the top set of communications show *noncontiguous data transmission between the compute nodes and the I/O nodes.* The second source of noncontiguity is *noncontiguous disk accesses*, as shown at the bottom, when I/O nodes access their local files. We try to solve these two issues.

3.1.1 PVFS List I/O

PVFS provides a list I/O interface to applications which can be used to perform the transfers in Figure 3.1 in a single operation. This interface confirms with the interface proposed by Thakur *et al.* in [90]. The following is the PVFS list I/O read interface (the write interface is similar):

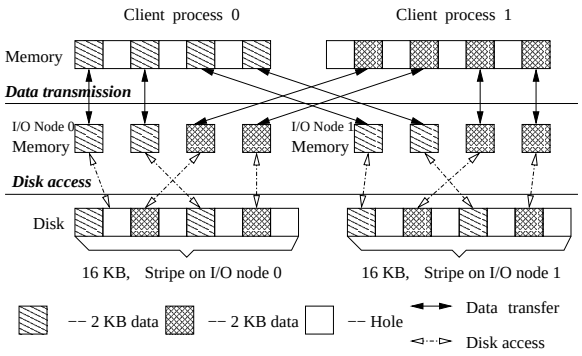


Figure 3.1: A PVFS list I/O example.

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

This interface allows a set of buffers to be used as read or write destinations in memory on the client and a set of offsets in the file on the I/O node. Noncontiguity in both the file and the memory is thus possible.

A naive implementation of list I/O would translate a list I/O request into a set of individual requests, each of which accesses one contiguous piece separately. Obviously, this would provide no advantages for list I/O.

PVFS has designed and implemented its list I/O in an efficient manner as described in [24]. The `pvfs_read_list` and `pvfs_write_list` functions take list I/O parameters and perform the noncontiguous access in a single PVFS operation. The current implementation is based on TCP/IP, a stream-based transport layer, therefore the buffer offset-length pairs are not required by the I/O nodes. When an I/O node receives a list I/O request with a number of file offset-length pairs, it services them individually; merge happens only when the actual file accesses from the same compute node are contiguous with each other.

This implementation effectively reduces the number of request and reply message pairs and increases data amount transfered in each pair of request and reply messages, significantly improving the data transfer efficiency. However, as it is based on TCP/IP, noncontiguous data transmission is not considered as an issue. Also, the I/O accesses to the local files in each I/O node are processed separately. Given a PVFS list read operation as shown in Figure 3.1, the I/O nodes read each contiguous

piece of data from the local files. After each read, they initiate a socket `write` operation to send data to client processes. Each client process reads data into 4 different buffers using 4 operations. In total, eight `read` and `lseek` operations are used for file accesses. 8 socket `write` and `read` operations are used to transfer data between the client processes and the server I/O nodes.

3.1.2 Network Support for List I/O

Many conventional communication interfaces, including TCP/IP, only support data transmission in contiguous blocks, defined by a memory address and a length. Based on these interfaces, to move data from and into a list of buffers specified in the PVFS list I/O, two schemes can be used. The first scheme is to send and receive one message for each contiguous block of data. The second scheme is to pack noncontiguous data into a temporary buffer before transmitting it, and unpacking it when it has arrived.

Both schemes have been widely used for noncontiguous data transmission. For example, the current PVFS list I/O design follows the first scheme using the socket interface over TCP/IP, and the second scheme is a generic method deployed in MPICH [102] to transmit noncontiguous data. Communication performance suffers in the first scheme since the message startup costs accrue for each message. Furthermore, the data size in each message is small. In the second scheme, one additional copy is required on both the send and the receive sides; however, it does use one large message to transfer all data.

Performance issues in noncontiguous data transmission are often ignored in conventional networks because of their high overhead and low bandwidth. The message startup costs or the extra memory copy overheads do not have much impact on the communication performance when the network is comparatively slow. However, in low overhead and high bandwidth networks such as InfiniBand, these overheads have a significant impact on performance. For example, in our InfiniBand testbed, the network bandwidth is 820 MB/s and memory copy bandwidth is 1300 MB/s, therefore a scheme to pack, send, and unpack data can offer an aggregate bandwidth of only 362 MB/s.

Due to the emergence of high-performance networks, traditional methods used for noncontiguous data transmission become very inefficient. In section 3.2, we address how we can achieve efficient noncontiguous data transmission for list I/O over high-speed networks.

3.1.3 Disk Operations for List I/O

As shown in Figure 3.1, file regions specified by the offset-length pairs in a list I/O request may not be contiguous, as is found in many parallel applications. Thus even though an I/O node can receive a large number of file accesses in one list I/O

request, if these accesses were serviced individually, the performance would be quite low. There are three major factors which affect performance of file accesses on the I/O node:

- Most file systems favor large accesses; small requests can not obtain peak performance.
- The cost of making many read/write system calls, each for small amounts of data, is extremely high, even when caching is performed by the file system [91].
- Minimizing file seeks is important to maximize performance when doing multiple file accesses.

To reduce the effects of these factors, it is critical to make as few requests to the file system as possible, generating as many large accesses to the file system as possible. Data sieving is such a technique that enables an implementation to make a few, contiguous requests to the file system even if the user's request consists of several small, noncontiguous accesses [89]. However, data sieving reduces the number of file access calls and increases file access sizes at the cost of extra data read/written. For write data sieving, costs are also paid to perform read-modify-write and synchronization to prevent concurrent updates to the same file region. The extra data may actually be good for later requests, though [90].

In section 3.3, we address the feasibility of performing data sieving on the I/O node to service list I/O requests and describe a cost model used by the I/O nodes to decide whether to perform data sieving or to access each contiguous piece of data separately in the presence of the advantages and disadvantages given above.

3.2 Noncontiguous Data Transmission

PVFS list I/O allows a set of discrete memory buffers to be used as read or write destinations in memory on the client. A typical example of such buffers is rows in a subarray of a multidimensional array, separated by gaps (*noncontiguous buffers*). As previously noticed [108], buffers on the I/O nodes are usually contiguous. An important issue is to transfer data between PVFS list I/O buffers on the compute nodes and buffers on the server nodes.

3.2.1 Mechanism Tradeoffs

As discussed in section 3.1.2, two schemes have been widely used to transfer noncontiguous data: 1) send and receive one message for each contiguous block of data, 2) pack noncontiguous data into a temporary buffer before transmitting it, and unpack it after its arrival. We call them *Multiple Message* and *Pack/Unpack*, respectively. The left and middle panels in Figure 3.2 illustrate these schemes.

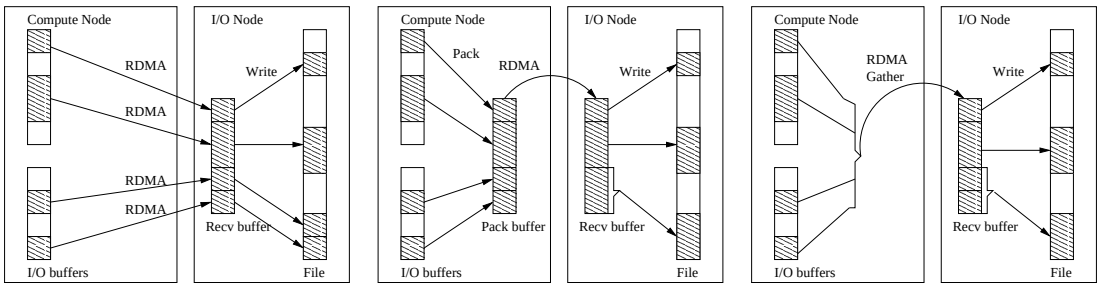


Figure 3.2: Noncontiguous data transfer. *Left: Multiple Message. Middle: Pack/Unpack. Right: RDMA Gather/Scatter.*

A third way exists to transfer noncontiguous data in modern communication networks such as InfiniBand that support RDMA Gather/Scatter operations. RDMA Write operations can gather multiple data segments together within one operation and place them in a single buffer on the receiver side. RDMA Read operations can read data from a single buffer on the peer side into multiple buffers on the local initiator. This gather/scatter functionality is a perfect match with the requirement of PVFS list I/O noncontiguous data transfer. The right panel in Figure 3.2 shows an example of RDMA gather write. In this *RDMA Gather/Scatter* scheme, the message startup costs which occur in the Multiple Message scheme can be reduced dramatically, since a large number of data segments can be specified in one operation. It also avoids data copies which are required in the Pack/Unpack scheme.

There are many tradeoffs among the three schemes, however, which complicates the design decision about when to use a particular scheme. These are listed in the following paragraphs.

Copy or memory registration. Buffers must be registered before any data transmission occurs in InfiniBand. This requires that all list I/O buffers be registered in both the Multiple Message and the RDMA Gather/Scatter schemes, and that the temporary buffer in the Pack/Unpack scheme be registered. Sometimes it is desirable to unregister these buffers after the completion of noncontiguous I/O access as well. A tradeoff exists between choosing to accept the overhead of an extra copy versus the overhead of memory registration and possible deregistration.

Communication startup overhead. The number of communication operations is different in these three schemes. In the Multiple Message scheme, it is equal to the number of list I/O buffers. In the Pack/Unpack scheme, only one transfer is required. In the RDMA Gather/Scatter scheme, some number of segments, 64 currently in InfiniBand, can be gathered into a single communication. Choosing fewer, larger messages results in better performance.

Buffer alignment. Networks which use RDMA are sensitive to buffer alignment and can generate large delays to compensate for misaligned buffers. Since the Pack/Unpack

scheme itself allocates a temporary buffer for RDMA operations, this buffer can be aligned. However, it is possible that list I/O buffers given by users may not be aligned and cause the performance of the Multiple Message and RDMA Gather/Scatter schemes to suffer.

Application buffer access patterns. The costs of memory registration and deregistration can be amortized across multiple operations by registration caching mechanisms such as pin-down cache [41]. But if the application chooses buffers in such a way that caching is not very frequent, performance of the Multiple Message and RDMA Gather/Scatter schemes might be hurt. It is likely that a Pack/Unpack implementation will reuse the same buffer and not be affected.

Since it is clear that the Multiple Message scheme will likely perform poorly compared to the other two, it is ignored now for clarity. From the tradeoffs listed above, though, it is not clear which of the remaining two schemes will be better. The answer depends on the total effects of the above factors in each scheme. We use the following test to show the performance of noncontiguous data transmission with these two schemes.

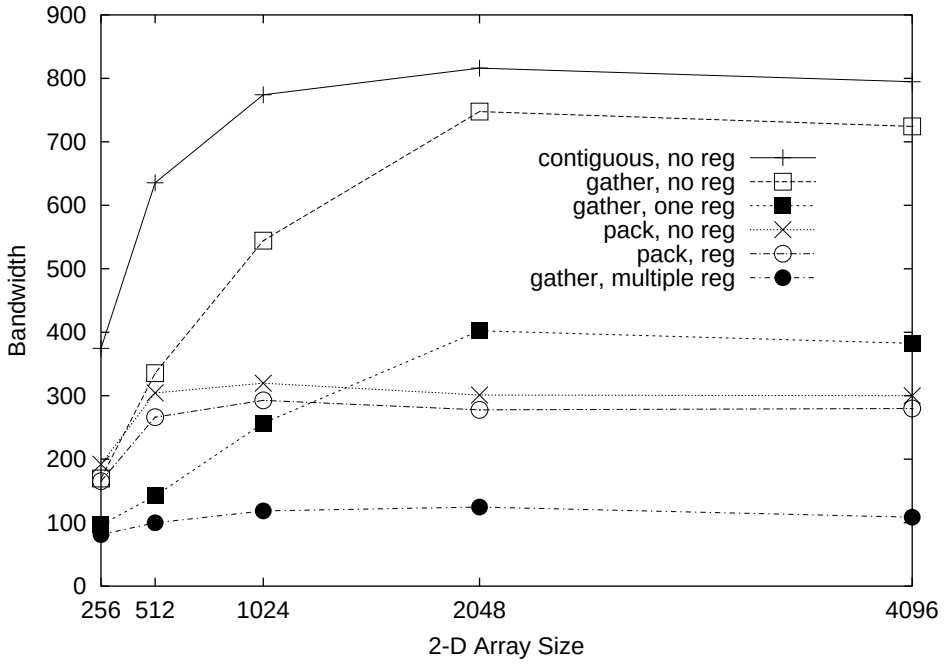


Figure 3.3: Bandwidth achieved in various transfer schemes.

In Figure 3.3, we show the bandwidth achieved in transferring a 2-D subarray from a compute node to an I/O node in our testbed. We consider the following scenario

which is a common case of I/O access patterns in scientific applications. A 2-D array of varying size is distributed across 4 processes using a block distribution in both dimensions. One of the subarrays is then sent using different schemes.

In the Pack/Unpack scheme, the temporary buffer can be allocated from a pre-registered buffer pool or from the system. In the former case, registration and deregistration are not needed. These two cases are termed as *pack, no reg* and *pack, reg*, respectively. In the RDMA Gather/Scatter scheme, two ways to register list I/O buffers are considered. One is to register each list I/O buffer separately, termed as *gather, multiple reg* in the graph. Another is to register the memory region which covers all list I/O buffers from a subarray, termed as *gather, one reg*. We also show its best case, where memory registrations are always found in the cache, called *multiple, no reg* in the graph. Finally, the maximum achievable bandwidth obtained by a single write is labeled *contiguous, no reg* in the graph.

Several observations can be made from Figure 3.3. First, the packing and memory registration costs have a dramatic impact on performance. Second, the Pack/Unpack scheme performs comparatively better when the array size is small, Third, the RDMA Gather/Scatter scheme has the potential for high performance if registrations are handled well.

The above test results show that the RDMA Gather/Scatter scheme is very promising when the costs of memory registration and deregistration can be controlled in a certain range. The issue of how we can reduce the costs of memory registration and deregistration is addressed in Chapter 4.

3.3 Efficient Noncontiguous File Access on the I/O Node

In this section, we set out to answer this question: *how a single I/O node can efficiently perform file accesses to service a list I/O request.* This is the second issue in noncontiguous I/O access. Combined with efficient noncontiguous data transmission as discussed in Section 3.2, a complete solution is achieved that can offer high performance noncontiguous I/O access in PVFS.

3.3.1 Active Data Sieving on the I/O node

As previously mentioned, in many parallel applications, each process tends to access a number of relatively small, noncontiguous portions of a file. Information of a large number of file accesses can be sent in one single list I/O request; however, performance would suffer if these accesses were serviced separately, as we discussed in Section 3.1.

We propose to apply data sieving on the I/O node to process PVFS list I/O requests. We refer to this as *Active Data Sieving (ADS)*. In data sieving, for read, instead of reading each contiguous portion separately, the I/O node reads a large

contiguous chunk into a temporary buffer. It then transfers the desired data to the list I/O buffers on the compute node. For write, a read-modify-write is performed locally. First, the I/O node reads a large contiguous chunk of data into a temporary buffer. Second, it copies data received from the compute nodes into appropriate locations in the temporary buffer. Then, it writes that temporary buffer back to the file. In general, the portion of the file being accessed must be locked to prevent concurrent updates, but that is not a concern here since the I/O node has exclusive access to its own local data. Active Data Sieving fits nicely with scatter/gather capable networks such as InfiniBand where just parts of the buffer can be sent naturally to the requesting client.

