DESIGNING HIGH-PERFORMANCE AND SCALABLE CLUSTERED NETWORK ATTACHED STORAGE WITH INFINIBAND

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Ranjit Noronha, MS

* * * * *

The Ohio State University

2008

Dissertation Committee:

Dhabaleswar K. Panda, Adviser

Ponnuswammy Sadayappan

Feng Qin

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

The Internet age has exponentially increased the volume of digital media that is being shared and distributed. Broadband Internet has made technologies such as high quality streaming video on demand possible. Large scale supercomputers also consume and create huge quantities of data. This media and data must be stored, cataloged and retrieved with high-performance. Researching high-performance storage subsystems to meet the I/O demands of applications in modern scenarios is crucial.

Advances in microprocessor technology have given rise to relatively cheap off-the-shelf hardware that may be put together as personal computers as well as servers. The servers may be connected together by networking technology to create *farms* or *clusters of workstations (COW)*. The evolution of COWs has significantly reduced the cost of ownership of high-performance clusters and has allowed users to build fairly large scale machines based on commodity server hardware.

As COWs have evolved, networking technologies like InfiniBand and 10 Gigabit Ethernet have also evolved. These networking technologies not only give lower end-to-end latencies, but also allow for better messaging throughput between the nodes. This allows us to connect the clusters with high-performance interconnects at a relatively lower cost.

With the deployment of low-cost, high-performance hardware and networking technology, it is increasingly becoming important to design a storage system that can be shared across all the nodes in the cluster. Traditionally, the different components of the file system have been stringed together using the network to connect them. The protocol generally used over the network is TCP/IP. The TCP/IP protocol stack in general has been shown to have poor performance especially for high-performance networks like 10 Gigabit Ethernet or InfiniBand. This is largely due to the fragmentation and reassembly cost of TCP/IP. The cost of multiple copies also serves to severely degrade the performance of the stack. Also, TCP/IP has been been shown to reduce the capacity of network attached storage systems because of problems like incast.

In this dissertation, we research the problem of designing high-performance communication subsystems for network attached storage (NAS) systems. Specifically, we delve into the issues and potential solutions with designing communication protocols for high-end single-server and clustered server NAS systems. Orthogonally, we also investigate how a caching architecture may potentially enhance the performance of a NAS system. Finally, we look at the potential performance implications of using some of these designs in two scenarios; over a long haul network and when used as a basis for checkpointing parallel applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Humanity has increasingly become dependent on digital media for work and pleasure. The Internet has transformed the way we work and play and has revolutionized the way information is created and distributed. With the exponential growth in the amount of information created by the digital age, storing this information has increasingly become a challenge. These challenges are complicated by legal requirements which require the retrieval of this information reliably many years into the future. Also, with the wide availability and use of broadband Internet, streaming digital media content has become widespread. This increasingly raises the bar on performance requirements for the storage subsystems.

At the other end of the spectrum, supercomputers are epitomized in the public consciousness by the chess competition between Kasparov and Big Blue [59]. These supercomputers are being used at many sites to perform scientific simulations of nuclear and atomic phenomena. Advances in microprocessor technology have given rise to relatively cheap hardware and may be put together as personal computers as well as servers. The servers may be connected together by networking technology to create *farms* or *clusters* *of workstations (COW)*. The evolution of COWs has significantly reduced the cost of ownership of high-performance systems and has allowed users to build fairly large scale supercomputers based on clustered hardware [30, 94]. Figure 1.1 shows IBM RoadRunner; the first cluster to reach a petaflop using off-the-shelf components. Simulations on these large scale systems, store and generate vast amounts of data. High-performance, timely access to application data becomes crucially important as it is often a limiting factor in the performance of these systems.

As clusters increase in sheer size, faults in these large scale systems become increasingly common [84, 37]. To protect application runs from these faults in a timely manner, these long running applications and programs need to be checkpointed at regular intervals, so that the applications may be reliably restarted in the case of failures. Storing these vast amounts of check-point data may pose a burden on the storage sub-system.

As COWs have evolved, commodity networking technologies like InfiniBand [8] and 10 Gigabit Ethernet [47] have also evolved. These networking technologies not only give lower end-to-end latencies, but also allow for better messaging throughput between the nodes as compared to traditional Fibre Channel and Gigabit Ethernet. These allow us to connect the clusters with high-performance interconnects at a relatively lower cost.

As clustering and networking technology have evolved, storage systems have also evolved to provide seamless, name space transparent, reliable and high-performance access to user-level data across clusters. However, these storage systems are not designed to take advantage of the properties of high-performance networks. In this dissertation, we look at the issues surrounding high-performance storage systems and networks. In this Chapter, we provide a background and overview of storage research. The rest of this Chapter is organized as follows. In Section 1.1, we discuss basic file system concepts. Next, Section 1.2 discusses storage architectures. Following that, Section 1.3 discusses some commonly used file systems. Finally, Section 1.4 surveys networking technology used in storage environments.

In Chapter 2, we discuss the problems addressed in this dissertation and also provide an overview of remaining Chapters.

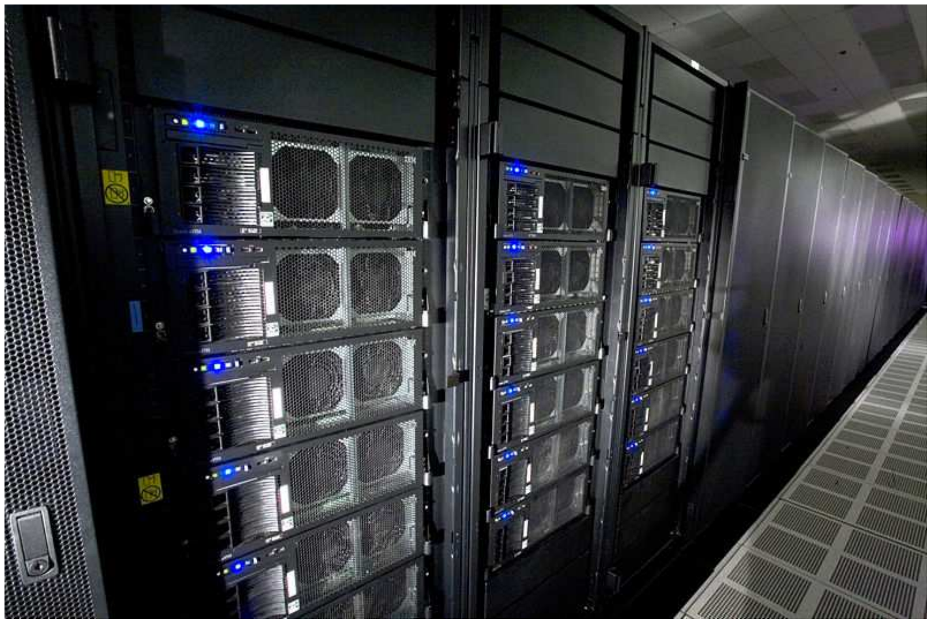


Figure 1.1: IBM RoadRunner: the first petaflop cluster. Courtesy, Leroy N. Sanchez, LANL [26], copyright, Appendix A

1.1 Overview of Storage Terminology

In this section, we briefly describe the commonly used concepts and termninology relating to file systems. Note that some terminology is used interchangeably in available literature. We will explain the differences and our usage in the text.

1.1.1 File System Concepts

Processes and users on UNIX systems and derivatives generally store information in files that reside in directories. Directories themselves could contain directories, which could lead to fairly deep levels of nesting. The majority of modern storage devices such as disk drives are not aware of file, directory or other process level storage abstractions. These storage devices are mostly responsible for storing and retrieving fixed sized units of data [54]. File systems are usually layered between the user and storage device. They provide the file and directory abstraction to the user. Besides maintaining the basics such as file permissions, providing and maintaining file pointers and descriptors, file systems are also responsible for the following:

- Moving information or data from processes to files and eventually to the underlying storage devices and vice-versa.
- Allowing appropriately authorized access to files and directories.
- Arbitrating between multiple processes accessing the same file.

- Managing the blocks on the underlying storage device(s). These features include listing sets of free blocks on the device, allocating these blocks to files and deallocating these blocks when the files shrink in size.
- Reducing the impact of slow access to and from the storage device.
- Keeping the file system consistent as storage devices fail and there are power outages.
- Providing data recovery when data is corrupted and fail-over modes in the face of irrecoverable errors.

Most file system face these issues, irrespective of whether they are on a single computer or distributed across a cluster of computers. We examine the concept of a file in Section 1.1.2. Following that, we look at how a file system organizes blocks available on a storage device. We then discuss how these blocks are used to store and access file and directory data. Finally, we examine the common caching and buffering techniques used to optimize access to data.

1.1.2 UNIX Notion of Files

A UNIX file consists of a sequence of bytes from 0..N-1, where N is the length of the file in bytes. A UNIX system does not assign any specific meaning to the bytes; it is up to the application to interpret the bytes. On typical UNIX systems, files usually have a user-id and group-id. It is possible to specify separate read, write or execute permissions for the user and group. In addition, it is also possible to specify these access permissions for everybody else not included in the group-id. Inorder to access the file, a process will request that a particular file be opened. Once the file system determines that a particular process has appropriate permissions to access a given file, a file pointer or descriptor is returned to the calling process. The file pointer has an implicit byte location in the file which starts at byte zero in the file and is incremented by read/write operations. This file pointer may also be explicitly changed by *seek* operations. A file may be accessed concurrently by multiple different processes. Each of these processes will have its own independent file descriptor. Conflicting writes from different processes are usually resolved by maintaining sequential consistency. Sequential consistency is generally easy to maintain in a local file system, but much harder to maintain in a parallel or distributed file system. Files may be of different types; regular files that contain process-level data, memory mapped files which allow processes to read and write to files that are mapped to locations in memory. Other types of files include socket files that are used for interprocess communication and device files that are used for communication between user-level and kernel-level entities. File types and their implementation are discussed comprehensively by Pate, et.al. [66]. File systems are usually implemented as part of the system kernel. The interface to access files is called the Virtual File System (VFS) interface and is a POSIX standard [88]. Additional information on the UNIX file model is available in [66, 92, 54, 97].

1.1.3 Mapping files to blocks on storage devices

File Systems may be layered on-top of a variety of different storage media such as magnetic tapes, magnetic disks, optical disks or floppy drives. The vast majority of file systems, including the ones we will consider in this dissertation are layered on-top of magnetic disks. We briefly discuss the architecture of magnetic disks. Storage media that is contained within the same enclosure as the compute device is usually referred to as Direct Attached Storage (DAS). DAS interfaces are usually based on SCSI or SATA. A magnetic disk is divided into a number of concentric tracks. In turn, each concentric track is divided into sectors or units of data. Because of the geometry of the disk, sectors on the outer track may be less densely packed as compared to those on the inner tracks. Modern disk drives may have the same density of sectors on the outer track as on inner tracks. As a result, the heads in modern disks may need to read and write data at different rates depending on the area of the disk they are accessing. The minimum unit of data that may be accessed from a magnetic disk is a sector. Modifying only one byte of data on a disk will require the entire sector containing that byte to be read into memory, modified and then written back to disk. This read-modify-write operation tends to be prohibitively expensive, and may reduce performance.

File systems must translate between the abstraction of a file and disk sectors. To achieve this, when a file is created, the file system will usually allocate a number of sectors on the disk for the file. We refer to this as a block. The list of blocks allocated to a file is stored in a data structure called an *inode* (index node). The inode also usually contains details about the file creation time, permissions and so on. As the file increases in size, the inode for the file might not be large enough to hold the references to all the blocks for the file. Some of the blocks references in the inode for the file might point to secondary inodes called indirect blocks, which are actually references to data blocks. This may be extended to a tertiary or higher level, depending on the size of the file. Directories are also represented with inodes. The entries in the inodes for the directory refer to inodes for file and other subdirectories. When a process requests a file, the file system needs to access each of the inodes along the path to the file. To reduce the impact of these lookups, most filesystems maintain an inode cache. The inode cache can dramatically improve the performance of extensively used inodes such as the inode corresponding to the root directory of the file system, or the system password file.

1.1.4 Techniques to improve file system performance

I/O access performance depends on the performance of the underlying disk drive. Since disk drives are mechanical devices, disk drive performance is ultimately limited by the seek time of the head as it swings across the disk platter and is positioned in the correct location for reading the data. It also depends on the rate at which data may be read from the disk, which depends on the speed of rotation of the disk platter. Clearly, there are physical limitations to the performance of magnetic disk drive technology. The file system may attempt to reduce the impact of the disk by placing blocks of the same file adjacent to each other on the disk. However, this only improves the access performance of sequential workloads, i.e. it does nothing to improve the performance of random workloads. Also, path lookup time requires fetching the inode block of each subdirectory in the path of the file. To reduce the access time of several operations, file systems usually maintain a data buffer cache. The buffer cache stores writes from a process and then writes it to the disk in stages. This process is called *write behind* and hides the latency of the disk from the process. Also, blocks that are reused from the buffer cache experience very little overhead on reads. In addition, sequential I/O access patterns may be optimized by “reading ahead” the data and storing it in the buffer cache. Since the buffer cache is located in volatile memory, a crash might lose write data. Some systems use NVRAM (Non Volatile RAM) to reduce the impact of loses due to a crash.

1.2 Overview of Storage Architectures

With problems in science and technology increasingly becoming larger, clusters built with commodity off-the-shelf components are increasingly being deployed to tackle these problems. As the number of nodes in the cluster increases in size, it is important to provide uniform storage performance characteristics to all the nodes across the cluster. To enable these types of performance characteristics, it becomes necessary to use clustering techniques on the storage system itself. Broadly, storage architectures may be classified into Direct attached Storage (DAS), Network Attached Storage (NAS), System Area Networks (SAN), Clustered NAS (CNAS) and Object Storage Systems (OSS). We discuss the salient features of each of these architectures next.

1.2.1 Direct Attached Storage (DAS)

Most modern operations systems such as Linux, BSD and Solaris that run on a generic computer box based on commodity components, store operating system configuration and application data on a local file system. The local file system in most operating systems is layered as shown in Figure 1.2. The applications generally run in user-space. Most local file systems are generally implemented within the operating system. To make I/O system calls, the application needs to trap into the operating system via system calls. The file system interface component of system calls, usually referred to as the Virtual File System (VFS) interface, is a POSIX standard [88]. Following that, the I/O call is processed by the *File System User Component*, which performs checks for user authorization, read/write access permission and quota checks. Following these checks, the *File System User Component* may attempt to satisfy the request through a cache. The *File System User Component* may also attempt to prefetch I/O data from the storage subsystem, especially if the access pattern is sequential. If the request cannot be satisfied through the local cache, it is passed to the *File System Storage Component*, which translates the request into sectors, blocks and cylinders that may be interpreted by the underlying storage device. The underlying storage device may be directly attached to the compute node through a SCSI interface [27], or via a System Area Network (SAN) [55] such as Fibre Channel [100]. There are a variety of different local file systems such as ext3 [10], reiserFS [25], NTFS [20], ufs [31], Sun ZFS [33], NetApp WAFL [32] and SGI XFS [86]. These file systems are designed and optimized for a variety of different properties; namely performance, scalability, reliability and recovery. In this dissertation, we mainly focus on file systems that span multiple different client and storage nodes in a distributed and clustered environment. A local file system is an important component of a distributed file system as it usually is responsible for managing the back-end storage nodes of the distributed file system. The characteristics of the local file system directly affect the properties of the distributed file system and may complicate the design of the distributed file system, depending on the specific qualities we would like to achieve.

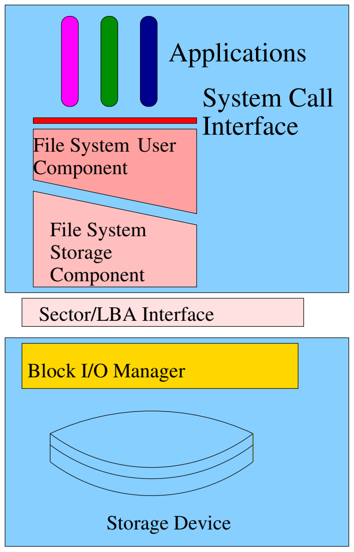


Figure 1.2: Local File System Protocol Stack. Courtesy, Welch, et.al. [40]

1.2.2 Network Storage Architectures: In-Band and Out-Of-Band Access

Depending on how metadata is accessed, storage systems may be categorized as In-Band (Figure 1.3) and out-of-band (Figure 1.4). In In-Band systems, the metadata is stored together with the regular data on the same file system. As a result, access to metadata may become a bottleneck, since it may be delayed by access to large data blocks. Metadata usually includes information such as the location of different parts of a file in addition to other information such as the permission and modification times of the file. Fast, high-speed access to metadata information usually helps improve file systems performance substantially.

By storing the metadata and data on separate servers (out-of-band), the metadata access time may be substantially improved.

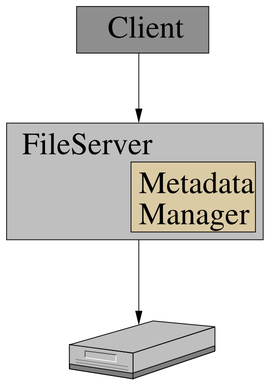


Figure 1.3: In-Band Access

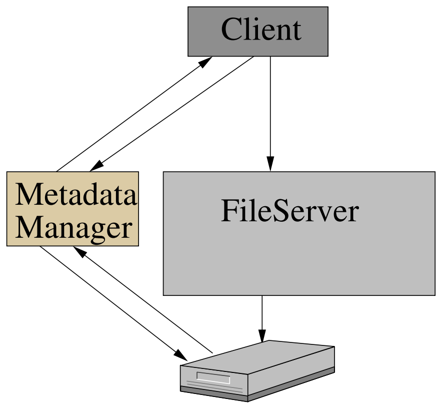


Figure 1.4: Out-of-Band Access

1.2.3 Network Attached Storage (NAS)

In Network Attached Storage (NAS) systems, the file system is divided into two components, a client and a server, which reside on different nodes. The client component of the file system sends I/O requests from the application to the file system server over a network such as Gigabit Ethernet or InfiniBand. The server processes the client requests in a manner similar to that described in Section 1.2.1 and then sends the responses back to the client. The server node generally contains most of the components of the local file system. The file system protocol stack in the case of NAS is shown in Figure 1.5. This implies that the NAS generally has a single server or head. By locating the same files in a single location, NAS helps solve the problem of multiple different copies and versions of the same file and data residing in different local file systems. However, NAS exposes a new set of problems not present with a local file system. These problems include network round-trip latency, possibility of dropped or duplicate non-idempotent I/O requests, increased overhead at the server head and limited scalability as the server increasingly becomes a bottleneck and finally, increased exposure to failures, since the NAS head is a single point of failure. Usually, the problems with performance and failures may be alleviated to some extent by using Non-Volatile memory (NVRAM) to store some transactions or components of transactions at the server [18]. Network File System (NFS) [36] and Common Interface File System (CIFS) are examples of widely deployed NAS systems.

1.2.4 System Area Networks (SAN)

System Area Networks (SANs) provide high-speed access to data at the block-level as shown in Figure 1.6. SANs may be implemented as either symmetric or asymmetric. In the symmetric implementation, each client runs part of the metadata server or block I/O manager. The clients use a distributed protocol to coordinate. The asymmetric implementation (pictured in Figure 1.6), uses a shared block-level manager, which the clients contact before attempting to access data through the SAN network. SAN may be used as either a component of a local file system, or re-exported through the server of a NAS. Traditional SANs went through Fibre Channel. As networks such as Giabit-Ethernet have become widespread, protocols such as iSCSI (SCSI over TCP/IP) have also become popular. Lately, with RDMA enabled networks such as InfiniBand becoming popular, protocols such as iSER (iSCSI over RDMA) have also been widely deployed.

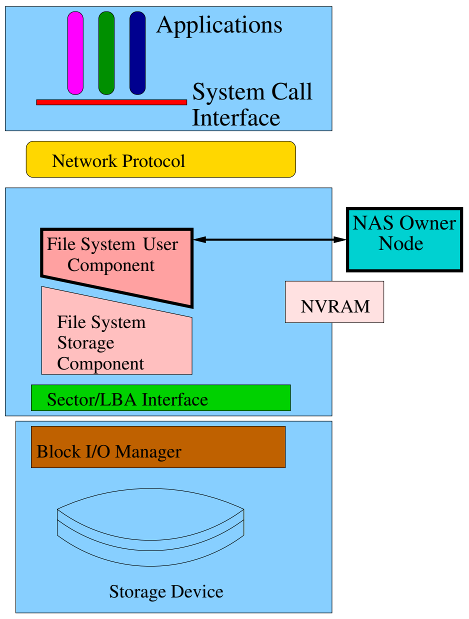


Figure 1.5: Network Attached Storage Protocols. Based on illustration by Welch, et.al. [40]

1.2.5 Clustered Network Attached Storage (CNAS)

The single server in traditional NAS architectures becomes a bottleneck with increasing number of clients. This limits the scalability of single headed NAS. To alleviate the scalability limitations, multi-headed or clustered NAS have evolved. In the clustered NAS architecture, shown in Figure 1.7 multiple NAS servers or heads share the same back-end storage. Since the data must still flow through the NAS server; multiple memory copies within the NAS server and network protocol stack and transmission overhead limit the

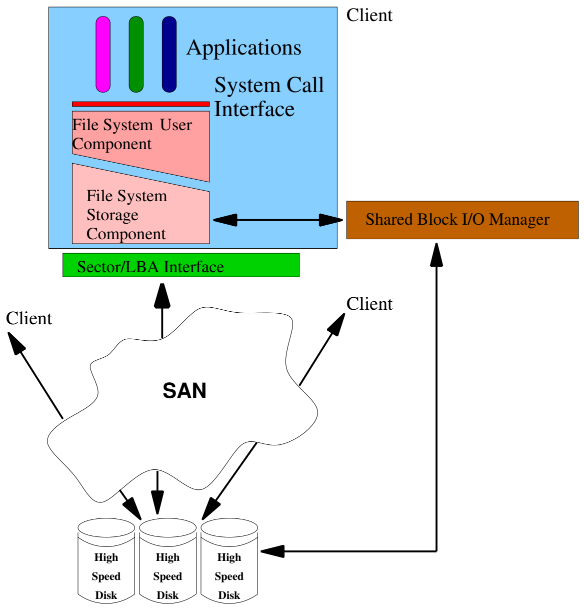


Figure 1.6: System Area Networks. Based on illustration by Welch, et.al. [40]

achievable I/O performance. In addition, because the NAS servers add a level of indirection between the storage subsystem and clients, this increases the cost of building such a system. Alternatively, the storage subsystem may be directly part of the NAS head, instead of as part of a separate SAN. This could help reduce the cost of the clustered NAS system, but depending on the the design of the clustered NAS, might reduce the broad availability of data to all NAS heads and introduce additional fault-tolerance problems.

Architecturally, clustered NASs may be designed in two different ways. The first way is the *SAN re-export* way shown in Figure 1.7. As discussed earlier, the SAN storage nodes are exposed to the client through the NAS heads. SAN re-exports suffer from the same problem as traditional SAN with respect to block allocation and writes. In addition, the re-exporting introduces additional coherency problems between the NAS servers. There are two potential solutions to the coherency problem, force Writes through to the SAN, or use the coherency protocol of the SAN. Examples of SAN re-export implementations include IBRIX [7], Polyserver [6] and IBM's GPFS [83].

The second technique of designing a NAS is the *Request Forwarding Approach* shown in Figure 1.8. In the *Request Forwarding Approach*, each NAS head is responsible for access to a subset of the storage space. The clients can forward requests to the NAS heads with which it communicates. The NAS heads, can choose to either service the request directly, or forward the request to the owner of the storage area. The benefits of directly servicing the request is that if it is available in the cache, it can be directly serviced from the cache. This allows the NAS heads to be used for caching popular requests. However, one needs to worry about maintaining the caches coherent. Forwarding the request to the NAS head eliminates the requirement to maintain the caches coherent. However, the limited amount of cache and the bottleneck of a single server may reduce the effectiveness of this approach. Examples of a Clustered NAS that use the forwarding model are NetApp GX [65], Isilon [12] and IBRIX [7].

1.2.6 Object Based Storage Systems (OBSS)

Object Based Storage Systems are the latest evolution in storage technology. File Systems are layered on-top of object storage devices (OSD). File Systems use the OSD interfaces. The OSDs act as containers of data and file system attributes. OSDs contain features for block management and security. Unlike traditional block-oriented storage systems,

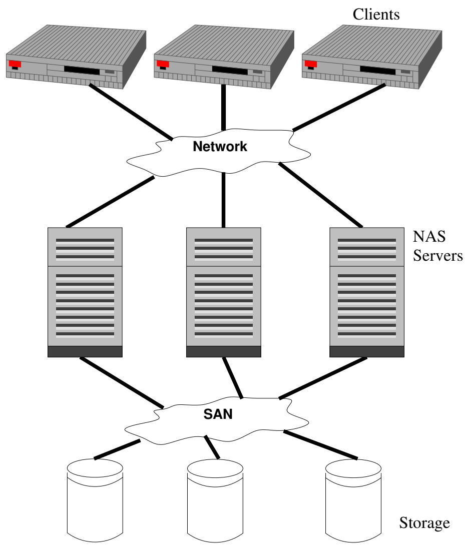


Figure 1.7: Clustered NAS: High-Level Architecture

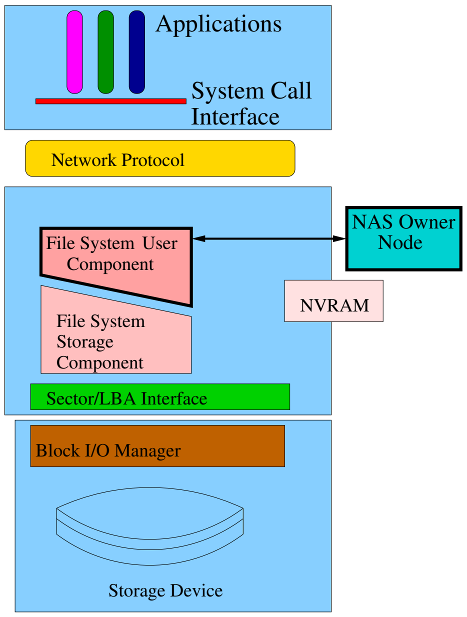


Figure 1.8: Clustered NAS Forwarding Model: Protocol Stack, Courtesy Welch, et.al. [40]

OSDs also understand high-level details such as data structure and layout. Usually, I/O data transfers are decoupled and takes place out-of-band directly between the clients and the OSDs. Also, clustering techniques similar to those in clustered NAS environments are used to improve scalability and performance. Examples of Object Based Storage Systems include Lustre [43] and the Panasas file system [46].

1.3 Representative File Systems

In this section, we discuss some representative file systems. We mainly focus on the Network File System (NFS) and Lustre.

1.3.1 Network File Systems (NFS)

Network File System (NFS) [36] is ubiquitously used in most modern UNIX based clusters to export home directories. It allows users to transparently share file and I/O services on a variety of different platforms. The file system model consists of a hierarchy of directories with an optional set of files in each of these directories. The leaf directories do not contain additional entries for directories. NFS is based on the single server, multiple client model and is an example of a NAS as described in Section 1.2.3. Communication between the NFS client and the server is via the Open Network Computing (ONC) remote procedure call (RPC) [36] initially described in IETF RFC 1831 [36]. The first implementation of ONC RPC also called Sun RPC for a Unix type operating system was developed by Sun MicroSystems [28]. Implementations for most other Unix like operating systems including Linux have become available. RPC is an extension to the local procedure calling semantics, and allows programs to make calls to nodes on remote nodes as if it were a local procedure call. RPC traditionally uses TCP/IP or UDP/IP as the underlying communication transport. Since the RPC calls may need to propagate between machines in a heterogeneous environment, the RPC stream is usually serialized with the eXternal Data Representation (XDR) standard [36] for encoding and decoding data streams. NFS has seen three major generations of development. The first generation, NFS version 2 provided a stateless file access protocol between the server and client using RPC over UDP and exported a basic POSIX 32-bit filesystem. It was slow, especially for writing files. NFS version 3 added to the features of NFSv2 and provided several performance enhancements, including larger block data transfer, TCP-based transport and asynchronous write, among many others. It also allowed for 64-bit files and improved write caching. Most modern clusters today use NFSv3. The NFSv2 and NFSv3 protocols are described further in [81]. The revision to the NFSv3 protocol is NFSv4 [87]. Besides the functionality available in NFSv3, NFSv4 adds a number of additional mechanisms for security and performance along with other features. The security features include the Global Security Services (GSS) API called RPCSEG_GSS. NFSv3 used UNIX user ID security which proved inadequate in wide-are networks. The RPCSEG_GSS allows the use of both private key authentication schemes such as Kerboros version 5 as well as private key schemes. To improve performance, NFSv3 allows a client side cache to be maintained. However, the client side caches are not kept coherent, this may lead to inconsistencies in the caches. NFSv4 also does not keep the client caches coherent. It does allow files to be delegated to a client, if it is exclusively accessing that file. NFSv3 and NFSv4 are discussed further in Chapter 3. The time lag between the NFSv3 and NFSv3 protocols is 8 years with a similar gap between the NFSv3 and NFSv4 protocols. To reduce the gap for relatively small enhancements to the protocol, minor versioning was added to the NFSv4 protocol. NFSv4.1 is the first minor revision to the NFSv4 protocol, which deals with parallel NFS (pNFS) and sessions. It is discussed further in Chapter 6.

