# Better NFS through RDMA and Efficient Memory Registration

Ranjit Noronha* and Lei Chai* and Thomas Talpey†
and Dhabaleswar K. Panda*

* Network Based Computing Lab
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{noronha. chail, panda}@cse.ohio-state.edu

† Network Appliances
1601 Trapelo Road, Suite 16
Waltham, MA 02451A
{Thomas.Talpey}@netapp.com

January 25, 2007

Technical Report
OSU-CISRC-1/07-TR06

# Better NFS through RDMA and Efficient Memory Registration

Ranjit Noronha* Lei Chai* Thomas Talpey† Dhabaleswar K. Panda*

* Network Based Computing Lab
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{noronha. chail, panda}@cse.ohio-state.edu

† Network Appliances
1601 Trapelo Road, Suite 16
Waltham, MA 02451A
{Thomas.Talpey}@netapp.com

## Abstract

*NFS over RDMA implementations on two operating systems are shown to achieve 10 gigabit Infiniband wire saturation with careful management of memory registration. While NFS is highly desirable in grid computing environments for its familiar file API and ease of management, performance issues in its implementations over Ethernet protocol stacks have impeded its use in them. We show that NFS version 3 clients and servers on two architecturally distinct operating systems can be layered atop the new RPC/RDMA protocol, and achieve outstanding performance through the use of appropriate memory registration techniques. With NFS/RDMA and Infiniband, we demonstrate performance in excess of 700MB/s on OpenSolaris and 900MB/s on Linux, at CPU utilizations of only 10%.*

## 1. Introduction

Since its inception in the mid-80's, the Network File System (NFS) [38] protocol has become ubiquitous as a means for sharing files over the network. It has been deployed on a variety of architectures and platforms such as OpenSolaris and Linux [6, 4], including workstations and high performance computing clusters. Because of its popularity, NFS has evolved through several generations which added features, performance enhancements and pushes for standardization [38, 35, 40] to its repertoire. Despite the rapid development of different cluster file systems [50, 18, 7, 39, 43], incarnations of NFS continue to be used as the primary file system for maintaining shared directories (such as users' home directories), largely because of its reliability and ease of deployment.

Traditionally, high-speed networks such as Myrinet and Quadrics have been based on proprietary vendor specific technology. Recently, implementations of high performance networks [30, 2] based on open standards such as InfiniBand [24] and 10 Gigabit Ethernet [23, 21] have become widely available. These implementations allow for low-latency messaging of a few micro-seconds and high-bandwidth communication. For example, the Mellanox InfiniBand 4X Single Data Rate (SDR) HCA allows for a unidirectional throughput of up to 1 GBps and half ping-pong latency of less than 4$\mu$s [30]. Supporting these high performance networks are open source vendor agnostic stacks such as the OpenFabrics communication stack for Linux [33] and the InfiniBand Transport Layer (IBTL) [6] for OpenSolaris. These stacks allow for network independent, low overhead access to communication primitives.

In addition to the low latency, high bandwidth communication, many of these networks also have Remote Direct Memory Access (RDMA) capabilities. RDMA operations allow two peers to read and write data directly from each others address space. RDMA operations unlike more traditional send/receive channel semantics, are inherently one-sided, not involving the CPU at the remote end. To access the address space of a remote process through RDMA operations usually requires the creation, exchange and use of RDMA Steering Tags.

With the wide deployment of standardized hardware with associated open source software to access the hardware, it is natural to ask whether a common file system protocol such as NFS can be designed to take advantage of RDMA mechanisms. While there have been other attempts to design file systems and protocols such as PVFS[47] and DAFS [48, 19] with RDMA, these systems were built for specialized purposes. In addition, protocols such as DAFS use proprietary interfaces which limit their usage to certain applications such as databases which have been specifically designed to use them. In addition, both of them are deployed in user-space, and do not offer the protection afforded by a kernel based implementation. In addition, while there is an implementation of NFS/RDMA [16] for the OpenSolaris operating system, it suffers from security and performance limitations due to the RDMA Read based design and multiple buffer copies. Our experiments reveal that on two Opteron 2.2 GHz systems with x8 PCI-Express SDR InfiniBand adapters capable of a unidirectional bandwidth of 900 MegaBytes/s (MB/s), the IOzone [3] multi-threaded Read bandwidth saturates at just under 375 MB/s.

In this paper, we take on the challenge of designing and implementing a high performance NFS over RDMA for OpenSolaris. This implementation should allow a user to transparently enjoy the benefits of improved throughput and low overhead of a high-performance RDMA enabled network. In addition, the implementation should be designed with the security of the server in mind. Finally, we would also like to provide an interoperable NFS/RDMA implementation across multiple operating systems which would allow a large class of applications to transparently enjoy the benefits of RDMA.