When a list I/O request arrives, the I/O node analyzes all file accesses in the request and decides whether it is beneficial to apply data sieving to process these accesses or not. A cost model could be very comprehensive and complicated to take access patterns, system parameters, disk characteristics and cache effects into account. Here, we deploy a simple but effective model in the design of ADS on the I/O node. The following parameters in table 3.1 are considered.

Table 3.1: Model Parameters

System Parameters	
B_{mem}	Memory bandwidth
$B_r(s)$	File read bandwidth without cache for size s
$B_w(s)$	File write bandwidth without cache for size s
O_r	File read overhead
O_w	File write overhead
O_{seek}	File seek overhead
O_{lock}	File lock overhead
O_{unlock}	File unlock overhead
Request Parameters	
N	Number of noncontiguous accesses
S_i	Size of the ith file access
S_{req}	Total size of wanted data
S_{ds}	Total size of data sieving data

Given these parameters, the costs to serve a PVFS list read and write without data sieving and with data sieving are represented by T_r, T_w, T_{dsr}, and T_{dsw}, respectively. In T_{dsw}, the costs for read, modify and write with synchronization are included.

$$T_{read} = N \times (O_r + O_{seek}) + \sum_{i=1}^{N} \frac{S_i}{B_r(S_i)}$$

$$T_{write} = N \times (O_w + O_{seek}) + \sum_{i=1}^{N} \frac{S_i}{B_w(S_i)}$$

$$T_{dsr} = O_r + O_{seek} + \frac{S_{ds}}{B_r(S_{ds})}$$

$$T_{dsw} = T_{dsr} + \frac{S_{req}}{B_{mem}} + O_{lock} + O_w + \frac{S_{ds}}{B_w(S_{ds})} + O_{unlock}$$

This model gives a conservative estimate of the costs for data sieving, since cache effects are not taken into account. This indicates that with this cost model, if data sieving is chosen by the I/O node, it is highly probable that data sieving is beneficial due to the added effect of caching discussed earlier.

3.3.2 Why not Just Use ROMIO Data Sieving?

Data sieving is used in ROMIO to handle independent noncontiguous accesses [91] on each process. In the latest ROMIO implementation over PVFS, it performs data sieving for noncontiguous reads if the corresponding hints are enabled. However, it does not perform data sieving for noncontiguous writes since the current PVFS does not support file locking. Although both ADS and ROMIO data sieving over PVFS tend to reduce the number of I/O calls to the local file system in the I/O node and to increase I/O access sizes, ROMIO data sieving is a *client* side implementation of data sieving, while ADS is a *server* side implementation of the same task. Particularly, there are a few key differences between the two.

First, list I/O with ADS enables list I/O write operations to take advantage of data sieving benefit. Second, the undesired data is not transfered through the network in list I/O with ADS. Third, data copies can be avoided on the compute node with ADS. In addition, ADS is more capable of making better decisions on whether to perform data sieving or access contiguous data segments separately since in ROMIO data sieving, the compute node may be not aware of underlying file system data layouts and parameters.

MPI-IO applications can choose to use either list I/O with ADS or ROMIO Data Sieving by setting file system hints to override the defaults.

3.4 Performance Results

This section presents performance results from a range of benchmarks on our implementation of PVFS over InfiniBand. Based on our previous work [108], we added noncontiguous data transmission and active data sieving. Our implementation is based on PVFS version 1.5.6. The InfiniBand interface is VAPI [57], which is a user-level programming interface developed by Mellanox and compatible with the InfiniBand Verbs specification. We use both PVFS and MPI-IO micro-benchmarks as

well as applications to quantify our design choices in noncontiguous data transmission and noncontiguous file accesses. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes, or 1024×1024 bytes.

3.4.1 Effects of Data Transfer Mechanism

We design a PVFS-level micro-benchmark to show the effects of the design choice whether to use Pack/Unpack or RDMA Gather/Scatter to transfer noncontiguous data between the compute nodes and I/O nodes. In this test, there are four I/O nodes and four compute nodes. Each process wants to write or read variable sizes of data using PVFS list I/O operations. The number of noncontiguous data segments is set to 128. The size of each segment is equal, and varies from 128 bytes to 8 kB.

Three design choices are compared: Pack/Unpack, RDMA Gather/Scatter, and the hybrid scheme which we use in our final design. Figure 3.4 shows that Pack/Unpack works better when the total request size is not large, while RDMA Gather/Scatter performs better when the request size is large. The hybrid scheme we choose combines these two schemes and works well in both cases.

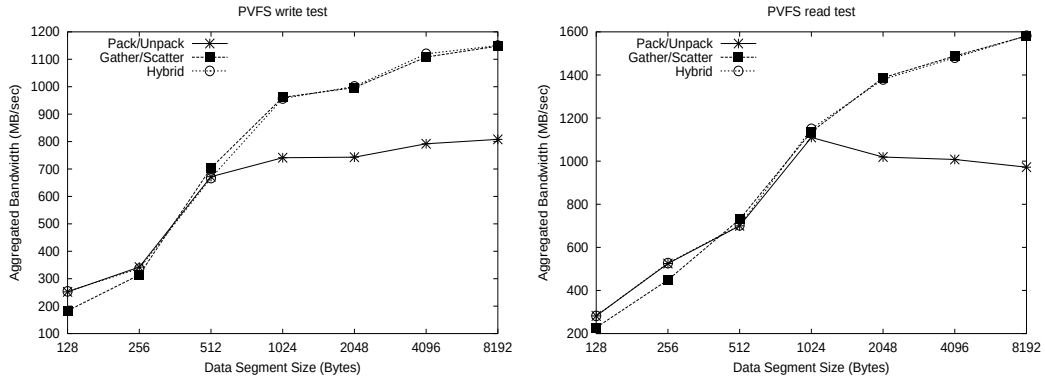


Figure 3.4: Performance of noncontiguous data transfer schemes.

3.4.2 MPI-IO Noncontiguous Access Benchmarks

To evaluate the impact of Active Data Sieving on the performance of PVFS list I/O operations, we designed a benchmark using MPI-IO. The file view is one dimensional block column distribution. As shown in Figure 3.5, the file accesses are noncontiguous: each process accesses only one unit out of every four in the file. In this test, we vary

the size of the array from 512 to 8192, thus the numbers of columns touched by each process changes from 128 to 2048.

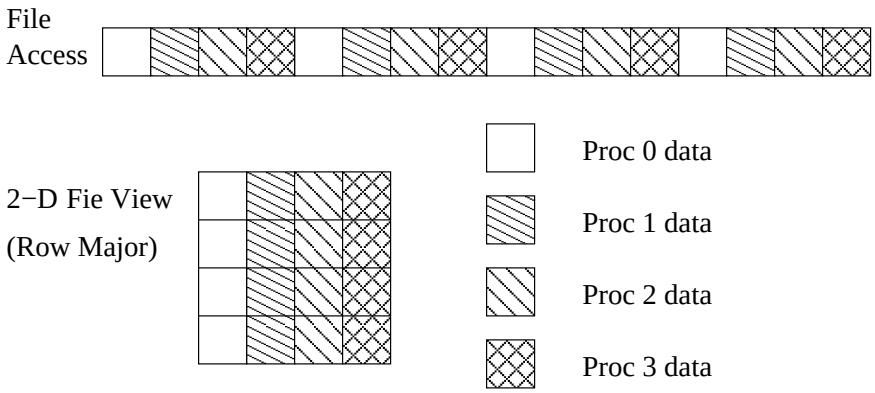


Figure 3.5: Accesses in the file view with one-dimensional block column distribution.

In the test, we set different hints to enable the potential techniques: Multiple I/O, ROMIO Data Sieving, and PVFS list I/O. In the PVFS list I/O method, we also test using Active Data Sieving or not. We compare the performance of these four methods for read, where the data is in cache or uncached, and write, where the data is potentially flushed to disk.

Figure 3.6 shows write results for each method. As mentioned earlier, ROMIO Data Sieving over PVFS for noncontiguous write is actually implemented with the Multiple I/O method, thus their performance is nearly identical. The next feature to notice is that using list I/O always outperforms ROMIO Data Sieving by a factor of anywhere from 3.5–12.1 depending on the array size. This is true both when considering just the network transfer aspects (no sync), and when considering the full time to commit the data to disk. Regarding the two list I/O curves, it can be seen that using ADS shows a significant benefit in the small array size range. Starting at 2048, the model used by the I/O node decides that there is no benefit to be gained from using ADS, hence the curves merge.

Figure 3.7 shows read results for each method. In these graphs it can be seen that list I/O is comparable to, or outperforms ROMIO Data Sieving. As the array size increases, ROMIO Data Sieving must transfer the whole array from the I/O nodes to the compute nodes, and its performance suffers while list I/O selectively transfers only the data required by each compute node. Using ADS again improves performance in the small array cases. In the read without cache case, ROMIO Data Sieving is comparable to list I/O for a wide range of array sizes, up to 2048, because disk access time dominates in this range. Eventually the overheads of reading three times as much data and sending it across the network catch up with ROMIO Data Sieving

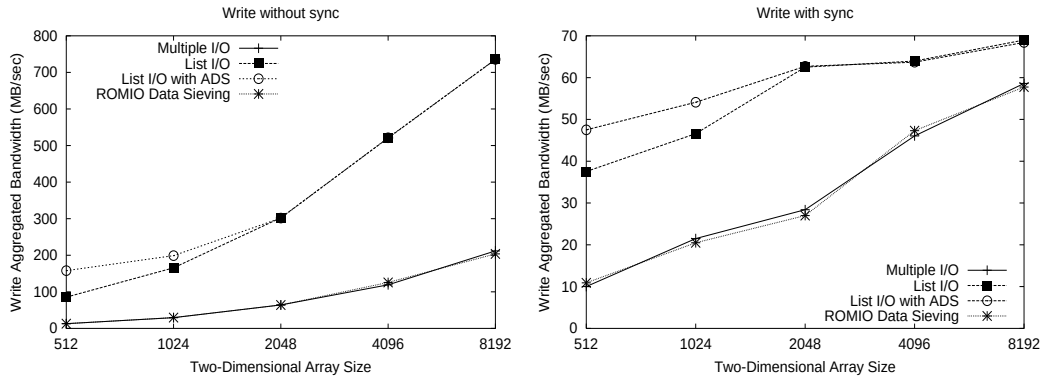


Figure 3.6: Write results with different methods in the block-column file view.

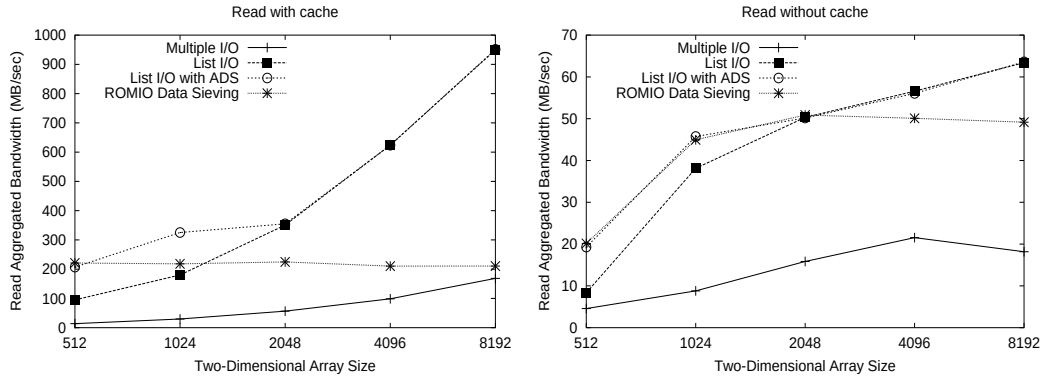


Figure 3.7: Read results with different methods in the block-column file view.

and its performance falls off. However, list I/O with ADS can decide that there is no benefit to perform data sieving and then can access each file region separately in the large array cases.

3.4.3 MPI-IO Tiled Access Test

The test application *mpi-tile-io* [76] 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 bits, leading to a file size of 9 MB.

The access pattern in this test is noncontiguous in file space but contiguous in memory. We consider the same four cases as in the previous test: Multiple I/O, List I/O, List I/O with ADS, and ROMIO Data Sieving for both read and write, again either considering or ignoring disk effects.

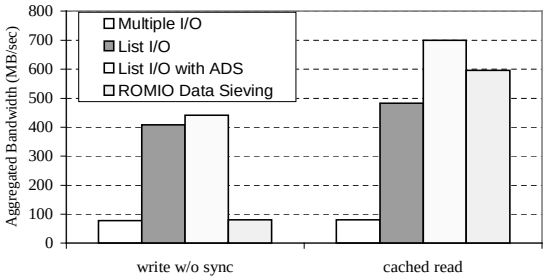


Figure 3.8: Tiled I/O, without disk effects.

Figure 3.8 shows the results for the four test cases when data is written without sync and read from the file cache. Compared to the Multiple I/O case, List I/O with ADS has a factor of 5.7 improvement for write, and 8.8 for read. Compared to the List I/O case, it has 8.4% improvement for write, and 45% improvement for read. Write is more costly than read with ADS, due to the need to perform a read-modify-write cycle. Compared against ROMIO Data Sieving, list I/O with ADS still has a factor of 5.7 improvement for write, but just 18% improvement for read.

Figure 3.9 shows the results for the four test cases when the disk is the bottleneck in data transfers. For write, list I/O with ADS still outperforms the other methods. For read, ROMIO Data Sieving now outperforms list I/O with ADS. There are two reasons. First, the increased network transfer time in ROMIO Data Sieving does not matter when the disk dominates. Second, list I/O with ADS generates 6 pairs of

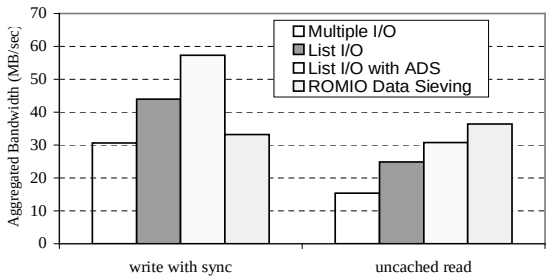


Figure 3.9: Tiled I/O, with disk effects.

request and reply messages, compared to just one with ROMIO Data Sieving, adding more overhead to the entire operation.

It is expected that the list I/O with ADS will improve with increasing number of file accesses in one list I/O request. Currently, we use the default value in PVFS which is 128, but a larger number can be used to decrease the number of request and reply pairs needed to complete the operation.

3.4.4 NAS BTIO Benchmark

The BTIO benchmark was recently added into the 2.4 version of NAS Parallel Benchmarks (NPB) and is used to test the output capabilities of high-performance computing systems, especially parallel systems. It is based on the Block-Tridiagonal problem of the NPB Suite. The details of the numerical algorithm, data partition, and data distribution can be referred to [72].

There is a very high degree of fragmentation in data sets of the BT problem. The main access pattern in BTIO is noncontiguous in memory and in the file. Thus, this test can be used for us to quantify our design choices in both noncontiguous data transmission and noncontiguous file accesses. Results for a class A problem size are shown in Table 3.2, where we show the total problem execution time and the I/O overhead, which is the amount of time the benchmark spends performing I/O operations. It can be seen that list I/O with ADS performs best even for this complex application.