1.3.2 Lustre File System

Lustre is an open source all software based, POSIX compliant clustered object based file system. Lustre uses a decoupled metadata and data paths as shown in Figure 1.9. The client contacts the Metadata server (MDS) to create, open, close and lock a file. Once the information is obtained from the MDS, the clients can directly access the data through the

Object Storage Servers (OSTs). Lustre demonstrates good performance on large sequential data I/O transfers, because of the parallelism of the OSTs. In addition, Lustre also demonstrates fairly good throughput on file creations and accesses to small files. This is largely achieved through modifications to the underlying local disk file system ext3, that bunches a number of small files together on the same disk block, instead of spreading out these files on different areas of the disk. The disk seeks at the OSTs and MDSs still limit the file access performance. To alleviate some of these problems, Lustre has both client and server side caches. These caches are kept coherent through multi-state locks maintained by the MDS. Lustre supports multiple different networks natively; InfiniBand [49], Quadrics Elan4 [74], and Myrinet MX, in addition to socket based TCP/IP based networks. Similar to NFS, Lustre also uses RPC to communicate between the clients, MDS and OSTs. To achieve reliability and high-availability, Lustre allows the MDS to failover to another standby server. In addition, OSTs that fail will not impact clients that do not have data stored on that particular OST. OSTs may also be taken offline for an upgrade or rebooted; clients that depend on that OST will experience a delay. Lustre has largely been used in HPC environments for large sequential workloads such as checkpointing. It is also used as a scratch space file system. In the July 2008 rankings of the TOP500 list of supercomputers, 6 of the top 10 use Lustre [30].

1.4 Overview of Networking Technologies

In this section, we discuss some important networks that have become prominent in high-performance storage.

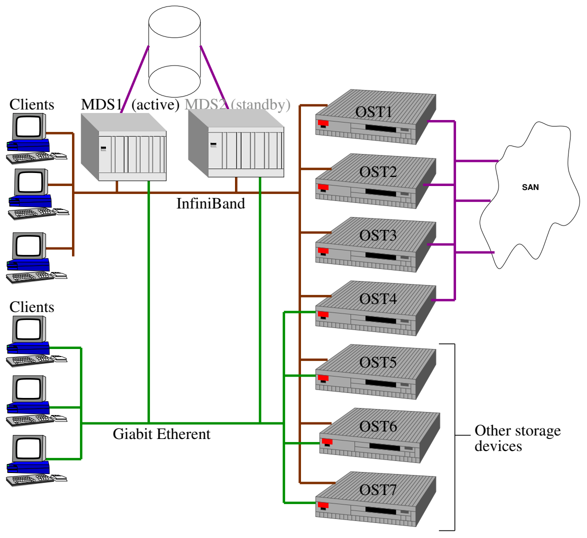


Figure 1.9: Lustre Architecture

1.4.1 Fibre Channel (FC)

Fibre Channel (FC) is a ANSI standard used primarily for storage area networks (SAN) in enterprise networks. It runs primarily at Gigabit speeds over both twisted copper wire pairs and fiber-optic cables. FC mainly uses SCSI commands which are transported over FCs native protocol Fibre Channel Protocol (FCP). FCP is responsible for transporting SCSI commands between the *SCSI initiator*, which is usually a client and the *SCSI target*, which is usually a storage device. In order to achieve this, FC allows for three different types of topologies; point-to-point, loop and fabric shown in Figure 1.11. Point-to-point involves connecting two devices back-to-back. For loop topology, the devices are connected in the form of a ring. In fabric topology, the devices are connected to a fabric switch, similar to ethernet. FCP is a layered protocol divided into five layers FC0-FC4. FC0-FC2 form the physical layers of Fibre Channel and are referred to as FC-PH. FC3 is used to implement encryption and RAID. FC4 is a protocol mapping layer and implements protocols such as SCSI. Fibre Channel runs at speeds from 1 Gbit/s to 20 Gbit/s, though some of the higher-speeds are not compatible with the lower speeds. Additional information on FC is available in [55, 2]. Figure 1.10 shows some examples of fibre channel connectors.

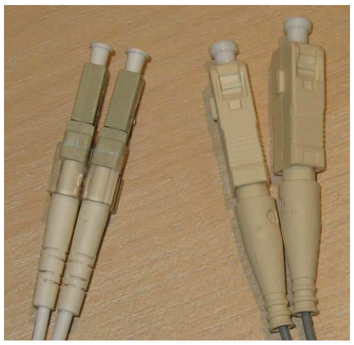


Figure 1.10: Fibre Channel Connectors [3]

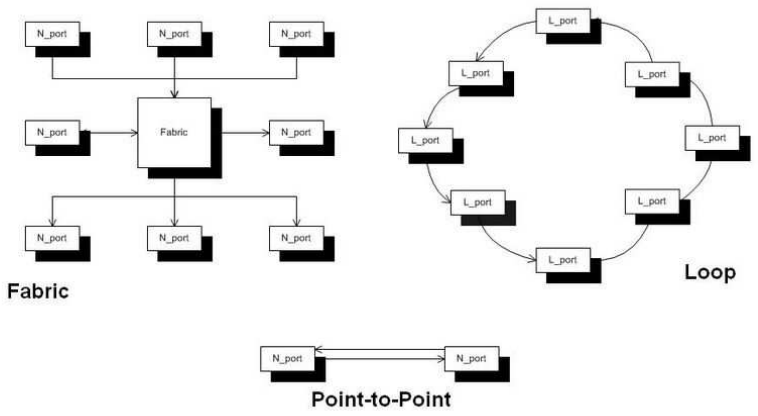


Figure 1.11: Fibre Channel Topologies [3]

1.4.2 10 Gigabit Ethernet

10 Gigabit Ethernet (10 GbE) is the successor to the Gigabit Ethernet standard. It runs at 10 times the speed of Gigabit Ethernet. It is defined by IEEE standard 802.3-2008. 10 Gigabit Ethernet is currently a switch based technology and does not support Carrier Sense Multiple Access/Carrier Detect (CSMA/CD) for shared access. The 10 GbE standard allows for different physical layer implementations. Currently, physical layers

exist for LAN and WAN environments. For cabling, 10 GbE supports CX-4 and fibre optical cables. 10 Gigabit Ethernet network cards exist in both TCP/IP offload (TOE) and non-TOE mode. Chelsio S310E is an example of a TOE [44] while Myri 10G is an example of non-offload 10 GbE NIC [24]. iWARP is a protocol which allows the user to use Remote Direct Memory Access (RDMA) over TCP/IP. TCP/IP is a compute intensive protocol whose performance is limited, especially on high-performance interconnects [39]. When iWARP is used with a TOE NIC, the performance limitations of TCP/IP can be significantly reduced or even eliminated under certain circumstances [77]. In addition, this allows RDMA to be used over Ethernet and in WAN environments. Internet SCSI (iSCSI), allows SCSI commands, which traditionally have been used in Fibre Channel environments to be used in TCP/IP based environments [50]. Like iWARP, iSCSI suffers from considerable overhead because of TCP/IP. By using a TOE NIC, this overhead may be reduced considerably. iSCSI may then be used to build SANs with capabilities similar to that of Fibre Channel, but at considerably lower cost.

1.4.3 InfiniBand

The InfiniBand Architecture (IBA) [49] is an open specification designed for interconnecting compute nodes, I/O nodes and devices in a system area network. In an InfiniBand network, compute nodes are connected to the fabric by Host Channel Adapters (HCA's). InfiniBand allows communication through several combinations of connection-oriented, connectionless, reliable and unreliable communication semantics. InfiniBand enables the application programmer to use kernel-bypass RDMA operations. These operations enable two appropriately authenticated processes to directly Read and Write from each others processes memory. InfiniBand is a popular interconnect in High Performance Computing (HPC) environments with 24% of the most powerful supercomputers in the June 2008 ranking of the TOP 500 list using InfiniBand [30]. InfiniBand implementations run at several speeds from Single Data Rate (SDR), with 2 GB/s bidirectional bandwidth up to Quad Data Rate (QDR), that is 8 GB/s bidrectional bandwidth. Processes using InfiniBand may also communicate with very low latency. Current generation InfiniBand adapters can communicate with a process-to-process latency of under 2 μs[15]. The IBA standard allows I/O nodes to be directly connected to the InfiniBand fabric by using a Target Channel Adapter (TCA). iSCSI may be implemented directly over RDMA and is called iSER. iSER eliminates the overhead of TCP/IP present in iSCSI type protocols. iSER allows us to build SAN type networks with InfiniBand [89]. Implementations of InfiniBand include those from Mellaox Technologies [14] and PathScale [9]. InfiniBand has also made its foray into the wide area network [78]. Figure 1.12 and Figure 1.13 are pictures of an InfiniBand Fibre cable and switch respectively.

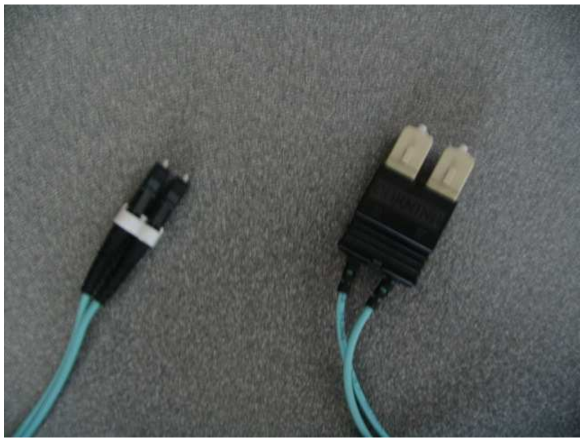


Figure 1.12: InfiniBand Fibre Cables

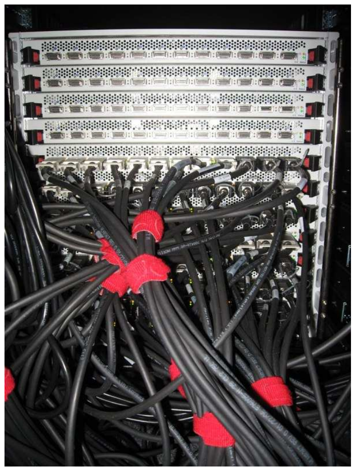


Figure 1.13: InfiniBand Switch

CHAPTER 2

PROBLEM STATEMENT

Commodity clusters have come a long way with the first petascale cluster being unveiled [26]. The applications on these systems consume and generate vast quantities of data. The storage requirements from these applications have stretched current storage architectures to their breaking point. We identify the following broad limitations in current storage systems:

- **Protocol Overhead with high-speed interconnects:** Most storage systems and architectures use protocols such as TCP/IP and UDP/IP. These protocols were designed for slower speed networks such as Gigabit ethernet. The memory bandwidth of machines is usually order of magnitude higher than the gigabit ethernet network bandwidth. As a result, the two or more memory copies in the protocol stack and checksumming for reliability do not severely impact the ability of the CPU to keep the network busy. However, with 10 Gb/s networks such as InfiniBand, the ratio of CPU speed to network speed has shifted towards the network. The CPU is no longer able to keep the network pipeline filled. The problem only becomes worse as networks move to Double Data Rate (DDR) InfiniBand and Quad Data Rate (QDR) [58].

- **Novel Communication Features of High-Speed Interconnects:** Existing Communication Protocols are unable to take advantage of the communication architecture of high-performance interconnects such as InfiniBand Remote Direct Memory Access (RDMA) and iWARP. These high-speed interconnects offer kernel bypass, low overhead, zero-copy, reliable in-order messaging. Existing storage protocol stacks are not designed to take advantage of these novel network features.

- **Scalability of network protocol stacks in parallel file system environments:** To meet the storage performance demands of large clusters, clustered storage NAS (CNAS) systems have evolved. These CNAS systems have multiple data servers and one or more metadata servers. Storing data in parallel on one or more different data servers offer several advantages; most important among them is the benefit of multiple streams of data from different servers. However, as the number of data servers increases, the multiple streams of data into a single node cause protocols such as TCP/IP to suffer from problems such as incast [46]. In incast, the overhead from multiple incoming data streams causes reliability timeouts in the TCP/IP stack, forcing extraneous retransmissions and reducing the throughput to a paltry percentage of the peak aggregate network bandwidth.

- **Mechanical Limitations of Magnetic Disks:** The vast majority of existing storage devices are magnetic disks. This is largely because of their vast storage capacity and reasonable cost. However, as with all mechanical devices, magnetic disks are fundamentally limited by the rotational speed of the disk platters in the bandwidth they can deliver to end-applications. Techniques such as RAID [67] might be able to help improve performance by stringing multiple disks together. However, the aggregate bandwidth of RAID arrays is still limited to less than or equal to the sum of the bandwidth of each individual disk. In addition, RAID-arrays usually exhibit fairly poor performance on random access patterns and write operations [53]. As CPU and main memory increases in speed, there is widening gap between what applications require and what can be offered by the storage subsystem.

- **Limitations in Storage Systems In Long-Haul Networks:** Geographically separated large scale clusters are becoming increasingly common [19]. Different components of large scale parallel applications may increasingly run in geographically disjoint portions of the cluster. Increasingly, the links between these geographically separated systems are being connected by high-performance interconnects such as InfiniBand and 10 Gigabit Ethernet [77]. Traditional storage systems are typically not designed and optimized for use in these geographically separated clusters connected by high-performance interconnects.

- **Fault-tolerance and large scale HPC applications:** Large scale applications generally run for long periods of time and generally solve valuable problems in science and technology. To guard against unforeseen consequences such as faults in computer systems, most large-scale applications use checkpoint schemes at regular intervals of time. Depending on the the characteristics of the underlying storage subsystem, checkpointing might dramatically increase the running time of the application.

We now discuss our research approach to solving these problems in the next section.

2.1 Research Approach

To help overcome these limitations of storage subsystems above, we use the following six research approaches shown in Figure 2.1; *RDMA enabled communication subsystem for NFSv3 (RPC_v3 with RDMA)*, *RDMA enabled transport for NFSv4 (RPC_v4 with RDMA)*, NFS with RDMA in an InfiniBand WAN environment, *RDMA enabled communication subsystem for parallel NFS (pNFS)*, *Caching Techniques to improve the performance of a NAS* and *Checkpointing with large-scale applications*. We describe each of these components below.

- **RDMA enabled communication subsystem for NFSv3 (RPC_v3 over RDMA).** In this component, we explore the trade-offs, merits and demerits of different strategies for enabling the communication substrate Remote Procedure Call (RPC) in the Network File System (NFSv3) with RDMA. We investigate the security, performance and scalability implications of two different designs; namely a server oriented design and a combination of a client and server oriented design. We also explore a variety of communication buffer pinning and registration techniques.

- **A communication subsystem for NFSv4 with RDMA (RPC_v4 over RDMA).** We investigate how the techniques researched for NFSv3 may be applied to NFSv4, the successor protocol to NFSv3. The NFSv4 protocol differs from NFSv3 in the methods, techniques and semantics used to provide performance. In our research, we look at how our existing RPC over RDMA design may be optimized for NFSv4.

- **NFS communication subsystem in a WAN environment.** Geographically separated HPC clusters with high-speed links are increasingly being deployed. As part of this research, we investigate the issues surrounding storage systems in a wide area network (WAN) environment. Specifically, we evaluate the performance of RPC over RDMA in an InfiniBand WAN environment.

- **RDMA enabled communication subsystem for pNFS.** Here we research how to design RPC over RDMA for NFSv4.1. NFSv4.1 is composed of parallel NFS (pNFS) and sessions. The pNFS architecture consists of different components; a metadata server, clients and data-servers. pNFS is architected to communicate with a variety of different file system architectures; file, block and object. We mainly focus on designing a high-performance RPC over RDMA for pNFS file architecture. We look at the trade-offs of different RPC over RDMA connections from client to metadata server, from client to the data servers and from the metadata server to the data server. We also provide a comprehensive evaluation of the design in terms of both micro-benchmarks and applications.

- **Caching techniques to improve the performance a NAS.** Modern storage systems are limited by the performance of disk based storage, which is the most common back-end for storing data. As CPU speeds increase dictated by Moore's law, multi-core environments and improving memory bandwidth, the gap between CPU and disk performance continues to widen. This translates into poor performance from the storage subsystem. As a result, applications that are I/O bound are directly impacted by these limitations. Parallel file systems are especially impacted by these limitations, since the data is usually stored on the back-end on multiple disks. To reduce the impact of these limitations, we propose, design and evaluate a data caching architecture for a parallel/distributed file system. The caching architecture is especially designed for environments where there is considerable read and write I/O sharing.

- **Impact of high-performance networks on distributed and parallel file systems.** Large scale applications can run for hours and days at a time. These applications perform crucial roles in scientific and engineering fields. Their failures can have considerable security, environmental and economic impact. Unfortunately, with large scale commodity clusters, the likelihood and actual occurrence of faults is considerably magnified. To guard against these faults, most applications employ some form of checkpointing. To reduce the complexity of checkpointing, application transparent, system-level checkpointing is generally provided. System-level checkpointing generally depends on the performance of the storage subsystem. As a part of this dissertation, we evaluate the impact of the storage subsystem on the performance of system-level checkpointing.

2.2 Dissertation Overview

In this dissertation, we propose, design and evaluate a high-performance storage architecture over InfiniBand. Chapter 3 presents our design for Network File System (NFS), NFSv3 over InfiniBand. Following that, in Chapter 4, we look at designing NFSv4 over InfiniBand. Next, in Chapter 5, we evaluate these protocols in a wide area network (WAN)

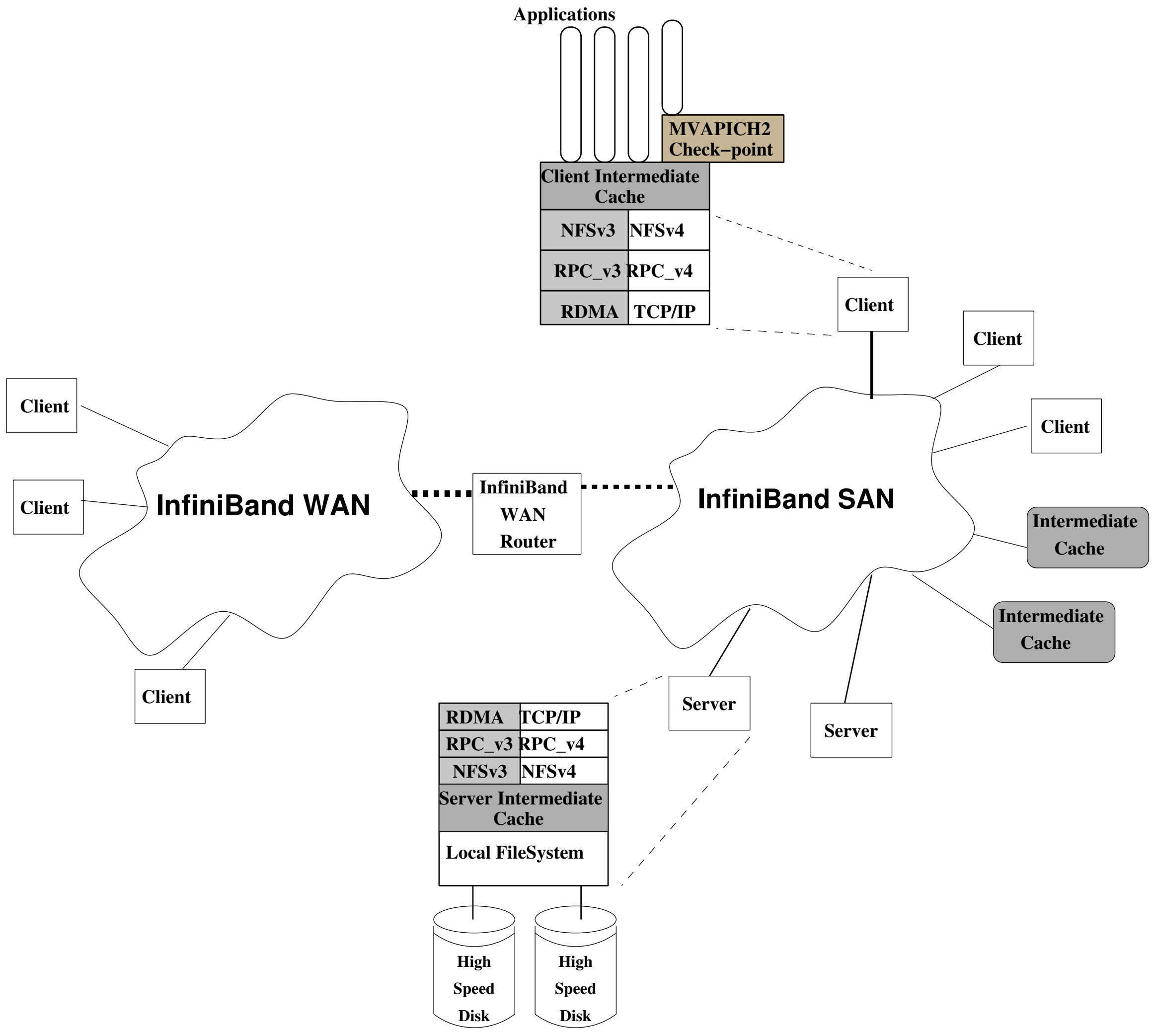


Figure 2.1: Proposed Architecture

scenario. Chapter 6, looks at our design for parallel NFS and sessions. Following that, Chapter 7 provides a discussion of the caching architecture for high-performance parallel and storage systems. After that, Chapter 8 evaluates the impact of high-performance I/O on checkpointing of parallel applications. Next, Chapter 9 provides the details of the open-source design availability. Finally, Chapter 10 presents our conclusions and future work.

CHAPTER 3

SINGLE SERVER NAS: NFS ON INFINIBAND

The Network File System (NFS) has become the dominant standard for sharing file in a UNIX clustered environment. In this Chapter, we focus on the challenges of designing a high-performance communication substrate for NFS.

3.1 Why is NFS over RDMA important?

The Network File System (NFS) [36] protocol has become the *de facto* standard for sharing files among users in a distributed environment. Many sites currently have terabytes of storage data on their I/O servers. I/O servers with petabytes of data have also debuted. Fast and scalable access to this data is critical. The ability of clients to cache this data for fast and efficient access is limited, partly because of the demands on main memory on the client, which is usually allocated by memory hungry application such as in-memory database servers. Also, for medium and large scale clusters and environments, the overhead of keeping client caches coherent quickly becomes prohibitively expensive. Under these conditions, it becomes important to provide efficient low-overhead access to data from the NFS servers.

NFS performance has traditionally been constrained by several factors. First, it is based on the single server, multiple client model. With many clients accessing files from the NFS server, the server may quickly become a bottleneck. Servers with 64-bit processors commonly have a large amount of main memory, typically 64GB or more. File systems like XFS can take advantage of the main memory on these servers to aggressively cache and prefetch data from the disk for certain data access patterns [91]. This mitigates the disk I/O bottleneck to some extent. Second, limitations in the Virtual file system (VFS) interface, force data copying from the file system to the NFS server. This increases CPU utilization on the server. Third, traditionally used communication protocols such as TCP and UDP require additional copies in the stack. This further increases CPU utilization and reduces the operation processing capability of the server. With an increasing number of clients, protocols like TCP also suffer from problems like incast [46], which forces timeouts in the communication stack, and reduces overall throughput and response time. Finally, an order of magnitude difference in bandwidth between commonly used networks like Gigabit Ethernet (125 MB/s) and the typical memory bandwidth of modern servers (2 GB/s or higher) can also be observed. This limits the stripping width, resulting in more complicated designs to alleviate this problem [46].

Modern high-performance networks such as InfiniBand provide low-latency and high-bandwidth communication. For example, the current generation Single Data Rate (SDR) NIC from Mellanox has a 4 byte message latency of less than 3μs and a bi-directional bandwidth of up to 2 GB/s for large messages. Applications can also deploy mechanisms like

Remote Direct Memory Access (RDMA) for low-overhead communication. RDMA operations allow two appropriately authorized peers to read and write data directly from each others address space. RDMA requires minimal CPU involvement on the local end, and no CPU involvement on the remote end. Since RDMA can directly move data from the application buffers on one peer into the applications buffers on another peer, it allows designers to consider zero-copy communication protocols. Designing the stack with RDMA may eliminate the copy overhead inherent in the TCP and UDP stacks. Additionally, the load on the CPU can be dramatically reduced. This benefits both the server and the client. Since the utilization on the server is reduced, the server may potentially process requests at a higher rate. On the client, additional CPU cycles may be allocated to the application. Finally, an RDMA transport can better exploit the latency and bandwidth of a high-performance network.

An initial design of NFS/RDMA [42] for the OpenSolaris operating system was designed by Callaghan, et.al.. This design allowed the client to read data from the server through RDMA Read. An important design consideration for any new transport is that it should be as secure as a transport based on TCP or UDP. Since RDMA requires buffers to be exposed, it is critical that only trusted entities be allowed to access these buffers. In most NFS deployments, the server may be considered trustworthy; the clients cannot be trusted. So, exposing server buffers makes the server vulnerable to snooping and malicious activity by the client. Callaghan's design exposed server buffers and therefore suffered from a security vulnerability. Also, inherent limitations in the design of RDMA Read reduce the number of RDMA Read operations that may be issued by a local peer to a remote peer. This throttles the number of NFS operations that may be serviced concurrently, limiting performance. Finally, Callaghan's design did not address the issue of multiple buffer copies. Our experiments with the original design of NFS/RDMA reveal that on two Opteron 2.2 GHz systems with x8 PCI-Express Single Data Rate (SDR) InfiniBand adapters capable of a unidirectional bandwidth of 900 MegaBytes/s (MB/s), the IOzone [11] multi-threaded Read bandwidth saturates at just under 375 MB/s.

In this Chapter, we take on the challenge of designing a high performance NFS over RDMA for OpenSolaris. We discuss the design principles for designing NFS protocols with RDMA. To this end we take an in-depth look at the security and buffer management vulnerabilities in the original design of NFS over RDMA on OpenSolaris. We also demonstrate the performance limitations of this RDMA Read based design. We propose and evaluate an alternate design based on RDMA Read and RDMA Write. This design eliminates the security risk to the server. We also look at the impact of the new design on buffer management.

We try to evaluate the bottlenecks that arise while using RDMA as the underlying transport. While RDMA operations may offer many benefits, they also have several constraints that may essentially limit their performance. These constraints include the requirement that all buffers meant for communication must be pinned and registered with the HCA. Given that NFS operations are short lived, bursty and unpredictable, buffers may have to be registered and deregistered on the fly to conserve system resources and maintain appropriate security restrictions in place on the system. We explore alternative designs that may potentially reduce registration costs. Specifically, our experiments show that with appropriate registration strategies, an RDMA Write based design can achieve a peak IOzone Read throughput of over 700 MB/s on OpenSolaris and a peak Read bandwidth of close to 900 MB/s for Linux. Evaluation with an Online Transaction Processing (OLTP) workload show that the higher throughput of our proposed design can improve performance up to 50%. We also evaluate the scalability of the RDMA transport in a multi-client setting, with a RAID array of disks. This evaluation shows that the Linux NFS/RDMA design can provide an aggregate throughput of 900 MB/s to 7 clients, while NFS on a TCP transport saturates at 360 MB/s. We also demonstrate that NFS over an RDMA transport is constrained by the performance of the back-end file system, while a TCP transport itself becomes a bottleneck in a multi-client environment.

In this Chapter, we investigate and contribute to the following:

- A comprehensive discussion of the design considerations for implementing NFS/RDMA protocols.
- A high performance implementation of NFS/RDMA for OpenSolaris, and a discussion of its relationship to a similar implementation for Linux.
- An in-depth performance evaluation of both designs.
- Design considerations for the relative limitations and potential solutions to the problem of registration overhead.

- Application evaluation of the NFS/RDMA protocols, and the impact of registration schemes such as Fast Memory Registration and All Physical Registration, and a buffer registration cache design on performance.

- Impact of RDMA on the scalability of NFS protocols with multiple clients and real disks supporting the back-end file system.

The rest of the Chapter is presented as follows. Section 3.2 provides an overview of the InfiniBand Communication model. Section 3.3 explores the existing NFS over RDMA architecture on OpenSolaris and the Linux. In Section 3.4, we propose our alternate design based on RDMA Read and RDMA Write and compare it to the original design based on RDMA Read only. Section 3.5 presents the performance evaluation of the design. Finally, Section 3.6 presents our summary.

3.2 Overview of the InfiniBand Communication Model

InfiniBand uses the Reliable Connection (RC) model. In this model, each initiating node needs to be connected to every other node it wants to communicate with through a peer-to-peer connection called a queue-pair (send and receive work queues). The queue pairs are associated with a completion queue (CQ). The connections between different nodes need to be established before communication can be initiated. Communication operations or Work Queue Requests (WQE) operations are posted to a work queue. The completion of these communication operations is signaled by *completion events* on the completion queue. InfiniBand supports different types of communication primitives. These primitives are discussed next.

3.2.1 Communication Primitives

InfiniBand supports two-sided communication operations, that require active involvement from both the sender and receiver. These two-sided operations are known as *Channel Semantics*. One-sided communication primitives, called *Memory Semantics*, do not require involvement by the receiver. *Channel* and *Memory semantics* are discussed next.

Channel Semantics: Channel semantics or Send/Receive operations are traditionally used for communication. A receive descriptor or RDMA Receive (RV) that points to a pre-registered fixed length buffer, is usually posted on the receiver side to the receive queue before the RDMA Send (RS) can be initiated on the senders side. The receive descriptors are usually matched with the corresponding send in the order of the descriptor posting. On the sender side, receiving a completion notification for the send indicates that the buffer used for sending may be reused. On the receiver side, getting a receive completion indicates that the data has arrived and is available for use. In addition, the receive buffer may be reused for another operation.