While investigating the issues of designing efficient NFS over RDMA for OpenSolaris and Linux, we are also taking into account the advocated IETF RPC RDMA draft [15] requirements of NFS security and interoperability. For these purposes, we demonstrate the inappropriateness of the current RDMA Read based design of OpenSolaris NFS over RDMA. For example, such RDMA Read based design exposes part of the server memory space to any client. This likely puts the server at risk from misbehaving or even malicious clients because of length limitations in protection keys for RDMA buffers [24]. An RDMA Read based design may significantly limit the number of concurrent RDMA operations. Over InfiniBand, this number is as small as eight (for current generations of InfiniBand HCAs) compared to virtually unlimited for RDMA Write [30]. Performance evaluation with an RDMA Write based design shows a maximum improvement of 47% in IOzone Read throughput compared to a RDMA Read based design. In addition, CPU utilization is also reduced from over 24% with the RDMA Read design to under 5% with the RDMA Write based design. We provide an optimized design, which conforms to NFS direct data placement and RPC over RDMA drafts [14, 15]. In this design, and compliant with these documents, only the NFS server is allowed to initiate RDMA operations, including RDMA Read and RDMA Write as appropriate.

We also 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 which 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 in the system. Specifically, our experiments show that with appropriate registration strategies, a 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.

In this paper we make the following contributions:

- 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.

The rest of the paper is presented as follows. Section 2 provides an overview of NFS and InfiniBand. In Section 4, we describe the detailed design of NFS over RDMA, focusing on the utilization of RDMA Write in the new RPC/RDMA transport and its conformance to the IETF standards [14, 15]. In Section 6, we provide the initial performance evaluation of the design. Section 7 discusses the breakdown of time in the RDMA Write design. Section 8 discusses different registration strategies. We discuss related work in Section 9. Finally, Section 10 concludes the paper and discusses future work.

# 2. Background

In this section, we provide an introduction to NFS and InfiniBand.

### 2.1. Network File System (NFS)

Network File System (NFS) [9] is ubiquitously used in most modern clusters. It allows users to transparently share file and IO services on a variety of different platforms. 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) described in IETF RFC 1831 [41]. The first implementation of ONC RPC also called Sun RPC for a Unix type operating system was developed by Sun MicroSystems. 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 or UDP as the underlying communication transport. RDMA transport enabled RPC specifications and protocols have become available [15]. 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 (IETF RFC 1832 [42]) for encoding and decoding data streams. NFS has seen three major generations of development. The first generation, NFS version 2 (RFC 1094 [5]), provided a stateless file access protocol between the server and client using RPC over UDP. NFS version 3 [35] (RFC 1813 [13]) 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. The latest version NFS version 4 [1] specification was developed by IETF (RFC 3530 [40]) and includes features for improved access and performance.

### 2.2. InfiniBand

The InfiniBand Architecture (IBA) [24] is an open specification designed for interconnecting compute nodes, IO 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 and reliable communication semantics. The RPC/RDMA protocol specifies the use of Reliable Connected mode (RC). In this model, each initiating node (client in RPC/RDMA) needs to be connected to every other node it wants to communicate with (server) 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. This connection establishment can take place either through an out-of-band channel over another control network, or with the help of a connection manager (CM). Typically, the standard Infiniband CM daemon facility is used.

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. The sender may either choose to poll the completion queue for completions, block on the completion queue, or opt to receive an interrupt when there is a completion, by registering a completion handler. Communication in InfiniBand uses the traditional channel semantics (send/receive operations), as well as memory semantics such as the Remote Direct Memory Access (RDMA) operations. Communication buffers need to be registered with the InfiniBand HCA. Implementations of InfiniBand stacks include the Open Fabrics [33] and OpenSolaris IBTL [6].

## 3 InfiniBand Communication Model

InfiniBand supports channel semantics as well as memory semantics for reliable communication. Communication operations in InfiniBand require buffers to be registered. We discuss the communication operations as well as memory registration in the following section.

### 3.1 Communication Primitives

**Channel Semantics:** Channel semantics or Send/Receive operations are traditionally used for communication. A receive descriptor or RDMA Receive (RV) which 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. 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 semanics or remote Direct Memory Access (RDMA) are one-sided operations initiated by one of the peers connected by a queue pair. The peer which 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. To prepare a region for a memory operation, the passive peer may need to perform 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 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 Posted), Protection and Exchange Message Buffer Address and Steering Tag Exchange (Rendezvous) for the receive buffer is shown in Table 1.

**Table 1. Communication Primitive Properties**

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

### 3.2 Memory Registration

Communication operations in InfiniBand require memory areas to be registered [24]. Registration is a multi-stage operation. Registration involves assigning physical pages to the virtual area. If physical pages have already 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 pinning them. Both these operations may be performed by the operating system virtual memory system. 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* which 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 upon how much load is at the HCA, the organization of the TPT, allocation strategies and overhead in the TPT, etc [31]. As a result of the combined effects of these operations, registration is an expensive operation and may constitute a considerable overhead, especially when it is in the critical path. Figure 1 shows the half ping-pong latency of a message with and without registration costs included. To reduce the cost of memory