We profiled the I/O characteristics of this test for the above five I/O methods. Due to space limitations, we briefly describe the profiling information here, details can be found in [111]. In both list I/O cases, the number of request messages is reduced to 1360, a significant reduction compared to the Multiple I/O method (163840) and the Data Sieving method (82040). In List I/O with ADS, the number of the file access call pairs (lseek, write) and (lseek, read) on each I/O node is also reduced to 7680, compared to Multiple I/O (163840), List I/O (163840), and ROMIO Data Sieving (85060). This reduction is attributed to Active Data Sieving on the I/O node.

Table 3.2: BTIO Performance

case	Time (s)	I/O overhead (s)
no I/O	165.6	0
Multiple I/O	180.0	14.4
Collective I/O	169.6	4.0
List I/O	168.2	2.6
List I/O with ADS	167.7	2.1
Data Sieving	177.3	11.7

3.5 Summary of Efficient Noncontiguous I/O Access Support

In this section, we address two issues involved in noncontiguous I/O accesses in cluster file systems over high performance networks: noncontiguous data transmission and noncontiguous disk accesses. For noncontiguous data transmission, we propose a novel approach, *RDMA Gather/Scatter*, to transfer noncontiguous data between the clients and the I/O servers. For noncontiguous disk accesses in the I/O server nodes, we have implemented a new scheme termed as *Active Data Sieving* to reduce disk access costs for a large number of small and noncontiguous accesses. Unlike other data sieving implementations, a cost model is used by the I/O nodes to actively and intelligently decide whether it is beneficial to perform data sieving or not.

We have designed and incorporated these approaches in a version of PVFS over InfiniBand. Our results show a performance improvement of up to 1.5 times for the RDMA Gather/Scatter approach on PVFS list I/O performance compared to the other approaches. Intelligent and active data sieving on the I/O node achieves a factor of 1.3–1.9 improvement on small noncontiguous I/O accesses. The NAS BTIO benchmark performance results show that our approach attains a 20% improvement compared to the best result across all other approaches in an environment which is a complex combination of noncontiguous data transmission and noncontiguous I/O accesses.

The approaches proposed in this paper for noncontiguous data transmission in noncontiguous I/O access has been used in MPI Datatype communication [112]. It also can be natually applicable to noncontiguous data movement such as database multiple data segment transfer in other networked storage systems. The design of Active Data Sieving is not network specific.

CHAPTER 4

COMMUNICATION BUFFER MANAGEMENT

RDMA enables direct data movement between two buffers across networks. However, it requires that both buffers be registered. Memory registration and deregistration are kernel involved operations and are commonly expensive. The costs of memory registration and deregistration make it necessary to have a particular component, namely communication buffer manager, to provide efficient memory registration and deregistration and manage those registered buffers in networked storage systems.

In this section, we first look at the memory registration and deregistration costs on InfiniBand and their impact on the network communication performance. Second, we present a scheme to achieve efficient memory registration and deregistration on a single buffer. Third, we present a scheme to achieve efficient memory registration and deregistration on a list of buffers. Fourth, we discuss how to manage the communication buffer on the server side for better communication performance. Finally, we use PVFS over InfiniBand to evaluate the performance of our proposed schemes.

4.1 Costs of Memory registration and Deregistration

Memory registration includes three steps. First, the registered buffer is pinned with the system and becomes non-swappable. Second, address translation from virtual address to physical address. Third, information bookkeeping and access control are recorded by the NIC. Both registration and deregistration are commonly expensive. Figure 4.1 shows memory registration and deregistration costs in the current InfiniBand SDK [57]. Figure 4.2 shows their impact on the network bandwidth. We use an MPI bandwidth test to show the impact. With on-the-fly registration and deregistration, we can see that for large messages, 35% performance is degraded. Especially small and medium messages suffer more. For example, when the message size is 4096 bytes, only one tenth of peak bandwidth is delivered (450 MegaBytes vs. 48 MegaBytes). Therefore, without efficient memory registration and deregistration, the benefits of RDMA become questionable.

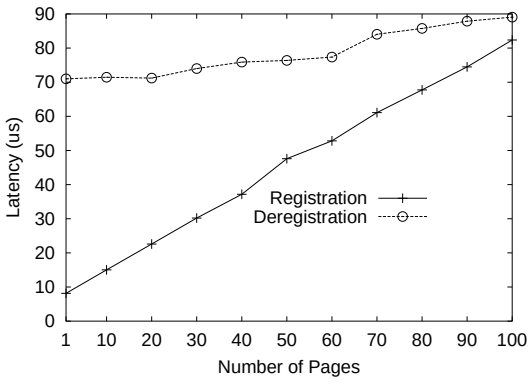


Figure 4.1: Costs of Memory Registration and Deregistration over InfiniBand Network

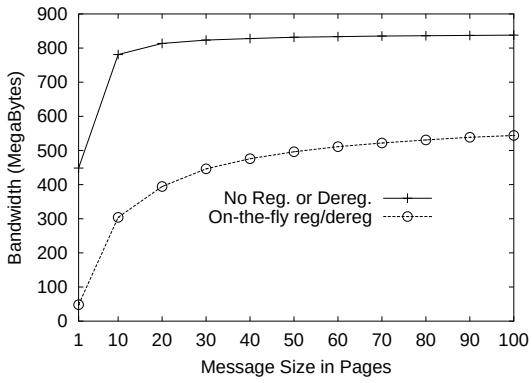


Figure 4.2: Impact of Memory Registration and Deregistration on the bandwidth of InfiniBand Network .

4.2 Memory Registration and Deregistration on Single Buffers

Memory registration and deregistration have been identified as an issue in RDMA networks by several research studies. Basu *et al.* [104] show how the NIC and host-level software can collaborate to manage large amounts of host memory. Tezuka *et al.* [41] propose a pin-down cache to reduce memory registration and deregistration overhead for zero-copy communication. 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.

In I/O intensive applications, a large number of buffers are used for I/O operations. For example, the database server may use all of memory available as cache to read and write data. Scientific I/O intensive applications also use a large number of different I/O buffers. In these applications, 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.

Zhou *et al.* [115] propose a *batched deregistration* scheme to deregister a certain number of buffers in a region in one operation. In their scheme, the registration is dynamic. However, the deregistration is batched. A certain number of deregistration operations are performed in one call.

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.

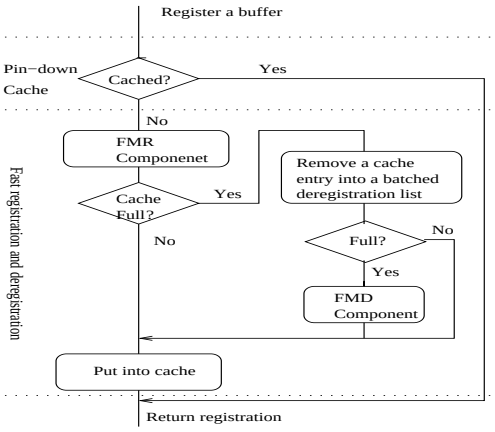


Figure 4.3: Fast Memory Registration and Deregistration (FMRD).

As shown in Figure 4.3, 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 [57]. 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. Their impact on PVFS performance is quantified in details in Section 4.5.

4.3 Memory Registration and Deregistration on a List of Buffers

The I/O systems often provide some interfaces to enable a list of I/O requests with a single function call. For example, POSIX *lio_listio*, DAFS *dap_async_listio* and PVFS *pvfs_read/write_list* all support noncontiguity in both application memory regions and file regions. For simplicity, we call these interfaces *list I/O interface*. The list I/O interface provides opportunity to the underlying system to optimize data movement and file/disk accesses. However, it also incurs noncontiguous data movement in networked storage systems.

Noncontiguous data movement may happen to a single request due to other reasons, even without the list I/O operations. For example, in a typical database server, I/O access buffers on the database server are from its cache, which are commonly divided into pages and may not be contiguous for a request. On the server side, the requested data may be also cached in multiple pages, which may not be contiguous. Data movement between these buffers is noncontiguous.

Memory registration and deregistration on a list of buffers are a challenge in noncontiguous data movement. We discuss its issues and propose our approaches to these issues. We evaluate the performance of our approaches by implementing PVFS list I/O over InfiniBand.

4.3.1 Issues

The registration and deregistration operation costs can be modeled as follows:

$$T = a \times T_p + b$$

where a is the per-page overhead, and b is the per-operation overhead. Usually, b is much larger than a. The complication for registering and deregistering a list of buffers comes from the number of registration and deregistration operations on list I/O buffers and the total size of memory space to be registered and deregistered.

Individual registration and deregistration on each buffer incur high per-operation overhead. Reducing the number of buffers needed to be registered as much as possible is critical. On the other hand, the total size of memory space to be registered and deregistered should also be considered. We could register or deregister the whole memory region which covers all list I/O buffers with a single operation. The total amount of memory registered increases because even unused areas are included. But the benefit of reducing the number of buffer registration calls in this way may not arise because more time is required to perform the registration as the memory size grows.

Based on these observations, the reigning design principle which dictates how to perform memory registration and deregistration on a list of I/O buffers is to reduce

the number of buffers as much as possible, while also minimizing the total size of memory regions.

4.3.2 Optimistic Group Registration

The first, naive scheme is to register the entire memory region which covers all the list I/O buffers. Although the number of registration operations is thus reduced to one, there are two practical problems. The "holes" between two buffers may not really have been allocated by the application, causing the registration call to fail. Or, even if the whole memory region has been allocated, the total size of "holes" may be so large that no benefit is gained over simply registering each of the buffers separately.

The second scheme is to group list I/O buffers into several memory regions. This scheme overcomes the problems in the naive scheme by the following two steps. In the first step, it controls the sizes of memory regions which are going to be registered by sorting and grouping buffer regions to avoid attempting to allocate truly large "holes" of memory between buffers. This avoids the failure of the naive scheme to gain any benefit over individual registration. In the second step, it queries the operating system to find out if a "hole" which was not rejected by the first step is actually in the process allocated memory space and thus safe to register. Where they are not, buffers again must be registered independently.

The third scheme is a combination of the previous two. First, it sorts and groups list I/O buffers into candidate regions for registration. Then, it optimistically attempts to register each memory region as the first scheme does, but if the operating system denies one of these registrations, it must query the operating system to find out actual boundaries of application memory allocation and register exactly those. We call this scheme *Optimistic Group Registration*. It is quite efficient in the common case where all list I/O buffers come from one or more bigger buffers, but is also safe by virtue of relying on queries to the operating system if it must.

4.3.3 Our Design and Implementation over InfiniBand

We use the Optimistic Group Registration scheme for two reasons. First, in common cases, list I/O buffers are from one encompassing allocation, such as rows in a subarray of a multidimensional array. Second, it is transparent to PVFS applications. There are three steps in this scheme: 1) group list I/O buffers into candidate regions, 2) optimistically register each region, and 3) to filter out "holes" which resulted in registration failures, if any.

The following equation is used to sort and group list I/O buffers. The cost of registering a buffer is modeled as $T = a \times p + b$, where a is the registration cost per page, b is the overhead per operation, and p is the size of the buffer in pages. The same cost equation can be applied to deregister a buffer with different values of a and b. In our testbed, we found the costs per page in buffer registration and

deregistration to be 0.77μs and 0.23μs, respectively. The overheads per registration and deregistration operations are 7.42μs and 1.1μs, respectively. According to this cost model, a tradeoff can be made between the number of operations and the buffer size. In our implementation, we compare the cost to register a large combined region which includes extra unneeded "holes" against the cost to perform multiple small regions to determine candidate groupings.

These candidate memory regions are optimistically registered, one at a time, in the second step. If all registration operations are successful, the procedure is finished. This is the common case in most applications.

When an optimistic registration fails, if there are not too many buffers inside the failed region, we simply allocate them as given. But if there are many buffers which would make that too expensive, we query the operating system to find the "true" holes in virtual memory space. There are a few ways to find out this information. In Linux, one can read from the file /proc/$pid/maps, but that is quite slow. Instead we added a system call which walks the virtual memory structures in the kernel to find the same information. This system call requires about 70 μs when querying about 1000 holes compared to 1100 μs when reading from /proc. A third mechanism is system independent and involves using signal handling to catch segmentation violations while reading from one word on each page in order to find if the pages are resident or not. Alternatively, some systems support the *mincore* system call which perhaps will provide the same information. We have investigated these last two and plan to use them to deploy PVFS on other systems.

4.4 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. 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 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 RDMA networks, 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.

4.5 Impact of Memory Registration and Deregistration

In this subsection, we show the impact of memory registration and deregistration on the performance of an implementation of PVFS over InfiniBand.

4.5.1 Fast Memory Registration and Deregistration (FMRD)

We run pvfs-test program again with three different memory registration and deregistration schemes. Results are presented in Figure 4.4. 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 4.4 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% improvement compared to pin-down cache when buffer reuse ratio is 20%.

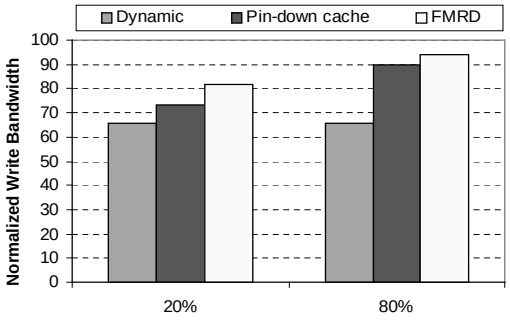


Figure 4.4: Effects of Memory Registration and Deregistration

4.5.2 Optimistic Group Registration Performance

This test is designed to study the impact of Optimistic Group Registration on the PVFS list I/O performance. The test writes a 2-D integer array of size 2048×2048 into one file in row-major order. The array is distributed across 4 processes using a block distribution in both dimensions. Each process writes its subarray into the file contiguously at different non-overlapping file locations.

Four cases are considered. The first case is the ideal one where no registration is needed. This happens when all buffer registrations have been previously cached. The second case is individual registration and deregistration on each buffer. The third case is to use the Optimistic Group Registration scheme to register list I/O buffers that come from the subarray. The fourth case is similar to the third case, except that the list I/O buffers are not all part of the same large array. We take 1024 buffers from several arrays, and intentionally create 10 holes which are not allocated yet between these buffers. By this, we can see the costs for registration failures and querying the operating system in the Optimistic Group Registration scheme. We call these four test cases "Ideal", "Indiv." "OGR" and "OGR+Q", respectively.

Table 4.1 lists the write bandwidth, the number of registrations, and the overhead for registration in each test case. Compared to the ideal case, the other three cases have 57%, 6% and 13% dregradation, respectively in write without sync. In write with sync, when disk access time is dominant, however, the overhead of memory registration and deregistration in the individual case still results in 11% degradation.

The number of registration operations and their costs are also shown in the table. It can be observed that Optimistic Group Registration reduces costs of registration on list I/O buffers dramatically. In addition, a faster file system leads to a larger impact from memory registration and deregistration.

Table 4.1: Optimistic Group Registration Impact

case	no sync (MB/s)	sync (MB/s)	# reg	overhead (μs)
Ideal	1010	82	0	0
Indiv.	424	73	1024	5254
OGR	950	≈82	1	227
OGR+Q	879	≈82	11	496

4.6 Summary of Communication Buffer Management

Memory registration and deregistration for networks with remote DMA capabilities adds a new dimension to data movement for I/O intensive applications in the networked storage system. In a contiguous data movement, we have observed that up to 35% performance can be degraded if dynamic registration and deregistration are not avoided. In a non-contiguous data movement, even more performance can be degraded. We have shown that our two-level Fast Memory Registration and Deregistration (FMRD) can effectively reduce the registration and deregistration costs and work well with I/O intensive applications. The Optimistic Group Registration scheme reduces these costs significantly on a list of buffers. Both schemes enable RDMA-based data movement mechanisms to take full advantage of RDMA operations.