Memory Semantics: Memory semantics or Remote Direct Memory Access (RDMA) are one-sided operations initiated by one of the peers connected by a queue pair. The peer that initiates the RDMA operation (*active peer*) requires both an address (either virtual or physical), as well as a steering tag to the memory region on the remote peer (*passive peer*). The steering tag is obtained through memory registration [49]. To prepare a region for a memory operation, the passive peer may need to do memory registration. Also a message exchange may be needed between the active and passive peers to obtain the message buffer addresses and steering tags. RDMA operations are of two types, *RDMA Write* (RW) and

RDMA Read (RR). *RDMA Read* obtains the data from the memory area of the passive peer and deposits it in the memory area of the active peer. RDMA Write operations on the other hand move data from the memory area of the active peer to corresponding locations on the passive peer.

A comparison of the different communication primitives in terms of Security (Receive Buffer Exposed), Involvement of the receiver (Receive Buffer Pre-Posted), Buffer protection (Steering Tag) and finally, Peer Message Exchanges for Receive Buffer Address and Steering Tag (Rendezvous) is shown in Table 3.1.

	Channel Semantics	Memory Semantics
Receive Buffer Exposed		✓
Receive Buffer Pre-Posted	✓	
Steering Tag		✓
Rendezvous		✓

Table 3.1: Communication Primitive Properties

3.3 Overview of NFS/RDMA Architecture

NFS is based on the single server, multiple client model. Communication between the NFS client and the server is via the Open Network Computing (ONC) remote procedure call (RPC) [36]. RPC is an extension to the local procedure calling semantics, and allows programs to make calls to nodes on remote nodes as if it were a local procedure call. Callaghan et.al. designed an initial implementation of RPC over RDMA [41] for NFS. This existing architecture is shown in Figure 3.1. The Linux NFS/RDMA implementation has a similar architecture. The architecture was designed to allow transparency for applications accessing files through the Virtual File System (VFS) layer on the client. Accesses to the file system go through VFS, and are routed to NFS. For an RDMA mount, NFS will make the RPC call to the server via RPC over RDMA. The RPC Call generally being small will go as an inline request. Inline requests are discussed in the next section. In the rest of the Chapter, we use the terms RPC/RMDA and NFS/RDMA interchangeably.

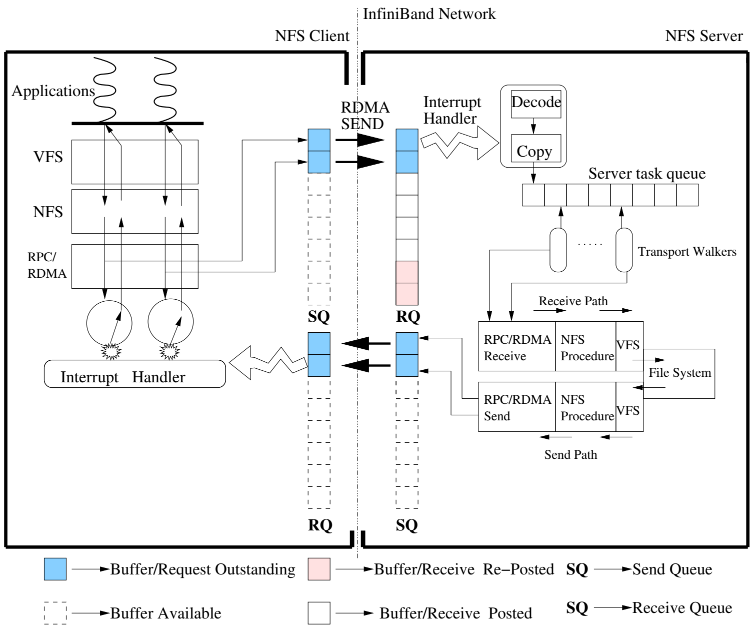
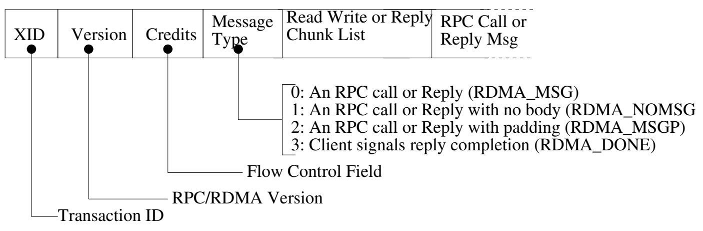


Figure 3.1: Architecture of the NFS/RDMA stack in OpenSolaris

3.3.1 Inline Protocol for RPC Call and RPC Reply

The RPC Call and Reply are usually small and within a threshold, typically less than one 1KB. In the RPC/RDMA protocol the call and reply may be transferred *inline* via a copy based protocol similar to that used in MPI stacks such as MVAPICH [15]. The copy based protocol uses the channel semantics of InfiniBand described in Section 3.2.1. During startup (at mount time), after the InfiniBand connection is established, the client and server each will establish a pool of send and receive buffers. The server posts receive buffers from the pool on the connection. The client can send requests to the server up to the maximum pool size using RDMA Send operations. This exercises a natural upper limit on the number of requests that the client may send to the server. The upper limit can be adjusted dynamically; this may result in better resource utilization. The OpenSolaris server sets this limit at 128 per client; the Linux server sets it at 32 per client. At the time of making the RPC Call, the client will prepend an RPC/RDMA header (Figure 3.2) to the NFS Request passed down to it from the NFS layer as shown in Figure 3.1. It will post a receive descriptor from the receive pool for the RPC Call, then issue the RPC Call to the server through an RDMA Send operation. This will invoke an interrupt handler on the OpenSolaris server that will copy out the request from the receive buffer and repost it to the connection. (The Linux server does not do the copy, and reposts the receive descriptor at a somewhat later time.) The request will then be placed in the servers task queue. A transport context thread will eventually pick up the request that will then be decoded by the RPC/RDMA layer on the server. Bulk data transfer chunks will be decoded and stored at this point. The request will then be issued to the NFS layer. The NFS layer will then issue



XID
Version
Credits
Message Type
Read Write or Reply Chunk List
RPC Call or Reply Msg
0: An RPC call or Reply (RDMA_MSG)
1: An RPC call or Reply with no body (RDMA_NOMSG
2: An RPC call or Reply with padding (RDMA_MSGP)
3: Client signals reply completion (RDMA_DONE)
Flow Control Field
RPC/RDMA Version
Transaction ID

Figure 3.2: RPC/RDMA header

the request to the file system. On the return path from the file system, the request will pass through the NFS layer. The NFS layer will encode the results and make the RPC Reply back to the client. The interrupt handler at the client will wake up the thread parked on the request and control will eventually return to the application.

3.3.2 RDMA Protocol for bulk data transfer

NFS procedures such as READ, WRITE, READLINK and READDIR can transfer data whose length is larger than the inline threshold [36]. Also, the RPC call itself can be larger than the inline data threshold. The bulk data can be transferred in multiple ways. The existing approach is to use RDMA Read only and is referred to as the Read-Read design. Our approach is to use a combination of RDMA Read and RDMA Write operations and is called the Read-Write design. We describe both these approaches in detail. Before we do that, we define some essential terminologies.

Chunk Lists: These lists provide encoding for bulk data whose length is larger than the *inline threshold* and should be moved via RDMA. A chunk list consists of a single counted array of segments of one or more lists. Each of these lists is in turn a counted array of zero or more segments. Each segment encodes a steering tag for a registered buffer, its length and its offset in the main buffer. Chunks can be of different types; *Read chunks*, *Write chunks* and *Reply chunks*.

- *Read chunks* used in the Read-Read and Read-Write design encode data that may be RDMA Read from the remote peer.
- *Write chunks* used in the Read-Write design are used to RDMA Write data to the remote peer.
- *Reply chunks* used in the Read-Write design are used for procedures such as READDIR and READLINK, and are used to RDMA Write the entire NFS response.

The *RPC Long Call* is typically used when the RPC request itself is larger than the inline threshold. The *RPC Long Reply* is used in situations where the RPC Reply is larger than the inline size. Other bulk data transfer operations include *READ* and *WRITE*. All these procedures are discussed in the next section.

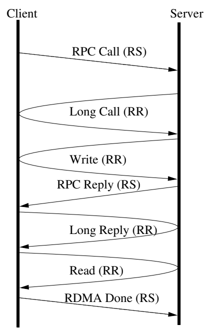


Figure 3.3: Read-Read Design

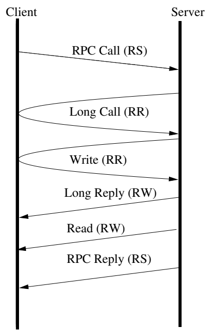


Figure 3.4: Read-Write Design

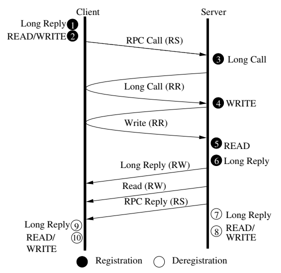


Figure 3.5: Registration points (Read-Write)

3.4 Proposed Read-Write Design and Comparison to the Read-Read Design

In this section, we discuss our proposed Read-Write design, which is based on a combination of RDMA Read and RDMA Write. We also compare with the original Read-Read based design, which is based on RDMA Read. We discuss the limitations of the Read-Read based design. Following that, we also discuss the advantages of the Read-Write design. We look at registration strategies and designs in Section 3.4.3. The Read-Read based design is show in Figure 3.3. The Read-Write design is shown in Figure 3.4.

RPC Long Call: The RPC Long Call is typically used when the RPC request itself is larger than the inline threshold. In this case, the client encodes a chunk list along with a RDMA_NOMSG flag in the header shown in Figure 3.2. It is always combined with other NFS operations. The RPC Long Call is identical in both the Read-Read and Read-Write based designs. If the RPC Call message is larger than the inline size, the RPC Call from the client includes a Read Chunk List. The message type in the header in Figure 3.2 is set to RDMA_NOMSG. When the server sees an RDMA_NOMSG message type, it decodes the read chunks encoded in the RPC/RDMA header and issues RDMA Reads to fetch these chunks from the client. The data from these chunks constitutes the remainder of the header (the fields *Read, Write or Reply Chunk List* onwards in Figure 3.2, which are overwritten by the incoming data). The remainder of the header usually constitutes other NFS procedures as discussed in Section 3.3.2 is then decoded.

NFS Procedure WRITE: The NFS Procedure WRITE is similar in both the Read-Read and Read-Write based designs. For an NFS procedure WRITE, the client encodes a

Read chunk list. On the server side, these read chunks are decoded, the RDMA Reads corresponding to each segment are issued and the server thread blocks till the RDMA Reads complete. The operation is then handled by the NFS layer. Once the operation completes, control is returned to the RPC layer, that sends an RPC Reply via the inline protocol described in Section 3.3.1. In the simplest case, an NFS Procedure WRITE would generate an RPC Call (RS) from the client to the server, followed by the WRITE (RR) issued by the server to fetch the data from the client, and finally, the RPC Reply (RS) from the server to the client.

NFS Procedure READ: In the Read-Read design the NFS server needs to encode a *Read chunk list* in the RPC Reply for an NFS READ Procedure. The RPC Reply is then returned to the client via the inline protocol in Section 3.3.1. The client decodes the Read chunk lists and issues the RDMA Reads. Once the RDMA Reads complete, the client issues an RDMA_DONE to the server, that allows it to free its pre-registered buffers. So, the simplest possible sequence of operations for an NFS Procedure READ is; RPC Call (RS) from the client to the server, followed by an RPC Reply (RS) from the server to the client, then a READ (RR) issued by the client to fetch the data from the server, and finally, an RDMA_DONE (RS) from the client to the server.

In the Read-Write design, for a NFS READ procedure, the client needs to encode a Write chunk list in the RPC Call. The server decodes and stores the Write chunk list. When the NFS procedure READ returns, the data is RDMA written back to the client. The server then sends the RPC Reply back to the client with an encoded Write Chunk List. The client uses this Write chunk list to determine how much data was returned in the READ call. So, the simplest possible protocol operations would be; RPC Call from the client to the server, then a Read (RW) from the server to the client, and finally, an RPC Reply (RS) from the server to the client.

NFS Procedure READDIR and READLINK (RPC Long Reply): The RPC Long Reply is typically used when the RPC Reply is larger than the inline size. The RPC Long Reply is used in both the Read-Read and Read-Write designs but the mechanisms are different. It may either be used independently, or combined with other NFS operations.

The design of the NFS procedure READDIR/READLINK in the Read-Read design is similar to the NFS Procedure READ in the Read-Read design. The server encodes a *Read chunk list* in the RPC Reply, that the client decodes. The client then issues RDMA Read to fetch the data from the server. Once the RDMA Reads complete, the client issues an RDMA_DONE to the server which allows the server to free its pre-registered buffers.

NFS Procedure READDIR and READLINK in the Read-Write design follows the design of the NFS READ procedure in the Read-Write design. The client needs to encode a Long Reply chunk list in the RPC Call. The server decodes and stores the Long Reply chunk list. When the NFS procedure returns, the server uses the long reply chunk to RDMA Write the data back to the client. The server then sends the RPC Reply back to the client with an encoded Long Reply Chunk List. The client uses this chunk list to determine how much data was returned in the READDIR/READLINK call. In the simplest case, an RPC Long Reply would entail the following sequence; RPC Call from the client to the server, then a Long Reply (RW) from the server to the client, and finally, an RPC Reply (RS) from the server to the client.

Zero Copy Path for Direct I/O for the NFS READ procedure: In addition to the basic design, we also introduce a zero copy mechanism for user space addresses on the NFS READ procedure path. This eliminates copies on the client side and translates into reduced CPU utilization on the client.

3.4.1 Limitations in the Read-Read Design

The Read-Read design has a number of limitations in terms of Security and Performance, and we discuss these issues in detail.

Security:

Server buffers exposed: An important design consideration for an RDMA enabled RPC transport is that it must not be less secure than other transports such as TCP. In the Read-Read design, the server side buffers are exposed for RDMA operations from the client. Since the steering tags are 32-bits [49] in length, a misbehaving or malicious client might attempt to guess them and thereby possibly read a buffer for which it did not have access to.

Malicious or Malfunctioning clients: The client needs to send an *RDMA_DONE* message to the server to indicate that the buffers used for a Read or Reply chunk may be freed up. A malicious of malfunctioning client may never send the *RDMA Done* message, essentially tying up the server resources.

Performance:

Synchronous RDMA Read Limitations: The RDMA Read issued from the NFS/RDMA server are synchronous operation. Once posted, the server typically has to wait for the

RDMA Read operation to complete. This is because the InfiniBand specification does not guarantee ordering between a RDMA Read and a RDMA Send on the same connection [49]. This may add considerable latency to the server thread.

Outstanding RDMA Reads: The number of RDMA Read that can be typically serviced on a connection is governed by two parameters, the Inbound RDMA Read Queue Depth (IRD) and the Outbound RDMA Read Queue Depth (ORD) [49]. The IRD governs the number of RDMA Read that can be active at the remote peer; the ORD governs the number of RDMA Read that might be actively issued concurrently from the local peer. In the current Mellanox implementation of InfiniBand, the maximum allowed value for IRD and ORD is typically 8 [61]. So, parallelism is reduced at the server, especially for multi-threaded workloads.

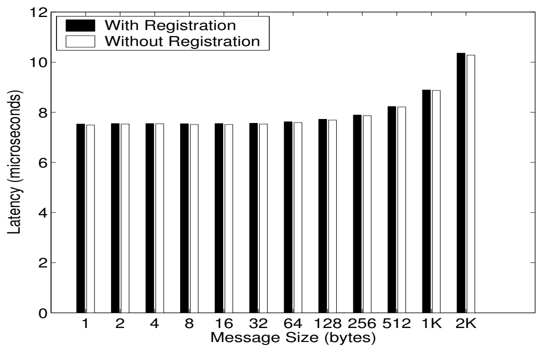
3.4.2 Potential Advantages of the Read-Write Design

The key design difference between the Read-Read (Figure 3.3) and Read-Write (Figure 3.4) protocol is that RPC long replies and NFS READ data may be directly issued from the server. To enable these, the client needs to encode either a Write chunk list or a long reply chunk list (Section 3.3.2). Moving from a Read-Read based design to a Read-Write based design has several advantages. The Mellanox InfiniBand HCA has the ability to issue many RDMA Write operations in parallel [61]. This reduces the bottleneck for multi-threaded workloads. Also, since completion ordering between RDMA Write and RDMA Sends is guaranteed in InfiniBand [49], the server does not have to wait for the RDMA Writes from the long reply or the NFS READ operation to complete. The completion generated by the RDMA Send for the RPC Reply will guarantee that the earlier RDMA Writes have completed. This optimization also helps reduce the number of interrupts generated on the server. The RDMA_DONE message and its resulting interrupt is also eliminated. The generation of the send completion interrupt on the server is sufficient to guarantee that the RDMA operations from the buffers have completed and they may be deregistered. A similar guarantee also exists at the client, when an RPC Call message is received. The elimination of an additional message helps improve performance. Since the server buffers are no longer exposed and the client cannot initiate any RDMA operations to the server, the security of the server is now enhanced. One potential disadvantage of the Read-Write design is that the client buffers are now exposed and may be corrupted by the server. Since the server is usually a trusted entity in an NFS deployment, this issue is less of a concern. The final advantage of the Read-Write design is that the server no longer has to depend on the RDMA_DONE message from the client to deregister and release it buffers.

3.4.3 Proposed Registration Strategies For the Read-Write Protocol

InfiniBand requires memory areas to be registered for communication operations [49, 63]. Registration is a multi-stage operation. Registration involves assigning physical pages to the virtual area. Once physical pages have been assigned to the virtual area, the virtual to physical address translation needs to be determined. In addition, the physical pages need to be prepared for DMA operations initiated by the HCA. This involves making the pages unswappable by the operating system, by pinning them. The virtual memory system may perform both these operations. In addition, the HCA needs to be made aware of the translation of the virtual to physical addresses. The HCA also needs to assign a *steering tag* that may be sent to remote peers for accessing the memory region in RDMA operations. The virtual to physical translation and the steering tag are stored in the HCA's Translation Protection Table (TPT). This involves one transaction across the I/O bus. However, the response time of the HCA may be quite high, depending on the load on the HCA, the organization of the TPT, allocation strategies, overhead in the TPT, and so on [63]. Because of the combination of these factors, registration is an expensive operation and may constitute a considerable overhead, especially when it is in the critical path. Deregistering a buffer requires the actions from registration to be done in reverse. The virtual and physical translations and steering tags need to be flushed from the TPT (this involves a transaction across the I/O bus). Once the TPT entries are invalidated, each of them is released. The pages may then be unpinned. If the physical pages were assigned to the virtual memory region at the time of registration, this mapping is torn down and the physical pages are released back into the memory pool.

Figure 3.6 shows the half ping-pong latency at the InfiniBand Transport Layer (IBTL) on OpenSolaris of a message with and without registration costs included (the setup is the same as described earlier). Half ping-pong latency is measured by using two nodes. Node 1 sends a message (ping) of the appropriate size to node 2. Node 2 then responds with a message of the same size (pong). The latency of the entire operation over 1,000 iterations is measured, averaged out and then divided by 2. The latency without registration is measured by registering a buffer of the appropriate size on both sides before starting the communication loop (that is timed). The latency with registration is measured by registering and unregistering the buffer on both nodes inside the communication loop.



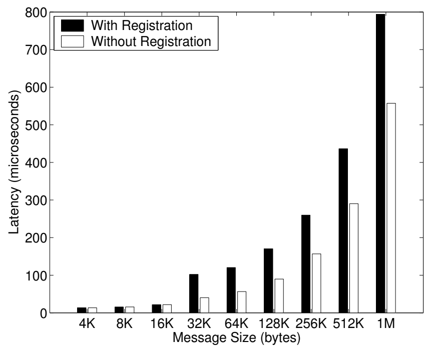


Figure 3.6: Latency and Registration costs in InfiniBand on OpenSolaris

The registration/deregistration points in the RDMA transport are shown in Figure 3.5. For example, an NFS procedure READ in the Read-Write protocol described earlier, requires a buffer registration at points 2 and 5, and a deregistration at points 8 and 10. From Figure 3.5, we can see that the registration overhead comes about mainly because the transport has to register the buffer and deregister the buffer on every operation at the client and server. The registration occurs once at the client, and then at the server in the RPC call path. Following that, deregistration happens once at the server, and then once at the client. To reduce the cost of memory registration, different optimizations and registration modes have been introduced. These include *Fast Memory Registration* [63, 49] and *Physical Registration* [49]. In addition, we propose a buffer registration cache. We discuss these next.

Fast Memory Registration (FMR): Fast Memory Registration [63, 49] allows for the allocation of the TPT entries and steering tags at initialization, instead of at registration time. The other operations of memory pinning, virtual to physical memory address translations and updating the HCA's TPT entries remain the same. The allocated entries in the TPT cache are then mapped to a virtual memory area. This technique is therefore not dependent on the response time of the HCA to allocate and update the TPT entries and consequently, may be considerably faster than a regular registration call. The limitations of FMR include the fact that it is restricted to privileged consumers (kernel), and the fact that the maximum registration area is fixed at initialization.

The Mellanox implementation of FMR [63] introduces additional optimizations to the InfiniBand specification [49]. Similar to the specification [49], it defines a pool of steering tags that may be associated with a virtual memory area at the time of registration. The difference arises at deregistration. The steering tag and virtual memory address is placed on a queue. When the number of entries in the queue crosses a certain threshold called the *dirty watermark*, the invalidations for the entries are flushed to the HCA. This invalidates the TPT entries for the particular set of steering tags and virtual addresses in the queue. While this optimization can potentially improve performance, this introduces a security restriction. While the entries in the queue have not been flushed, there is a window of vulnerability after the deregistration call is made. During this window, a remote peer with the steering tag can access the virtual memory area. We have incorporated FMR calls (Mellanox FMR [63]) in the regular registration path in RPC/RDMA. To allow FMR to work transparently, we use a fall-back path to regular registration calls in case the memory region to be registered is too large.

Design of the Buffer Registration Cache: An alternate registration strategy is to create a buffer registration cache. A registration cache [95] has been shown to considerably improve communication performance. Most registration caches have been implemented at the user level and cache virtual addresses. Caching virtual addresses has been shown to cause incorrect behavior in some cases [69]. Also, unless static limits are placed on the number of entries in the registration cache, the cache tends to expand endlessly, particularly in the face of applications with poor buffer reuse patterns. Finally, static limits may perform poorly depending on the dynamics of the application.

To alleviate some of these deficiencies, we have designed an alternate buffer registration cache on the server. As discussed earlier in Section 3.3, the NFS server state machine is split into two parts. The first part is on the RPC Call receive path where the NFS call is received and is issued to the file system. The second component is on return of control from the file system. Buffer allocation is done when the request is received on the server side and registration is executed when control returns from the file system. To model this behavior, we override the buffer allocation and registration calls and feed them to the registration cache module. This module allocates buffers of the appropriate size from a slab cache [51], for the request and then registers them when the registration request is made. If the buffer from the cache is already registered, no registration cost is encountered. The advantages of this setup are that the cache is no longer based on virtual address, and it is also linked to the systems slab cache, that may reclaim memory as needed. Since the server never sends a virtual address or steering tag to the client for any buffers in the registration cache, this is as secure as regular registration.

The server registration cache scheme described above can also be applied to the client side. However, in order to use the system slab cache, data needs to be copied from the application buffer to an intermediate NFS buffer. Therefore, compared with the zero-copy path mentioned in Section 3.4, there is an extra data movement involved in the registration cache scheme, and we need to carefully study the trade-off between data copy and memory registration. Since a malfunctioning server may compromise the integrity of the clients buffers, this approach should be used in which the server buffers are well tested.

All Physical Memory Registration: In addition to virtual addresses, communication in InfiniBand may also take place through physical addresses. Physical Registration takes two different forms, i.e. mapping all of physical memory and the *Global Steering Tag* optimization. Mapping all of physical memory involves updating the HCA's TPT entries to map all physical pages in the system with steering tags. This operation places a considerable burden on the HCA in modern systems which may have GigaBytes of main memory and is usually not supported. The *Global Steering Tag* available to privileged consumers (such as kernel processes) allows communication operations to use a special remote steering tag. The communication operation must use a physical addresses. The consumer must pin the memory before communication starts and obtain a virtual to physical mapping, but does not need to register the mapping with the HCA.

Physical Registration can considerably reduce the impact of memory registration on communication, but is restricted to privileged consumers. The issue of security also needs to be considered. The *Global Steering Tag* potentially allows unfettered access to remote peers, that may have obtained the Remote Steering Tag through earlier communication with the peer. It should be used in environments where a level of trust exists between the peers. In addition, the issue of *integrity* should be considered. The HCA is unable to do checks on incoming requests with physical addresses and an associated remote steering tag. Given that the peers can corrupt each others memory areas through a communication operation with an invalid physical address, the *Global Steering Tag* should be used in environments where sufficient confidence exists in the correctness of the communication sub-system. In the Linux implementation, we allocate a *Global Steering Tag* at initialization. By using the *Global Steering Tag*, we only need to do virtual to physical address translation when actually registering memory.

3.5 Experimental Evaluation of NFSv3 over InfiniBand

In this section, we evaluate our proposed RDMA design with NFSv3. We first compare the Read-Write design with the existing Read-Read design on OpenSolaris in Section 3.5.1 (Linux did not have a Read-Read design). Following that, Section 3.5.2 discusses the impact of different registration strategies on NFS/RDMA performance, both at the microbenchmark and at the application-level. Finally, in Section 3.5.3 we discuss how RDMA affects the scalability of NFS protocols in an environment where the server stores the data on a back-end RAID array and services multiple clients.

3.5.1 Comparison of the Read-Read and Read-Write Design

Figures 3.7 and 3.8 show the IOzone [11] Read and Write bandwidth respectively with direct I/O on OpenSolaris. Performance of the Read-Read design are shown as RR. Performance of Read-Write design are shown as RW. The results were taken on dual Opteron x2100's with 2GB memory and Single Data Rate (SDR) x8 PCI-Express InfiniBand Adapters [61]. These systems were running OpenSolaris build version 33. The back-end file system used was tmpfs which is a memory based file system. The IOzone file size used was 128 MegaBytes to accommodate reasonable multi-threaded workloads (IOzone creates a separate file for each thread). The IOzone record size was varied from 128KB to 1MB. From the figure, we make the following observations. For both the Read-Read and

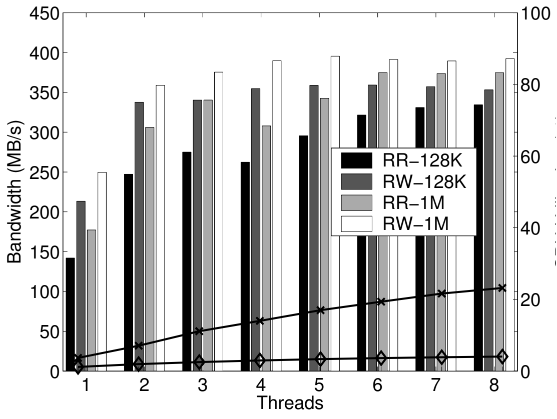


Figure 3.7: IOzone Read Bandwidth

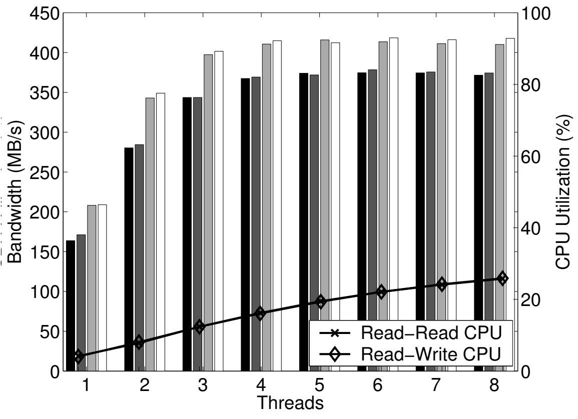


Figure 3.8: IOzone Write Bandwidth

Read-Write design, the bandwidth increases with record size. The RPC/RDMA layer in OpenSolaris does not fragment individual record sizes. The size on the wire corresponds exactly to the record size passed down from IOzone to the NFS layer. Larger messages have better bandwidth in InfiniBand. This translates into better IOzone bandwidth for larger record sizes. Since the size of the file is constant, the number of NFS operations is lower for larger record sizes. So, the improvement in bandwidth with larger record sizes is modest.

IOzone Write bandwidth is the same in both cases. This is to be expected as the NFS WRITE path through the RPC/RDMA layer is the same on the client and server for both the Read-Read and Read-Write designs. The WRITE bandwidth is slightly higher than the READ bandwidth is both designs because the server has to do additional work for READ procedures than for WRITE procedures, affecting its operation processing rate.

The Read-Write design performs better than the Read-Read design for all record sizes, for the READ procedure. The improvement in performance is approximately 47% with one thread at a record size of 128 KB, but decreases to about 5% at 8 threads. This improvement is primarily due to the elimination of the RDMA_DONE message as well as the improved parallelism of issued RDMA Writes from the server. The READ bandwidth for the Read-Read design saturates at 375 MB/s; the Read-Write design saturates at 400 MB/s. The bandwidth in both cases seems to saturate with an increasing number of threads, though the saturation in the case of the RDMA-Write design takes place much earlier than that of the Read-Read design.