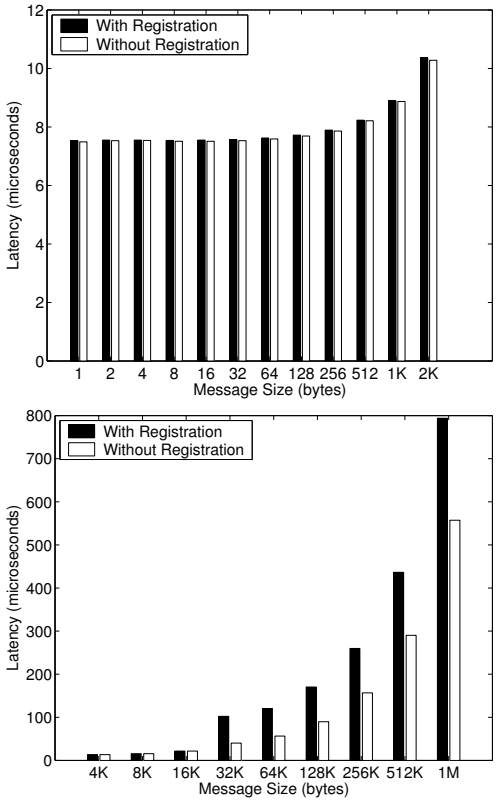



**Figure 1. Latency and Registration costs in InfiniBand on OpenSolaris**

registration, different optimizations and registration modes have been introduced. These include *Fast Memory Registration* [20, 31, 24] and *Physi-*

*cal Registration* [24].

**Fast Memory Registration (FMR):** Fast Memory Registration [20, 31, 24] 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 as a result can 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.

Additionally, the Mellanox implementation of FMR [20] introduces additional optimizations to the InfiniBand specification [24]. Similar to the specification [24], it defines a pool of steering tags which 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 becomes more than 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.

**Physical 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 needs to be considered. The *Global Steering Tag* potentially allows unfettered access to remote peers, which may have obtained the Remote Steering Tag through earlier communication with the peer. It should be used in environments where there is a level of trust between the peers. In addition, the issue of *integrity* should be considered. The HCA is unable to perform 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 there is sufficient confidence in the correctness of the communication sub-system.

# 4. NFS/RDMA OpenSolaris Architecture

Callaghan et.al. designed an initial implementation of NFS/RDMA [15] for the Solaris Operating System which subsequently became an open

source project OpenSolaris [6]. This existing architecture is shown in Figure 2. The architecture

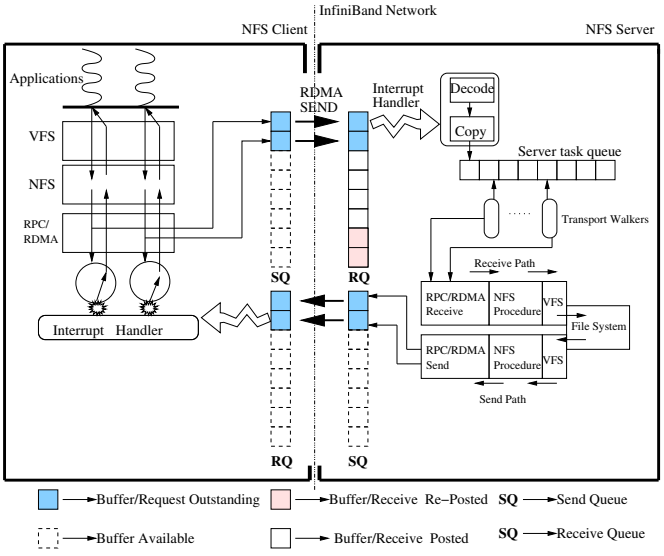



**Figure 2. Architecture of the NFS/RDMA stack in OpenSolaris**

was designed to allow transparency for applications accessing files through the Virtual File System (VFS) [8] layer on the client. Accesses to the file system through VFS are routed to NFS. If RDMA was the transport selected while doing the mount, NFS will make the RPC call through RPC over RDMA to the server. The RPC Call generally being small will go as an inline requests which are described in the next section.

### 4.1. 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 MPICH-GM [32]. The copy based protocol uses the channel semantics of InfiniBand described in Section 3.1. 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 may 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 which the client may send to the server. At the time of making the RPC Call, the client will prepend an RPC/RDMA header (Figure 3) to the NFS Request passed down to it from the NFS layer as shown in Figure 2. 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. On the OpenSolaris NFS server, this will invoke an interrupt handler which will copy out the request from the receive buffer and repost it to the connection. (The Linux server does not perform the copy, and reposts the receive descriptor at a somewhat later time.) The request will then be placed in the server's task queue. A transport context thread will eventually pick up the request which will then be decoded by the RPC/RDMA on the server. Bulk data transfer chunk will be decoded and stored at this point. The request will then be issued to the NFS layer which will then issue it to the file system. On the return path from the file system, the request will pass through the NFS layer. It will then 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.

### 4.2. RDMA Protocol for bulk data transfer

NFS procedures such as READ, WRITE, READLINK and READDIR may transfer data whose length is larger than the inline threshold [41]. In addition, the RPC call itself may be larger than the inline data threshold. There are multiple ways of transferring the bulk data. 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 ap-

proaches in detail. Before we do that, we define some essential terminology.

**Chunk Lists:** provide encoding for bulk data whose length is larger than the *inline threshold* and which should be moved via RDMA. A chunk list consists of a single counted array of segments or one or more lists. Each of these lists is in turn a counted array of zero or or more segments. Each segment encodes a steering tag for a registered buffer, its length and its offset in the main buffer. There are different types of chunks; *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 like READDIR and READLINK, and are used to RDMA Write the entire NFS response.