CHAPTER 5

UNIFYING CACHE MANAGEMENT AND COMMUNICATION BUFFER MANAGEMENT

This chapter presents the design, implementation, and evaluation of *Unifier* [113]. Unifier is a component in server applications such as network storage system servers and other I/O serving applications (e.g., Web servers). It enables efficient interaction and integration among the components of the server application and the system subsystems.

Unifier is designed to improve the performance of server applications. In particular, Unifier has three main goals. First, Unifier eliminates redundant data copying in the I/O path. Each data object can have only one single copy in the whole system which is shard by all application components and system subsystems safely and concurrently. Unifier also eliminates multiple buffering of data, thus the cache size is effectively increased. Second, Unifier serves as a buffer manager to provide buffers to RDMA operations in the emerging network technologies. Unifier tries to manage these communication buffers in a manner to reduce memory registration and deregistration costs as much as possible. Therefore, the server application can take full advantage of RDMA-capable networks such as InfiniBand. Third, Unifier provides an application-level cache to achieve cache adaptivity and application-specific cache optimization. It provides expressive interfaces to achieve better cooperation among components.

A prototype of Unifier was implemented as a stand-alone component. It has well-defined interfaces. It also allows flexible accesses to the underlying file and storage systems via various interfaces. This component can be deployed in a wide range of server applications as both an application-level cache manager and a communication buffer manager for RDMA operations. In this chapter, we focus on the design of Unifier over InfiniBand network and its deployment in an implementation of PVFS1 over InfiniBand [109, 110] which has been discussed in Chapter 2. Our central performance results are the performance of the PVFS1 implementation with Unifier, in addition to other micro-benchmarks to measure the cache performance itself.

5.1 Motivation

The advent of networking technologies and high performance transport protocols facilitates the service of storage over networks. These enabling technologies eliminate or reduce costs of memory copy, network access, interrupt, and protocol processing in the network subsystem. In Chapters 2, 3, and 4, we have discussed how we can take advantage of RDMA networks to reduce CPU overhead, increase I/O throughput, and improve server scalability in networked storage systems.

Another source of performance limitation in networked storage systems is the lack of integration among system subsystems and the storage server applications in the general-purpose operating system [35, 65, 6]. This often results in redundant data copying, multiple buffering, and other performance degradation [65]. Redundant memory copying leads to high CPU overhead and limited server throughput. In networks such as IBA which provides comparable performance to the memory system, this becomes even worse. Multiple buffering of data wastes memory. Consequently, the effective size of cache space is reduced, increasing cache miss rates and disk accesses. In addition, the narrow interface [35, 6, 5, 38, 83] between system subsystems and applications becomes a barrier to achieve efficient cooperation.

We proposed Unifier to achieve efficient interaction and integration between the components of storage server applications and the system subsystems.

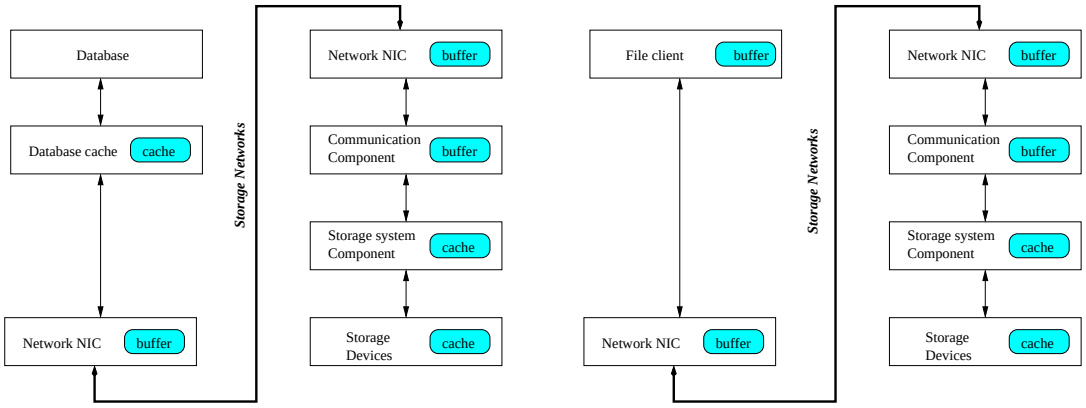
5.1.1 Data Path in Networked Storage Systems

Figure 5.1 shows two examples of data paths in the networked storage systems. In Figure 5.1(a), a database server is connected to a networked storage server. In Figure 5.1(b), it shows the data path in a typical network file system environment such as DAFS and PVFS in which there is no client cache. In the storage server side, we can see that there may be a copy between the communication buffer and the storage cache. In the general-purpose system, there are several methods to avoid such copy in some cases. We look at this issue further by analyzing PVFS data transfer over TCP/IP and issues on InfiniBand.

5.1.2 PVFS Data Transfer over TCP/IP

The I/O path in a PVFS I/O server combines both network I/O operations and file I/O operations. Therefore, the efficiency of PVFS I/O servers relies on performance of both operations, as well as the interaction between their associated subsystems: the network subsystem and the file system. In the implementation of PVFS over TCP/IP, three data transfer methods can be provided, reflecting different interactions.

Normal: In the *Normal* method, a PVFS server translates a PVFS read request into two separate calls: a file read call and a network write call. Similarly for a PVFS write request, it is translated to a network read call and a file write call. As analyzed



(a) A Database Server Example

(b) A Direct Access File Server Example

Figure 5.1: I/O Path In Networked Storage Systems.

in [88], there are usually four context switches. There are at least two data copies: copy between the user buffer and the file cache, and copy between the user buffer and the network buffer.

Mmap: The *Mmap* method maps the requested part of a file into the application user space using the system call `mmap(2)`. Then an application read or write on the mapped buffers results in a file read or write. This avoids data copy between the user buffer and the file cache. But the context switches remain same. The improvement comes at the cost of several constraints, complicated memory management, and error-prone pitfalls [88].

Sendfile: `sendfile(2)` is a system call providing direct data transfer between two file descriptors, including a TCP socket descriptor. Using sendfile, a PVFS server can do the file read and the network write together in one call. This reduces not only context switches, but also two data copies as mentioned in the Normal method. Over networks with Zero-copy TCP/IP implementation, the *Sendfile* method enables Zero-copy I/O path for transmitting data from the file to the network [88]. However, there is no support on *recvfile-like semantics*. That is, to serve a PVFS write request, the I/O server should follow either the Normal or the Mmap method.

5.1.3 Data Transfer Issues in PVFS over InfiniBand

In Chapter 2, we designed and implemented a version of PVFS over InfiniBand. Our results show that re-designing PVFS over the InfiniBand native transport layer is worthy with up to 3 times improvement over TCP/IP on the same IBA network when

performance of the local file system is well balanced compared to the network system. The Normal and Mmap methods can be applied to PVFS over InfiniBand when we use the InfiniBand native transport layer, while we cannot use the Sendfile method directly. In addition, there are several issues to be addressed to further improve PVFS performance.

Data copying between different components: I/O data is copied between the file cache and PVFS server communication buffers. This happens when the Normal method is used. It also happens when we want to avoid dynamic memory registration and deregistration in the Mmap method. Data copying incurs high per-byte overhead for PVFS read and write operations.

Explicit communication buffer pool: To avoid expensive dynamic memory registration and deregistration, an often used solution is to pre-register a list of buffers and to keep using them for all communication. To serve a large number of requests concurrently, a significant amount of memory space should be allocated. Since these buffers are not swappable, they actually reduce the effective size of main memory, and thus the size and hit rate of the server's file cache.

Data duplication in communication buffers: When we use an explicit communication buffer pool, a same data object may be in multiple communication buffers to serve different requests which access the same data object. This duplication reduces the efficiency of the communication buffers, leading to a possible increase of the communication buffers and service stalls.

Dynamic memory registration and deregistration: This happens when we use the Mmap method. As shown in Chapter 4, up to 35% performance can be degraded due to the costs of memory registration and deregistration.

These issues have a root in the lack of integration and interaction among the PVFS transport layer over InfiniBand, the file/storage component, and the underlying I/O subsystem. To solve these issues, we propose a component to unify the communication buffer space and the cache space. We deploy an application-level cache in this component. The cache space is directly used for communication. We call this component *Unifier*, working as both a cache manager and a communication buffer manager. It provides pre-registered communication buffers without reducing the effective cache size. It also offers other features to enable better cooperation with related components. We describe the detailed design of Unifier in Section 5.2 and a prototype implementation in Section 5.3.

5.2 The Design of Unifier

In this section, we present the design of Unifier. We start with its basic software architecture and its application programming interface (API), followed by its potential benefits and design issues.

5.2.1 Basic Software Architecture

Unifier is designed to provide efficient interaction between components in PVFS I/O servers. The basic architecture and its interaction with other components are shown in Figure 5.2.

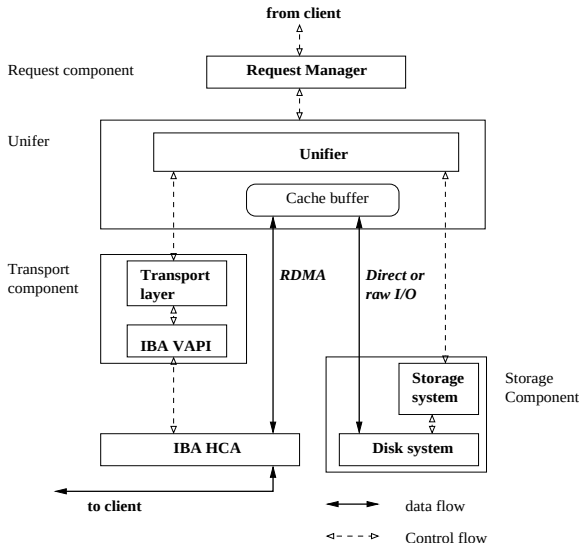


Figure 5.2: Basic software architecture of Unifier.

The control flow is shown by the dotted lines in Figure 5.2. Unifier, as a central hub, interacts with the request manager, the transport component, and the storage component. First, it receives requests from the request manger. Second, it provides cache buffers to serve these requests. Lastly, for a read request, it first talks to the storage component to read the requested data into its cache buffer if data is not cached. Then, it provides the same buffer to the transport component to transmit data to the client. For a write request, it first asks the transport component to receive data into its cache buffer. These data then is cached in the Unifier's cache buffer and flushed to the storage component at appropriate time.

The data flow is shown by the solid line. The data flow is simple. All data is placed in the Unifier's cache buffers. The cache buffers are also used by the transport component for communication, as well as the storage component for file and storage I/O operation. Given a data object, there is only one copy in the Unifier's cache buffers shared by all components safely and concurrently.

Unifier provides two main functionality. First, it acts as a cache manager, maintaining an application-level cache. It also hides the details of the storage component. Second, it acts as a buffer manager, providing buffers to the transport component.

The cache buffer pool is managed in a way to enable efficient RDMA operations. Further, it intends to optimize cache management for better network performance, such as buffer coalescing and variable cache units.

5.2.2 Unifier Interface

The underlying observation that shapes our design of the Unifier API is that a high-performance API should adopt the lessons learned from the design of the high-performance server architectures. As a result, we provide the following features in the Unifier API.

Supporting structured data access: Structured data access is a common access pattern in many applications. Native structured data access support in each component is a key for high performance [90, 110, 25]. The Unifier API should cater to this requirement and enable possible optimizations for structured data access.

Supporting asynchronous operations: Asynchronous operations provide opportunities to overlap I/O operations with other processing. Network I/O operations in IBA are asynchronous. File and storage systems have been evolving to provide asynchronous I/O support [11]. Unifier API should provide an interface to support asynchronous operations and to take advantage of the advances in both network and storage I/O.

A more expressive interface: Significant research work has pointed out that narrow interfaces in the existing systems have become a barrier for different subsystems to exchanging their semantic information to improve system performance [38, 83, 6]. A more expressive interface is expected, which allows more cross-subsystem optimizations and more flexible extended services.

Recognizing the importance of these features, we define a simple yet powerful Unifier's interface. This subsection briefly describes its interface. A complete discussion of the whole interface can be found in the PVFS2 document [70]. Currently, the interface includes five types of calls: 1) Post a request; 2) Check the request completion; 3) Query cache information; 4) Completion notification; 5) Release resources. As an example, we use `Unifier_post_read` to show how we achieve the aforementioned features in the Unifier API.

```
Unifier_post_read(int fd,
  ACCESS_Agg *    access_info ,
  BUFFER_Agg *    buffer_info,
  INFO_Agg *      semantic_info,
  COMP_Info *     comp_info)
```

In `Unifier_post_read`, *ACCESS_Agg* aggregates information of a structured access. This aggregate structure can be easily represented by an MPI Datatype if other components accept Datatype directly [25], or a representation of structured access.

INFO_Agg contains semantic information the caller wants to pass to Unifier. Currently, we only support cache policy selection and the cache unit size. We intend to extend this to convey more information to Unifier for optimization and for differential requirements. *COMP_Info* guides Unifier to set up the completion notification. The Unifier_post_read operation returns buffers which hold the requested data. We use *BUFFER_Agg* to aggregate a list of buffers. These buffers will be provided to the transport component for communication.

5.2.3 Potential Benefits

The primary goal of Unifier is to improve the performance of PVFS I/O servers. It offers the following potential benefits.

1. **Zero-copy I/O serving:** Unifier eliminates data copying between PVFS server components in the I/O path. Further, it maintains an application-level cache which enables the storage component to bypass the operating system file/storage cache without losing performance. Therefore, Unifier can achieve the minimal number of data copies to the extent permitted by the hardware. Zero-copy I/O serving path is easily achieved in a typical I/O server hardware setup over InfiniBand, as shown in Figure 5.2.

2. **Increased cache size:** Unifier eliminates all multiple buffering. Each object can have only one single copy in the Unifier's cache buffer. This actually increases the effective cache size, and thus the cache hit rate. Considering the increasing gap between the memory system and the disk system and the increasing gap between the network system and the disk system, a small increase in the cache hit rate can improve the performance of I/O intensive applications significantly.

3. **Reduced memory registration and deregistration costs:** A part if not all of the cache buffers in Unifier can be pre-registered for communication without any memory registration or deregistration cost on these buffers.

4. **Native structured data access support:** We bear the structured data access support in mind from the beginning when we design Unifier. This support not only fits application common access patterns well, but also provides tremendous optimization potential in both Unifier and other components. For example, the storage component can perform optimizations such as active sieving on a structured data access as discussed in Chapter 3.

In this Chapter, we focus on the above benefits. Many other potential benefits, such as providing cache information to the request scheduler for cache-aware scheduling, application-controlled caching policies, and moving hot data into the memory of the IBA Channel Adapter, are not discussed.

5.2.4 Design Issues

Unifier and the Unifier-based I/O server software architecture show very attractive potential benefits. However, several issues need to be addressed for this architecture to be used in real systems to achieve high performance. We consider the following three important issues, namely adaptive PVFS I/O server cache, buffer sharing, and the size of registered cache buffers.