Client CPU utilization was measured using the IOzone [11] +*u* option. The utilization corresponds to the percentage of the time the CPU is busy over the lifetime of the throughput test. Since the CPU utilization for different record sizes is the same, we show only a single line for the Read-Read and Read-Write designs in Figures 3.7 and 3.8. Client CPU utilization is lower for Read-Write than for Read-Read for the NFS READ procedure. In addition, the CPU utilization for the Read-Write design remains flat starting at only 2% at 1 thread increasing to about 5% at 8 threads. On the other hand, the CPU utilization for the Read-Read design increases from about 4% at 1 thread to about 24% at 8 threads. This is primarily because of elimination of data copies on the client direct I/O path. Due to the data copy from tmpfs to NFS across the VFS layer, server utilization is close to 100%. So, we do not present server CPU utilization numbers in the graph.

3.5.2 Impact of Registration Strategies

From Section 3.4.3, we see that registration can constitute a substantial overhead in the RPC/RDMA transport. We evaluate the impact of Fast Memory Registration (FMR) and buffer registration cache at the micro-benchmark and application-level. We also look at the performance benefits from the All Physical Registration mode in Linux.

Fast Memory Registration (FMR): We now look at the impact of FMR discussed in Section 3.4.3 on RPC/RDMA performance. The maximum size of the registered area was set to be 1MB. In addition, the FMR pool size was set to 512, which is sufficient for up to 512 parallel requests of 1MB. We evaluate the IOzone read and write bandwidth. Since the bandwidth from the different record sizes are similar, we present results with only a 128KB record size and a 128 MB file size. The results are shown in Figure 3.9 and Figure 3.10. FMR can help improve Read bandwidth from about 350 MB/s to approximately 400 MB/s, though this comes at the cost of increased client CPU utilization (Figure 3.9 shows an upper bound to CPU utilization shown by the legend CPU-Cache-Solaris. CPU Utilization for FMR is between that of CPU-Cache-Solaris and CPU-Register-Solaris). This increased client CPU utilization is to be expected, since the client is able to place more operations per second on the wire, due to the better operation response time from the server. Improvement in write bandwidth is modest, mainly because the time saving from the reduction in registration cost is dwarfed by the serialization of RDMA Reads, due to the limitation in the number of outstanding RDMA Reads (Section 3.4.1).

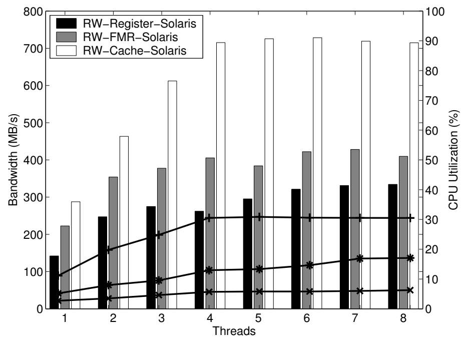


Figure 3.9: IOzone Read Bandwidth with different registration strategies on OpenSolaris

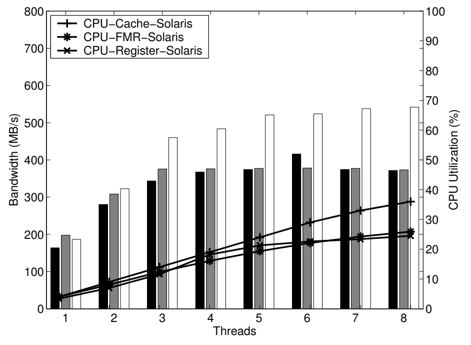


Figure 3.10: IOzone Write Bandwidth with different registration strategies on OpenSolaris

Buffer Registration Cache: The performance impact of the server registration cache on the IOzone Read and Write bandwidth is shown in Figure 3.9 and Figure 3.10 respectively. The registration cache dramatically improves performance for both the Read and Write bandwidth which goes up to 730 MB/s and 515 MB/s respectively. The client CPU utilization is also increased, though this is to be expected with an increasing operation rate from the client. Again, the limited number of outstanding RDMA Reads bounds the improvement in Write throughput. Figure 3.11 shows the impact of the client registration cache scheme on IOzone multi-thread Read test. From the figure we can see that it is beneficial to use the client registration cache when the record size is small. The peak Read throughput doubles for 2KB record size by using registration cache. For large record size, the dynamic registration scheme yields higher throughput for 1-4 threads, because large memory copy is more expensive than registration. Since there is an extra data copy involved, the client registration cache scheme consumes more CPU cycles as expected.

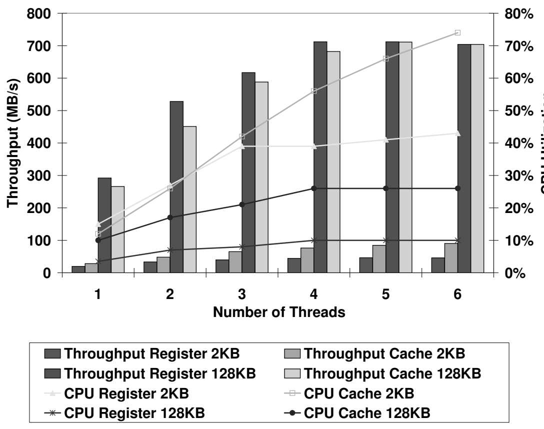


Figure 3.11: Performance Impact of Client Registration Cache on IOzone Read Tests

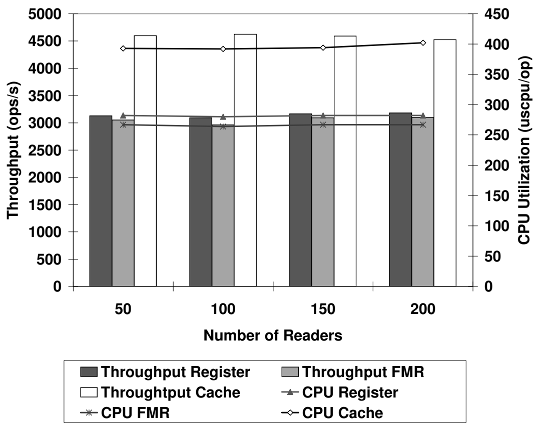


Figure 3.12: FileBench OLTP Performance

Impact of registration schemes on application performance: To evaluate the impact of memory registration schemes on application performance, we have conducted experiments using the online transaction processing (oltp) workload from FileBench [4]. We tune the workload to use the mean I/O size equal to 128KB. The results are shown in Figure 3.12. The bars represent the throughput (operations/sec) and the lines represent the client CPU utilization (us cpu/operation). From Figure 3.12 we can see that the registration cache scheme improves throughput by up to 50% compared with the dynamic registration scheme. This indicates that the improvement in raw read/write bandwidth has been translated into application performance. The CPU utilization is slightly higher because client registration cache involves an additional memory copy, as discussed above. The FMR scheme performs comparably with the dynamic registration scheme in this benchmark.

All Physical Memory Registration: From Figure 3.13 we can see that the *all physical memory registration* mode yields the best Read throughput on Linux. It degrades the Write performance compared with the FMR mode as shown in Figure 3.14 because in all-physical mode the client cannot do local scatter/gather and so has to build more read chunks, therefore, each write request issues multiple RDMA Reads from the server that hits the limit of incoming/outgoing RDMA Reads in InfiniBand.

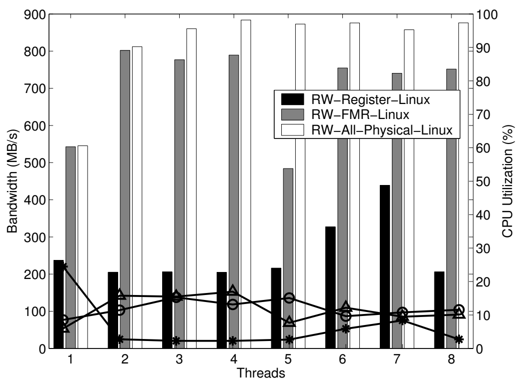


Figure 3.13: IOzone Read Bandwidth with different registration strategies on Linux

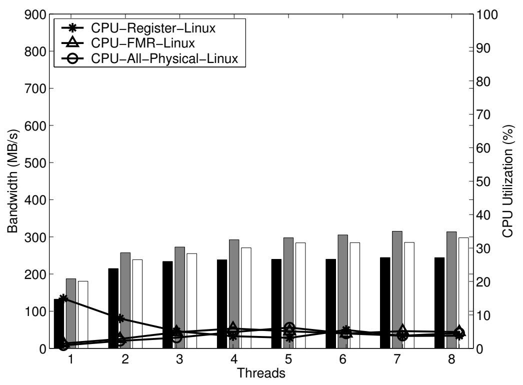


Figure 3.14: IOzone Write Bandwidth with different registration strategies on Linux

3.5.3 Multiple Clients and Real Disks

In this section, we discuss the impact of RDMA on an NFS setup with multiple clients. We start with a description of the multi-client setup. Following that, we look at the performance with two different server configurations.

Multi-client Experimental Setup: We use the Linux NFS/RDMA design with the *All Physical Memory Registration* mode described in Section 3.4.3 for multiple client experiments. The server and clients are dual Intel 3.6 Xeon boxes with an InfiniBand DDR HCA. The clients have 4GB of memory. The server was configured with 4GB and 8GB of memory for each of the experiments below. The server has eight HighPoint SCSI disks with RAID-0 stripping. The disk array is formatted with the XFS file system [91]. For each disk, the single threaded read bandwidth with direct I/O and a record size of 4 KB (NFS/RDMA fragments data requests into 4 KB chunks in the All Physical Memory Registration Mode) for a 1GB file is approximately 30 MB/s. We use a single IOzone process per client. A 1GB file size per process with a 1MB record size is used for all the experiments. We compare the aggregate Read bandwidth of the Linux NFS/RDMA (RDMA) implementation with the regular NFS implementation over TCP on InfiniBand (IPoIB) and Gigabit Ethernet (GigE).

Server with 4GB of main memory: Figure 3.15 shows the IOzone read bandwidth with multiple clients and a server with 4GB main memory. RDMA and IPoIB reach a peak aggregate bandwidth at three processes. RDMA peaks at 883 MB/s, while IPoIB reaches 326 MB/s. In comparison, GigE saturates at 107 MB/s with a single process and then the aggregate bandwidth goes down as the number of processes increases. The limited bandwidth of Gigabit Ethernet (peak theoretical bandwidth of 125 MB/s) may become a bottleneck with future high performance disks and server with large amounts of memory. Analysis with the help of the System Activity Reporter (SAR) tool show that the underlying XFS file system on the server is able to cache the data in memory up to three processes.

So, RDMA is able to achieve the best performance mainly because of elimination of data copies at the client and server. IPoIB needs additional data copies in the TCP stack that limits the potential number of NFS operations that may be in transit at any point in time. As the number of threads increases beyond three, some or all the data must be fetched from the disk. This results in a significant drop in bandwidth. Since, requests from different clients use different files, a disk seek needs to be performed for each request. So, disk seek times dominate and contribute to the overall reduction in throughput.

Server with 8GB of main memory: Figure 3.16 shows the IOzone read bandwidth with 8GB on the server. Clearly, XFS is able to take advantage of the larger memory and cache data for multiple processes. RDMA is able to maintain a peak bandwidth of above 900 MB/s up to seven threads, while IPoIB saturates at about 360 MB/s. At eight threads, the RDMA aggregate bandwidth drops to 624.38 MB/s, while the IPoIB bandwidth drops to 300 MB/s. From Figures 3.15 and 3.16, we can conclude that NFS/RDMA is limited by the ability of the back-end server to service data requests. NFS/TCP is a bottleneck on current generation systems. With servers with 64GB and larger main memories, NFS/RDMA is likely to be the obvious choice for a scalable deployment.

3.6 Summary

In this Chapter, we have designed and evaluated a RPC protocol for high performance RDMA networks such as InfiniBand on NFSv3. This design is based on a combination of RDMA Read and RDMA Write. The design principles considered include NFS server security, performance and scalability. To improve performance of the protocol, we have

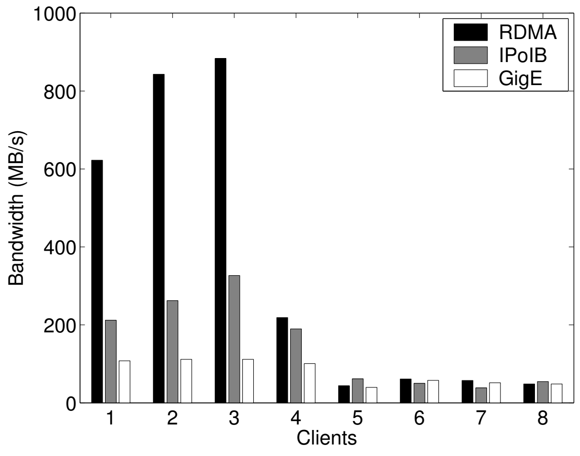


Figure 3.15: Multiple clients IOzone Read Bandwidth (4GB on the server).

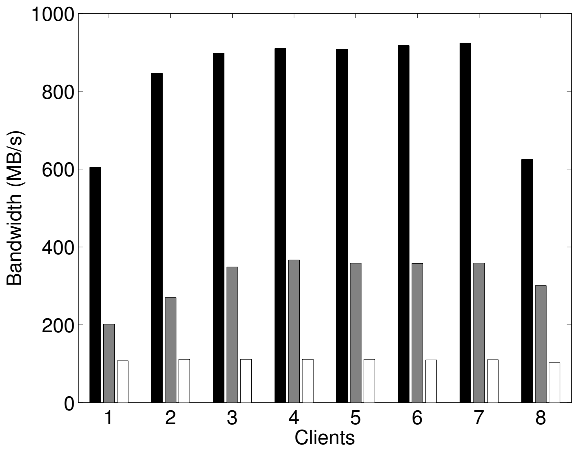


Figure 3.16: Multiple clients IOzone Read Bandwidth (8GB on the server).

incorporated several different registration mechanisms into our design. Our evaluations show that, the NFS/RDMA design can achieve throughput close to that of the underlying network. We also evaluated our design with an OLTP workload. Our design can improve throughput of the OLTP workload by 50%. Finally, we also studied the scalability of NFS/RDMA with multiple clients. This evaluation shows that the Linux NFS/RDMA design can provide an aggregate throughput of 900 MB/s to 7 clients, while NFS on a TCP transport saturates at 360 MB/s. We observe that a TCP transport is itself a bottleneck when servicing multiple clients. By comparison, NFS/RDMA is able to maintain throughput even with multiple clients; provided the back-end file system is able to sustain it.

CHAPTER 4

ENHANCING THE PERFORMANCE OF NFSV4 WITH RDMA

While NFSv3 has been widely deployed, NFSv4 has been gaining traction. Since, NFSv4 offers several advantages over NFSv3, it becomes crucial to design an RDMA transport for NFSv4. We investigate the challenges of designing an RDMA transport for NFSv4 on OpenSolaris.

NFSv4 introduced the concept of a single procedure COMPOUND, which can encompass several operations. This potentially reduces the number of trips to the server. However, it also potentially increases the size of each request sent to the server. A COMPOUND request may potentially have an unbounded length. Correspondingly, the response may also be unbounded. This raises a challenge in a design with RDMA, which requires the length to be precisely specified. Also, COMPOUND procedures may have interactions with other encompassed operations. We demonstrate a design of NFSv4 over an RDMA transport on InfiniBand, that can achieve an IOzone Read Bandwidth of over 700 MB/s and an IOzone Write Bandwidth of over 500 MB/s. It attains performance close to that of the corresponding design of NFSv3 over RDMA and significantly outperforms NFSv4 ove TCP.

4.1 Design of NFSv4 with RDMA

In this section, we discuss our design for NFSv4 over RDMA. We first discuss COMPOUND procedures, followed by Read and Write operations and finally, Readdir and Readlink operations.

4.1.1 Compound Procedures

We first give an example of a COMPOUND procedure, followed by our design.

Example of a COMPOUND procedure: One of the benefits of a COMPOUND operation is that it allows the concatenation of several operations into a single RPC call. On the downside, a COMPOUND operation makes simple operation larger than necessary. Another disadvantage is the error processing involved in a COMPOUND operations. An error in an operation in a COMPOUND will result in the server rejecting and not processing all the remaining operations. As a result, the client may have to send multiple requests to the server to resolve the error. The error processing in a COMPOUND operation is discussed in detail in the IETF RFC 3530 [85]. An example where COMPOUND operation may benefit is the initial opening of a file. The opening of a file requires the procedures LOOKUP (to convert the file name to a file handle), followed by OPEN (to store the file handle at the server) and finally, READ (to read the initial data from the file). Note that OPEN is a NFSv4 specific operation which stores state at the server not present in version 3 and earlier of the protocol. So, in version 3, the sequence we are trying to optimize would be LOOKUP followed by READ. If the COMPOUND is successful, three round-trip operations would have been achieved with a single round-trip to and from the server. However, if the lookup failed, bandwidth would have been wasted for sending the OPEN and READ operations. Whether or not this might have an impact depends on the properties of the underlying interconnect.

Design of a COMPOUND operation with RDMA: As discussed earlier, a COMPOUND procedure consists of a sequence of operations. So, the first issue is to transfer a COMPOUND operation from the client to the server. If the sequence of operations constitutes a small message (less than the inline threshold of 1KB), the request may go inline from the client to the server as a RDMA Send operation. If the sequence of operations constitutes a larger message, an RPC long call needs to be generated from the client to the server. In this case, the client will encode the data from the COMPOUND operation in a single RDMA chunk, that is registered. It will then do an RDMA send from the client to the server with the RDMA chunk encoded and the RDMA_NOMSG flag set. For a given implementation of NFSv4, the client may be aware of the potential length of the reply for all procedures. But, in the general case, the transport may be used for a minor revision to NFSv4 (currently this revision is NFSv4.1). There might be additional procedures introduced in a minor version, that could generate a long reply. To make the design more general, the client always encodes a long reply chunk for every COMPOUND procedure. To reduce the client memory usage, the client should only send a long reply for sequences of procedures that may generate a long reply.

The client should also take into account the preceding READDIR's and READLINK's when computing the length of the long reply. Additional chunks may be encoded to compensate for the preceding READDIR and READLINK procedures. When the server receives the RPC Call, it will decode and store the READ, WRITE and Reply chunks. Finally, control will be passed to the NFS layer for further processing. On return from the server, the RPC/RDMA will need to return the results of the COMPOUND procedure to the client. If the results are small, the results can be returned inline. Otherwise, the last reply chunk sent from the client is used to return the data to the client. So, the reply chunk sent from the server to the client may remain unused. From this, it may seem that the client may need to waste memory for COMPOUND operations. Also, these memory regions would need to be registered. But, with appropriate design strategies in the design, such as a server registration cache, or physical registration, the impact of resource allocation and/or memory registration is likely to be minimal.

4.1.2 Read and Write Operations

Read operations would follow the design of the READ procedure in NFSv3/RDMA, with one caveat. Write chunk lists now must account for the sequential ordering of Read operations in a COMPOUND procedure. The RPC layers at the client and the server need to take this into account, by appropriately marshaling the chunks at the client, storing them at the server and then using them to RDMA Write data back to the client. Similarly, Write Operations follow the design of the WRITE procedure in NFSv3. Sequential ordering of operations is taken into account.

4.1.3 Readdir/Readlink Operations

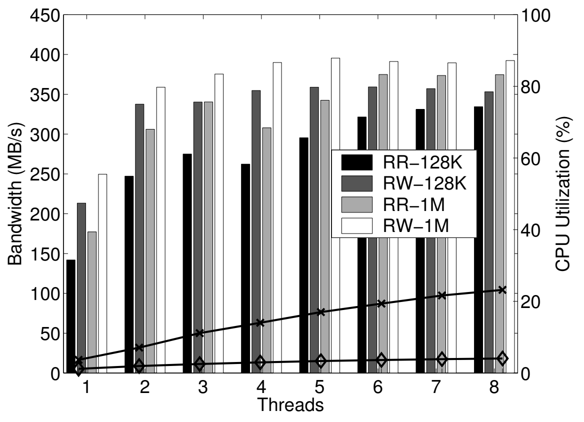
Readdir and Readink operations always encode a long reply chunk. This chunk needs to take into account the sequential ordering of operations in a COMPOUND procedure. The encapsulating COMPOUND procedure itself will include a long reply chunk as the last chunk in the list. The design should account for this interaction.

4.2 Evaluation of NFSv4 over RDMA

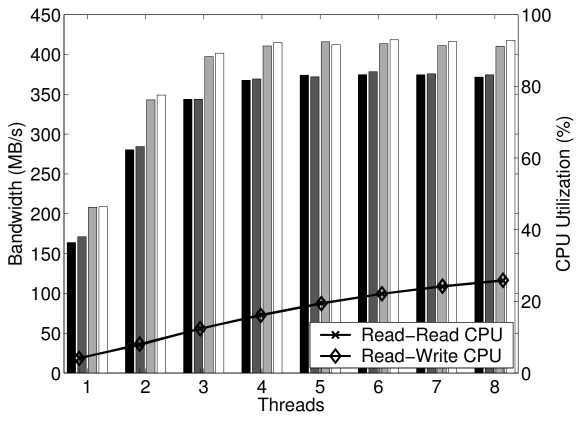
In this section, we discuss the experimental evaluation of NFSv4 with an RDMA transport. First, we describe the setup in Section 4.2.1. Following that we compare the performance of NFSv4/RDMA with NFSv3/RDMA in section 4.2.2. Finally, we compare NFSv4/RDMA with NFSv4/TCP in section 4.2.3.

4.2.1 Experimental Setup

We measure the IOzone [11] Read and Write bandwidth with direct I/O on OpenSolaris. The results were taken on dual Opteron x2100's with 2GB memory and Single Data Rate (SDR) x8 PCI-Express InfiniBand Adapters. These systems were running OpenSolaris build version 33. The back-end file system used was tmpfs which is a memory based file system. The IOzone file size used was 128 MegaBytes to accommodate reasonable multi-threaded workloads (IOzone creates a separate file for each thread). The IOzone record size was set to be 128KB.



(a) Read



(b) Write

Figure 4.1: IOzone Bandwidth Comparison

4.2.2 Impact of RDMA on NFSv4

Figures 4.1(a) and 4.1(b) shows the IOzone Read and Write bandwidth for NFSv4 and NFSv3 over an RDMA transport. From the figures, we can see that v4 performs comparably with v3. The Read bandwidth saturates at 714 MB/s for both NFSv4 and NFSv3. Similarly, Write Bandwidth saturates at 541.71 MB/s. It should be noted that NFSv4 performs slightly worse than NFSv3. This is to be expected because of the additional overhead of COMPOUND operation, as discussed in Section 4.1.1.

4.2.3 Comparison between NFSv4/TCP and NFSv4/RDMA

We also compare the performance of NFSv4 when the underlying transport is TCP. This is shown in Figures 4.1(a) and 4.1(b) (IOzone Read and Write Bandwidth, respectively).

For TCP, we used IPoIB (IP over InfiniBand) as the transport over the same InfiniBand link. Clearly, NFSv4/RDMA outperforms NFSv3/RDMA by an order of magnitude. While, NFSv4/RDMA saturates at over 700 MB/s for IOzone Reads, NFSv4/TCP saturates at just over 200 MB/s. Similarly, for Writes, NFSv4/RDMA saturates at 541.71 MB/s, while NFSv3/TCP saturates at slightly under 100 MB/s.

4.3 Summary

In this Chapter, we have designed and evaluated an RDMA transport for NFSv4. The design challenges included allowing COMPOUND operations which can potentially have an unbounded number of v3 operations to use RDMA operations, which are length limited. Evaluation shows that NFSv4 with RDMA has performance similar to NFSv3 for an RDMA transport. IOzone Read bandwidth saturated at over 700 MB/s, while IOzone Write bandwidth saturates at over 500 MB/s. NFSv4 over RDMA outperforms NFSv4 over TCP (IPoIB) by an order of magnitude.

CHAPTER 5

PERFORMANCE IN A WAN ENVIRONMENT

As cluster-of-clusters connected by high-speed networks become increasingly widespread, it becomes important to evaluate storage protocols in a Wide Area Network environment. We have evaluated NFS over RDMA using Obsidian Long Bow Routers. First, we provide an overview of InfiniBand WAN through router range extension.

5.1 InfiniBand WAN: Range Extension

Obsidian Longbows [21] allow for network transmission range extension. The range extension is primarily for native InfiniBand fabrics over modern 10 Gigabit/s Wide Area Networks (WAN). The Obsidian Longbows routers are usually deployed in pairs. Each pair establishes a point-to-point link between clusters with one Longbow at each end of the link. Figure 5.1 shows a typical installation of Longbow routers in a WAN environment. The Longbows use traditional IP based communication over SONET, ATM, 10 Gigabit Ethernet and dark fiber applications. Existing Longbows routers may run at a maximum speed of 8 Gigabits/s (SDR). To compensate for the remaining 2 Gbps bandwidth, these

Obsidian Longbow routers can also encapsulate a pair of 1 Gigabit/s Ethernet traffic across the WAN link.

In the basic switch mode, the Longbows appear as a pair of two-ported switches to the InfiniBand subnet manager as shown in Figure 5.1. Both the networks are then unified into one InfiniBand subnet. The InfiniBand network stacks are unaware of the unification, though a perceptible lag is added by the wire delays.

The Obsidian Longbow XR routers allow users to simulate WAN environments by adding delays on packet transmissions. The packets are queued, delayed and then transmitted over the WAN link. The delay is constant and may be used as a measure of the distance between the end-nodes in the WAN environment.

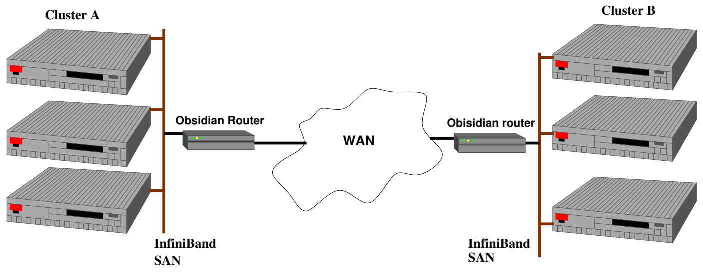


Figure 5.1: Two geographically separated clusters connected by LongBow routers

5.2 WAN Experimental Setup

The Obsidian Long Bow Routers connect two InfiniBand clusters and run at Single Data Rate (SDR) 4X speeds (1 GB/s unidirectional bandwidth). The router allows us to simulate different distances. We simulate with 0 kilometers, 2 kilometers and 200 kilometers, corresponding to a delay of 0μs, 10μs and 1,000μs respectively.

5.3 Performance of NFS/RDMA with increasing delay

The single client, multi-threaded IOzone Read Bandwidth of NFS over RDMA with different delays is shown in Figure 5.2. The InfiniBand HCAs on the local area network run at Double Data Rate (2GB/s unidirectional bandwidth) while the WAN connections run at SDR. As a result, the IOzone Read Bandwidth LAN is approximately 36% higher than the corresponding WAN bandwidth at 0 μs delay. For the WAN scenario, we observe that the peak bandwidth with increasing number of threads is approximately 700 MB/s at 0μs and 10μs. However, beyond that, at 1,000μs delay, the bandwidth drops to around 100 MB/s. The characteristics of the router are such that the bandwidth at 1,000μs for 4 KB messages is close to 100 MB/s. Since the NFS/RDMA design chunks the data into 4KB pieces, the bandwidth is correspondingly lower at a delay of 1,000μs.

5.4 NFS WAN performance characteristics with RDMA and TCP/IP

We also compare the performance of NFS/RDMA and NFS/IPoIB. We compare two different modes of IPoIB, Reliable Connection (RC) and Unreliable Datagram (UD). RC supports reliable in-order data transfers up to 2 GB and RDMA (TCP/IP MTUs are limited to 64KB) . Unreliable Datagram (UD) is restricted to unreliable 4KB MTUs. Figure 5.3 shows the throughput with a WAN delay of 10μs. NFS/RDMA bandwidth is 40% better than IPoIB-RC and 250% better than IPoIB-UD. The data copies are eliminated with NFS/RDMA, and this translates into better performance, since the TCP/IP protocol must deal with fragmentation and reassembly, reliability, congestion management and checksumming overhead. The situation is reversed with a delay of 1,000μs (Figure 5.4). Native InfiniBand protocols are not designed to work in WAN environments. Correspondingly, the NFS/RDMA bandwidth is lowest with a delay of 1,000μs.

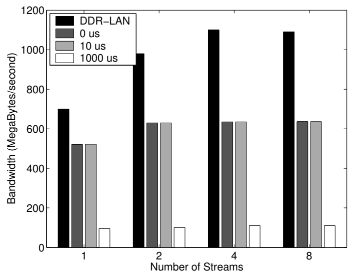


Figure 5.2: NFS/RDMA Performance With Different Delays

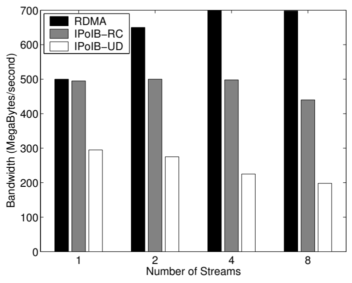


Figure 5.3: NFS at 10 μs delay, comparison between RDMA and IPoIB

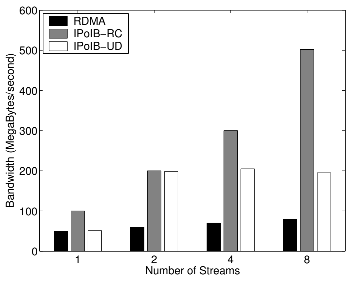


Figure 5.4: NFS at 1,000 μs delay, comparison between RDMA and IPoIB

5.5 Summary

Geographically disparate clusters with high-throughput InfiniBand links are becoming increasingly common. Storage protocols must be able to behave transparently in these scenarios and provide reasonable performance. In this Chapter, we evaluated the performance of the NFS over RDMA protocol in an InfiniBand WAN environment. Evaluations of the NFS/RDMA protocol in a WAN environment show that NFS/RDMA protocols provide better performance than TCP/IP over short distance of up to 10 kilometers, but perform worse than TCP/IP as distances reach into the 1,00-1,000 kilometer range.