**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. The RPC Long Call is used by both the Read-Read and Read-Write designs.

**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.

### 4.3. Read-Read Based Design

The current RPC/RDMA design in OpenSolaris [11] is shown in Figure 4(a). It is based on

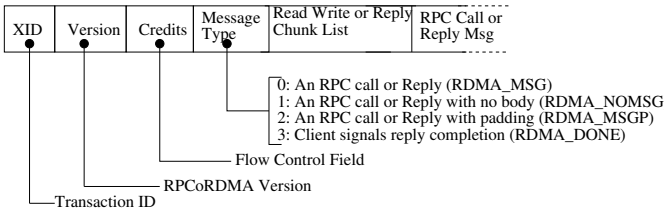



**Figure 3. RPC/RDMA header**

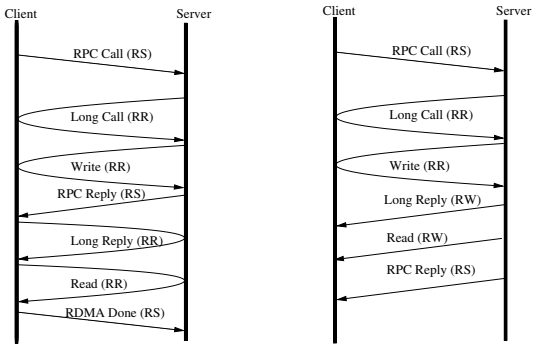
RDMA Read. As shown in the figure, the RPC Call and RPC Reply are handled using the *inline protocol* described in Section 4.1. The protocol for the bulk data transfer is as follows:

**RPC Long Call:** 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 is set to RDMA_NOMSG. If the message type is RDMA_NOMSG, the server 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). The remainder of the header is then decoded. If the message type is RDMA_NOMSG, the server decodes the read chunk encoded in the RPC/RDMA header (Figure 3) and issues RDMA Read to fetch these chunks from the client. The data from these chunks constitutes the data following the message header. The remainder of the header is then decoded.

**NFS Procedure Write:** 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 which then sends an RPC Reply via the inline protocol described in Section 4.1.

**NFS Procedure READ:** For a NFS READ procedure, the NFS server needs to encode a *Read chunk list* in the RPC Reply. The RPC Reply is then returned to the client via the inline protocol in Section 4.1. 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 which allows it to free its pre-registered buffers.

**NFS Procedure READDIR and READLINK (RPC Long Reply):** This is similar to the case for the NFS procedure READ. The server encodes a *Read chunk list* which is decoded by the client. 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.




(a) Read-Read (b) Read-Write

**Figure 4. Read-Read and Read-Write Based Designs**

### 4.4. 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 requirement for implementations of RDMA transports for NFS is that it must not be less secure than other transports such as TCP [17]. In the Read-Read design, the server side buffers are exposed for RDMA operations from the client. Since the steering tags are 32-bits [24] 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.

- *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 [24]. This may add considerable latency to the server thread.

- *Outstanding RDMA Reads:* The number of RDMA Read which 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) [24]. The IRD governs the number of RDMA Read which may be active at the remote peer, while the ORD governs the number of RDMA Read which 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 [30]. As a result parallelism is reduced at the server, especially for multi-threaded workloads.

## 5. Proposed Read-Write Design

To alleviate some of the deficiencies in the Read-Read design, the Read-Write design is specified by the IETF protocol. The Read-Write design essentially does not allow the client to issue any bulk RDMA data transfer operations. All bulk RDMA operations are issued from the server. This has several advantages. For one, the bulk data transfer buffers are no longer exposed on the server. This eliminates the possibility of misbehaving clients reading or writing to these buffers. The Read-Write design is shown in Figure 4(b).

**RPC Long Call:** The protocol for an RPC Long Call in the Read-Write design is exactly the same as in the Read-Read case discussed in Section 4.3. A Read Chunk is encoded in the RPC Call, which is decoded at the server. The Read Chunk is then read in by the server which then proceeds to decode it.

**NFS Procedure WRITE:** The protocol for a NFS WRITE procedure in the Read-Write design is exactly the same as in the Read-Read case discussed in Section 4.3. A Read Chunk is encoded in the RPC Call, which is decoded at the server. The Read Chunk is then read in by the server. The server then sends the RPC Reply back to the client.

**NFS Procedure READ:** 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.

**NFS Procedure READDIR and READLINK (RPC Long Reply):** For a NFS READDIR and READLINK procedure, the client needs to encode a Long Reply 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.

**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.

### 5.1. Advantages of the Read-Write Design

The difference between the Read-Read and Read-Write (Figure 4(b)), 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. 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 a large number of RDMA Write operations in parallel [30]. This reduces the bottleneck for multi-threaded workloads. In addition, since completion ordering between RDMA Write and RDMA Sends is guaranteed in InfiniBand [24], 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 exists at the client also when an RPC Call message is received. The elimination of an additional message helps improve performance. Finally, the control of server side buffer deregistration is no longer under the control of the client.