1. **Adaptive PVFS I/O server Cache:**

   Application-level cache has been popularly used in many server applications, such as database management applications, web server applications [94], and Grid data servers [10], We could borrow these designs into the design of PVFS I/O server cache. We could also reuse the design of the system cache for general-purpose systems. However, the reason why we consider the design of PVFS I/O server cache is an issue is that applications using PVFS have different I/O workload characteristics and I/O requirements from that on other systems [96]. Compared to database applications, PVFS applications may have more diversified access patterns. On the other hand, compared to applications on general-purpose systems, PVFS applications may have less variation in access patterns. Therefore, the design of PVFS I/O server cache should reflect these differences and provide high performance in general. An adaptive cache to cater to various requirements is expected.

   There is no "one size fits all" solution for a cache with fixed policies [82]. In our design, we attempt to increase the cache adaptivity from two aspects. First, we explicitly expose cache information to other components. Research work in [16, 6] has shown that applications can adapt their own behavior to that of the OS for improved performance with cache information. Unifier provides explicit cache information queries to enable adaptation. Second, we allow applications to specify their cache requirements. These requirements are passed down to Unifier. Consequently, different cache policies can be applied, different cache units can be used. Note that Unifier only provides best-effort services to these requirements. It is possible that some of them may be overruled [18].

2. **Buffer Sharing:**

   In Unifier, network read and write and file/storage read and write all share a single copy of a given data object. This results in problems of synchronization and consistency in buffer sharing. Techniques such as *immutable buffers* used in *IO-Lite* [65] can be used to solve these problems. Immutable buffers provide read-only buffer sharing to eliminate synchronization and consistency problems at the price of the data which can not generally be modified in place. As also mentioned in IO-Lite, immutable buffers is not suitable for scientific applications where in-place modification is a must.

Considering scientific applications being the main target of PVFS, we propose other means to solve the buffer sharing problems. We use an *allocate-release* model to manage and control sharing on the cache buffers. The main design points are as follows:

Single owner: The only owner of all cache buffers is Unifier. This implies that Unifier has control on all buffer sharing. This method reduces the design complexity significantly.

Allocate: Unifier allocates the cache buffers to each operation. When a conflict sharing occurs, the allocation will be deferred. When there is no conflict sharing, the same cache buffers may be allocated to several concurrent operations. This enables safe and concurrent sharing.

Release: When an operation is granted with the cache buffers, it should release these buffers to Unifier when it completes.

With this design, Unifier supports both read-only sharing as well as write sharing. I/O data can be modified in place if it is not currently shared. Therefore, Unifier provides not only the sendfile semantics over InfiniBand transport protocols, which transmits data in the cache buffers directly to the network without any copy, but also a *recvfile*-like support that data received by the network is placed directly into the cache buffers which are associated with a data object in file/storage systems.

There are three reasons why we support the recvfile-like semantics which is not supported by the operating system on the traditional network protocols. First, the IBA network performance is becoming comparable to the bandwidth of the memory system. Second, RDMA operations provide a "shared-memory illusion". To some extent, a process on a remote machine could be equally considered as a local process running on the same machine. Third, write sharing is very little in parallel applications [96]. A PVFS write can be done without affecting others. Therefore, providing recvfile-like support over InfiniBand can improve performance of PVFS writes without costs in common cases. Even when write sharing does occur, since the network performance is high, the cost to maintain writing sharing is low.

3. **The Size of Registered Cache Buffers:**

   Another main goal of Unifier is to reduce memory registration and deregistration cost imposed by RDMA operations. Ideally, a part (if not all) of the cache buffers can be registered and be always ready for RDMA operations. However, there are several tradeoffs to be addressed to achieve this objective. First, the size of Unifier's cache should be as large as possible. Unifier should use all free memory as cache to increase cache hit rate. Due to dynamic memory demands,

a static size may cause virtual memory penalties. Second, as many cache buffers as possible should be registered during the cache initialization. However, the size of registered cache buffers should be limited not to degrade the system performance. Because registered buffers are pinned and not swappable, the effective size of physical memory used for other purposes is reduced.

In our design, the cache buffers are divided into two groups: *Ready* and *Raw*. Ready buffers are registered and resident in the system during the Unifier's life time. The size of Ready buffers is projected conservatively according to the estimate of memory needed by a PVFS server application with its maximum support of outstanding requests. Raw buffers are not registered. Communication on these buffers needs on-the-fly registration and deregistration. The size of raw buffers is the total physical memory size subtracted by the size of Ready buffers and the size of memory needed by a PVFS server application with a light load. With this design, we can achieve a good tradeoff between the cost of memory registration and deregistration and the cost of potential virtual memory activities.

5.3 Implementation

This subsection gives an overview of the implementation of the Unifier component and its deployment in PVFS over InfiniBand.

Unifier is implemented as a user-level component in PVFS software architecture [70, 109]. As a prototype implementation, the cache implementation is mostly based on the file cache implementation in Linux 2.6. Our implementation supports variable cache unit sizes from 4 KBytes to 64 KBytes. Applications can advise Unifier to choose a cache unit size for a file when the file is first opened. Unifier provides both polling and callback completion notification. The callback completion notification depends on the support of callback completion notification provided by the underlying storage component. To support structured data access, our current implementation uses the <*offset, length*> to represent a structured data access and a list of cache buffers. This is compliant with both PVFS1 and PVFS2 implementation where the request manager interprets the high-level abstraction (e.g. MPI Datatype) of structured data access.

The deployment of Unifier in PVFS is straightforward, as shown in Figure 5.2. In the current implementation, Unifier provides explicit information queries to the request manager. However, how to make use of the cache information is under study. We are also working on the adaptive cache management.

Apparently, our design and implementation are not without problem. In the current implementation, we assume that the PVFS I/O server is multi-threaded. Then a single cache can be shared by all threads. Significant modifications are needed in Unifier to provide cache sharing for a multi-process server architecture. We also

consider a dedicated PVFS I/O server configuration for better performance. The total cache size and the size of pre-registered cache buffers depend on the dedicated configuration for accurate estimate.

5.4 Experimental Results

In this subsection, we provide three sets of results. First we show the basic results of the network, the file system, and the memory system. Next, we compare the micro-benchmark level performance of Unifier with the Normal and Mmap methods. Lastly, we analyze the performance of PVFS implementation over InfiniBand with the deployment of Unifier.

All our experiments used the following experimental testbed. A cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets 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-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_18_0000. We used the Linux 2.4.7-10 kernel. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes.

5.4.1 Basic System Performance Results

Performance realized by PVFS applications depends on the performance of three main subsystems: the network, the memory, and the file system. Table 5.1 compares the throughput of IBA VAPI Send/Recv, RDMA Write, RDMA Read, IPoIB, memory copy, file read and write with and without cache. In the IBA throughput tests, memory registration and deregistration costs are not included. In the memory copying test, the amount of data copied is 20 MBytes, much larger than L1 and L2 caches to eliminate the cache effect.

Memory registration and deregistration costs are crucial for us to leverage InfiniBand features. Figure 4.1 shows these costs with different buffer sizes using Mellanox fast memory registration extension in VAPI [57]. Note that much higher costs should be paid if we use VAPI regular memory registration facilities. Figure 4.2 shows the impact of dynamic memory registration and deregistration. This is the reason why we make great effort in Unifier to reduce these costs.

It can be seen that there is a large difference in bandwidth realizable over the network and the memory system compared to that which can be obtained to a disk-based file system without cache effect. 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

Table 5.1: Throughput of different subsystems

Subsystem	Throughput (MB/s)
VAPI Send/Recv	830
VAPI RDMA Write	830
VAPI RDMA Read	826
Memory Copying	596
File Read w/o cache	20
File Write w/o cache	25
File Read w/i cache	590
File Write w/i cache	183

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 and independent of any disk activities. We consider data is cached. The results 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 from a well-balanced storage system.

5.4.2 Performance of Micro-benchmarks

In this subsection, we present the designs of several micro-benchmarks to show the performance of Unifier. We put Unifier in a simple client-server environment, which is similar to the PVFS architecture but simpler. In these tests, a client sends one or more read or write requests to a server. The server then serves these requests using three different methods: *Normal*, *Mmap*, and *Unifier*, respectively. Details of Normal and Mmap methods are discussed Section in 5.1.2.

Cached read performance: We first measured the cached read performance of these three methods. In this test, all data is in the system cache in the Normal and Mmap method. All data is also in the *Ready* cache buffer in the Unifier method. We used this test to show the best case performance of all methods.

Figure 5.3 shows the results. The Normal method gives a peak bandwidth of 324 MBytes/sec. We see a small drop when the access sizes are larger than 128 KBytes, probably this is because the increase of the memory footprints affects the memory copy performance.

In the Mmap method, the memory registration and deregistration costs have a significant impact, particularly for small access sizes. When the access size increases, the costs of memory registration and deregistration become less than the cost of memory copy. Thus, this method performs better than the Normal method.

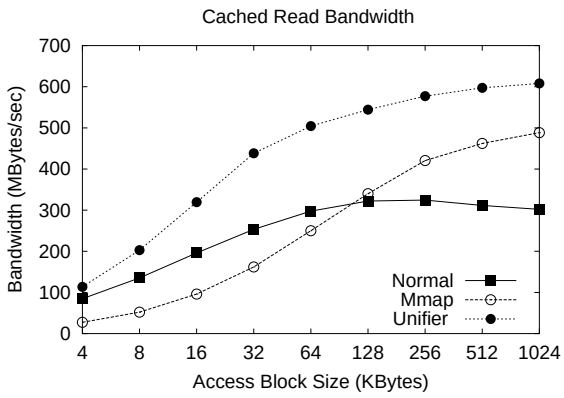


Figure 5.3: Cached read bandwidth.

In the Unifier method, data is cached in the Unifier Ready cache buffers. Thus, the server can RDMA write data directly to the client buffer from its Unifier's cache buffers. Unifier achieves an improvement of a factor of 2.1 over the Normal method, a factor of 1.3 over the Mmap method when the access size is large, a factor of up to 2.7 over the Mmap method when the access size is small.

Effects of cache size: As discussed earlier, the effective cache size in each method is different. Given a system with 512 MBytes physical memory, the maximum size of memory which can be used for cache is around 420 MBytes. In our test, the server application consumes around 60 MBytes. Then around 360 MBytes memory can contribute to cache data. The Mmap and Unifier methods can make full use of these 360 MBytes for caching. However, since we need some pre-registered communication buffers in the Normal method, we allocate 20 MBytes for this use, thus, the effective cache size is around 340 MBytes. Note that to allow the server to serve a large number of concurrent requests in a real PVFS configuration, even a larger buffer pool may be needed. In the Unifier method, the maximum size of Ready buffers allowed by the system is around 200 MBytes. So that around 160 MBytes of Raw buffers are in the Unifier cache, which requires dynamic registration and deregistration.

We used a *re-read* test to show the effects of cache size. In this test, the client reads a file whose size varies from 300 MBytes to 400 MBytes. This test reads a file sequentially with the block size of 128 KBytes. Then, it reads the same file again sequentially. The bandwidth achieved by the second read is reported in Figure 5.4. We can see that both the Mmap and the Unifier methods can still hold the entire file in the cache when its size is not larger than 360 MBytes, while the Normal method can not. When the file size increases to 380 MBytes, all methods suffer due to the disk-bound access on a normal IDE disk which can offer a read bandwidth of 20

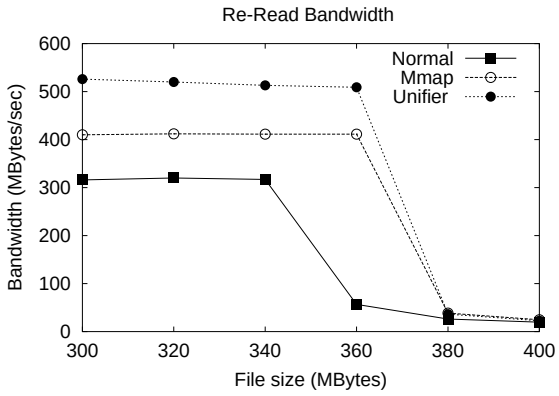


Figure 5.4: Effects of cache size.

MBytes/sec. All methods are comparable. This also shows that the Unifier cache can provide comparable performance to the system cache with the sequential workload.

5.4.3 Performance of PVFS1 with Unifier

The test program used is *pvfs-test*, which is included in the PVFS release package. We followed the same test method as described in [19]. That is, 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. The number of I/O nodes was fixed at four, and the number of compute nodes was varied from one to four.

Figure 5.5 shows the cached read performance with different methods deployed in an implementation of PVFS over InfiniBand VAPI as discussed in 2. The aggregate bandwidth realized by all clients is reported. There are two observations. First, PVFS with Unifier scales better than other two methods. This is due to the lower CPU overhead needed to serve each request in the Unifier method. In other methods, either memory copying or memory registration and deregistration consumes significant CPU cycles. Second, in terms of the peak bandwidth, the Unifier achieves an improvement of a factor of 1.7 over the Normal method, a factor of 1.3 over the Mmap method.

5.5 Summary

Unifier is designed to improve the performance of network storage server applications. It provides three notable features. First, Unifier eliminates redundant data copying and multiple buffering in the I/O path. It provides a single data sharing

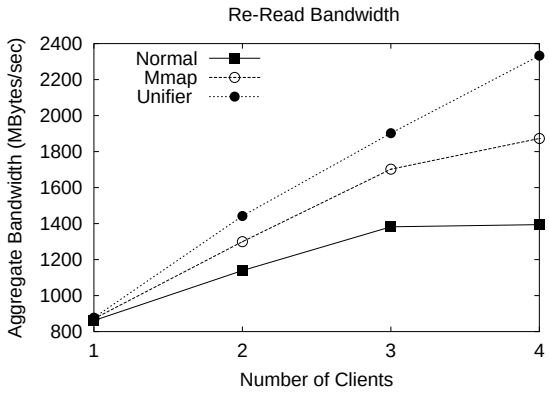


Figure 5.5: PVFS cached read performance.

among all components in a server application safely and concurrently over high performance networks such as InfiniBand. Second, the integration of communication buffer management and cache management reduces memory registration and deregistration costs as much as possible. This enables applications to take full advantage of RDMA operations. Third, Unifier provides means to achieve adaptation, application-specific optimization, and better cooperation among different components in a server application.

This chapter presents the design and implementation of Unifier. We also deployed and evaluated this component in a version of PVFS1 implementation over InfiniBand. Experimental results from a prototype implementation show performance improvements between 30 and 70% over two other methods often used in the PVFS I/O server implementation. Better scalability is achieved by the PVFS I/O servers. The Unifier method also increases the effective cache size due to the integration of communication buffers and the cache buffers, leading to increased performance.

CHAPTER 6

FAST DEMOTION-BASED EXCLUSIVE CACHING THROUGH DEMOTE BUFFERING

This chapter presents *Demote Buffering* to improve performance of Demotion-based exclusiving caching. With DEMOTE buffering, a small portion of buffer space is used to delay DEMOTE operations. When an evicted block needs to be sent to the server cache, the block is first placed in the DEMOTE buffer. DEMOTE buffering can mask the DEMOTE overheads, to smooth the variance (burstiness) in the DEMOTE traffic, and to provide more flexibility for optimizations. The design space for optimizations is broadened, including non-blocking network operations, remote direct memory access (RDMA), RDMA Gather/Scatter operations in networks such as InfiniBand [46, 69], and speculating demotions.