CHAPTER 6

CLUSTERED NAS: PNFS ON INFINIBAND

The explosive growth in multimedia, internet and other content have caused a dramatic increase in the volume of media that needs to stored, cataloged and accessed efficiently. In addition, high-performance applications on large supercomputers process and create petabytes of application and checkpoint data. Modern single-headed nodes with a large number of disks (single headed Network Attached Storage (NAS)) may not have the adequate capacity to store this data. Also, the single head or single server may potentially become a bottleneck with accesses from a large number of clients. Also, a failure of the node or the disk may lead to a loss of data.

To deal with several of these problems, clustered NAS solutions have evolved. Clustered NAS solutions attempt to store the data across a number of storage servers. This has a number of benefits. First, we are no longer limited to the capacity of a single node. Second, depending on the way data is striped across the nodes, with accesses from a large number of clients, the load will be more evenly distributed across the servers. Third, for large files, this architecture has the advantages of multiple streams of data from different nodes for better aggregate bandwidth for larger file sizes. Finally, clustered NAS allows data to be stored redundantly across a number of different nodes [12, 102, 46], reducing the likelyhood of data loss.

Even though clustered NAS provide several benefits in term of capacity, enhanced load capacity, better aggregate throughput and better fault-tolerance, they bring with them their own set of unique problems. First, since the data-servers have now been de-coupled, any given stream of data will require multiple network, usually TCP/IP, connections from the clients to the data servers and metadata servers. TCP/IP connections have been shown to have considerable overhead, mainly in terms of copying costs, fragmentation and reassembly, reliability and congestion control. In addition, with multiple streams of incoming data from multiple data-servers, TCP/IP connections have been shown to suffer from the problem of incast [70], which severely reduces the throughput. Second, TCP/IP with multiple copies and considerable overhead is unable to take advantage of the high-performance networks like InfiniBand and 10 Gbe [38, 78]. Third, with a single headed NAS, there is only a single point of failure, making it easier to protect the data on the NAS. However, with a clustered NAS, we have multiple data servers, with multiple failure points.

Modern high-performance networks such as InfiniBand provide low-latency and high-bandwidth communication. For example, the current generation ConnectX NIC from Mellanox has a 4 byte message latency of around 1μs and a bi-directional bandwidth of up to 4 GB/s for large messages. Applications can also deploy mechanisms like Remote Direct Memory Access (RDMA) for zero-copy low-overhead communication. RDMA operations allow two appropriately authorized peers to read and write data directly from each other's address space. RDMA requires minimal CPU involvement on the local end, and no CPU involvement on the remote end. Designing the stack with RDMA may eliminate the copy overhead inherent in the TCP and UDP stacks and reduce CPU utilization. As a result, a high-performance RDMA enabled network like InfiniBand might potentially reduce the overhead of TCP/IP connections in clustered NAS.

In the previous Chapter, we looked at how to design a Network File System (NFS) (which is a single headed NAS) with RDMA operations. for NFSv3 and NFSv4. In this Chapter, we propose, design and evaluate a clustered Network Attached Storage (NAS). This clustered NAS is based on parallel NFS (pNFS) with RDMA operations in InfiniBand. While other parallel and clustered file systems exist such as Lustre [102] exist, we choose pNFS since NFS is widely deployed and used. In this Chapter, we make the following contributions:

- An in-depth discussion of the tradeoffs in designing a high-performance pNFS with an RPC/RDMA transport.
- An understanding of the issues with sessions that provides exactly once semantics in the face of network faults and the trade-offs in designing pNFS with sessions over RDMA.
- A comprehensive performance evaluation with micro-benchmarks and applications of a RDMA enabled pNFS design.

Our evaluations show that by enabling pNFS with an RDMA transport, we can decrease the latency for small operations by up to 65% in some cases. Also, pNFS enabled with RDMA allows us to achieve a peak IOzone Write and Read aggregate throughput of 1,872 MB/s and 5,029 MB/s respectively using a sequential trace with 8 data servers. The RDMA enabled Write and Read aggregate throughput is 150% and 188% better than the corresponding throughput with a TCP/IP transport. Also, evaluation with a Zipf trace distribution allows us to achieve a maximum improvement of up to 27% when switching transports from RDMA to TCP/IP. Finally, application evaluation with BTIO shows that the RDMA enabled transport with pNFS performs better than with a TCP/IP transport by upto 7%.

The rest of the Chapter is presented as follows. Section 6.1 looks at the parallel NFS and sessions extensions to NFSv4.1. Following that, Section 6.2 looks at the design considerations for pNFS over RDMA. After that, Section 6.3 evaluates the performance of the design. Finally, Section 6.4 summarizes the contributions of this Chapter.

6.1 NFSv4.1: Parallel NFS (pNFS) and Sessions

In this section, we discuss pNFS and sessions, which are defined by the NFSv4.1 semantics.

Parallel NFS (pNFS): The NFSv4.1 [79] standard defines two main components; namely parallel NFS (pNFS) and sessions. The focus of pNFS is to make an NFSv4.1 client a front-end for clustered NAS or parallel file-system. The pNFS architecture is shown in Figure 6.1. The NFSv4.1 client can communicate with any parallel file using the *Layout* and *I/O driver* in concert with communications with the NFSv4.1 server. The NFSv4.1 server has multiple roles. It acts as a metadata server (MDS) for the parallel/cluster file system. It sends information to the client on how to access the back-end cluster file system.

This information takes the form of *GETDEVICEINFO*, which returns information about a specific data-server in the cluster file system, usually an IP address and port number that is stored by the client layout driver. The NFSv4.1 server is also responsible for communicating with the data servers for file creation and deletion. The NFSv4.1 server may either directly communicate with the data servers, or it may communicate with a metadata server, that is responsible for talking to and controlling the data servers in the parallel file system. The pNFS client uses the file layout and I/O driver for communicating with the data servers. The layout driver is responsible for translating READ and WRITE requests from the upper layer into the corresponding protocol that the back-end parallel/cluster file system uses; namely object, block and file. This is achieved through the additional NFS procedures *GETFILELAYOUT* (how the file is distributed across the data servers), *RETURNFILELAYOUT* (after a file is closed), *LAYOUTCOMMIT* (commit changes to file layout at the metadata server, after writes have been committed to data servers). Examples of pNFS designs for object based system include the PanFS file system [23]. Examples of file based systems include those from Sun [22] and Network Appliances [65].

NFSv4.1 and sessions: Sessions are aimed at making the NFSv4 non-idempotent requests resilient to network level faults. Traditionally, non-idempotent requests are taken care of through the Duplicate Request Cache (DRC) at the server. The DRC has a limited number of entries, and these entries are shared among all the clients. So, eventually some entries will be evicted from the cache. In the face of network-level partitions, duplicate requests that arrive that have been evicted from the DRC, will be re-executed. Sessions solves this problem by requiring each connection be alloted a fixed number of RPC slots in

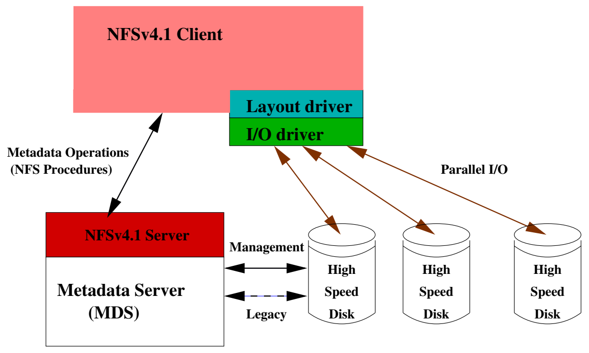


Figure 6.1: pNFS high-level architecture

the DRC. The client is only allowed to issue requests up to the number of slots in the connection. Because of this reservation policy, duplicate requests from the client to the server in the face of network-level partitions will not be re-executed. We will consider design issues with sessions and RPC/RDMA in the following section.

RPC/RDMA for NFS: The existing RPC/RDMA design for Linux and OpenSolaris is based on the Read-Write design discussed in Chapter 3. It consists of two protocols; namely the inline protocol for small requests and the bulk data transfer protocol for large operations. The inline protocol on Linux is enabled through the use of a set of persistent buffers; (32 buffers of 1K each for Send and 32 buffers of 1K each for receives on Linux). RPC Requests are sent using the persistent inline buffers. RPC replies are also received using the persistent inline buffers. The responses for some NFS procedures such as READ and READDIR might be quite large. These responses may be sent to the user via the bulk-data transfer protocol, which uses RDMA Write to send large responses from the server to the clients without a copy and RDMA Reads to pull data in from the client for procedures such as Write. The design trade-offs for RPC/RDMA are discussed further in Chapter 3.

6.2 Design Considerations for pNFS over RDMA

In this section, we examine the considerations for a high-performance design of pNFS over RDMA. First, we look at the detailed architecture of pNFS with a file layout driver.

6.2.1 Design of pNFS using a file layout

As discussed in Section 6.1, pNFS can potentially use an object, block or file based model. In this Chapter, we use the file-based model for designing the pNFS architecture. We now discuss the high-level design of the pNFS architecture.

pNFS Architecture: The detailed architecture is shown in Figure 6.2. The NFSv4.1 clients use a file layout driver which is responsible for communicating with the NFSv4 servers, that act as the data-servers. At the NFSv4.1 server, the sPNFS daemon runs in user-space. The sPNFS daemon communicates with the NFSv4.1 server in the kernel via the RPC PipeFS. The RPC PipeFS is essentially a wait queue in the kernel. The NFSv4.1 server enqueues requests from the clients via the control path, and these requests are then pushed to the sPNFS daemon via an upcall. The sPNFS daemon then processes each of these requests and makes a downcall into the kernel with the appropriate data/response for the requests. The NFSv4.1 requests which are sent up to the sPNFS daemon for processing

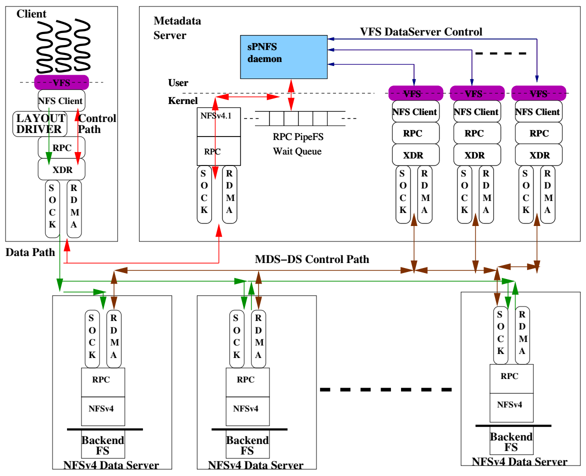


Figure 6.2: pNFS detailed design

include the NFSv4.1 procedures *GETFILELAYOUT*, *RETURNFILELAYOUT*, *LAYOUT-COMMIT* and *GETDEVICEINFO*. These procedures are discussed in Section 6.1. In-order to work on the processing of the requests, the sPNFS daemon mounts an NFSv3 directory from each of the data-servers. For example, when a file layout is requested (*GETFILELAY-OUT*), the sPNFS daemon may need to create the file on each of the data servers or open the file through the *VFS DataServer Control Path*.

sPNFS file creation: To create a file, the sPNFS daemon will open the file on the mount of each of the data servers in create mode. It will then do a *stat* to make sure that the file actually got created or exists. It will then close the file (the file handle is static). This traffic will propagate via RPC calls through the *MDS-DS control path*. Finally, it will return the set of open file descriptor to the NFSv4.1 server as part of the response to the upcall. The NFSv4.1 server will then reply to the NFSv4.1 client with the file layout. The client will then use the layout received (through the file layout driver) to communicate with the NFSv4 data servers using the *Data Paths*.

sPNFS file deletion: File deletes are initiated by the NFS REMOVE procedure. The REMOVE procedure is sent up to the sPNFS daemon through RPC PipeFS. The process of deleting a file is opposite to that of creation. The sPNFS daemon will try to delete each of the file from the mount points. Once this is achieved, sPNS will send a message to the NFS kernel thread about this.

6.2.2 RPC Connections from clients to MDS (*Control Path*)

The RPC connections from the clients to the MDS may be through either RDMA or TCP/IP. A majority of the communication from the clients to the metadata server is expected to be small operations or metadata intensive workloads. As a result, these workloads may potentially benefit from the lower latency of RDMA. However, since NFS and RPC are in the kernel, there is the cost of a context switch from user-space to kernel-space, in addition to the copying costs with the NFS and RPC stacks. Depending on factors such as the CPU speed and memory bus-bandwidth, these costs might dominate. Correspondingly, the lower latency of RDMA might not provide much of a benefit in these cases. Another important factor that needs to be considered is the memory utilization and scalability of the MDS. The MDS is required to maintain RDMA enabled RPC connections with all the clients. Each of these connection holds 32 1K send buffers and 32 2K receive buffers. These buffers are not shared across all the connections. With a very large number of client connections using RPC over RDMA, the MDS server might run out of buffers that might be appropriately utilized. In these cases, using RPC over TCP might be more appropriate for the majority of clients, though the high copying cost associated with TCP/IP connections needs to be considered. If an RDMA enabled RPC transport can provide adequate benefit for small operations, it might be appropriate to use a few connections with RDMA for some clients that communicate frequently with the MDS and a TCP enabled RPC transport for the remaining connections. A final factor that needs to be considered is the disconnect time for a RDMA enabled RPC transport. RDMA enabled RPC connections are disconnected after 2 minutes idle time. Reestablishing a RDMA enabled RPC connection is a very expensive operation because of the high-overhead of registering memory and reestablishing the eager protocol as discussed in Chapter 3. In comparison, RPC over TCP does not have such high-latencies for reestablishing the connections.

6.2.3 RPC Connections from MDS to DS (*MDS-DS control path*)

It might be potentially possible to use RPC over RDMA or RPC over TCP connections between the MDS and DSes. The *MDS-DS control path* allows the MDS to control the NFSv4 data-servers. This control is in the form of file creations and deletions. There are a number of factors that affect the choice of a RPC enabled with RDMA or TCP connection from the metadata server to the data servers. As discussed earlier, the sPNFS daemon is multi-threaded. As a result, there are expected to be a large number of requests in flight, in parallel. So, the lower potential latency of RPC with RDMA is likely to provide a benefit in completion of these requests. Also, the fixed number of buffers per connection is expected to provide a better flow-control mechanism for a large number of outstanding parallel requests. Finally, the number of data servers is relatively small in comparison to the number of clients. As a result, the *MDS-DS control path* is not likely to be severely affected by the buffer scalability issue that may potentially affect the *Control Path*.

6.2.4 RPC Connections from clients to data servers (*Data paths)*

The expected traffic patterns from the client to the data servers is expected to consist of small, large and medium size traffic. Since 32K is the maximum payload for the cached I/O case, this is likely to be the most common transfer over the network, depending on the stripe size of the file at the data servers. We also need to consider the case of buffer scalability. Since data-servers are expected to have connections from a large number of clients, and since each connection will have persistent buffers, this might cause a memory scalability issue. However, clients do not connect to a particular data server unless the data server is in the list of DSes returned in the file layout. As a result, not all clients will be connected to all data servers at any given time. Depending on the load on the back-end file system, using an RPC over RDMA connection from the data-servers to the client might not cause a large amount of overhead at the data-servers. Also, quiescent clients will be disconnected from the data-servers, further reducing the overhead. Since an RPC transport enabled with RDMA has been shown to provide considerable benefits via-viz large transfers, it might be beneficial to use RPC over RDMA between the clients and data-servers.

6.2.5 Sessions Design with RDMA

As discussed earlier, sessions provides exactly once semantics for all NFS procedures in the wake of network-level faults. To do this, sessions provide dedicated slots of buffers to each connection between the client and the servers. The client may only send requests upto a maximum number of slots per session. In order to design sessions with a RDMA enabled RPC transport, we associate the inline buffers in each connection with the minimum number of slots required from the connection. If the number of slots requested is lower than the number available, and the caller cannot accept a lower number, the session create request will fail. The disadvantages of the sessions design with RDMA is that advanced features of the InfiniBand network such as the Shared Receive Queue (SRQ) [80] cannot be used. SRQ enhances the buffer scalability by having the buffers shared across all the InfiniBand connections. When the number of buffers falls below a certain watermark, an interrupt may be generated to post more buffers. Since sessions require that slots be guaranteed per connection, SRQ cannot be used.

6.3 Performance Evaluation

In this section, we evaluate the performance of pNFS designed with an RPC over RDMA transport First, We discuss the experimental setup in Section 6.3.1. Following that, in Sections 6.3.2, 6.3.3 and 6.3.4, we look at the relative performance advantages of using an RDMA enabled RPC transport over a TCP/IP transport in different configurations involving the metadata server (MDS), Data Server (DS) and Client. Since sessions only requires reservation of RDMA inline buffers, we do not evaluate the sessions portion of the design.

6.3.1 Experimental Setup

To evaluate the performance of the RPC over RDMA enabled pNFS design (pNFS/RDMA), we used a 32-node cluster. Each node in the cluster is equipped with a dual Intel Xeon 3.6 GHz CPU and 2GB main memory. For InfiniBand communication, each node uses a Mellanox Double Data Rate (DDR) HCA. The nodes are equipped with SATA drivers, which are used to mount the backend ext3 filesystem. A memory based filesystem ramfs is also used for some experiments. The pNFS with sockets uses IP over InfiniBand (IPoIB) and we refer to this transport as pNFS/IPoIB. We use pNFS/IPoIB and pNFS/TCP interchangeably. All experiments using IPoIB are based on Reliable Connection mode (IPoIB-RC) and an MTU of 64KB, unless otherwise noted. We explicitly use pNFS/IPoIB-UD to explicitly mean an unreliable datagram mode of transport. IPoIB-UD uses a 2K MTU size.

6.3.2 Impact of RPC/RDMA on Performance from the client to the Metadata Server

The clients communicate with the MDS using either NFSv4 or NFSv4.1 procedures. As Section 6.1 mentions, the vast majority of NFSv4.1 requests from the clients to the MDS are expected to be procedures such as *GETDEVICEINFO*, *GETDEVICELIST*, *GETFILELAYOUT* and *RETURNFILELAYOUT*. These small procedures will potentially carry small and medium size payloads. For example, *GETFILELAYOUT* returns a list of file handles, which is only a small amount of payload. A file handle can be encoded with no more than 16 bytes of information (although a native file handle size may vary depending on platforms). One of the largest deployments of a parallel file system Lustre [102] in recent times is the TACC [94] cluster with 8,000 nodes containing 64,000 cores, serviced by a bank of 1,000 data server nodes. With 1,000 data server nodes and the assumption that a file is stripped across all the data server nodes, the payload from *GETFILELAYOUT* will only be 16K. Also, some of these operations such as *GETDEVICEINFO* are only executed at mount time and are not in the critical path. On the other hand, operations such as *CREATED*, *GETFILELAYOUT*, *RETURNFILELAYOUT* are executed every time a file is created, opened and closed. With a workload consisting of a large number of such operations (metadata intensive workloads) RPC/RDMA is likely to provide some benefit. Also, *LAYOUTCOMMIT* is executed once a WRITE operation completes and is likely to be in the critical path for workloads dominated by write operations.

To understand the relative performance of small operations when switching transports from RPC/TCP to RPC/RDMA, we measured the latency of issuing a *GETFILELAYOUT* (at the RPC layer) from the client to the MDS and the time required for it to complete, averaged over 1024 times, while the payload from the MDS to the client was varied from 1 to 32K bytes. A 32K message can contain the information for more than 2,000 file handles and might be considered large for contemporary, high-performance parallel file system deployments. The measured latency is shown in Figure 6.3. As shown in Figure 6.3, the

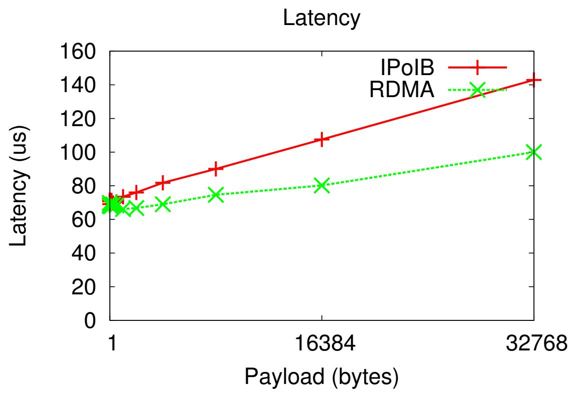


Figure 6.3: Latency for small operations (GETFILELAYOUT)

latency with a 1 byte payload is 68μs with RPC/RDMA and 71μs with RPC/TCP. The relatively low improvement in performance is because the high access latency of the disk which is a dominant portion of the latency. With larger access, the disk blocks are prefetched because of sequential access and the performance improvement from using RPC/RDMA is increased by up to 65%. The performance benefit of the RPC/RDMA connection from the client to the MDS is taken in the context of the inline buffers, which need to be statically allocated per-client at the MDS. With an increasing number of clients, the RPC/RDMA connections may consume considerable memory resources. Since the MDS is likely to be the target of a mainly metadata intensive workload, it becomes imperative to maintain a large number of inline buffers in order to guarantee a high throughput performance.

6.3.3 RPC/RDMA versus RPC/TCP on metadata server to Data Server

The connection from the MDS to the DSes may also consist of RPC/RDMA. The sPNFS daemon controls the DSes by mounting the exported directories from the data servers. The sPNFS daemon creates, open and deletes files in the exported directories. These calls are translated through the VFS layer to RPC/RDMA calls. Thus the scalability of these calls is directly impacted by the time required by the RPC operations to complete. To gain insight into the relative scalability of the RPC/RDMA and RPC/TCP transports, we measured the performance of create portion of the sPNFS daemons operation. In this multi-process benchmark, each process is synchronized in the start phase by a barrier. After being released from the barrier, each process performs a stat operation on the target file to check its state, then opens this file in creation mode. These two operations are followed by a chmod to set the mode of this file, and a close operation to close this file. The close operation is a portion of the process to open a pNFS file, and it is included to avoid running out of open file handles, a limited operating system resource. Each process performs each of these operations on every one of the DS mounts. The time required for 1024 of these operations is measured and averaged out. This test is performed for a RPC/RDMA and RPC/TCP transport from the MDS to the DSes. These numbers are shown in Figures 6.4 and 6.5.

In Figures 6.4 and 6.5, we observe the following trends. RPC/RDMA performs worse than RPC/TCP (indicated as IPoIB) for 1 process. Note that in this case, we are measuring the time at the VFS level, whereas in Section 6.3.2, we are measuring the time at the RPC layer. In the current scenario, the IPoIB-RC driver uses a ring of 128 receive buffer of

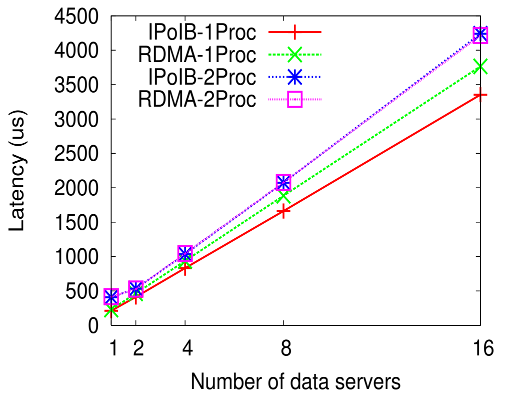


Figure 6.4: Latency for 1 and 2 processes

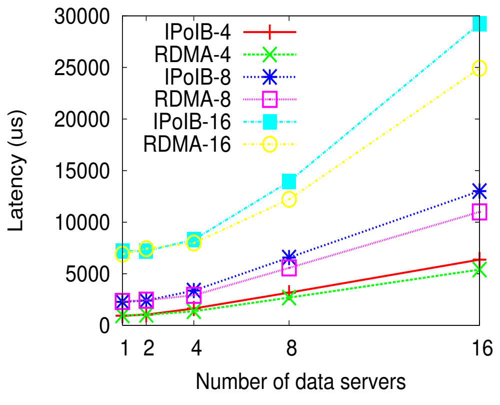


Figure 6.5: Latency for 4,8 and 16 processes

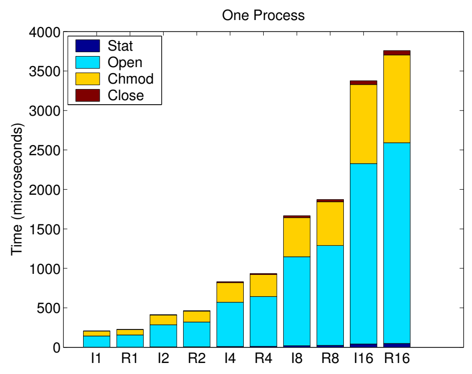


Figure 6.6: Timing breakdown (1 process)

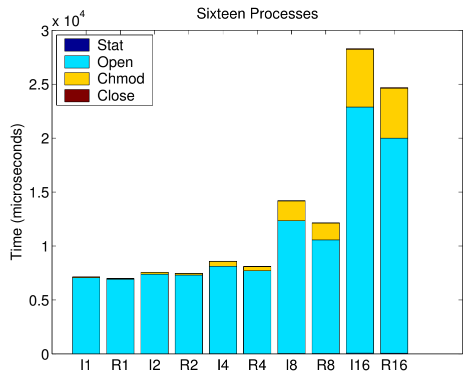


Figure 6.7: Breakdown 16 processes (I-IPoIB, R-RDMA)

size 64K and 64 send buffers. On the other hand, RPC/RDMA uses 32-buffers of 1K. As a result, with an increasing number of data servers, and 1 process, more create and stat operations can be issued in parallel with IPoIB-RC than with RPC/RDMA (we issue 1,024 create operations and 1,024 stat operations for a total of 2,048 operations). However, with

IPoIB-RC all 128 receive buffers are shared across all the connections using SRQ [80]. With RPC/RDMA, each connection from an MDS to a DS is allocated a set of 32-buffers. As a result, when the number of connections increases, RPC/RDMA has a dedicated set of buffers in which to receive messages, while IPoIB has a fixed number of buffers, and this might result in dropped messages with IPoIB. Also in RPC/IPoIB, there are up to 5 copies from the application to the IP-level. With RPC/RDMA, there are up to 3 copies from the application downto the RPC/RDMA layer. With an increasing number of processes, the larger number of copies in the case of IPoIB begins to dominate and IPoIB performs worse than RDMA. The copying cost with IPoIB and 1 client does not totally consume the CPU and so is not the dominant factor. As a result, with 1 process, RPC/TCP is able to perform better than RPC/RDMA. At 2 processes per-node, RPC/RDMA and RPC/TCP perform comparably with an increasing number of data-servers. At 4 processes/node and above with RPC/RDMA, the time required to perform the create operations is lower than RPC/TCP. At 16 processes at the MDS, the improvement with 16 DSes there is a maximum decrease in latency of 15%. The trends we have observed indicate that RPC/RDMA will perform better than RPC/TCP with a larger volume of operations. We have conducted a test with 32 client threads with both RPC/RDMA and RPC/TCP. RPC/RDMA exhibits similar degree of improvement over RPC/TCP.

Also, Figures 6.6 and 6.7 show the timing breakdown for open (with create), stat, chmod and close at one and 16 processes at the MDS with varying number of data servers. In the x-axis, the legend R-n stands for n data servers using RPC/RDMA, and I-n stands for n data servers using RPC/TCP. As expected, the time for open (with create) dominates.

The time for open is higher with RPC/RDMA than with RPC/TCP at 1 process. At 16 processes, RPC/TCP overhead becomes dominant and the time for open (with create) is lower with RPC/RDMA than with RPC/TCP.

6.3.4 RPC/RDMA versus RPC/TCP from clients to DS

We measure the relative performance impact of changing the transport from RPC/RDMA to RPC/TCP from the client to the data-servers. To measure the performance impact, we use three different benchmarks: sequential throughput with IOzone, throughput of a Zipf trace and a parallel application BTIO.

Sequential Throughput

We use IOzone [11] in cluster mode to measure the performance of a sequential workload modeling the throughput from the client to the DSes. 8 nodes act as data servers, 8 nodes act as clients, and 1 node is designated ad the metadata server. Each client node hosts one IOzone process. The benchmark is run on both the IPoIB Reliable Connection mode (IPoIB-RC) and IPoIB Unreliable Datagram mode (IPoIB-UD) to compare against RPC/RDMA. The IOzone record size is kept at 32KB, the default cached I/O maximum size and the total file size per client used is 512MB. The Write and Read throughput while varying the number of data servers and clients (aggregate throughput) is shown in Figures 6.8 and 6.9 respectively.

For Write, RPC/RDMA begins to outperform RPC/TCP as the number of data server is increased beyond two. At 8 data servers and 8 clients, RPC/RDMA reaches its peak write throughput of 1,872 MB/s, which is 22% higher than IPoIB-RC and 150% higher than IPoIB-UD. For Read, there is an improvement in performance for all cases. Using RPC/RDMA achieves a peak read throughput of 5,029 MB/s at 8 clients and 8 data servers, which outperforms IPoIB-RC by 89% and IPoIB-UD by 188%.