# 6. Comparison of the Read-Read and Read-Write Design

Figure 5 and figure 6 shows the IOzone [3] 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 [30]. These systems were running OpenSolaris build version 33. The back-end file system used was tmpfs which is a memory based file system [28]. 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 can make the following observations:

For both the Read-Read and 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. As a result 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 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, while the Read-Write design saturates at 400 MB/s. The bandwidth in both cases seems to saturate with increasing number of threads, though the saturation in the case of the RDMA-Write design takes place much earlier than in the case of the Read-Read design.

Client CPU utilization was measured using the IOzone [3] $+u$ 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 only show a single line for the Read-Read and Read-Write designs in Figure 5 and Figure 6. 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 iesign 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.

Figures 7 and 8 shows the IOzone Read and Write performance respectively on Linux with the same hardware configuration as described above with regular memory registration and deregistration. The peak Read bandwidth is 440MB/s and the peak Write bandwidth is 240MB/s.

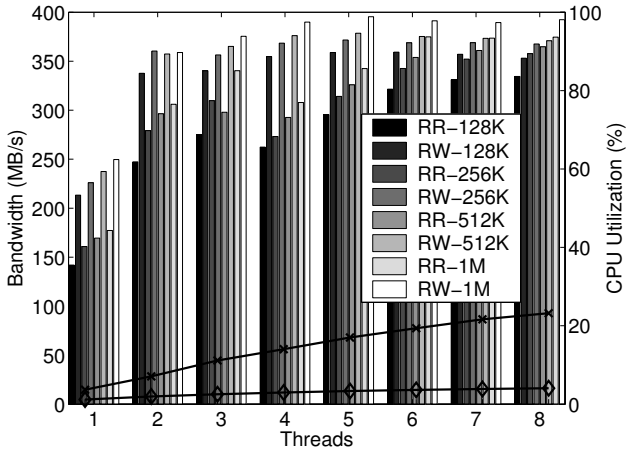



**Figure 5. IOzone Read Bandwidth on Solaris**

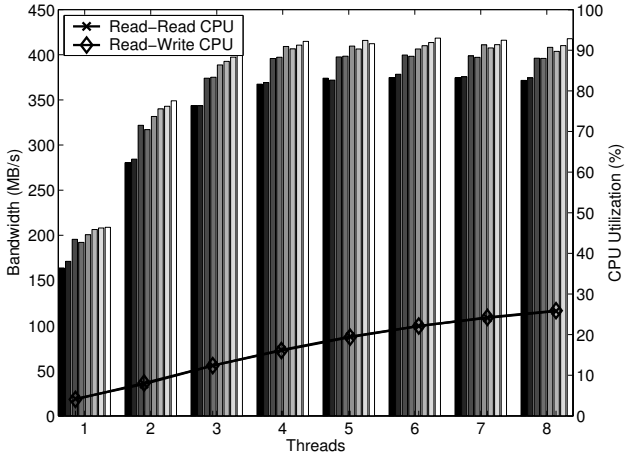



**Figure 6. IOzone Write Bandwidth on Solaris**

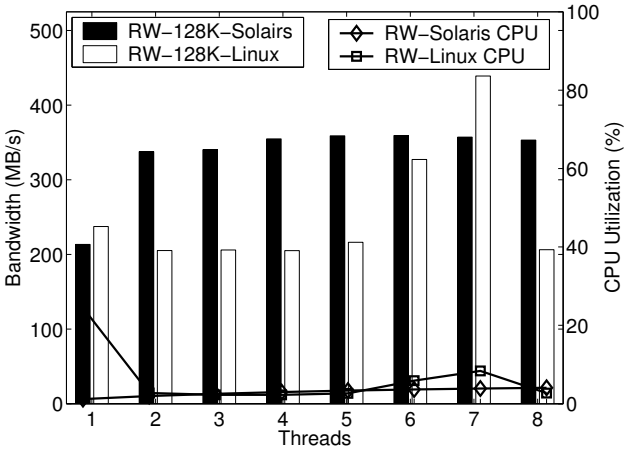



**Figure 7. IOzone Read Bandwidth: Solaris vs. Linux**

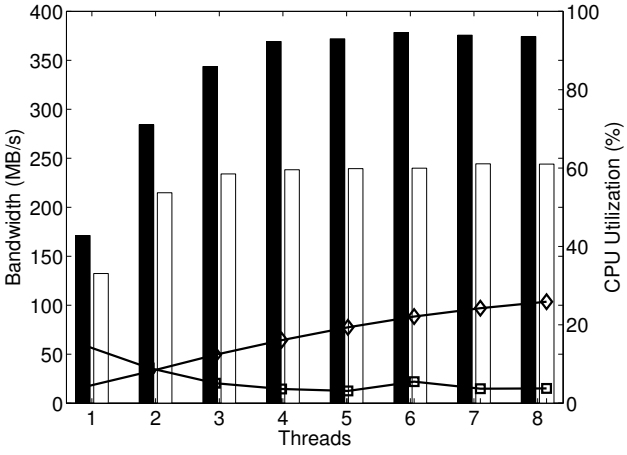



**Figure 8. IOzone Write Bandwidth: Solaris vs. Linux**

# 7. Understanding the overhead in the Read-Write design

While the Read-Write design shows improved performance over the Read-Read design, the improvement in performance is less than expected. To investigate this further, we instrumented the