6.1 Multi-level Cache Hierarchy in Networked Storage Systems

Caching is designed to shorten access paths for frequently referenced items, and so improve the performance of the overall file and storage systems. With the increasing gap between processors and disks, and decreasing memory price, modern file and storage servers typically have large caches up to several or even tens of gigabytes to speed up I/O accesses [107]. In addition, the clients of these servers also devote a large amount of memory for caching [115, 4, 61, 62]. Multiple clients may share file and storage resources through various storage networks. A typical scenario is a two-level hierarchy: the lower level cache can be a high-end disk array cache or a cluster file server cache, and the upper level can be a database server cache or a file client cache. We call a lower level cache a *server cache*. In contrast, we call an upper level cache as a *client cache*.

6.1.1 Inclusive and Exclusive Caching

Most cache placement and replacement policies used in multi-level cache systems maintain the *inclusion property*: any block in an upper level buffer cache is also in

a lower level cache. The drawbacks of inclusive caching have been observed in a rich set of literature [60, 116, 107, 23]. To aggregate the cache size of the multi-level cache hierarchy to achieve *exclusive caching*, different approaches have been proposed. Wong and Wilkes [107] recently proposed a simple operation called *DEMOTE* as an additional interaction between a client cache and a disk array cache to achieve *exclusive re-read cache*. The DEMOTE operation is used to transfer evicted data blocks from the client buffer cache to the disk array cache. Then the server cache uses different cache replacement policies for the demoted blocks and blocks recently read from disks to yield exclusive caching. Chen *et al* [23] proposed an eviction-based cache replacement to achieve exclusive caching. Using the eviction-based cache placement policy, the server cache tries to reload blocks from disks when these blocks are evicted from the client cache. Unlike the Demotion-based exclusive caching, the eviction-based cache placement attempts to retain the current interface between the client and the storage server. An extra software is installed in the client side to send eviction information to the storage server, Thus, a change in the interface to the storage server is also needed to receive the eviction information. Recent work on a non-invasive exclusive caching mechanism [7] achieves a high degree of exclusivity through gray-box methods: using system information to construct an approximate image of the contents of file system cache and uses this image to determine the exclusive set of blocks that should be cached in the storage server. This approach considers no change to the storage and file system interfaces as central to achieving cache exclusivity in multi-level cache hierarchy.

6.2 Demotion-Based Exclusive Caching

Wong and Wilkes [107] proposed a simple operation called *DEMOTE* as an additional interaction means between a client cache and a disk array cache to achieve *exclusive re-read cache*. The DEMOTE operation is used to transfer evicted data blocks from the client cache to the disk array cache. To achieve exclusive caching, the server cache should choose appropriate cache placement and replacement policies to manage blocks demoted from the client cache and blocks that have been read from the disk. One of exclusive cache schemes discussed is shown in Figure 6.1. The client cache uses the LRU policy to read blocks from the server cache. The server cache puts blocks it has sent to a client at the head (earliest to be discarded) of its LRU queue, while puts demoted blocks from the client cache at the tail. This cache management policy most closely achieves a single unified LRU cache [107]. Ideally, exclusive caching has the potential to double the effective cache size with client and server caches of equal size.

When a client is to discard a clean block from its cache to make space for other blocks, it first sends the block metadata in advance of the block data [106]. This control message can be used to avoid transferring the block data in some cases. For

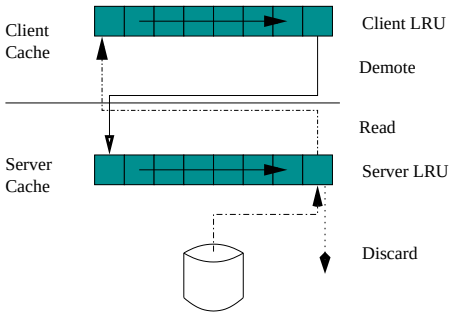


Figure 6.1: Cache management in the DEMOTE exclusive caching.

example, from the metadata, the server can determine whether or not it has cached the block, and if it has it can signal to the client to abort the transfer.

Demotion-based exclusive caching relies on transferring demoted blocks through network between the clients and the server. Further, the DEMOTE approach performs eager DEMOTE operations with assumption that the network is fast. This method offers design simplicity. However, it introduces the following performance overheads, which may offset the benefits of exclusive caching for some workloads and networks.

- Eager DEMOTE operations increase request access time. A request may be delayed because it needs to wait for the completion of a DEMOTE operation to make space for it. That is, the average client cache miss penalty will be increased due to the cost of DEMOTE operations.
- DEMOTE operations increase the network traffic. In the worst case, each read incurs a DEMOTE operation, the network traffic is more than doubled (some control traffic is also counted).
- Eager DEMOTE operations provide little design space for optimizations. A DEMOTE operation must be finished before a demand request can have space in the client cache. Therefore, features such as non-blocking network operations can not be applied. Besides, each block needs a control message. In addition, the size of cache blocks is usually small (e.g. 4 kB or 8kB), one cache block per DEMOTE operation may not utilize the network bandwidth efficiently.

We propose DEMOTE buffering to hide and/or reduce these overheads. DEMOTE buffering provides more flexibility for optimizations. These optimizations can make better use of network features, such as non-blocking network operations and gather/scatter operations. A control message in DEMOTE buffering can contain multiple blocks' metadata. This can reduce control traffic and amortize control cost over multiple blocks.

6.3 DEMOTE Buffering

In this section, we first describe the structure of DEMOTE buffering and its potential benefits. Then we discuss its design issues.

6.3.1 The Architecture of DEMOTE Buffering

In DEMOTE buffering, a small memory space, the *DEMOTE buffer*, is used on the client side for buffering demoted blocks, as shown in Figure 6.2. DEMOTE buffering works as follows: when a client encounters a cache miss and is about to demote a clean block from its cache (e.g. to make space for a READ), it first moves the demoted block into the DEMOTE buffer. This operation is a local operation. Then it initiates a request to the server cache to read the demanded block. Unless the number of demoted blocks in the DEMOTE buffer is up to a certain threshold, requests will not encounter any overhead of DEMOTE operations.

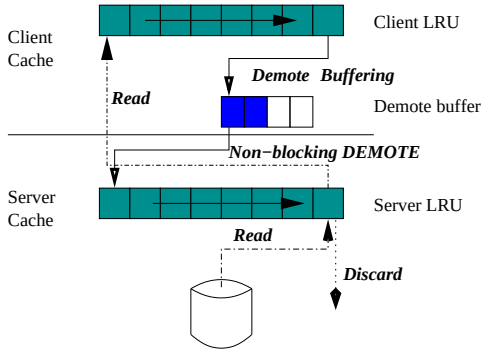


Figure 6.2: Architecture of DEMOTE Buffering.

Double-buffering technique is used in our DEMOTE buffering mechanism. The threshold could be half of the DEMOTE buffer size in blocks. Thus, when the DEMOTE buffer is half full, non-blocking DEMOTE operations are initiated.

When client cache misses are too bursty, it is possible that there is no space left in the DEMOTE buffer when a client needs to demote a block. The client cache then checks the completion of previously initiated demotions and reclaims resources for future demoted blocks.

6.3.2 Benefits of DEMOTE Buffering

DEMOTE buffering has the following potential benefits.

1. **Reduced visible DEMOTE cost:** DEMOTE buffering tries to hide the cost of DEMOTE operations by increasing the overlap between demotions and other activities. This is achieved through buffering demoted blocks in the DEMOTE buffer and using non-blocking I/O to perform DEMOTE operations.

2. **Exploiting idle network bandwidth:** DEMOTE buffering provides opportunities to use idle network bandwidth to transfer demoted blocks in the DEMOTE buffer. The network link between a client and a server is free when the client cache hits occur or the client is not performing I/O operations. If DEMOTE operations occur during this period, the impact of the increased traffic on the system performance is minimized. This benefit can be realized easily with one-sided communication such as RDMA operations in the InfiniBand network. For example, the server can monitor the network usage and then initiate RDMA Read operations to retrieve demoted blocks from a client's DEMOTE buffer when the link is free.

3. **Better network utilization:** In the DEMOTE buffering mechanism, multiple control messages can be aggregated into one single message for all buffered blocks. The control messages can be piggybacked with request and reply messages. Effective scheduling or batching on the transmission of multiple blocks in the DEMOTE buffer can be performed. For example, RDMA Gather/Scatter operations in InfiniBand can be used to retrieve multiple blocks in one operation, even though they are not contiguous. This can achieve better network utilization.

4. **More flexibility for optimizations:** Demote buffering enables more flexibility for optimizations. One example is speculating demotion. When the client cache misses are too bursty for the DEMOTE buffering to hide the DEMOTE costs, the client can only send metadata information of the demoted blocks to the server. The server then speculates about cold blocks [23] which are accessed infrequently and thus are unnecessary to demote from the clients. After the client receives the server's speculation information, it can replace those unnecessarily demoted blocks with the newly demoted blocks directly. Speculating demotion can be applied without DEMOTE buffering, however, the control message cost is significant. In Demote buffering, the control message cost can be amortized effectively across multiple blocks.

In summary, Demote buffering has the potential to hide the demotion cost through overlapping communication and other activities, to reduce the number of communication operations, to achieve better utilization of the network bandwidth, and to allow more flexible optimizations to reduce the impact of demotion cost on the system performance.

6.3.3 Design Issues in DEMOTE Buffering

The DEMOTE buffering mechanism shows very attractive potential benefits over the eager DEMOTE mechanism, however, several issues need to be addressed for this mechanism to be used in real systems to achieve high performance.

- **Reducing DEMOTE buffering overhead:** There are two ways to buffer a demoted block. One is to copy the demoted blocks into the DEMOTE buffer. The method can have demoted blocks in a contiguous memory space which can be used to optimize communication in some networks. There is no change to the client cache space. These advantages are achieved at the cost of memory copy. The second one is to exchange the positions of a free block in the DEMOTE buffer and a demoted block in the client cache. There is no memory copy. However, the client cache space and the DEMOTE buffer space change with time. Noncontiguous data transmission may occur [110]. Tradeoff must be made between the cost of memory copy and the performance of noncontiguous data transmission in the studied network.

- **Tuning the size of the DEMOTE buffer:** The size of the DEMOTE buffer affects the ability of the DEMOTE buffering mechanism to mask the demotion overhead. From the server cache point of view, a DEMOTE operation is similar to a WRITE operation, and the DEMOTE buffer is similar to a write-behind buffer. Ideally speaking, the DEMOTE buffer size must be large enough to cover the burstiness of the client cache misses. This is similar to a write-behind buffer to cover the variance in the write workload [95, 77]. On the other hand, the DEMOTE buffer should not consume too much memory since most memory should be devoted to the client cache.

- **Maintaining cache hits:** DEMOTE buffering introduces a new complication: the delayed demotions may result in cache misses in the client and server caches. To address this issue, first the DEMOTE buffer should be considered as a part of the client cache. When a client cache miss occurs, the client should look at the DEMOTE buffer to see if the requested block is stored. Thus, from maintaining the client cache hit point of view, the DEMOTE buffer works as a victim buffer in victim caching [48]. In a single-client system, this method solves the issue completely. However, it is a little bit more complicated in a multi-client system.

- **Handling remote client cache hits:** In a multi-client system, it is possible that the demoted blocks in one client's DEMOTE buffer may be expected by others. That is, the delayed demotions might decrease the server cache hits for some workloads. One solution to this problem is to let a server cache maintain

a directory of which blocks are in which client's DEMOTE buffer. Since the number of blocks in a DEMOTE buffer is small, the total space for this directory is limited. Then when a request from a client encounters a server cache miss, the server can potentially retrieve the block from a different client which is holding it in its demote buffer. Data transfer between clients is also possible, as with Cooperative Caching [31, 62, 98, 29].

6.4 Performance Evaluation

We developed a simulator to simulate the DEMOTE buffering between the clients and the server over various networks. For comparison, the original DEMOTE mechanism is also simulated. Our simulator is built over *fscachesim* [107] by adding communication details. The simulator takes synthetic workloads and traces as input.

6.4.1 Experimental setup

Our experimental testbed consists of a cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets 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-x86-0.1.2-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. Each node has a Fujitsu Ultra3 SCSI (Model MAM3184MC) disk, which is a 18.4GB and 15,000 rpm drive. We used the Linux RedHat 7.2 operating system.

We perform tests over three different networks: TCP/IP over Fast Ethernet (referred as *FE*), IBNice (TCP/IP over InfiniBand), and native InfiniBand (using the VAPI library). We intend to use these three networks to represent roughly 0.1 Gb/s, 1 Gb/s, and 10 Gb/s networks. Table 6.1 lists their minimum latency, maximum bandwidth, and effective bandwidth when the message size is 4 kB bytes.

Table 6.1: Network Performance

	FE	IBNice	VAPI
Latency (μs)	68	36	5.4
eff. Bandwidth (MiB/s)	11	109	718
max. Bandwidth (MiB/s)	11.2	130	831

6.4.2 Single-client synthetic workloads

Three synthetic workloads: Random, Sequential, and Zipf [15], are generated using the *fscachesim* package [107]. The size of cache blocks is assumed 4 kB, the client and server caches each have 16384 blocks.

We follow the following method as mentioned in [107] to compare DEMOTE buffering and DEMOTE mechanisms. We perform simulations over different networks with the above synthetic workloads. The size of the working set is 32768 blocks, 32767 blocks, and 49152 blocks for the Random, Sequential and Zipf workloads, respectively. In each test, the caches are "warmed up" with a working-set size set of READs. After the warm-up, another 10 timed READs are initiated. Time to randomly access a 4 kB disk block is set to 10 ms, the same value set in [107]. The DEMOTE buffer size is set to 10 cache blocks. The client cache hit ratios are 50%, 0%, and 86% for Random, Seqential, and Zipf workloads, respectively; with server cache hit ratios of 46%, 100%, and 9%. Note that these ratios are expressed as fractions of the total client READs. Since the cache hit results are important for us to understand the performance of DEMOTE buffering, we put these results in Table 6.2 for clearer reference.

Table 6.2: Client and server cache hit rates for single-client synthetic workloads.

Workload	client	server
Seq	0%	100%
Random	50%	46%
Zipf	86%	9%

The main metric for evaluating DEMOTE buffering is the mean latency of a READ at the client. DEMOTE buffering achieves same cache hits and server hits as DEMOTE does. The results in Table 6.3 show the mean latency of READs in each workload with DEMOTE (*DE* for short) and DEMOTE Buffering (*DB* for short) mechanisms.

It can be observed that DEMOTE buffering achieves the highest speedup (1.44x) for the Sequential workload. This is because there is no cache hit on the client, and all accesses are cached in the server. Each access results in sending a demoted block to the server and receiving a block from the server.

In Random workload, the number of demote operations is 50% of the size of the working set due to 50% client cache hit ratio. There are 4% blocks needed to be read from disk. DEMOTE buffering achieves considerable improvement on Fast Ethernet and IBNice. However, the benefit diminishes on VAPI because the total demotion overheads are less significant.

DEMOTE buffering achieves the least improvement in the Zipf workload. This is actually expected, because of the highest client cache hit ratio and the lowest server cache hit ratio. The client cache hit ratio is 86%, indicating that there are only 14% blocks needed to be demoted. The server cache hit ratio is 9%, indicating that there are 5% blocks needed to be read from disk. Since the disk access time is much higher than the network access time, the total demotion overheads become less significant in the Zipf workload.