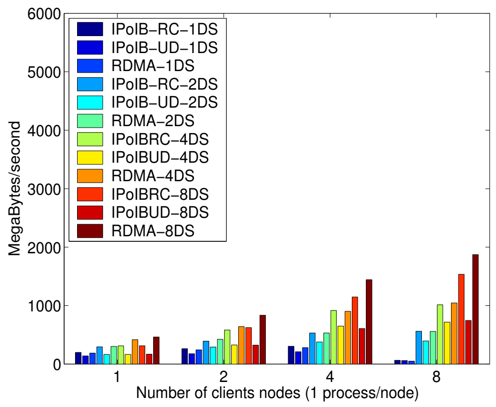


Figure 6.8: IOzone Throughput (Write)

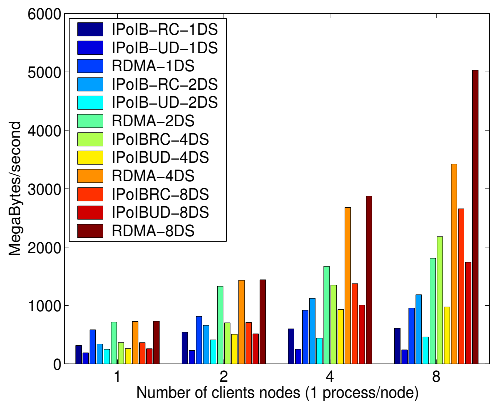
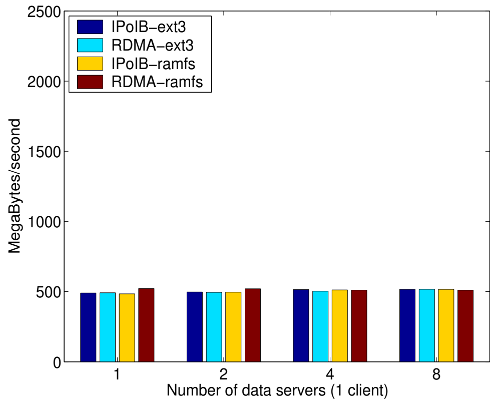


Figure 6.9: IOzone Throughput (Read)

Throughput with a Zip Trace

Zipf's law [103], named after the Harvard linguistic professor George Kingsley Zipf (1902-1950), is the observation that frequency of occurrence of some event (P), as a function of the rank (i) when the rank is determined by the above frequency of occurrence, is a power-law function $P_i \sim 1/i^a$ with the exponent α close to unity. Zipf distributions have been shown to occur in a variety of different environments such as word distributions in documents, web-page access patterns [56] and file and block distributions in storage sub-systems [34].

We modified IOzone to issue write and read requests, where the size and location of the Read or Write request follows a Zipf distribution [103] with an α=0.9. We used IOzone to measure the throughput of the trace on a single node with one thread, issuing requests where the location and I/O size of the issued request follows a Zipf distribution. We used a 512MB file size on both an ext3 as well as a ramfs backend file system. We compare pNFS/RDMA with pNFS/IPoIB-RC while varying the number of data servers. The results for Write are shown in Figure 6.10, while the results for Read are shown in Figure 6.11. We observe that the RPC transport used does not have a large impact on performance for Writes. Disk Filesystem Write performance is generally sensitive to the performance of the backend storage subsystem. The large majority of disks exhibit poor random Write performance [96]. Also, depending on the organization of the in-memory file system, ramfs based systems have also been shown to exhibit poor performance for random Write operations [35]. Correspondingly, for the Zipf based Write distribution, we see a very poor throughput of around 500 MB/s for both pNFS/RDMA and pNFS/IPoIB-RC. On the other hand, the IOzone Read throughput is impacted by the underlying RPC transport. With a *ramfs* based file system, we see an improvement of 22% from pNFS/IPoIB-RC to pNFS/RDMA with 1 data server. The improvement in throughput from pNFS/IPoIB-RC to pNFS/RDMA increases to 27% at 8 data-servers. We are also able to achieve a peak throughput of 2073 MB/s with the Zipf trace at 8 data-servers. Since, the Zipf trace has an element of randomness, a portion of the Read data is cached at the client. As a result we see some amount of cache effect in addition to network-level transfers, which reduces the potential performance improvement with pNFS/RDMA. Kanevsky, et.al. [34] observed a similar effect when using a Zipf trace with NFS/RDMA using a single server. Using the suggested technique of reduced caching at the client for NFS/RDMA, it may be possible to further enhance the performance of pNFS/RDMA when a workload has a Zipf distribution.



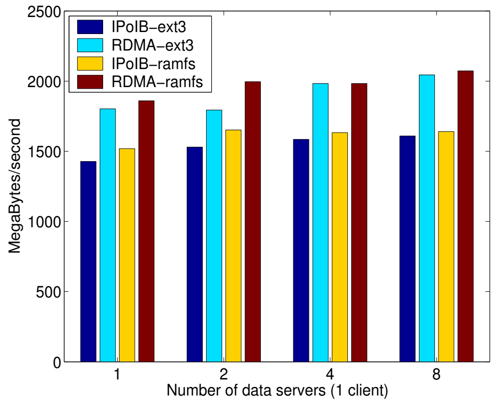


Figure 6.10: Zipf trace throughput (Write)   Figure 6.11: Zipf trace throughput (Read)

Performance with a Parallel Scientific Application NAS BTIO

The NAS Parallel Benchmarks (NPB) [99] suite is used to measure the performance of Computational Fluid Dynamic (CFD) codes on a parallel machine. One of the benchmarks BT measures the performance of block-triangulation equations in parallel. In addition to the computational phase of BT, BTIO adds additional code for check-pointing and verifying the data from the computation. The user may choose from three different modes of performing I/O; namely simple mode, Fortran I/O mode and Full MPI I/O mode [62]. We use the Full MPI I/O mode in which MPI collective calls are used to aggregate Read and Write operations. We run BTIO with a class A size (that uses a 64x64x64 array) over pNFS/RDMA and pNFS/IPoIB. The results with an ext3 back-end file system at the data servers are shown in Figure 6.12.

In these experiments, we measured the performance of BTIO [99] (Million Operations/second) while varying the number of data servers from one to eight and with one, sixteen and sixty-four processes (BTIO requires a square number of processes). For the sixteen process case, we use eight client nodes (2 processes/node). For the 64 process case, we also use eight client nodes (eight processes per node). We observe the following trends at eight data servers. First, pNFS/IPoIB-RC and pNFS/RDMA perform comparably at one

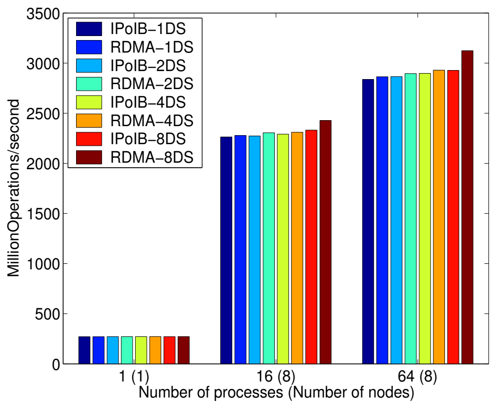


Figure 6.12: Performance of BTIO with ext3

process on one node, irrespective of the number of data severs. As the number of processes increases, pNFS/RDMA begins to perform better than pNFS/IPoIB-RC. This trend is more easily seen with eight data servers and we discuss the trend for eight data servers. At 16 processes (2 processes/node), BTIO over a pNFS/RDMA transport performs upto 4% better than over a pNFS/IPoIB-RC transport. At 64 processes (8 processes/node), this increases to approximately 7%. In full MPI I/O mode, MPI collective calls are used to aggregate smaller reads and writes from different processes into fewer write and read operations with larger sizes. As a result, the bandwidth of the transport impact the performance of BTIO. This becomes apparent at larger number of processes/node and a greater number of data-servers because the copying cost and contention is the dominant factor with the TCP/IP transport.

6.4 Summary

In this Chapter, we propose, design and evaluate a high-performance clustered NAS. The clustered NAS uses parallel NFS (pNFS) with an RDMA enabled transport. We consider a number of design considerations and trade-offs, in particular, buffer management at the client, DS and MDS, scalability of the connections with increasing number of clients and data servers. We also look at how an RDMA transport may be designed with sessions which gives us exactly once semantics. Our evaluations show that enabling pNFS with a RDMA transport, we can decrease the latency for small operations by up to 65% in some cases. Also, pNFS enabled with RDMA allows us to achieve a peak IOzone Write and Read aggregate throughput of 1,800+ MB/s (150% better than TCP/IP) and 5,000+ MB/s

(188% improvement over TCP/IP) respectively, using a sequential trace and 8 data servers. Also, evaluation with a Zipf trace distribution allows us to achieve a maximum improvement of up to 27% when switching transports from RDMA to TCP/IP. Finally, application evaluation with BTIO shows that the RDMA enabled transport with pNFS performs better than a transport with TCP/IP by up to 7%.

CHAPTER 7

CACHING IN A CLUSTERED NAS ENVIRONMENT

With the dawn of the Internet age, the rapid growth of multi-media and other traffic, there has been a dramatic increase in the amount of data that needs to be stored and accessed. In addition, commercial and scientific applications such as data-mining and nuclear simulations generate and parse vast amounts of data during their runs. To meet the demand for access to this data, single server file systems such as NFS [76] and GlusterFS [5] and parallel file systems such as Lustre [102] over high-bandwidth interconnects like InfiniBand with high-performance storage disks at the storage servers have become common-place. However, even with these configurations, the performance of the file system under a variety of different workloads is limited by the access latency to the disk. With a large number of requests to non-contiguous locations of the disk, the ability of the file system to cope with these types of requests is severely limited. In addition, parallel striping of parallel data provides limited benefit in environments with a lot of small files.

To reduce the load on the disk and enhance the performance of the file system, several different types of caching strategies and alternatives have been proposed [57, 52]. Generally, in most file systems, a cache exists at the server side. It might be part of the distributed file system, such as with Lustre [102], or it might reside in the underlying file system such as with NFS. The server side cache will generally contain the latest data. The server side cache may be used to reduce the number of requests hitting the disk, and also provide enhancements when there is a fair amount of read/write data sharing. The server side cache is generally limited in size and shared by a large number of I/O threads. In addition, the limited size of the cache in-concert with policies like LRU can reduce the performance of the server side cache.

In addition to a server side cache, file system protocols like NFS [76] and Lustre [102] also provide a client side cache. A client side cache may provide a large benefit in terms of performance when most of the data is accessed locally, such as in the case of a user home directory. However, client side caches introduce cache coherency issues when there is sharing of data between multiple clients. NFS does not offer strict cache coherency and uses coarse timeouts to deal with the issue. Lustre [102] on the other hand uses locking with the metadata server acting as a lock manger to implement client cache coherency. Writes are flushed before locks are released. With a large number of clients, the overhead of maintaining locks and keeping the client caches coherent increases. GlusterFS [5] does not provide a client side cache in the default configuration.

In this Chapter, we propose, design and evaluate an InterMediate Caching architecture (IMCa) between the client and the server for the GlusterFS [5] file system. We maintain a bank of independent cache nodes with a large capacity. The file system is responsible for storing information from a variety of different operations in the cache. Keeping the information in the cache bank up-to-date is achieved through a number of different hooks at the client and the server. Through these hooks, the client attempts to fetch the information for different operations from the cache, before trying to get it from the back-end file server.

We expect multiple benefits from using this architecture. First, the file system clients can expect to retain the benefits of a client side cache with a small penalty bounded by network round-trip latency. With the advent of low latency, high-performance networks like InfiniBand which offer low latency messaging, the penalty associated with this is likely to be low. Second, since the number of caches is small in comparison to the number of clients, and these caches are lockless, keeping the caches coherent is considerably cheaper. Finally, we expect to reap the benefits of a client cache without the associated scalability and coherency issues.

Our preliminary evaluations shows that we can improve the performance of file system operations such as stat by up to 82% over the native design and upto 86% over a filesystem like Lustre. In addition, we also show that the intermediate cache can improve the performance of data transfer operations with both single and multiple clients. Finally, in environments with read/write sharing of data, we can see an overall improvement in file system performance. Finally, IMCa helps us to achieve better scalability of file system operations.

The rest of this Chapter is organized as follows. Section 7.1 describes the background work. After that, Section 7.2 tries to motivate the need for a bank of caches. In Section 7.3 we discuss the design issues. Following that, Section 7.4 presents the evaluation. Finally, summary is presented in Section 7.5.

7.1 Background

In this section, we discuss the file system GlusterFS and the dynamic web-content caching daemon MemCached.

7.1.1 Introduction to GlusterFS

GlusterFS [5] is a clustered file-system for scaling the storage capacity of many servers to several peta-bytes. It aggregates various storage servers or bricks over an interconnect such as InfiniBand or TCP/IP into one large parallel network file system. GlusterFS in its default configuration does not stripe the data, but instead distributes the namespace across all the servers. Internally, GlusterFS is based on the concept of translators. Translators may be applied at either the client or the server. Translators exist for Read Ahead and Write Behind. In terms of design, a small portion of GlusterFS is in the kernel and the remaining portion is in userspace. The calls are translated from the kernel VFS to the userspace daemon through the Filesystem in UserSpace (FUSE).

7.1.2 Introduction to MemCached

Memcached is an objects based caching system [45] developed by Danga Interactive for LiveJournal.com. It is traditionally used to enhance the performance of database application or websites with dynamic content that are heavily loaded. Memcached is usually run as a daemon on spare nodes. Memcached listens for requests on a user specified port. The amount of memory used for caching is specified at startup. Internally, memcached implements Least Recently Used (LRU) as the cache replacement algorithm. Memcached uses a lazy expiration algorithm; i.e. objects are evicted when the cache is full and a request is made to add an object to the cache, or a request to fetch a data element from the cache is made and the time for the object in the cache has expired. Memory management is based on slab cache allocation to reduce excessive fragmentation. Memcached currently limits the maximum size of the object to be stored to 1MB and the maximum length of the key to 256 bytes. The Memcache daemon may be accessed through TCP/IP connections. Clients usually change data elements in memcached through a *T(key, data)* tuple. The API consists of the functions set, replace, delete, prepend and append. A number of libraries are available for accessing memcached daemons; one of them libmemcache is a C based library [13].

7.2 Motivation

We now consider the motivation for using intermediate caching architecture in a file system. We look at some common problems in file system design that could potentially be solved through the use of a caching layer.

Single Server Bandwidth Drop With Multiple Clients. Protocols like NFS/RDMA attempts to offer the improved bandwidth of networks like InfiniBand to NFS. However, NFS servers usually store most of the data on the disk. The server is constrained by the ability of the disk to match the bandwidth of the network. Since the disk is usually much slower than the network, the benefit from using NFS/RDMA is reduced. The effect of this is shown in Figure 7.1(a) and Figure 7.1(b), which show the multi-client IOzone Read throughput with different transports, namely NFS/RDMA (RDMA), NFS/TCP on InfiniBand (IPoIB) and finally, NFS/TCP on Gigabit ethernet (GigE). In Figure 7.1(a), 4GB server memory is used; in Figure 7.1(b), 8GB server memory is used. The bandwidth available to the clients seems to be related to the amount of memory on the server and falls off as the server runs out of memory and is forced to fetch data from the disk.

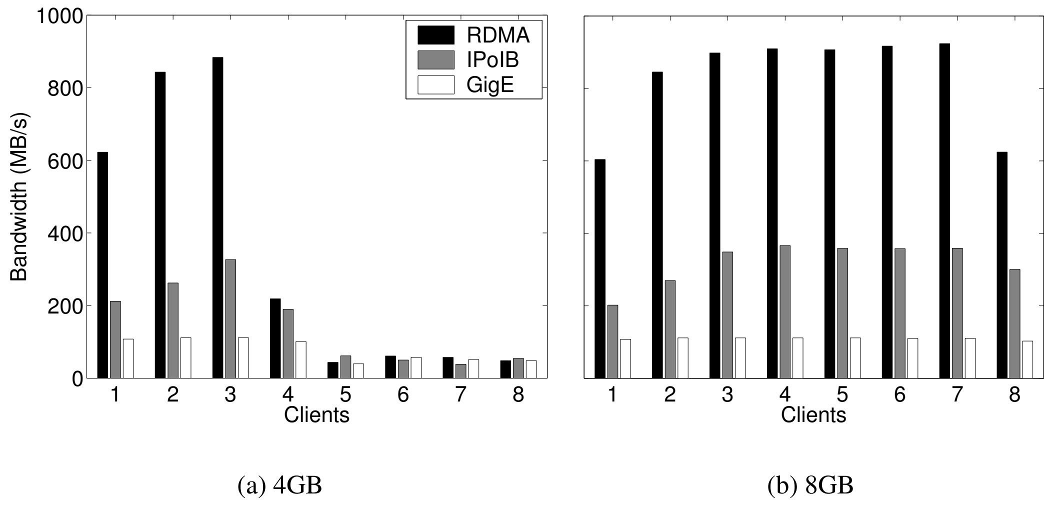


Figure 7.1: Multiple clients IOzone Read Bandwidth with NFS/RDMA [76]

Parallel I/O Bandwidth From Multiple Servers. Parallel I/O attempts to use the aggregate bandwidth of multiple servers. Since the back-end server ultimately uses real disks, the benefits of parallel I/O bandwidth are ultimately mitigated especially for multiple streams that access data spread on different portions of the disk causing increased disk seeking, reducing performance.

Performance For Small Files. Delivering good performance for small files is generally difficult. In data-center environments a large number of small files are used [64]. Data striping techniques generally used in parallel file system are of limited use for small files. Storing files on multiple independent servers can help reduce contention for small files, but still exposes these files to the limits of the disk on these servers.

Cache Coherency Problems. In file system environments, a client side cache usually provide best performance. Client caches may be coherent such as with Lustre [102] or non-coherent, such as with NFS [76]. Non-coherent client side caches are more scalable but have limited use in environments with read/write sharing. Coherent client side cache may be used in environments with sufficient read/write sharing. However, they have limited scalability.

Server load problems. Reducing the load on the server is generally crucial to improving the scalability of file system protocols. RDMA is generally proposed as a communication offload technique to reduce the impact of copying in protocols like TCP/IP. However, RDMA cannot eliminate other copying overheads such as those across the VFS layer and other file system related overheads. Using an intermediate cache layer may help mitigate the effect of some of these problems. We will now look at the design and implementation of a layer of caching nodes.

7.3 Design of a Cache for File Systems

In this section, we consider the design of the Intermediate memory caching (IMCa) architecture for the GlusterFS [5] file system. First, we look at the overall block level architecture of IMCa in Section 7.3.1. Following that, we look at the potential non-data file system operations that could be optimized in Section 7.3.2. In Section 7.3.3, we look at the potential optimizations for data operations. Finally, we discuss some of the potential advantages and disadvantages of IMCa in Section 7.3.4.

7.3.1 Overall Architecture of Intermediate Memory Caching (IMCa) Layer

The architecture of IMCa is shown in Figure 7.2. The architecture consists of three components: CMCache (Client Memory Cache), MemCached (MCD) array and SMCache (Server Memory Cache). The first component CMCache (Client Memory Cache) is located at the GlusterFS client.

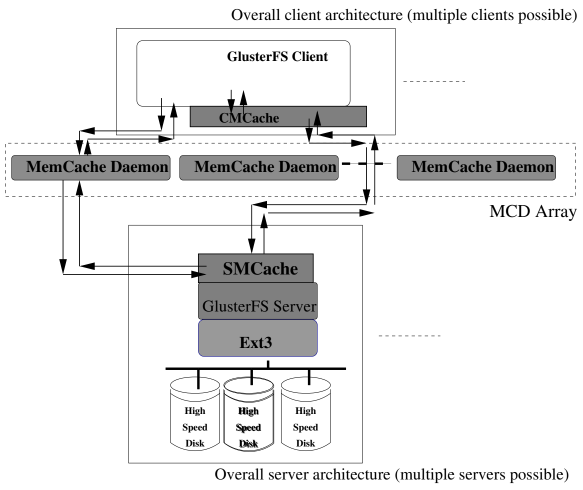


Figure 7.2: Overall Architecture of the Intermediate Memory Caching (IMCa) Layer

Client Memory Cache (CMCache): This is responsible for intercepting file system operations at the client. It is implemented as a translator on the GlusterFS client as discussed in Section 7.1. Once these operations are intercepted CMCache determines whether these requests have any interaction with the caching layer or not. If there is no interaction, CMCache will propagate the request to the server. Interactions are generally in two forms. In the first form, it may be possible to process the request from the client directly by contacting the MCDs. In this case, CMCache will contact the MCDs and attempt to directly return the results for the requests. CMCache communicates with the MCDs through TCP/IP.

MemCached MCD Array (MCD): This consists of an array of MemCached daemons running on nodes usually set aside primarily for IMCa. The daemons may reside on nodes that have other functions, since MCDs tends to use limited CPU cycles. To obtain maximum benefit from using IMCa, the nodes should be able to provide a sufficient amount of memory to the daemons while they are running.

Server Memory Cache (SMCache): This is the final component of IMCa. It is located on the GlusterFS server. SMCache is implemented as a translator at the GlusterFS server. SMCache is divided into two parts. The first part of SMCache intercepts the calls coming from GlusterFS clients. Depending on the type of operation from the GlusterFS client, it may either pass the operation directly to the underlying file system, or perform certain transformations on it before passing it to the underlying file system. The GlusterFS file system uses the asynchronous model of processing requests as discussed in Section 7.1. Initially, requests are issued to the file system and later when they complete, a callback handler is called that processes these responses and returns the results back to the client. The second part of SMCache maintains hooks in the callback handler. These hooks allow SMCache to intercept the results of different operations and send them to MCDs if needed. SMCache communicates with the MCDs using TCP/IP.

7.3.2 Design for Management File System Operations in IMCa

We now consider some of the design trade-offs for different management file system operations.

Stat operations: These are included in POSIX semantics. Stat applies to both files and directories. Stat generally contains information about the file size, create and modify times, in addition to other information and statistics about the file. Stat operations are a popular way of determining updates to a particular file. For example, in a producer-consumer type of application, a producer will write or append to a file. A consumer may look at the modification time on the file to determine if an update has become available. This avoids the need and cost for explicit synchronization primitives such as locks. This approach is used in a number of web and database applications [64]. Since the data structures for the stat operations are generally stored on the disk, stat operations usually have considerable latency. It is natural to consider stat functions for cache based functionality. We have designed a cache based functionality for stat. At open, MCD is updated with the contents of the stat structure from the file by SMCache. The key used to locate a MCD consists of the absolute pathname of the file, with the string *:stat* appended to it. SMCache uses the default CRC32 hashing function in libmemcache [13] to locate the appropriate MCD. For every read and write operation, the stat structure in the MCD is replaced with the most recent value of stat by SMCache. CMCache then intercepts stat operations, attempts to fetch the stat information from the MCD if available, and return it to the client. If there is a miss, which might happen if the stat entry was evicted from the MCD for example, the stat request propagates to the server.

Create operations: These usually require allocation of resources on the disk. There is not much potential for cache based optimizations. *Create* operations are directly forwarded from the client to the server without any processing.

Delete operations: These operations usually require removal of items from the disk. The potential for optimizations with *delete* operations is limited. Delete operations are forwarded by the client to the server without any interception. When *delete* operations are encountered, we remove the data elements from the cache to avoid false positives for requests from clients.

7.3.3 Data Transfer Operations

There are two types of file system operations that generally transfer data; i.e. *Read* and *Write*. To implement *Read* and *Write* with IMCa, CMCache intercepts the *Read* and *Write* operations at the client. Before we discuss the protocols for these operations, we look at the issue of cache blocking for file system operations.

Need for Blocks in IMCa

Most modern disk based file systems store data as blocks [60]. Parallel file systems also tend to stripe large files across a number of data servers using a particular stripe width. Generally, the larger the block size, the better bandwidth utilization from the disk and network subsystems. Smaller block sizes on the other hand tend to favor lower latency, but also tend to introduce more fragmentation. IMCa uses a fixed block size to store file system data in the cache. Since IMCa is designed as a generic caching layer and should provide good performance for a variety of different file sizes and workloads; the block size should be set appropriately keeping these limits in mind. It should be kept small enough so that small files may be stored more efficiently. It should also be kept large enough to avoid excessive fragmentation and reasonable network bandwidth utilization. MemCached [45] has a maximum upper limit of 1MB for stored data elements as discussed in Section 7.1. This places a natural upper bound on the size of data that may be stored in the cache. Depending on the blocksize, IMCa may need to fetch or write additional blocks from/to the MCDs above and beyond what is requested. This happens if the beginning or end of the requested data element is not aligned with the boundary defined by the blocksize. This is shown in Figure 7.3. As a result, data access/update from/to the MCDs become more expensive. This is discussed further in Section 7.3.3.

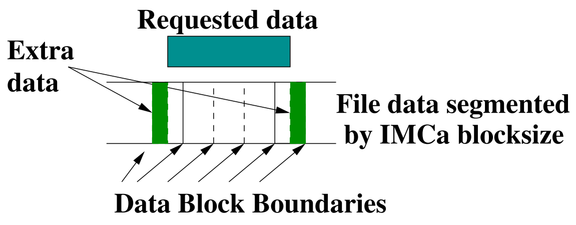
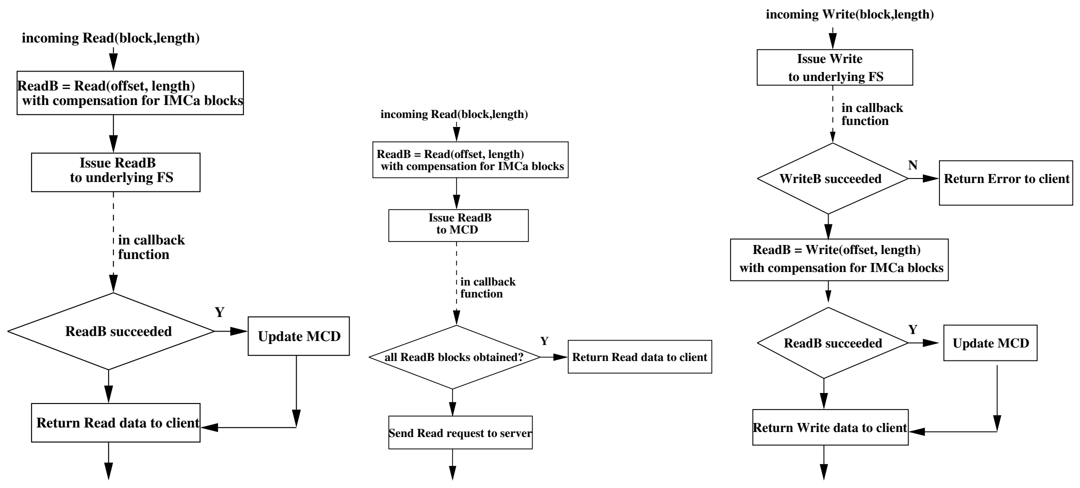


Figure 7.3: Example of blocks requiring additional data transfers in IMCa


incoming Read(block,length)
ReadB = Read(offset, length) with compensation for IMCa blocks
Issue ReadB to underlying FS
in callback function
ReadB succeeded
Y
Update MCD
Return Read data to client
incoming Read(block,length)
ReadB = Read(offset, length) with compensation for IMCa blocks
Issue ReadB to MCD
in callback function
all ReadB blocks obtained?
Y
Return Read data to client
Send Read request to server
incoming Write(block,length)
Issue Write to underlying FS
in callback function
WriteB succeeded
N
Return Error to client
ReadB = Write(offset, length) with compensation for IMCa blocks
ReadB succeeded
Y
Update MCD
Return Write data to client

(a) Read (SMCache) (b) Read (CMCache) (c) Write (SMCache)

Figure 7.4: Logical Flow for Read and Write operations in IMCa

Design for Data Transfer Operations:

We now look at the protocols for Read and Write data transfer operations in IMCa. We also consider the supporting functionality for data transfer operations such as Open and Close.

Open: On the open, in CMCache, the absolute path of the file and the file descriptor is stored in a database, so that this information may be accessed at a later point. At the server, the MCDs are purged of any data relating to the file when the *Open* operation is received.

Read: The algorithm for Read requests in CMCache is shown in Figure 7.4(b). On a *Read* operation, CMCache appends the absolute path of the file (which was stored during the Open) with the offset in the file to generate a key. Since IMCa is based on a static block size; the size of the Read data requested from the MCD may be equal to or greater than the current Read request size. CMCache will generate keys that consist of the absolute pathname for the file, that was stored during the open and the offsets from the Read request, taking into account the IMCa blocksize. CMCache uses the keys to access the MCDs and fetch the blocks. If there is a miss for any one of the keys, CMCache will forward the Read request to the GlusterFS server. The cost of a miss is more expensive in the case of IMCa, since it includes one or more round-trips to the MCD, before determining that there might be a miss. The SMCache Read algorithm is shown in Figure 7.4(a). Because of the IMCa block size, the *Read* operation may potentially require the server to read additional data from the underlying file system. Once the Read operation returns from the filesystem, the server will append the full file path name with the block offset and update the MCDs with the data. The server may need to send several blocks to the MCDs servers. Using an additional thread to update the MCDs at the server may potentially reduce the cost of Reads at the server.

Write: Write operations are persistent. This means that the Write operations must propagate to the server where they need to be written to the filesystem. CMCache does not intercept *Write* operation. At the server, the Write operation is issued to the file system as shown in Figure 7.4(c). When the write operation completes, Read(s) are issued to the underlying file system by SMCache that cover the Write area, accounting for the IMCa blocksize. When the data is available, the Read(s) are sent to the MCDs. Since there may be multiple overlapping *Writes* to a particular record and because of the IMCa requirement of a fixed block-size, neither CMCache nor SMCache can directly send the *Write* data to the

MCDs. Write latency may be potentially increased by the additional update of the MCDs at the server. Using an additional thread as with Reads can reduce the cost of this update.

Close: Closes propagate from the client directly to the server without any interception. When the close operation is intercepted by SMCache, it will attempt to discard the data for the file from the MCDs.