RPC/RDMA layer to determine where the time went for the IOzone Read bandwidth test in Section 6. This instrumentation was done with the help of the OpenSolaris dtrace tool [10]. The instrumentation measured the breakdown in time for the RPC/RDMA layer for each individual NFS operation. These results are shown in Figure 10. The registration points are shown Figure 9. The x-axis shows a block of eight (1 to 8 threads) for each record size. From the timing breakdown, we can make the following observations. Registration is a considerable overhead on the server, taking up to 16% of the per operation time. At a record size of 128KB, registration overhead increases from about 5% at 1 thread to about 24% at 4 threads and then decreases slightly to about 16% at 8 threads. A similar trend may be observed for other record sizes. Comparatively the server deregistration is about 1-2% of the time per operation irrespective of the number of threads at 128KB. It decreases with increasing record sizes and is less than 1% for a 1MB record size. The share of the client registration and deregistration overhead is much lower by comparison (less than 0.1% of the per operation time). The time spent in tmpfs by comparison dominates starting at about 83% at 1 thread, decreasing to about 79% at 8 threads.

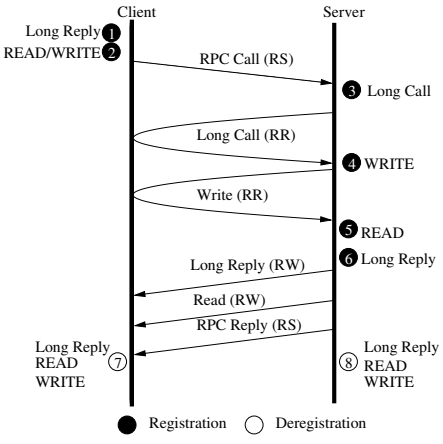



**Figure 9. Registration points in the Read-Write protocol**

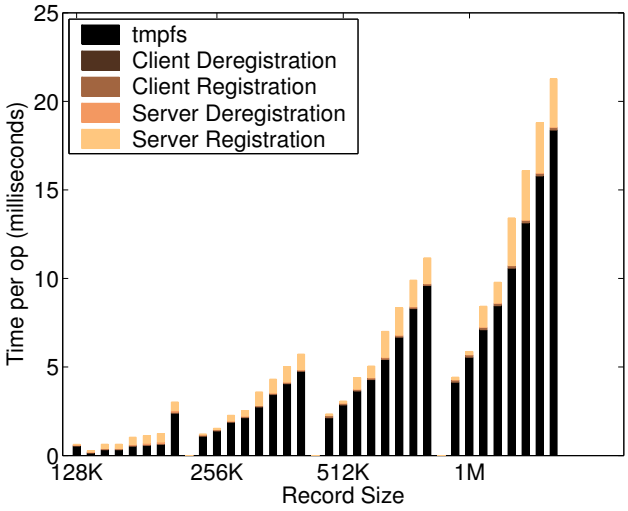



**Figure 10. The breakdown of time per operation (op). Each block of bars on the x-axes shows the trend for the record size when increasing the number of threads from 1 (left) to 8 (right)**

## 8. Registration Strategies

From Section 7, we see that registration can constitute a substantial overhead in the RDMA transport especially at the server. This overhead comes about mainly because the transport has to register the buffer and deregister the buffer on every operation. The registration occurs once at the client, and then at the server in the RPC call path. The deregistration occurs once at the server and then at the client in the RPC Reply path. To reduce the impact of the registration, we explore several different registration strategies, namely Fast Memory Registration (FMR) and a buffer registration cache.

**Fast Memory Registration (FMR):** From our earlier discussion in Section 3.2, we saw that Fast Memory Registration (FMR), reduces the overhead of memory registration by allocating entries in the HCA's TPT tables as well as protection keys, at the time of initialization. These entries are then mapped into the virtual space of the host

and may be directly accessed without the intervention of the HCA. As a result, virtual to physical memory address translations on the HCA can be updated without having to wait for the HCA to respond to a registration request. The limitations of FMR include the requirement that the maximum registration area be fixed and the restriction that this may only be used in the kernel (which is acceptable for a NFS client/server kernel implementation).

To measure the impact of FMR on performance, we have incorporated FMR calls (Mellanox FMR [20]) in the regular registration path in RPC/RDMA. To allow FMR to work transparently, there is a fall-back path to regular registration calls in case the memory region to be registered is too large. 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 again evaluated the IOzone read and write bandwidth. Since the bandwidth from the different record sizes are similar, we only present number with a 128KB record size and a 128 MB file size. The results are shown in Figure 11 and figure 12 [1]. 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 11 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 as a result of the better operation response time from the server. FMR also helps improve IOzone write bandwidth.