Table 6.3: Mean READ latencies and speedups over DEMOTE for single-client synthetic workloads (DE: DEMOTE; DB: Demote Buffering)

		Seq (ms)	Random (ms)	Zipf (ms)
FE	DE	1.21	0.99	0.84
	DB	1.09 (1.11x)	0.92 (1.08x)	0.83 (1.01x)
IBNice	DE	0.26	0.52	0.73
	DB	0.18 (1.44x)	0.46 (1.13x)	0.71 (1.03x)
VAPI	DE	0.078	0.41	0.704
	DB	0.056 (1.4x)	0.40 (1.03x)	0.70 (1.01x)

We note that performance gain achieved by using DEMOTE buffering varies with the network performance, disk performance, as well as workload access patterns. It depends how significant the total demotion overheads are. It can be expected that workloads with high client cache miss rates can benefit more from DEMOTE buffering. In addition, the faster disks are, the more significance DEMOTE buffering is. Furthermore, the performance gain is closely related to the performance gap between network and disk. In Table 6.3, we see DEMOTE buffering achieves less improvement on VAPI than on IBNice, this is because VAPI performs 6.5 times better than IBNice and tens of times better than disk, the total demotion overheads over VAPI are less significant than those over IBNice.

6.4.3 Effectiveness of DEMOTE buffering

To show the effectiveness of DEMOTE buffering directly, we profiled the total demotion overheads visible to the client in our tests. Results are shown in Table 6.4. With a small DEMOTE buffer (10 blocks), up to 34% demotion overheads are reduced by DEMOTE buffering.

Note that in the above tests, one request is initiated immediately after the completion of the previous one. This incurs the highest burstiness of DEMOTE operations in all workloads studied. It is possible that client cache misses are too bursty to mask

Table 6.4: Total demotion overheads for single-client synthetic workloads

		Seq (s)	Random (s)	Zipf (s)
FE	DE	12.76	10.65	1.91
	DB	8.30	7.44	1.52
IBNice	DE	9.45	4.70	1.70
	DB	6.39	3.23	1.14
VAPI	DE	0.95	1.81	0.33
	DB	0.70	1.30	0.26

the DEMOTE overhead. That is, some of requests should wait for the completion of non-blocking demotions for spaces. Thus, the results in Table 6.4 are actually the worst-case results for DEMOTE buffering. In the next subsection, we show that DEMOTE buffering can achieve better overlap if we reduce the access rate.

6.4.4 Effects of Burstiness

The sequential workload results in the most bursty demote operations since each READ incurs a DEMOTE operation. To study how well DEMOTE buffering can mask the DEMOTE overheads under different client cache miss burstiness, we put some computation delay after each client read request. We expect that the larger the delay is, the better DEMOTE buffering can overlap demotions with the computation delay. Consequently, the visible DEMOTE overheads to the client decrease. In contrast, each read in the eager DEMOTE approach must pay the demotion cost no matter what the delay is.

The results in Figure 6.3 validate our expectations. The overhead visible to the client is expressed as the fraction of the total overhead visible to the client in the eager DEMOTE, shown in y-axis. The delays between two consecutive requests are shown in x-axis. First, DEMOTE buffering effectively reduces the demotion overheads visible to the client even with small delays. Second, the delay at which the best overlap is achieved is adversely proportional to the network bandwidth. For example, when the delay is 500 μs, DEMOTE buffering on Fast Ethernet can offer the best overlap, while similar benefits arises with 100 μs for IBNice, and 50 μs for VAPI. Third, another interesting observation is that still 36% and 32% overheads are visible to the client on Fast Ethernet and IBNice even with large delays, while only 12% overhead visible to the client on VAPI. This is due to different CPU overheads needed to transfer data in the three networks studied. Both fast Ethernet and IBNice use the kernel-based TCP/IP stack which incurs substantial overheads due to context switching

and memory copies. In contrast, VAPI provides user-level networking which requires much less interference from the host CPUs and operating system.

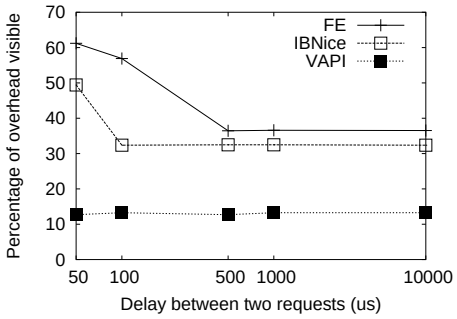


Figure 6.3: Demotion overhead visible to the client with different request rates.

6.4.5 The Single-client DB2 workload

We used the DB2 workload [97] to evaluate the benefits of DEMOTE buffering for real-life workloads. The DB2 traces were generated by an eight-node IBM SP2 system. The size of data set is 5.2 GB. The eight client nodes access disjoint sections of the database. For single-client test, the eight access streams are combined into one. Unlike the above-mentioned tests, in this and the next tests, the disk accesses are not simulated. We used a Fujitsu Ultra3 SCSI (Model MAM3184MC) disk, which is a 18.4GB and 15,000 rpm drive. DB2 exhibits a behavior between the sequential and random workload styles [107]. The results of mean latencies achieved with different DEMOTE buffers are shown in Figure 6.4. Data points with zero block are results of DEMOTE. Overall, a 1.10 to 1.15x speedup over DEMOTE is achieved for Fast Ethernet and IBNice on InfiniBand, a 1.05x speedup for VAPI on InfiniBand. The mean latencies on IBNice and VAPI have little sensitivity to the size of DEMOTE buffer.

6.4.6 The Multi-client HTTPD workload

The HTTPD workload [97] was collected on a seven-node IBM SP2 parallel web server serving a 524 MB data set. There is a significant portion of blocks shared by clients. In this study, we evaluate the impact of DEMOTE buffering on the server cache hit ratio. For comparison, we implemented two methods to process server cache misses. One is to reload the missed blocks from disks, referred as *Reload*. The second one tries to retrieve the missed blocks first from the clients' DEMOTE

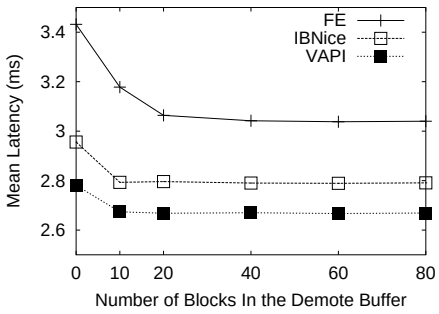


Figure 6.4: Mean latencies of the DB2 workload

buffers, referred as *Retrieve*. To keep the server updated of which blocks are in clients' DEMOTE buffers, the client eagerly piggybacks metadata information of the demoted block to the server cache in the request message.

In our test, each client has an 8 MB cache. The server cache is 64 MB. This configuration results in 62% client cache hits on average, 12% server cache hits, and 16% server cache misses with the DEMOTE approach. In DEMOTE buffering, when the DEMOTE buffer is less than 40 blocks, the server cache misses remain same. When the DEMOTE buffer size is set to 80, the average client cache hit ratio is 63%, 9% server cache hit ratio, and 18% server cache miss ratio. Figure 6.5 shows the aggregated throughput (the total number of HTTPD requests finished per second) of seven clients with different DEMOTE buffer sizes. We use "1" to represent the Reload method, and "2" for the Retrieve method. In Fast Ethernet (FE), Reload performs better than Retrieve because of the poor network performance. In both IBNice and VAPI, the network access is much faster than the disk access; hence, retrieve performs better than Reload. Although there is some increase of the server cache misses when the DEMOTE buffer becomes large, the DEMOTE buffering with Retrieve scheme still provides better performance than the eager DEMOTE approach whose results are shown by data points with zero block in the figure. A speedup up to 1.08x can be achieved. For example, the eager DEMOTE can support 9859 ops/sec on IBNice, while DEMOTE buffering can support 10670 ops/sec, a factor of 1.08 improvement.

6.5 Summary

DEMOTE buffering is proposed to hide the cost of DEMOTE operations by increasing the overlap between demotions and other activities in demotion-based exclusive caching. It also provides more flexibility for optimizations, such as non-blocking operations, aggregate of control messages, gather/scatter network operations, and

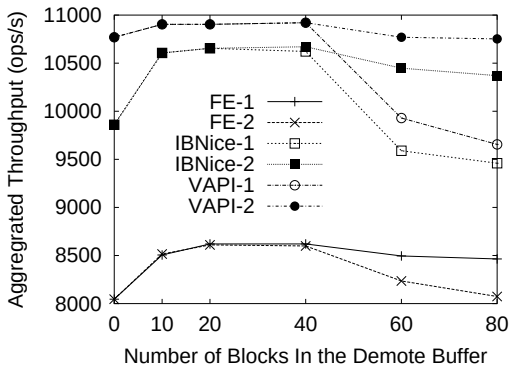


Figure 6.5: HTTPD workload aggregate throughput

speculating demotions. Results of experiments with synthetic workloads demonstrate that 1.11-1.44x speedups are achieved for the Sequential workload, up to 1.13x speedups for the Random workload. Simulation results with real-life workloads validate the benefits of DEMOTE buffering by 1.08-1.15x speedups over the DEMOTE approach.

The exclusive caching motivates us to aggregate resources from the NICs, main cache, and the disk/disk array cache.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this dissertation we have designed a high performance and highly adaptive networked storage system over emerging networks to cater to the requirements of different applications. We approached this problem by focusing on the management of communication and memory in the I/O path of networked storage systems. Our research has shown that emerging networking technologies and storage architectures have a profound impact on the design and implementation of networked storage software. Specifically, the RDMA-based networks provide high speed and low CPU overhead data movement for networked storage systems. They also increase the system scalability. However, RDMA operations have introduced new issues such as memory registration and deregistration and communication buffer management. This dissertation investigates approaches to solve these new issues and evaluates different tradeoff to make the most out of the RDMA benefits for networked storage systems.

Secondly, this dissertation presents a novel architecture to provide efficient interaction between different components such as the communication subsystem, the file system, the file cache in both clients and servers. The lack of efficient interaction between these components in the general operating systems is one of performance bottlenecks in networked storage systems. The low level primitives provided by RDMA networks can be used to achieve the best performance. However, building the networked storage systems on these low level primitives deteriorates the existing interaction without new designs. We approached this problem by providing integrated communication buffer management and cache management. This approach not only provides efficient interaction between storage server application components and different subsystems, but also takes advantage of the high performance of low level primitives of RDMA networks.

Thirdly, this dissertation introduces a buffering mechanism to achieve efficient exclusive caching in multi-level cache hierarchy which is often formed in networked storage systems. The exclusive cache memory management makes the better use of memory resources in different cache levels.

The main conclusion of this dissertation is that using innovative methods to manage communication and memory can significantly improve performance and scalability of a networked storage system. To achieve this requires studying and taking advantage of the new features in the emerging networking technologies and storage architectures, understanding new issues and performance bottlenecks, investigating different approaches, and evaluating various tradeoffs.

7.1 Summary of Research Contributions

We presented communication and memory management in networked storage systems. Below, we summarize the contributions and results of this dissertation.

7.1.1 Contiguous Data Movement using RDMA

In Chapter 2, we presented the design, implementation and performance evaluation of PVFS over InfiniBand. We studied how to leverage the emerging InfiniBand technology to improve I/O performance and scalability of cluster file systems. 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, including communication management for choosing appropriate transfer mechanisms and communication buffer management.

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.

7.1.2 Non-Contiguous I/O Access Support

We presented our Non-Contiguous I/O access support in Chapter 3. In this chapter, we addressed two issues involved in noncontiguous I/O accesses in cluster file systems over high performance networks: noncontiguous data transmission and noncontiguous disk accesses. For noncontiguous data transmission, we propose a novel approach, *RDMA Gather/Scatter*, to transfer noncontiguous data between the clients and the I/O servers. For noncontiguous disk accesses in the I/O server nodes, we have implemented a new scheme termed as *Active Data Sieving* to reduce disk access costs for a large number of small and noncontiguous accesses. Unlike other data sieving implementations, a cost model is used by the I/O nodes to actively and intelligently decide whether it is beneficial to perform data sieving or not.

We have designed and incorporated these approaches in a version of PVFS over InfiniBand. Our results show a performance improvement of up to 1.5 times for

the RDMA Gather/Scatter approach with Optimistic Group Registration on PVFS list I/O performance compared to the other approaches. Intelligent and active data sieving on the I/O node achieves a factor of 1.3–1.9 improvement on small noncontiguous I/O accesses. The NAS BTIO benchmark performance results show that our approach attains a 20% improvement compared to the best result across all other approaches in an environment which is a complex combination of noncontiguous data transmission and noncontiguous I/O accesses.

7.1.3 Efficient Memory Registration and Deregistration

Memory registration and deregistration for networks with remote DMA capabilities adds a new dimension to data movement for I/O intensive applications in the networked storage system. In a contiguous data movement, we have observed that up to 35% performance can be degraded if dynamic registration and deregistration are not avoided. We designed a two-level architecture as Fast Memory Registration and Deregistration (*FMRD*) scheme to reduce the registration and deregistration costs. Our performance evaluation shows that FMRD outperforms other existing approaches. Particularly, it works much better than others with I/O intensive applications.

Memory registration and deregistration on a list of buffers in a non-contiguous I/O access is even more challenging due to the large number of buffers and the gaps between these buffers. Our scheme, Optimistic Group Registration (OGR), maintains a good tradeoff between the number of registration and deregistration operations and the total size of memory space to be registered and deregistered. Our performance evaluation shows that Optimistic Group Registration is a necessity for us to achieve efficient non-contiguous I/O access over InfiniBand.

7.1.4 Unified Buffer and Cache Management

Integration and cooperation among different components are a general issue for many systems. Our work has demonstrated that the conventional Operating Systems and applications provide little support to achieve efficient integration and cooperation, especially for networked storage systems. We designed Unifier to improve the performance of network storage server applications. It provides three notable features. First, Unifier eliminates redundant data copying and multiple buffering in the I/O path. It provide a single data sharing among all components in a server application safely and concurrently over high performance networks such as InfiniBand. Second, the integration of communication buffer management and cache management reduces memory registration and deregistration costs as much as possible. This enables applications to take full advantage of RDMA operations. Third, Unifier provides means to achieve adaptation, application-specific optimization, and better cooperation among different components in a server application.

Chapter 5 presents the design and implementation of Unifier. We also deployed and evaluated this component in a version of PVFS1 implementation over InfiniBand. Experimental results from a prototype implementation show performance improvements between 30 and 70% over two other methods often used in the PVFS I/O server implementation. Better scalability is achieved by the PVFS I/O servers. The Unifier method also increases the effective cache size due to the integration of communication buffers and the cache buffers, leading to increased performance.

7.1.5 Fast Demotion-Based Exclusive Caching through Demote Buffering

In Chapter 6, we present DEMOTE buffering to to hide the cost of DEMOTE operations to achieve efficient exclusive caching. DEMOTE buffering increases the overlap between demotions and other activities in demotion-based exclusive caching. It also provides more flexibility for optimizations, such as non-blocking operations, aggregate of control messages, gather/scatter network operations, and speculating demotions. Results of experiments with synthetic workloads demonstrate that 1.11-1.44x speedups are achieved for the Sequential workload, up to 1.13x speedups for the Random workload. Simulation results with real-life workloads validate the benefits of DEMOTE buffering by 1.08-1.15x speedups over the DEMOTE approach.