7.3.4 Potential Advantages/Disadvantages of IMCa

In this section, we discuss the potential advantages and disadvantages of IMCa.

Fewer Requests Hit the Server: The data server is generally a point of contention for different requests. In addition to communication contention, there may be considerable contention for the disk. IMCa may help reduce both these contentions at the server.

Latency for Requests Read From the Cache is Lower: With considerable percentage of Read sharing as well as Read/Write sharing patterns, a large number of requests could potentially be fielded directly from the MCDs. This might help reduce the latency for these patterns, in addition to reducing the load on the server.

MCDs are self-managing: Each cache in the MCD implements LRU. As the caches fill up, unused data will automatically be purged from the MCDs. There is no need to manage the cache by the client or the server. This reduces the overhead of IMCa. Additional caching nodes can be easily added. IMCa can transparently account for failures in MCDs.

Failures in MCDs do not impact correctness: Writes are always persistent in IMCa and are written successfully to the server filesystem before updating the MCDs. Irrespective of node failures in the MCDs, correctness is not impacted.

Additional Nodes Elements Needed Especially For Caching: MCDs needs an array of nodes on which to run the daemons. These nodes might be used for other purposes such as storing file system data or running web services.

Cold Misses Are Expensive: Reads on the client require one or more accesses to the MCDs depending on the blocksize and the requested Read size. If any of these accesses results in a miss, the Read needs to be propagated to the server. As a result, misses are more expensive than in a regular file system.

Additional Blocks/Data Transfer Needed: In IMCa data is stored in blocksizes to act as a tradeoff between bandwidth, latency, utilization and fragmentation. If the block size is set too large, small Read requests will be penalized, requiring additional data to be transferred from the MCDs. If the block size is set too small, large requests might require multiple trips to the MCDs to fetch the data.

Overhead and Delayed Updates: IMCa hooks into both Read/Write functions at the server through SMCache. Read/Write data from the server needs to be fed to the MCDs before it is returned to the client in non-threaded mode. This may result in additional overhead at the server and updates from the MDCs being delayed.

7.4 Performance Evaluation

In this section, we attempt to characterize the performance of IMCa in terms of latency and throughput of different operations. First, we look at the experimental setup.

7.4.1 Experimental Setup

We use a 64 node cluster connected with InfiniBand DDR HCAs. Each node is an 8-core Intel Clover based system with 8GB of memory. The GlusterFS server runs on a node with a configuration identical to that specified above; it is also equipped with a RAID array of 8-HighPoint Disks on which all files used in the experiment reside. IP over InfiniBand (IPoIB) with Reliable Connection (RC) is used as the communication transport between the GlusterFS server and client; as well as between the components of IMCa namely SMCache, CMCache and the MCD array. The MCDs run on independent nodes and are allowed to use upto 6GB of main memory. Unless explicitly mentioned, SMCache and CMCache use a CRC32 hashing function for storing and locating data blocks on the MCDs. For comparison, we also use the default configuration of Lustre 1.6.4.3 with a TCP transport over IPoIB. The Lustre metadata server runs on a node separate from the data servers (DS).

7.4.2 Performance of Stat With the Cache

We look at the performance of the *stat* operation with IMCa as discussed in Section 7.3.2.

Stat Benchmark: The benchmark used to measure the performance of stat consists of two stages. In the first stage (untimed), a set of 262,144 files is created. In the second stage (timed) of the benchmark, each of the nodes tries to perform a stat operation on each of the 262,144 files. The total time required to complete all 262,144 stats is collected from each of the nodes and the maximum time among all of them is reported.

Performance With One MCD: The results from running this benchmark is shown in Figure 7.5. Along the x-axis the number of nodes is varied. The y-axis shows the time in seconds. Legend *NoCache* corresponds to GlusterFS in the default configuration (no client side cache). Legend *MCD (x)* corresponds to GlusterFS with x MemCached daemons running. From Figure 7.5, we can see that without the cache, the time required to complete the stat operations increases at a much faster rate than with the cache nodes. With a single MCD, the time required to complete the stat operations increases at a much slower rate. At 64 clients, with 1 MCD, there is an 82% reduction in the time required to complete the *stat* operations as compared to without the cache. GlusterFS with a single MCD outperforms Lustre with 4 DSs by 56% at 64 clients.

Performance With Multiple MCDs: With an increasing number of MCDs, there is a reduction in the time needed to complete the *stat* operations. However, with an increasing number of MCDs, there is a diminishing improvement in performance. For example, at 64 nodes, there is only a 23% reduction in time to complete the stat operation from 4 to 6 MCDs. The statistics from the MCDs show that the miss rate with increasing MCDs beyond 2 is zero. This seems to suggest that 2 MCDs provide adequate amount of cache memory to completely contain the stat data of all the files from the workload. There is little stress on the MCDs memory sub-system beyond two MCDs. The overhead of the communication protocol TCP/IP is alleviated to some extent by going beyond two MCDs. Using four and six MCDs provide some benefit as may be seen from Figure 7.5. At 64 nodes, using GlusterFS with 6 MCDs, the time required to complete the *stat* operation is 86% lower than Lustre with 4 DSs.

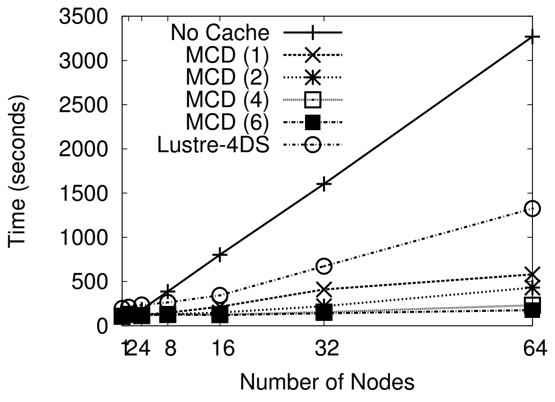


Figure 7.5: Stat latency with multiple clients and 1 MCD

7.4.3 Latency: Single Client

In this experiment, we measure the latency of performing read and write operations.

Latency Benchmark: In the first part of the experiment, data is written to the file in a sequential manner. For a given record size *r*, 1024 records of record size *r* are written sequentially to the file. The *Write* time for that record size is measured as the average time of the 1024 operations. We measure the Write time of record sizes from 1 byte to a maximum record size in multiples of 2. In the second stage of the benchmark, we go back to the beginning of the file and perform the same operations for *Read* operations, varying the record size from 1 byte to the maximum record size, with the time for the Read being averaged over 1024 records for each given record size.

Read Latency with different IMCa block sizes: The results from the latency benchmark for *Read* is shown in Figure 7.6(a) and Figure 7.6(b). For IMCa, we used block sizes of 256 bytes, 2K and 8K bytes. For the Read latency shown in Figure 7.6(a), for a record size of 1 byte, there is a reduction of upto 45% in latency using one MCD over using No-Cache, with a block size of 2K, and a 31% reduction in latency with an 8K IMCa block size. With an IMCa block size of 256, the reduction in Read latency increases to 59%. As discussed in Section 7.3, even for a Read operation of 1 byte, the client needs to fetch a complete block of data from the MCDs. So, we must fetch data in multiples of the minimum record size of IMCa. Smaller block sizes help reduce the latency of smaller Reads, but degrade the performance of larger Reads, since CMCache must make multiple trips to the MCDs. This may be seen in Figure 7.6(a), where beyond a record size of 8K, NoCache has lower latency than IMCa with a block size of 256 and has the lowest latency overall as the record size is further increased (Figure 7.6(b)). Since no Read at the client results in a miss from the MCDs, no read requests propagate to the server. We use a block size of 2K for the remaining experiments.

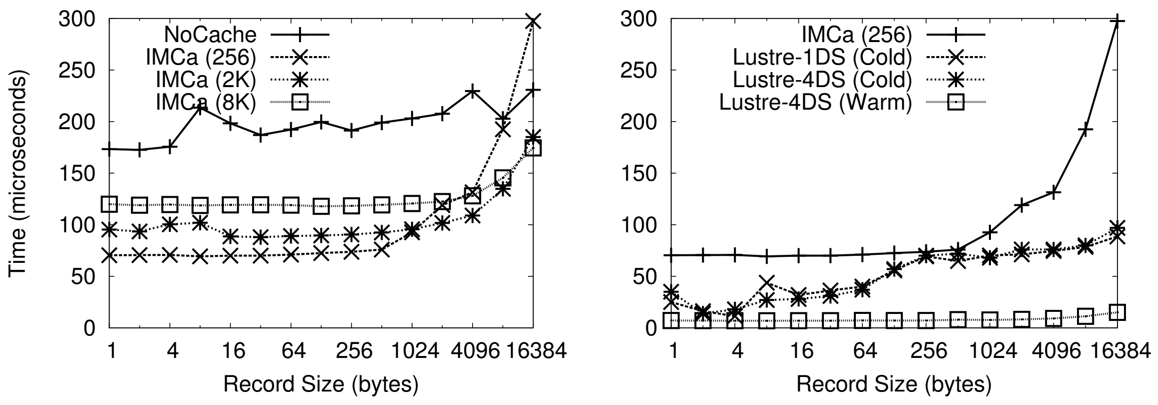
Comparison with Lustre: We use one and four data servers with Lustre, denoted by 1DS and 4DS respectively. Also, we use two different configurations for Lustre, warm cache (Warm) and cold cache (Cold). For the warm cache case, the Write phase of the benchmark is followed by the Read phase of the benchmark without any intermediate step. For the cold cache case, after the Write phase of the benchmark, the Lustre client file system is unmounted and then remounted. This evicts any data from the client cache. Clearly, the warm cache case denoted by *Lustre-4DS (Warm)* provides the lowest Read latency in all cases (Figure 7.6(a)), since Reads are primarily satisfied from the local client cache (results for larger record sizes with a warm cache are not shown). The cold cache forces the client to fetch the file from the data servers. So, *Lustre-1DS (Cold)* and *Lustre-4DS (Cold)* are closer to IMCa in terms of performance. We discuss these results further in [75].

Write Latency: The Write latency is shown in Figure 7.6(c) with an IMCa block size of 2K. *Write* introduces an additional Read operation in the critical path at the server (Section 7.3). Correspondingly, Write latency with IMCa is worse than the NoCache case. By offloading the additional Read to a separate thread, the additional latency of the Read may be removed from the critical path and the Write latency can be reduced to the same value as without the cache. IMCa provides little benefit for Write operations because of the need for Writes to be persistent (Section 7.3.3). Correspondingly, we do not present the results for Write for the remaining experiments.

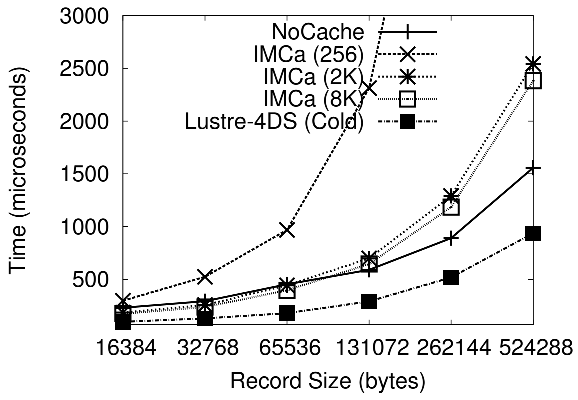
7.4.4 Latency: Multiple Clients

The multi-client latency tests starts with a barrier among all the processes. Once the processes are released from this barrier, each process performs the latency test (with separate files), described in Section 7.4.3. The Write and Read latency components as well as each record size for Read and Write is separated by a barrier. The latency for a particular record size is the average of the times reported by each process for the given record size.

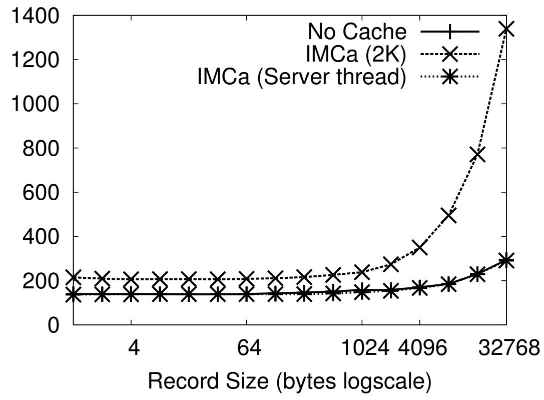
We present the numbers for the Read latency with 32 clients each running the latency benchmark, while the MCDs are being varied. These latency numbers are shown in Figure 7.7(a) (Small Record sizes) and Figure 7.7(b) (Medium Record Sizes). From the figure, we can see that there is reduction of 82% in the latency when four MCDs are introduced over the NoCache case for a 1 byte Read. Clearly, IMCa provides additional benefit in the case of multiple clients as compared to the single client case. In addition, with 32 clients,



(a) Read (Small)



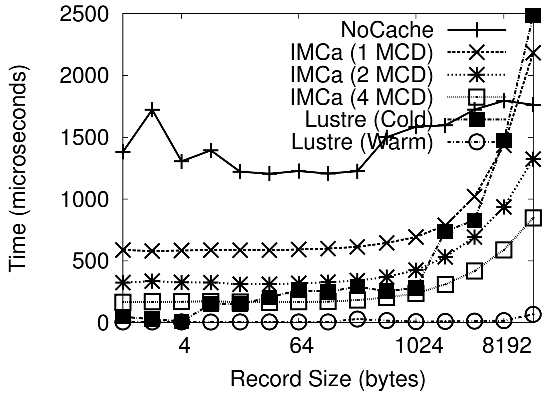
(b) Read (Medium)



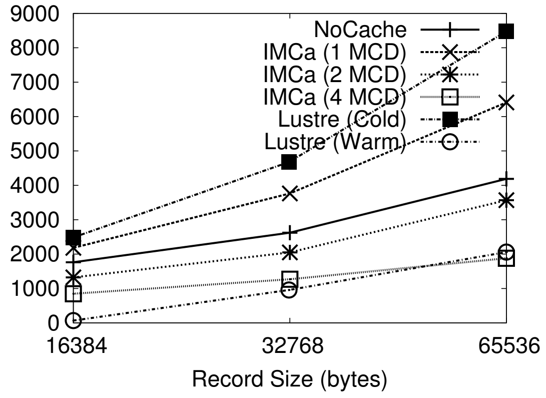
(c) Write

Figure 7.6: Read and Write Latency Numbers With One Client and 1 MCD.

and a single MCD, statistics taken from the MCDs show that there are an increasing number of MCD capacity misses. These capacity misses are reduced by increasing the number of MCDs. The trend of increasing capacity misses may be seen more clearly while varying the clients and using a single MCD. These Read latency number are shown in Figure 7.8(a) and



(a) Read (Small)

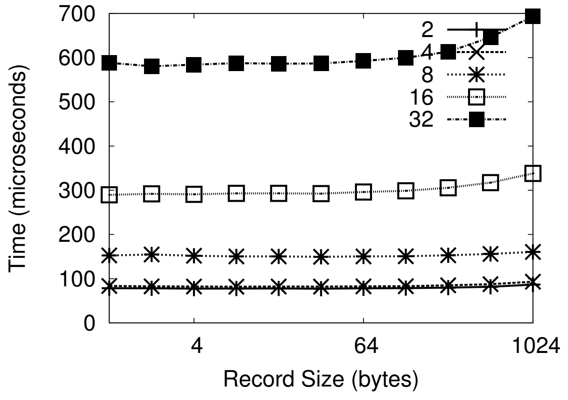


(b) Read (Medium)

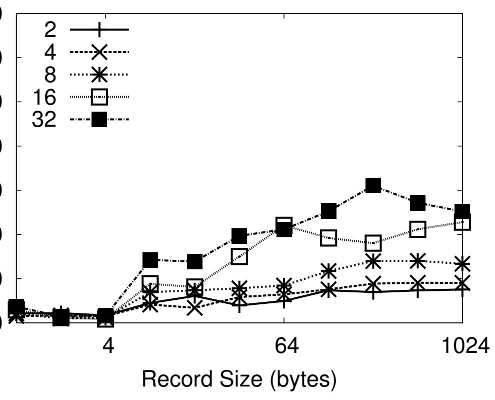
Figure 7.7: Read latency with 32 clients and varying number of MCDs. 4 DSs are used for Lustre.

Figure 7.8(c). The Read latency at 32 clients is higher than with one client and increases with increase in record size.

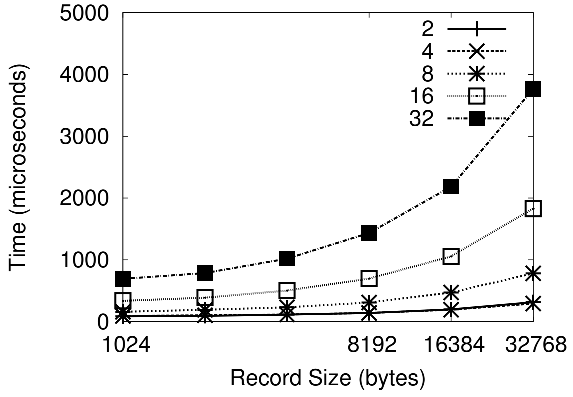
We also compare with Lustre at 32 clients (Figure 7.7(a) and Figure 7.7(b)). With a cold cache, for small Reads less than 32 bytes, *Lustre (Cold)* has lower latency than *IMCa (4MCD)*. After 32 bytes, *IMCa (4 MCD)* delivers lower latency than *Lustre (Cold)*. IMCa with 1 and 2 MCDs also provide lower latency than Lustre beyond 8K and 2K respectively. Finally, *Lustre (Warm)* again produces the lowest latency overall. However, the latency for *IMCa (4 MCDs)* increases at a slower rate with increasing record size and at 64K, IMCa (4 MCDs) has lower latency than *Lustre (Warm)*. Similar trends can also be seen with varying number of clients (Figure 7.8(b) and Figure 7.8(d)).



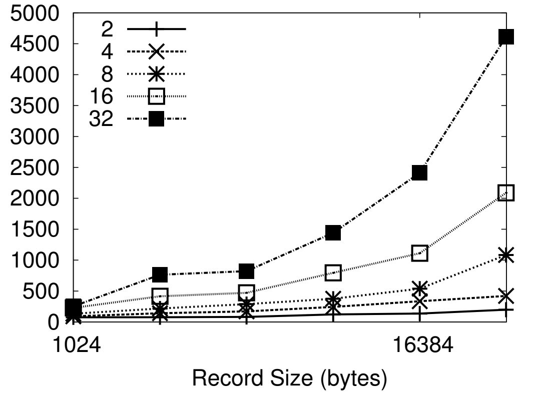
(a) Read (Small)-IMCa



(b) Read (Small)-Lustre (4DS)



(c) Read (Medium)-IMCa



(d) Read (Medium)-Lustre (4DS)

Figure 7.8: Read latency with 1 MCD and varying number of clients

7.4.5 IOzone Throughput

In this section, we discuss the impact of IMCa on the I/O bandwidth. One of the benefits of a parallel file system with multiple data servers over a single server architecture such as NFS is the striping and advantage of improved aggregate bandwidth from multiple data streams from multiple data servers. This is especially true with larger files and larger record size. Using multiple caches in MCD, it might be possible to gain the advantage of multiple parallel data servers, while using a single I/O server. We use IOzone to measure the Read throughput of a 1GB file, using a 2K block size. We replace the standard CRC32 hash function used by libmemcache [13] with a static modulo function (round-robin) for distributing the data across the cache servers using a 2K block size. We measured the IOzone Read throughput with 1, 2 and 4 MCDs. These results are shown in Figure 7.9. From these results, it can be seen that we can achieve a IOzone Read Throughput of upto 868 MB/s with 8 IOzone threads and 4 MCDs. This is almost twice the corresponding number without the cache (417 MB/s) and *Lustre-1DS (Cold)* (325 MB/s). Clearly, adding additional Cache servers helps provide better IOzone Read Throughput.

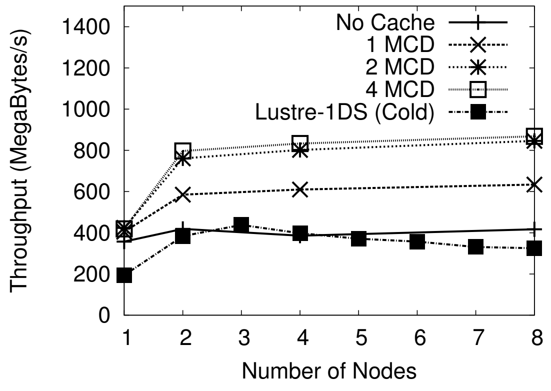


Figure 7.9: Read Throughput (Varying MCDs)

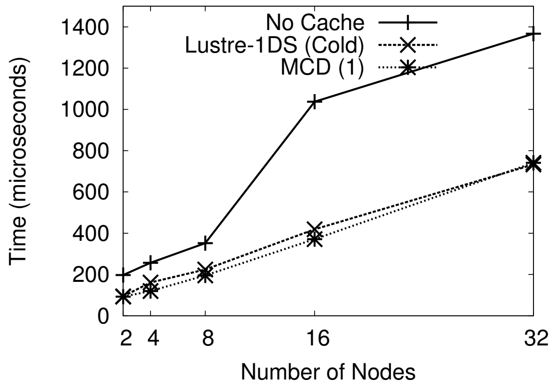


Figure 7.10: Read Latency to a shared file

7.4.6 Read/Write Sharing Experiments

To measure the impact of IMCa in an environment where file data is shared, we modified the latency benchmark described in Section 7.4.3 so that all the nodes use the same file. In the write phase of the benchmark, only the root node writes the file data. In the read phase of the benchmark, all the processes attempt to read from the file. Again, as with the multi-client experiments (Section 7.4.4), the Read and Write portions, as well the portions for each record size are separated with barriers.

We measure the read latency, with and without IMCa and compare with *Lustre-1DS (Cold)*. With IMCa, we use one MCD. The read latency is shown in Figure 7.10. At 32 nodes, there is a 45% reduction in latency with IMCa over the NoCache case. Also, as may be seen from Figure 7.10, IMCa provides benefit, that increases with an increase in the number of nodes. Since we are using a single MCD, with all the clients trying to read the data from the MCD in the same order, we see that the time even with IMCa increases linearly. With a greater number of MCDs, we expect better performance. Because of space limitations, we do not present the numbers for multiple MCDs here (they are available in [75]). IMCa with 1 MCD provides slightly higher latency compared to Lustre-1DS (Cold) upto 16 nodes. However, at 32 nodes, IMCa with 1 MCD has slightly lower latency than Lustre-1DS (Cold).

7.5 Summary

In this Chapter, we have proposed, designed and evaluated an intermediate architecture of caching nodes (IMCa) for the GlusterFS file system. The cache consists of a bank of MemCached server nodes. We have looked at the impact of the intermediate cache architecture on the performance of a variety of different file system operations such as stat, Read and Write latency and throughput. We have also measured the impact of the caching hierarchy with single and multiple clients and in scenarios where there is data sharing. Our results show that the intermediate cache architecture can improve stat performance over only the server node cache by up to 82% and 86% better than Lustre. In addition, we also see an improvement in the performance of data transfer operations in most cases and for most scenarios. Finally, the caching hierarchy helps us to achieve better scalability of file system operations.

CHAPTER 8

EVALUATION OF CHECK-POINTING WITH HIGH-PERFORMANCE I/O

The petaflop era has dawned with the unveiling of the InfiniBand based RoadRunner cluster from IBM [26]. The RoadRunner cluster is a hybrid design consisting of 12,960 IBM PowerXCell 8i processors and 6,480 AMD Opterons processors. As we move towards an exaflop, the scale of clusters in terms of number of nodes will continue to increase. To take advantage of the scale of these clusters, applications will need to scale up in terms of number of processes running on these nodes. Even as applications scale up in sheer size, a number of factors come together to increase the running time of these applications. First, communication and synchronization overhead increases with larger scale. Second, applications data-sets continue to increase at a prodigious rate soaking up the increase in computing power. ENZO [48], AWM-Olsen [101], PSDNS [1] and MPCUGLES [93] are examples of real-world petascale applications that consume and generate vast quantities of data. In addition, some of these applications may potentially run for hours or days at a time. The results from these applications may potentially be delayed, or the application may not run to completion for a number of reasons. First, each node has a *Mean Time*

Between Failures (MTBF). As the scale of the cluster increases, it becomes more likely that a particular node will become inoperable because of a failure. Second, with increasing scale, different components in the application, software stack and hardware will be exercised to their limit. As a result, bugs which would normally not show up at smaller scales may show up at larger scale causing the application to malfunction and abort. To avoid losing a large number of computational cycles because of faults or malfunctions, it becomes necessary to save intermediate application results or application state at regular intervals. This process is know as checkpointing. Saving checkpoints at regular intervals, allows the application to be restarted in the case of a failure from the nearest checkpoint instead of from the beginning of the application. The overhead of checkpointing depends on the checkpoint approach used, in concert with the frequency and granularity of checkpointing. In addition, the characteristics of the underlying storage and file system play a crucial role in the performance of checkpointing. There are several different approaches to taking a checkpoint and these are discussed next in Section 8.1. Following that, in Section 8.2, we examine the impact of the storage subsystem on checkpoint-level performance. Finally, we conclude this Chapter and summarize our finding in Section 8.3.

8.1 Overview of Checkpoint Approaches and Issues

Several different approaches exist for checkpointing. They differ in the approach taken for checkpoint initiation, blocking versus non-blocking as well as checkpoint size and content. We will discuss each of these issues next. Following, that we discuss an implementation of checkpoint/restart in a popular parallel program message passing library MVAPICH2.

8.1.1 Checkpoint Initiation

There are two potential approaches to initiating a checkpoint; application initiated and system initiated. In application initiated checkpointing, the application decides when to start a checkpoint and requests the system to initiate the checkpoint. This is usually referred to as a synchronous checkpoint. Asynchronous checkpointing is also possible. In asynchronous checkpointing, the application may request that the checkpoint be initiated by the system, at regular time intervals. The other potential approach to initiating a checkpoint is system-level initiation. In system-level checkpointing, the system directly initiates the application checkpoint, without interaction with the application. This might happen for example, if the system administrator wanted to migrate all the jobs from one particular machine to another, or a particular machine needs to be upgraded or rebooted. This might also happen if the job initiator requested a checkpoint be initiated.

8.1.2 Blocking versus Non-Blocking Checkpointing

A checkpoint may require the application be paused while the checkpoint is taken. This approach is usually easy to implement, though it might cause considerable overhead especially for long running and large scale parallel applications. Another approach is to allow for a non-blocking checkpoint to take place while the application is running. This approach is more complicated to implement, since the parallel application state might change while the checkpoint is being obtained. A potential solution to this problem is to make the data segments of the application read-only and use the technique of *Copy On Write (COW)*. Besides reducing the overhead on the application, non-blocking checkpointing allows the checkpoint I/O to be scheduled when the I/O subsystem is lightly loaded.

8.1.3 Application versus System Level Checkpointing

There are two potential techniques for taking a checkpoint. The first approach requires the application to take its own checkpoint. Advantages include reduced checkpoint size, since not all data within the application need to be checkpointed. Disadvantages include the additional effort required to develop and debug the checkpointing components of the application as well as the need to keep track of the data structures that changed between checkpoint intervals. Transparent application-level checkpointing may be achieved through compiler techniques [90]. Additionally, a hybrid approach is possible where the application participates in the creation of a checkpoint but is assisted by a user-level library [71]. The other approach is system-level checkpointing, where the checkpoint is performed transparently without application modification through the system, usually the kernel such as with Berkley Lab Checkpoint Restart (BLCR) [68]. BLCR has been combined with several Message Passing Libraries (MPI) such as LAM/MPI [98] and MVAPICH2 [73]. We discuss the MVAPICH2 checkpoint/restart facility next.

MVAPICH2 Checkpoint/Restart Facility

MVAPICH2 is a message passing library for parallel applications with native support for InfiniBand [15]. It includes support for checkpoint/restart (C/R) for the InfiniBand Gen2 device [73, 72]. Support for C/R is achieved through interaction between the MVAPICH2 library and the kernel-level module BLCR [68]. Taking a checkpoint involves bringing the processes to a consistent state through coordination with the following steps. First, all the processes coordinate with one another, and the communication channel is flushed and then locked. Following that, all InfiniBand connections are torn-down. Next, the MVAPICH2 library on each node requests the BLCR kernel module to take a blocking checkpoint on the process. After that, this checkpoint data is written to an independent file; one file per process. Finally, all processes again coordinate to rebuild the InfiniBand reliable connections and take care of any inconsistencies in the network setup that might have occurred and unlock the communication channels. Finally, the application continues execution. Checkpointing in the MVAPICH2 library may involve considerable overhead, because of the coordination requirement, the application execution suspension and the need to store checkpoint data on the storage subsystem. MVAPICH2 supports both attended (user-initiated) and unattended, interval based checkpointing. Next, we will evaluate the impact of the storage subsystem on the performance of checkpointing in MVAPICH2. We do not discuss or evaluate restart performance. Please refer to the work by Panda, et.al. [73, 72] for a discussion on restart performance.

8.2 Storage Systems and Checkpointing in MVAPICH2: Issues, Performance Evaluation and Impact

In this section, we attempt to understand the impact of the storage subsystem on checkpoint performance.