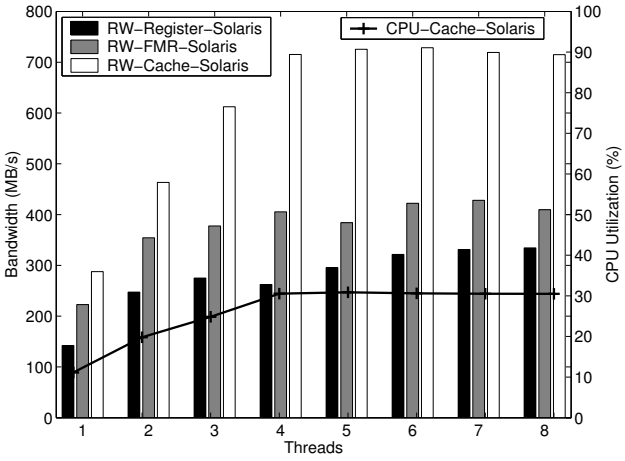



**Figure 11. IOzone Read Bandwidth with different registration strategies on Solaris**

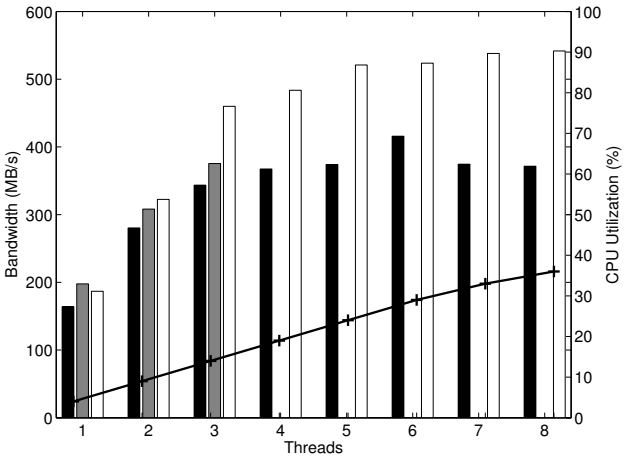



**Figure 12. IOzone Write Bandwidth with different registration strategies on Solaris**

**Buffer Registration Cache:** Another registration strategy is to create a buffer registration

[1] Problems in the FMR implementation on OpenSolaris prevent us from getting a full set of numbers at higher number of threads

cache. A registration cache [46] 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 [37]. In addition, unless static limits are placed on the amount of entries in the registration cache, the cache tends to grow 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 implemented a buffer registration cache on the server. As discussed earlier in Section 4, the NFS server state machine is split into two parts. The first part in 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 performed when control is returned 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 from a slab cache [25], 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, which 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 performance impact of the server registration cache on the IOzone Read and Write bandwidth is shown in Figure 11 and Figure 12 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 CPU utilization is also increased, though this is to be expected with an increasing op rate at the client.

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 5, 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.

Figure 13 shows the performance of the client registration cache scheme when doing IOzone multi-thread Read test. From the figure, we can see that if the record size is small, it is more beneficial to use the registration cache to get higher throughput. The peak READ throughput when using the registration cache scheme is 100% higher than that when using the zero-copy scheme. If the record size is large, memory copy is more expensive than registration, so for a small number of threads the zero-copy scheme yields higher throughput. But as long as the client has enough number of requests, the data copy time can be overlapped by network transaction time, so the registration cache scheme can achieve the same throughput as the zero-copy scheme. Of course since there is an extra data movement involved, the client registration cache scheme consumes more CPU cycles as expected. Therefore, it depends on the application characteristics (CPU intensive vs. IO intensive) and system configuration (plenty vs. limited CPU resources) to determine which scheme is beneficial to use.

**All Physical Memory Registration:** The Linux NFS/RDMA implementation provides another registration mode called *all physical memory registration*. As discussed in Section 3.2,

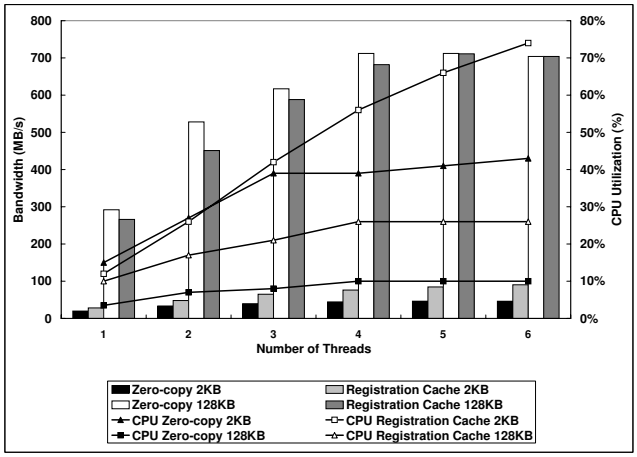



**Figure 13. Performance Impact of Client Registration Cache**

the memory registration process involves two main steps: pinning down the buffers and doing address translation. Since the buffers used for IO operations are anyway required to be pinned down, only address translation is needed when doing memory registration. Linux provides an interface that allows us to register all the physical pages in one operation. Therefore, we can register all the physical memory in the initialization phase, and only need to do virtual to physical address translation when actually registering memory.

From figure 14 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 15 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.