7.2 Future Research Directions

Networked storage systems have become a mainline solution in data-centers and high performance computing systems. Many interesting research directions are still left to pursue. Below we describe some of these areas of future research.

Storage service with performance guarantees — Networked storage systems provide storage consolidation to applications for both ease of administration and economics of scale. This leads to sharing of storage resources across multiple workloads, corresponding to different applications/customers. Storage resources such as cache space, disk arms, disk controller cycles, and network bandwidth are shared and contended for, as shown in Figure 7.1. One of the central challenges in such a shared and networked storage environment is to provide *performance isolation* and to achieve *Quality of Service (QoS)* [40, 14, 105]. Independent workloads should be isolated from each other and their performance should be guaranteed in a predictable manner. For example, clients 1, 2 and 3 in Figure 7.1 expect that the storage server provides them I/O service with their required quality. On the other hand, the storage server must differentiate its services to different applications.

Providing storage service with QoS in networked storage systems has its unique challenges compared to QoS in networking and QoS in local storage systems. It

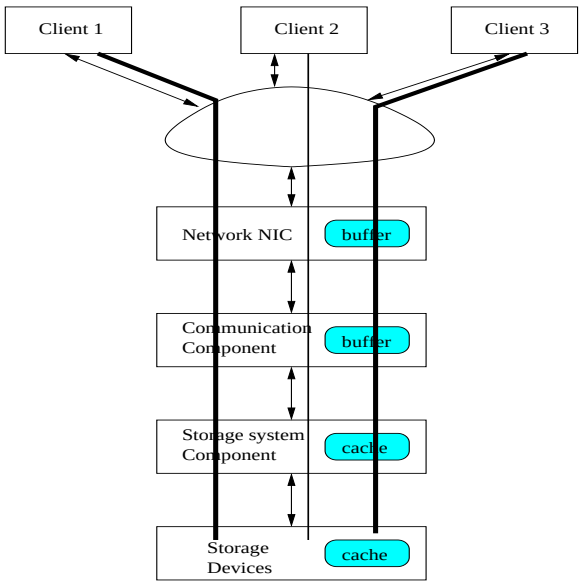


Figure 7.1: Resource Sharing in Networked Storage Servers.

is clear that networked storage systems involves multiple subsystems including both networking and local storage systems. Research needs to be done to provide QoS awareness in all involved subsystems and integrate them together to provide QoS to end workloads. Specifically, the communication subsystem, the file cache, the file system, the storage devices must be able to cooperate with each other. QoS features in emerging networking technologies such as InfiniBand should have a profound impact on the development of networked storage systems with QoS.

Efficient interaction and cooperation between different components — Networked data servers are machines that manage clients' data and provide access to these data for clients through networks. Based on the type of data, networked data servers can be categorized into file servers, storage servers, web servers, database servers, video servers, and so on.

There may be many differences between these networked data servers. However, several salient common features are shared. First, the interaction between the communication component and the file/storage component is very intensive. Second, caches in different levels and components form a multi-level cache hierarchy. Figure 7.2 shows a typical I/O architecture in networked data servers. In each component, an amount of memory is used either for caching data or for communication.

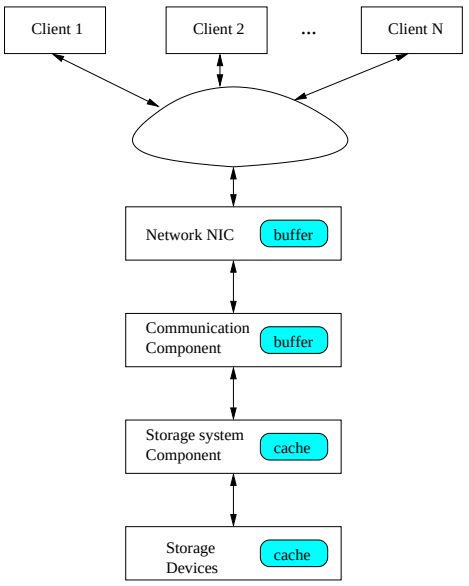


Figure 7.2: Basic I/O Architecture in Networked Data Servers.

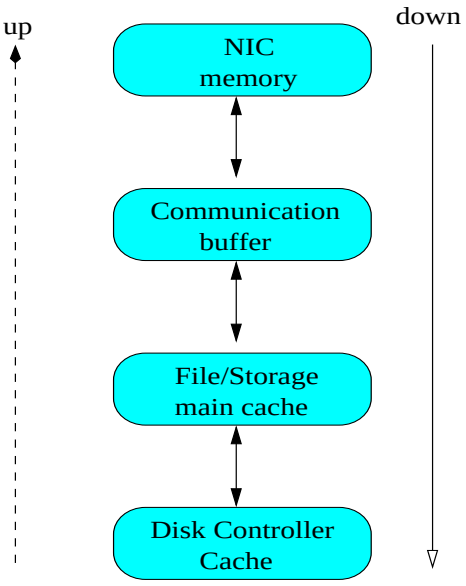


Figure 7.3: Down/Up Information Stream in I/O Data Path.

How to make better use of these memory is important to the overall performance of networked data servers. We proposed a notion, *InfoCache*, to provide flexible memory management in each component and interactions among multiple components. As shown in Figure 7.3, each cache receives information in the down-stream path, and exposes internal state information in the up-stream path. These in/out information can be used to achieve adaptation, intelligent scheduling decision, efficient interaction and integration, and better aggregation and management of memory resources.

As a part of the InfoCache project, our *Unifier* work described in Chapter 5 shows the power of information exchanged in different memory/cache levels. Essentially, Unifier takes the cache buffer information and uses the cache buffers in the communication component. Zero-copy in the whole I/O path is achieved. Thus, the end users can take full advantage of RDMA-based data movement.

Our basic idea is to make better use of the limited memory resources in each component. We continue this project with the following studies:

- **Cache-aware Scheduling:** The cache state information can be exposed to make better request scheduling. Applications and/or the server scheduler component can determine which data access is likely to be fast and thus re-order their requests.

- **Semantic Prefetching:** We attempt to use the semantic information to achieve effective prefetching. We try to explore the correlation information of independent accesses in workloads and use these information to guide prefetching policy. Therefore, the cache space can be used in a more efficient manner. For example, information can be used to provide across-file prefetching in I/O workloads such web access. A homepage file and its associated document and image files have useful correlation for prefetching. On the other hand, we may also need to limit the prefetching in a file/record level in workloads. This idea is partly motivated by the work [20].

- **An aggregate and cooperative management of the NIC memory, the storage system cache and the disk/disk array cache:** With the increasing size of the NIC memory, the storage system cache, and the disk/disk array cache, an aggregate and cooperative management is expected to make better use of these resources and to achieve the expected performance commensurate to the aggregate cache size.

BIBLIOGRAPHY

[1] A. Ching, A. Choudhary, K. Coloma, W.-K. Liao, R. Ross and W. Gropp. Noncontiguous I/O Accesses Through MPI-IO. In *Proceedings of CCGrid2003*, Tokyo, Japan, May 2003.

[2] American National Standard of Accredited Standards Committee ANSI NCITS T11.1 Technical Committee. Information Technology – SCSI on Scheduled Transfer Protocol (SST) Work Draft. July 2001.

[3] 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.

[4] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[5] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[7] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[8] Martha Bancroft, Nick Bear, Jim Finlayson, Robert Hill, , Richard Isicoff, and Hoot 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.

[9] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[10] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abdelsalam Heddaya, and Sulaiman A. Mirdad. Application-level document caching in the Internet. In *Proceedings of the 2nd International Workshop in Distributed and Networked Environments (IEEE SDNE '95)*, Whistler, British Columbia, 1995.

[11] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous IO support in Linux 2.5. In *Ottawa Linux Symposium*, Ottawa, ON, Canada, Jul 2003.

[12] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[13] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[14] E. Borowsky, R. Golding, A. Merchant, L. Schrier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve qos. In *5th International Workshop on Quality of Service (IWQoS 97)*, 1997.

[15] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[16] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 29–44, June 2002.

[17] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 196–208. ACM Press, 2003.

[18] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file c aching. In *Proc. First USENIX Symposium on Operating Systems Design an d Implementation*, pages 165–178, November 1994.

[19] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev 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.

[20] Enrique V. Carrera and Ricardo Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proceedings of the 10th International Symposium on High-Performan ce Computer Architecture (HPCA)*, February 2004.

[21] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A study of iSCSI extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 209–219. ACM Press, 2003.

[22] J. Chase, A. Gallatin, and K. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[23] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *USENIX Annual Technical Conference*. Usenix, June 2003.

[24] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[25] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[26] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[27] Cluster File Systems, Inc. Lustre: Scalable Clustered Object Storage. http://www.lustre.org/, June 2002.

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

[29] Francisco Matias Cuenca-Acuna and Thu D. Nguyen. Cooperative caching middleware for cluster-based servers. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2001.

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

[31] Michael Dahlin, Randy Wang, Tom Anderson, and David Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, November 1994.

[32] DAT Collaborative. uDAPL and kDAPL API Specification V1.0, June 2002.

[33] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *Second USENIX Conference on File and Storage Technologies*. USENIX, April 2003.

[34] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[35] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[36] J. Satran et al. "iSCSI", Internet Draft draft-ietf-ipsiscsi-20.txt, February 2003.

[37] IEEE P802.3ae 10Gb/s Ethernet Task Force. 10 Gigabit Ethernet Standard. http://grouper.ieee.org/groups/802/3/ae/index.html.

[38] Gregory R. Ganger. Blurring the Line Between Oses and Storage Devices. CMU SCS Technical Report CMU-CS-01-166, December 2001.

[39] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of COMPCON Spring'96: IEEE Computer Society International Conference*, February 1996.

[40] Pawan Goyal, Divyesh Jadav, Dharmendra S. Modha, and Renu Tewari. Cachecow: Qos for storage system caches. In *Eleventh International Workshop on Quality of Service (IWQoS 03), Monterey, CA*, 2003.

[41] 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.

[42] J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3nd edition, 2003.

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

[44] Justin Hurwtize and Wu chun Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. In *Hot Interconnects 11*, August 2003.

[45] IBM. StorageTank. http://www.almaden.ibm.com.

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

[47] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation on top of GPFS. In *Supercomputing 2001*, Nov. 2001.

[48] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[49] Clint Jurgens. Fibre channel: A connection to the future. *Computer*, 28(8):88–90, 1995.

[50] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[51] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Washington, DC, November 1994. IEEE Computer Society Press.

[52] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXcluster: a closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, 1986.

[53] Rob Latham and Rob Ross. PVFS, ROMIO, and the noncontig Benchmark. http://www.mcs.anl.gov/romio/noncontig-perf.pdf, April 2003.

[54] 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.

[55] Mellanox Technologies. Mellanox InifniBand Technologies. http://www.mellanox.com.

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

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

[58] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8), 2003.

[59] Message Passing Interface Forum. MPI-2: A Message Passing Interface Standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[60] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. Technical Report 91-3, University of Michigan Center for IT Integration (CITI), August 1991.

[61] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[62] Srivatsan Narsimhan, Sohom Sohoni, and Yiming Hu. A Log-Based Write-Back Mechanism for Cooperative Caching. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[63] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

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

[65] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[66] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[67] Patrick Beng T Khoo and Wilson Yong H Wang. Introducing a flexible data transport protocol for network storage applications. In *Proceedings of the 20th IEEE Symposium on Mass Storage Systems*, 2002.

[68] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[69] Timothy M. Pinkston, Alan F. Benner, Michael Krause, Irv M. Robinson, and Thomas Sterling. InfiniBand: The "De Facto" Future Standard for System and Local Area Networks or Just a Scalable Replacement for PCI Buses? *Cluster Computing 6, pp. 95-103*, 2003.

[70] PVFS2 Developer Team. Parallel Virtual File System, Version 2. http://www.pvfs.org/pvfs2/, Nov. 2003.

[71] Renato John Recio. Server i/o networks past, present, and future. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 163–178. ACM Press, 2003.

[72] Rob F. Van Der Wijngaart and Parkson Wong. NAS Parallel Benchmarks I/O Version 2.4. http://www.nas.nasa.gov/Research/Reports/Techreports/2003/nas-03-002-abstract.html.

[73] Allyn Romanow and Stephen Bailey. An overview of rdma over ip. In *First International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[74] Allyn Romanow, Jeff Mogul, Tom Talpey, and Stephen Bailey. RDMA over IP Problem Statement. *Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-rddp-problem-statement-02.txt*, June 2003.

[75] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application i/o on linux clusters. In *Proceedings of SuperComputing Conference*, 2001.

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

[77] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 405–420, San Diego, CA, January 1993.

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

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

[80] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[81] Rich Seifert. *Gigabit Ethernet: Technology and Applications for High-Speed LANs*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1998.

[82] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *To appear in Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, San Jose, CA, October 2002.

[83] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, CA, March 2003.

[84] E. Smirni and D.A. Reed. Workload characterization of input/output intensive parallel applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, June 1997.

[85] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, Syracuse, NY, 1996. IEEE Computer Society Press.

[86] SNIA (Storage Networking Industry Association). Shared Storage Model. www.snia.org/tech_activities/shared_storage_model, 2002.

[87] Steven R. Soltis. The Design and Implementation of a Distributed File System based on Shared Network Storage. PhD dissertation. University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, August 1997.

[88] Dragan Stancevic. Zero copy I: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

[89] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[90] Rajeev Thakur, William Gropp, and Ewing 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.

[91] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *to appear in Parallel Computing*, 2002.

[92] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, California, 1994.

[93] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[94] Brian Tierney, William Johnston, and Jason Lee. A Cache-Based Data Intensive Distributed Computing Architecture For "grid" Applications. In *CERN School of Computing*, Sept. 2000.

[95] Kent Treiber and Jai Menon. Simulation study of cached RAID5 designs. In *Proceedings of the First Conference on High-Performance Computer Architecture*, pages 186–197. IEEE Computer Society Press, January 1995.

[96] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, 1997.

[97] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report, CS-TR-3802, University of Maryland, College Park, May 1997.

[98] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43. ACM Press, 1998.

[99] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[100] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[101] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters, and Grids: A Case Study. In *Supercomputing 2003: The International Conference for High Performance Computing and Communications*, Nov. 2003.

[102] W. Gropp and E. Lusk and N. Doss and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard.

[103] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Aug. 1997.

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

[105] John Wilkes. Traveling to rome: QoS specifications for automated storage system management. *Lecture Notes in Computer Science*, 2092:75–??, 2001.

[106] Theodore M. Wong. Personal Communication, May 2003.

[107] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Monterey, CA, June 2002. USENIX.

[108] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. Technical Report, OSU-CISRC-0/03-TR, April 2003.

[109] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[110] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[111] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. Technical Report, OSU-CISRC-05/03-TR26. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/, May 2003.

[112] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance Implementation of MPI Datatype Communication over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[113] Jiesheng Wu, Pete Wyckoff, Dhabaleswar K. Panda, and Rob Ross. Unifier: Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid2004)*, Chicago, IL, April 2004.

[114] ANSI X3T9.3. Fiber Channel- Physical and Signaling Interface (FC-PH), 4.2 edition. November 1993.

[115] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai 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.

[116] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, Boston, Massachusetts, June 2001. USENIX.