8.2.1 Storage Subsystem Issues

Taking a checkpoint is an expensive operation in MVAPICH2. This is mainly because of the need to exchange coordination messages between all the processes. As noted earlier, the final step in the checkpoint stage involves dumping the checkpoint data to a file. Since, the data must be synced to the file before the checkpoint is completed, the time required to sync the checkpoint data to the file is an important component of the checkpoint time. The characteristics of the underlying storage architecture or file system plays an important role in the time required to sync the checkpoint to the file. In Section 8.2.2, we evaluate the impact of two different file systems namely NFS and Lustre on the performance of checkpointing. In Section 8.2.3, we evaluate the impact of changing application size on checkpoint performance. Following that in Section 8.2.4, we look at the impact of increasing system size on the checkpoint performance. For all experiments, we used a 64-node cluster with InfiniBand DDR adapters and with RedHat Enterprise Linux 5 on all nodes. The cluster is equipped with four storage nodes with RAID HighPoint drives.

8.2.2 Impact of Different File Systems

In the first experiment, we measured how MVAPICH2 Checkpoint Performance is impacted by different file systems. We expect that a local file system such as ext3 will offer the best performance for dumping a checkpoint. However, the data stored on a local node is subject to the vagaries of hardware and other unforeseen consequences on that particular node. For example, if the disk fails on that particular node, the checkpoint data will be lost and it will not be possible to restart the application from a reasonable checkpoint. Correspondingly, there will be no perceptible benefit to using the checkpointing approach. Another, possibility is to use the NFS directory mount. Most modern UNIX based clusters have NFS mounted directories. Since NFS is based on the multiple client, single server architecture, a synchronous checkpoint will force the data to the NFS server, where it will be immune to client node failures. However, with large scale parallel applications, the single server in NFS might become a bottleneck and cause an unnecessary delay in performing the checkpoint and subsequently in the execution of the application. Finally, it is possible to use a parallel file system such as Lustre with data striped across multiple data servers to reduce the time for the parallel write. We have evaluated these three options and they are shown in Figure 8.1. We evaluated two applications BT and SP from the NAS Parallel Benchmarks [16]. We used class C for both applications and ran them at a size of 16-nodes. We used ext3 as the local file-system (local), NFS over TCP (NFS) and Lustre 1.6.4.3 with four data servers using native InfiniBand as the underlying transport for the experiments.

The number at the top of the bars corresponds to the application execution time in seconds (as reported by the benchmark), the number in the middle corresponds to the checkpoint data size in MegaBytes, per process and per checkpoint and finally, the number at the bottom in each bar corresponds to the number of checkpoints taken. We used 30 and 60 seconds as the checkpoint interval. From Figure 8.1, we notice the following trends. First, the running time of the application is worse with NFS than the local file system and Lustre in all cases. The difference is largest with a 30 second checkpoint interval. With BT (30 second interval) the execution time with NFS is 2.47 times that of ext3. At a 60 second interval, the execution time with NFS is 1.34 times that of ext3. Similarly, with SP, at a 30 second interval, execution time with NFS is 2.04 times that of ext3 and at 60 second intervals is 1.29 times that of ext3. Clearly, the single server of NFS becomes a bottleneck as each process tries to write the checkpoint to the same server. The single server bottleneck is also the reason behind the dramatic reduction in application execution when the checkpoint interval time is increased from 30 seconds to 60 seconds (1.93 with BT and 1.73 times with SP).

The second important trend we observe is that parallel application execution time with checkpointing enabled is lower when using Lustre with four data servers as compared to the local file system. Parallel application execution time with BT, when using Lustre is approximately 11% lower than ext3 irrespective of checkpoint interval time. For SP, the application execution time with Lustre is approximately 17% lower than ext3 at 30 second intervals and approximately 8% lower at 60 second intervals. We attribute this better performance to two factors. First, the RAID drives on the storage nodes used by Lustre can achieve IOzone bandwidth slightly over 400 MB/s as compared to the disks on the client nodes which can attain approximately 70-80 MB/s. In addition, the write data is striped across four data servers, giving us the advantage of parallel I/O. As a result, checkpoint execution time is lower with Lustre. We use Lustre, with four data servers for all remaining experiments.

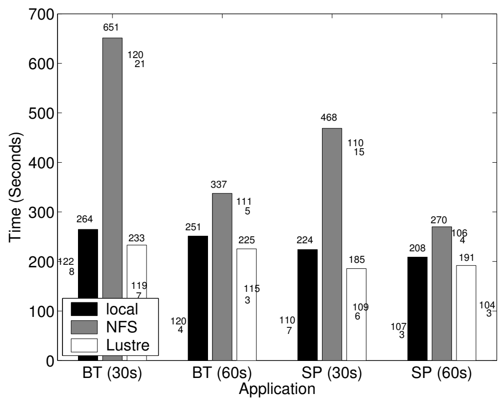


Figure 8.1: Impact of different file systems on checkpoint performance at 16 nodes. Refer to Section 8.2.2 for an explanation of the numbers on the graph.

8.2.3 Impact of Checkpointing Interval

We also evaluated the effect of changing the checkpoint interval. We keep the file system the same in all cases, Lustre with four data servers, which delivers the best performance as discussed in Section 8.2.2. We used BT (class C) and SP (class D) for the evaluation. The performance at 36-nodes (one process/node) is shown in Figure 8.2 and at 16-nodes (also one process/node) is shown in Figure 8.3. The checkpointing intervals chosen are 0-seconds (checkpointing disabled), 30-seconds and 60-seconds. We can observe the following trends. First, as expected when checkpointing is disabled, we get the lowest application execution time. When the checkpoint interval time is set to 30-seconds, the application execution time is maximum. With BT, at 36 nodes the increase in time is approximately 9% and at 16-nodes, the increase in time is approximately 5%. For SP, the correponding increase in times are approximately 7% and 5% respectively. When the checkpoint time interval is increased to 60-seconds, the difference in application time with no-checkpoint and with checkpointing is lower as compared to that with a 30-second interval. The increase in time for BT at 36 nodes is 3% and at 16 nodes is approximately 2%. Similarly, for SP the corresponding increase in time is approximately 2% and 3%. Clearly, using a checkpoint interval of 60-seconds with a Lustre file system using four data servers introduces very little overhead, even though checkpointing is a blocking operation. We need to consider that the storage space at the data servers is a finite resources. Too many checkpoints might fill up the data server storage space. To deal with this situation, MVAPICH2-C/R allows us to maintain the last N checkpoints, deleting the ones prior to that.

8.2.4 Impact of System Size

Finally, we evaluate the overhead of checkpointing with MVAPICH2-C/R on BT performance as the system size is increased. For a slightly different scenario from the ones used previously, we used class D (the largest class available with NAS) and two different system sizes, 16-nodes (1 process/node) and 64-nodes (1 process/node). The results

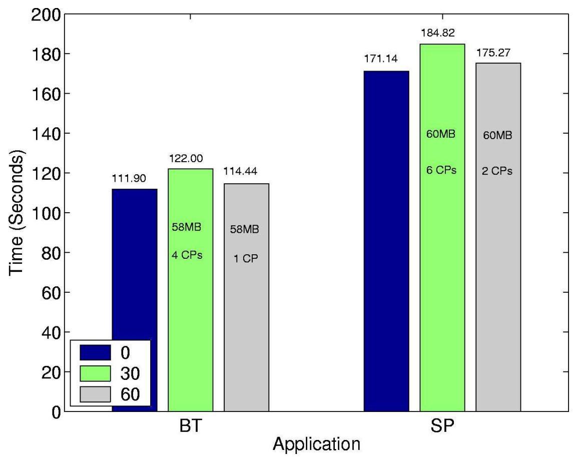


Figure 8.2: Checkpoint Time: 36 processes

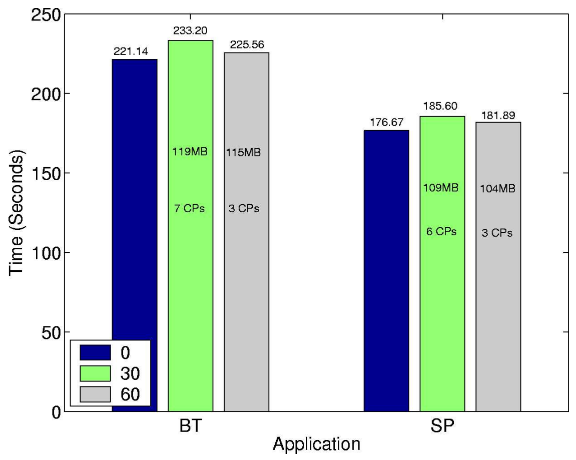


Figure 8.3: Checkpoint Time: 16 processes

from these experiments are shown in Figure 8.4. We used two different checkpoint interval scenarios, No checkpoints, which establishes an upper bound on the parallel application performance. The second interval scenario involves taking checkpoints at two minute intervals (120 seconds). At 16-nodes, the increase in application time is approximately 22% and at 64-nodes is approximately 35%. The number of checkpoints created (the application needs to run for a longer time) and the size of each checkpoint are much larger at class D as compared to class C. This is responsible for the increase in the time required for checkpointing. Clearly, we need to take into account the application in-memory core size while deciding on the checkpoint interval and use longer checkpoint intervals for larger application sizes. Finally, with increasing system size, there is an improvement in parallel application performance, especially for a strong scaling application like BT. However, the reduction in application execution time exacerbates the checkpointing time component.

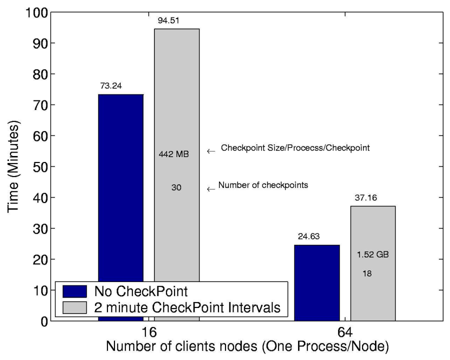


Figure 8.4: Checkpoint Time With BT class D
Lustre with 4 dataservers

8.3 Summary

In this section, we evaluated the role of the storage subsystem plays in the process of checkpointing. We find primarily that the type of file system and disk storage directly impacts the performance of checkpointing. Of the three file systems NFS, ext3 and Lustre evaluated, Lustre was shown to perform the best because of striping which spreads a large write across different I/O data servers in parallel. The type of the underlying storage I/O disk also plays an important role in the overhead introduced by checkpointing. We also observe that by increasing the checkpoint interval time, the time lag introduced by checkpointing may be reduced to a negligible level. Finally, with strong scaling applications, with an increase in the number of processes, application execution time is reduced. Checkpointing time as a percentage component of the application parallel execution time is increased and this exacerbates the checkpointing overhead.

CHAPTER 9

OPEN SOURCE SOFTWARE RELEASE AND ITS IMPACT

In this dissertation, we have proposed, designed and evaluated communication substrates for a single server network file-system (NFS/RDMA) and a clustered file-system (pNFS/RDMA) over InfiniBand. The research performed for the NFS/RDMA code (NFSv3 and NFSv4) is opensource and is available as a part of the OpenSolaris kernel [29]. The OpenSolaris kernel and storage are used by a large community of academic, research and commercial enterprise [17, 82]. In addition, the OpenSolaris design is interoperable with the Linux NFS/RDMA design, which is available as part of the Linux kernel. Finally, the pNFS/RDMA design will be made available as an opensource project.

CHAPTER 10

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this dissertation, we have researched a high-performance NFS over InfiniBand (NFSv3 and NFSv4), NFS with RDMA in a WAN environment, pNFS over InfiniBand, an intermediate caching substrate for a distributed file system and finally an evaluation of a high-performance checkpoint for a parallel file system. As discussed in the previous Chapter, a majority of these designs are available in an open-source manner. Even though these research components are primarily oriented towards storage subsystems, they may very easily be applied to other programming paradigms such as MPI.

10.1 Summary of contributions

Overall, our main contributions include:

10.1.1 A high-performance single-server network file system (NFSv3) over RDMA

Our design includes clients and servers for both Linux and OpenSolaris as well interoperability mechanisms between these two implementations. We have also compared the tradeoffs and measured the performance of a design which exclusively uses RDMA Read as well as a design which uses a combination of RDMA Reads and RDMA Writes. These results show that the RDMA Read/RDMA Write based design performs better than the RDMA Read only based design in terms of IOzone Read bandwidth and CPU utilization. In addition, the RDMA Read/RDMA Write design exhibits better security characteristics as compared to the RDMA Read only based design. In addition, we show that the peculiar nature of communication in NFS protocols, force memory registration overhead in InfiniBand to the surface, limiting performance. Special registration modes as well as a buffer cache can considerably help enhance performance, though these mechanisms themselves have their own limitations, particularly in terms of security.

10.1.2 A high-performance single-server network file system (NFSv4) over RDMA

Our design for RPC/RDMA was also enhanced for NFSv4. NFSv4 enables higher performance through mechanisms like COMPOUND operations. We research the challenges of designing an RPC over RDMA protocol for COMPOUND operations. COMPOUND operations are potentially unbounded in length, while RDMA operations in InfiniBand are required to have a fixed length. Our evaluations show that, because of the overhead of COMPOUND operations in InfiniBand, NFSv4 performance is slightly lower than NFSv3 performance.

10.1.3 NFS over RDMA in a WAN environment

We also investigated the NFS over RDMA protocol in a WAN environment. These evaluations show that the RPC over RDMA protocol out-performs TCP/IP up to distances of 10 km, but beyond that, because of limitations in the InfiniBand RDMA protocol in WAN environments, does not perform as well as the TCP/IP protocol. We attribute this drop in performance mainly to the limited buffering available on existing InfiniBand DDR NICs. We expect the performance to be significantly better with the ConnectX NICs.

10.1.4 High-Performance parallel network file-system (pNFS) over InfiniBand

We also propose, design and evaluate a high-performance clustered NAS. The clustered NAS uses parallel NFS (pNFS) with an RDMA enabled transport. We consider a number of design considerations and trade-offs, in particular, buffer management at the client, DS and MDS, scalability of the connections with increasing number of clients and data servers. We also look at how an RDMA transport may be designed with sessions which gives us exactly once semantics. Our evaluations show that enabling pNFS with a RDMA transport, we can decrease the latency for small operations by up to 65% in some cases. Also, pNFS enabled with RDMA allows us to achieve a peak IOzone Write and Read aggregate throughput of 1,800+ MB/s (150% better than TCP/IP) and 5,000+ MB/s (188% improvement over TCP/IP) respectively, using a sequential trace and 8 data servers. Also, evaluation with a Zipf trace distribution allows us to achieve a maximum improvement of up to 27% when switching transports from RDMA to TCP/IP. Finally, application evaluation with BTIO shows that the RDMA enabled transport with pNFS performs better than a transport with TCP/IP by up to 7%.

10.1.5 Intermediate Caching Architecture

In this portion of the dissertation, we have proposed, designed and evaluated an intermediate architecture of caching nodes (IMCa) for the GlusterFS file system. The cache consists of a bank of MemCached server nodes. We have looked at the impact of the intermediate cache architecture on the performance of a variety of different file system operations such as stat, Read and Write latency and throughput. We have also measured the impact of the caching hierarchy with single and multiple clients and in scenarios where there is data sharing. Our results show that the intermediate cache architecture can improve stat performance over only the server node cache by up to 82% and 86% better than Lustre. In addition, we also see an improvement in the performance of data transfer operations in most cases and for most scenarios. Finally, the caching hierarchy helps us to achieve better scalability of file system operations.

10.1.6 System-Level Checkpointing With MVAPICH2 and Lustre

As a part of the dissertation, we evaluated the role the storage subsystem plays in the process of checkpointing. We find primarily that the type of file system and disk storage directly impacts the performance of checkpointing. Of the three file systems evaluated, NFS, ext3 and Lustre, Lustre was shown to perform the best, because of striping which spreads a large write across different I/O data servers in parallel. The type of the underlying storage I/O disk also plays an important role in the overhead introduced by checkpointing. We also observe that by increasing the checkpoint interval time, the time lag introduced by checkpointing may be reduced to a negligible level. Finally, with strong scaling applications, with an increase in the number of processes, application execution time is reduced. The checkpoint component as a percentage of the overall application execution time is higher and correspondingly, checkpoint overhead is exacerbated.

10.2 Future Research Directions

We also propose the following future research directions:

10.2.1 Investigation of registration modes

As part of the future work, we would like to study how support for upcoming registration modes like *memory windows* will impact performance. Memory windows decouple registration into different stages, allowing us to use a pipeline for registration. This may allow the user to get the best of both worlds, secure registration of large areas and reduced overhead because of the pipelining.

10.2.2 Scalability to very large clusters

In addition we would like to study how the shared receive queue (SRQ) support in InfiniBand will impact performance. SRQ allows to us to cut down on the number of communication buffer per connection. This reduces the memory footprint of the communication stack with increasing cluster scale.

10.2.3 Metadata Parallelization

Metadata parallelization allows the file system to efficiently create and mutate millions of objects in a distributed file system. This is especially relevant to environments where millions of small files may be created per second. These scenarios occur in both scientific as well as enterprise environment. In addition, metadata parallelization may potentially utilize primitives on the NIC to enhance performance.

10.2.4 InfiniBand based Fault Tolerance for storage subsystems

As part of future directions, we would like to explore how to design a fault tolerant pNFS enabled with RDMA. pNFS allows us to use multi-pathing to enable redundant data-servers. Alternatively, an RDMA enabled pNFS design may take advantages of network-level features like Automatic Path Migration (APM) in InfiniBand. APM allows alternate paths in the network to be utilized when faults in the network cause the connection to break. We would also like to explore how the shared receive queue (SRQ) optimization may be used with an RDMA enabled RPC transport that uses sessions. Sessions require the reservation of slots or RDMA eager buffers per RPC connection. Dedicating a fixed number of buffers might have an impact on the scalability of larger systems deployed with pNFS. Finally, we would like to evaluate the scalability of our RDMA enabled pNFS design.

10.2.5 Cache Structure and Lookup for Parallel and Distributed File Systems

As part of future work, we plan to investigate different hashing algorithms for distributing the data across the cache servers. In addition, we would also like to look at how network mechanisms like Remote Direct Memory Access (RDMA) in InfiniBand can help reduce the overhead of the cache bank and also provide stronger coherency. We also plan on researching how the set of cache servers may be integrated into a file system such as Lustre, where it can potentially interact with the client and server caches. Finally, we would also like to study the relative scalability of a coherent client side cache and a bank of intermediate cache nodes.

10.2.6 Helper Core to Reduce Checkpoint Overhead

We also propose to use helper cores to reduce checkpoint overhead. Multicore clusters are becoming increasingly popular. Because of limitations with applications, such as power of two scaling and scheduling limitations, some of the cores on each nodes may go unused. Also, as multicore systems evolve, some of the cores may be heterogeneous and designed for specific tasks. Some of these cores may be used for different tasks such as checkpoint compression or incremental checkpointing.

APPENDIX A

LOS ALAMOS NATIONAL LABORATORY COPYRIGHT NOTICE FOR FIGURE RELATING TO IBM ROADRUNNER

BIBLIOGRAPHY

[1] Direct Numerical Simulation (DNS). http://www.cfd-online.com/Wiki/Direct_numerical_simulation_(DNS).

[2] Fibre Channel over Ethernet (FCoE): Technology and Futures. http://www.fibrechannel.org/.

[3] Fibre Channel Wiki Page. http://en.wikipedia.org/wiki/Fibre_Channel.

[4] FileBench. http://www.solarisinternals.com/si/tools/filebench/index.php.

[5] GlusterFS: Clustered File Storage that can scale to petabytes. http://www.glusterfs.org/.

[6] HP PolyServe. http://h18006.www1.hp.com/products/storage/software/polyserve/support/hp_polyserve.html.

[7] IBRIX FileSystem. http://www.ibrix.com/.

[8] Infiniband Trade Association. *www.infinibandta.org*.

[9] InfiniPath InfiniBand Adapter. http://www.pathscale.com/infinipath.php.

[10] Introducing ext3. http://www-128.ibm.com/developerworks/linux/library/l-fs7.html.

[11] IOzone Filesystem Benchmark. In *http://www.iozone.org*.

[12] Isilon IQ X-Series Brochure. http://www.isilon.com/.

[13] libmemcache. http://people.freebsd.org/ seanc/libmemcache.

[14] Mellanox Technologies. http://www.mellanox.com.

[15] MPI over InfiniBand Project. In *http://nowlab.cse.ohio-state.edu/projects/mpi-iba/*.

[16] NAS Parallel Benchmarks. http://www.nas.nasa.gov.

[17] NFS/RDMA Transport Update and Performance Analysis. http://opensolaris.org/os/project/nfsrdma/.

[18] Non-volatile random access memory. http://en.wikipedia.org/wiki/NVRAM.

[19] NSF Research Infrastructure Project. http://www.cse.ohio-state.edu/nsf_ri.

[20] NTFS. http://en.wikipedia.org/wiki/NTFS.

[21] Obsidian Research Corp. http://www.obsidianresearch.com.

[22] OpenSolaris Project: NFS version 4.1 pNFS. http://opensolaris.org/os/project/nfsv41/.

[23] pNFS over Panasas OSD. http://www.pnfs.org/.

[24] Quick Reference to Myri-10G PHYs. http://www.myri.com/Myri-10G/documentation/Myri-10G_PHYs.pdf.

[25] ReiserFS. http://en.wikipedia.org/wiki/ReiserFS.

[26] RoadRunner Super Computer. http://www.lanl.gov/roadrunner/.

[27] SCSI STA. http://www.scsita.org/.

[28] Sun MicroSystems, Inc. http://www.sun.org.

[29] The Open Solaris Project. http://www.opensolaris.org.

[30] Top 500 Super Computer Sites. In *http://www.top500.org/*.

[31] Unix File System (UFS). http://en.wikipedia.org/wiki/Unix_File_System.

[32] Write Anywhere File Layout (WAFL). http://en.wikipedia.org/wiki/Write_Anywhere_File_Layout.

[33] ZFS File System. http://en.wikipedia.org/wiki/ZFS.

[34] Alexandros Batsakis and Randal Burns and Arkady Kanevsky and James Lentini and Thomas Talpey. AWOL: an adaptive write optimizations layer. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[35] An-I Wang and Geoffrey Kuenning and Peter Reiher and Gerald Popek. The Effects of Memory-Rich Environments on File System Microbenchmarks. In *Proceedings of the 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Montreal, July 2003.

[36] B. Callaghan. NFS Illustrated. In *Addison-Wesley Professional Computing Series*, 1999.

[37] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. An analysis of data corruption in the storage stack. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[38] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High-Performance Sockets over Protocol Offload Engines. Technical Report LA-UR-05-2635, Los Alamos National Laboratory, June 2005.

[39] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *ISPASS '04*.

[40] Brent Welch and Marc Unangst. Cluster Storage and File System Technology. In *Tutorial T3 at USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, 2007.

[41] 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: Experience, Lessons, Implications*, pages 196–208. ACM Press, 2003.

[42] Brent Callaghan and Tom Talpey. RDMA Transport for ONC RPC. http://www1.ietf.org/proceedings_new/04nov/IDs/draft-ietf-nfsv4-rpcrdma-00.txt, 2004.

[43] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.lustre.org/docs.html.

[44] Chelsio Communications. http://www.chelsio.com/.

[45] Danga Interactive. Memcached. http://www.danga.com/memcached/.

[46] Abbie Matthews David Nagle, Denis Serenyi. The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage. In *Proceedings of Supercomputing '04*, November 2004.

[47] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of 10-Gigabit Ethernet: From Head to TOE. Technical Report LA-UR-05-2635, Los Alamos National Laboratory, April 2005.

[48] G. Bryan. Fluid in the universe: Adaptive Mesh Refinement in cosmology. In *Computing in Science and Engineering*, volume 1, pages 46–53, March/April 1999.

[49] Infiniband Trade Association. http://www.infinibandta.org.

[50] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). http://tools.ietf.org/html/rfc3720#section-8.2.1.

[51] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel. In *USENIX Summer 1994 Technical Conference*, 1994.

[52] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[53] Hai Jin. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[54] John M. May. *Parallel I/O For High Performance Computing*. Morgan Kaufmann, 2001.

[55] Jon Tate and Fabiano Lucchese and Richard Moore. *Introduction to Storage Area Networks*. ibm.com/redbooks, July 2006.

[56] Lee Breslau and Pei Cao and Li Fan and Graham Phillips and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM (1)*, pages 126–134, 1999.

[57] M. Dahlin, et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[58] M. Koop and W. Huang and K. Gopalakrishnan and D. K. Panda. Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand. In *16th IEEE Int'l Symposium on Hot Interconnects (HotI16)*, August 2008.

[59] John Searle (III) Director: Vikram Jayanti Marc Ghannoum. *Game Over - Kasparov and the Machine*. ThinkFilm, 2003.

[60] John M. May. *Parallel I/O for high performance computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[61] Mellanox Technologies. InfiniHost III Ex MHEA28-1TC Dual-Port 10Gb/s InfiniBand HCA Cards with PCI Express x8. http://www.mellanox.com/products/infinihost_iii_ex_cards.php.

[62] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[63] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. 8 2006. Accepted for publication at Euro-Par 2006 Conference.

[64] S. Narravula, P. Balaji, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. Panda. Supporting strong coherency for active caches in multi-tier data-centers over infiniband. In *SAN-03 Workshop (in conjunction with HPCA)*, 2004.

[65] Network Appliance. Netapp Ontap GX. http://www.netapp.com/us/products/storage-systems/data-ontap-gx/data-ontap-gx-tech-specs.html.

[66] Steve D. Pate. *UNIX Filesystems: Evolution, Design and Implementation*. Wiley Publishing Inc., 2003.

[67] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM Sigmod Conference*, 1988.

[68] Paul H. Hargrove and Jason C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *SciDAC*, 6 2006.

[69] Pete Wyckoff and Jiesheng Wu. Memory Registration Caching Correctness. In *CCGrid*, May 2005.

[70] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[71] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. Technical report, Knoxville, TN, USA, 1994.

[72] Q. Gao, W. Huang, M. Koop, and D. K. Panda. Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand. In *Int'l Conference on Parallel Processing (ICPP)*, XiAn, China, 9 2007.

[73] Q. Gao, W. Yu, W. Huang and D. K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In *International Conference on Parallel Processing (ICPP)*, August 2006.

[74] Quadrics, Inc. Quadrics Linux Cluster Documentation. http://web1.quadrics.com/onlinedocs/Linux/Eagle/html/.

[75] R. Noronha and D.K. Panda. IMCa: A High-Performance Caching Front-end for GlusterFS on InfiniBand. Technical Report OSU-CISRC-3/08-TR09, The Ohio State University, 2008.

[76] R. Noronha and L. Chai and T. Talpey and D.K. Panda. Designing NFS With RDMA for Security, Performance and Scalability. Technical Report OSU-CISRC-6/07-TR47, The Ohio State University, 2007.

[77] S. Narravula and A. Mamidala and A. Vishnu and G. Santhanaraman and D.K. Panda. High Performance MPI over iWARP: Early Experiences. In *ICPP'07: Proceedings of the 2007 International Conference on Parallel Processing*, 9 2007.

[78] S. Narravula and H. Subramoni and P. Lai and R. Noronha and D.K. Panda. Performance of HPC middleware over InfiniBand WAN. In *International Conference on Parallel Processing (ICPP)*, 9 2008.

[79] M. Eisler S. Shepler and D. Noveck. NFS Version 4 Minor Version 1. http://tools.ietf.org/html/draft-ietf-nfsv4-minorversion1-19.

[80] S. Sur, L. Chai, H.-W. Jin and D. K. Panda. Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*, Rhode Island, Greece, 4 2006.

[81] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. pages 379–390, 1988.

[82] Sandia National Labs. Scalable IO. http://www.cs.sandia.gov/Scalable_IO/.

[83] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02*, pages 231–244. USENIX, January 2002.

[84] Bianca Schroeder and Garth A. Gibson. Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you? *Trans. Storage*, 3(3):8, 2007.

[85] Spencer Shepler, Brent Callaghan, D. Robinson, R. thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 Protocol. http://www.ietf.org/rfc/rfc3530.txt.

[86] Silicon Graphics, Inc. CXFS: An Ultrafast, Truly Shared Filesystem for SANs. http://www.sgi.com/products/storage/cxfs.html.

[87] Spencer Shepler. NFS Version 4 (the inside story). http://www.usenix.org/events/fast05/tutorials/tutonfile.html.

[88] S.R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *1986 Summer USENIX Conference*.

[89] Storage Networking Industry Association. iSCSI/iSER and SRP Protocols. http://www.snia.org.

[90] V. Strumpen. Compiler Technology for Portable Checkpoints. submitted for publication (http://theory.lcs. mit.edu/ strumpen/porch.ps.gz). citeseer.ist.psu.edu/strumpen98compiler.html, 1998.

[91] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, January 1996.

[92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[93] Mahidhar Tatineni and Mahidhar Tatineni. SDSC HPC Resources. https://asc.llnl.gov/alliances/2005_sdsc.pdf.

[94] Texas Advanced Computing Center. 62,976-core Ranger Cluster. http://www.tacc.utexas.edu/resources/hpcsystems/.

[95] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. pages 308–315.

[96] Tom Barclay and Wyman Chong and Jim Gray. A Quick Look at Serial ATA (SATA) Disk Performance. Technical Report MSR-TR-2003-70, Microsoft Corporation, 2003.

[97] W. Richard Stevens. *Advanced Programming in the UNIX environment*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[98] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proceedings of the* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, Long Beach, CA, USA, March 26-30, 2007.

[99] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

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

[101] Y. Cui, R. Moore, K. Olsen, A. Chorasia, P. Maechling, B. Minister, S. Day, Y. Hui, J. Zhu, A. Majumdar and T. Jor dan. Enabling very large earthquake simulations on Parallel Machines. In *Lecture Notes in Computer Science*, 2007.

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

[103] George Kingsley Zipf. Human Behavior and the Principle of Least Effort. Addison-Wesley Press, 1949.