## 9. Related Work

There have been numerous studies in the optimization of NFS protocols. In this section, we

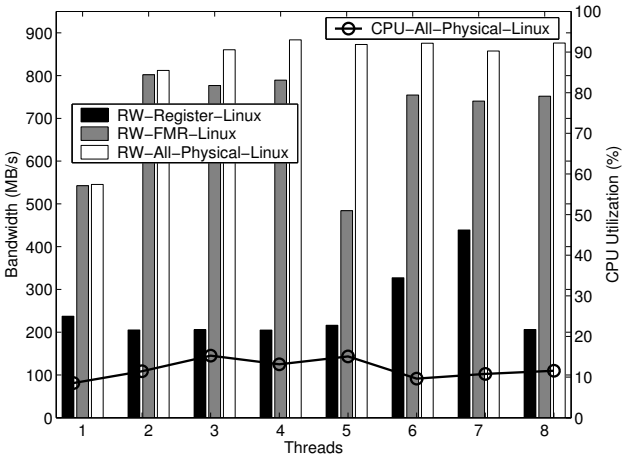



**Figure 14. IOzone Read Bandwidth with different registration strategies on Linux**

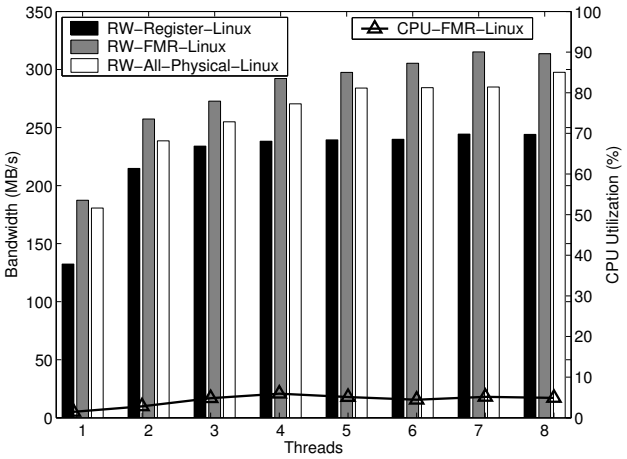



**Figure 15. IOzone Write Bandwidth with different registration strategies on Linux**

discuss about some of the most related work in the evolution of NFS, as well as its relation to the advances in high performance networks and other network-based storage protocols.

Xu et. al. [49] investigated the performance benefits of client caching to concurrent read sharing over NFS. Peng et. al. [36] showed that a network-centric reorganization of the buffer cache can improve the NFS performance. Radkov et. al. [34] compared the performance of file-based NFS protocol and block-based iSCSI protocol and noted that aggressive meta-data caching can benefit the NFS protocol.

Martin et. al. [29] studied the sensitivity of NFS to high performance networks by introducing controlled delays into live systems in the late 90's. They observed that NFS was more sensitive to processor overhead rather than networking latency and bandwidth. However, the emergence of high speed networks with direct access protocols such as RDMA lead to both the design of new network file system and the revision of traditional network file systems to enable file accesses over RDMA-capable networks. For example, iSER was recently proposed by IETF as an extension for Internet Small Computer Systems Interface (iSCSI) protocol [26, 44]. DAFS [19, 27] designed a user space file system library that allows applications to bypass operating system kernel and take advantage of high performance user-level network directly. Goglin et. al. [22] replaced the RPC protocol of NFS with Myrinet GM protocol to achieve Optimized Remote File System Accesses (ORFA). Callaghan et. al. [12] provided an initial implementation NFS over RDMA on Solaris. An RDMA read based RPC transport is implemented as a proof of concept to show the performance benefit of NFS over RDMA compared to TCP.

This work has identified the performance shortcomings in the work done by Callaghan et. al. [12]. In addition, this work attempts to provide an implementation for OpenSolaris which is compliant with the IETF drafts [14, 15] for wide interoperability with available implementations for Linux [45].

# 10 Conclusions and Future Work

In this paper, we have designed and implemented NFS/RDMA protocols for high performance RDMA networks like InfiniBand. Our implementations include clients and servers for both Linux and OpenSolaris as well interoperability mechanisms between these two implementations. We have also compared the trade-offs 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 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.

As part of the future work, we would like to study how support for upcoming registration modes like *memory windows* will impact performance. In addition we would like to study how the shared receive queue (SRQ) support in InfiniBand will impact performance. Finally, we would like to investigate how RDMA operations in high speed networks will help improve performance in implementations of pNFS.

# References

[1] General Information and References for the NFSv4 protocol. In *http://www.nfsv4.org/*.

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

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

[4] Linux Online. In *http://www.linux.org*.

[5] NFS: Network File System Protocol Specification. http://www.ietf.org/rfc/rfc1094.txt.

[6] The Open Solaris Project. In *http://www.opensolaris.org*.

[7] The Parallel Virtual File System, version 2. http://www.pvfs.org/pvfs2.

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

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

[10] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX 2004 Annual Technical Conference*, 2004.

[11] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.

[12] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. 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.

[13] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. http://www.ietf.org/rfc/rfc1813.txt.

[14] B. Callaghan and T. Talpey. NFS Direct Data Placement. http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-nfsdirect-02.txt.

[15] B. Callaghan and T. Talpey. RDMA Transport for ONC RPC. http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-rpcrdma-02.txt.

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

[17] B. Callaghan and M. Wittle. NFS RDMA Requirements. http://ietfreport.isoc.org/all-ids/draft-callaghan-nfsrdmareq-00.txt, 2003.

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

[19] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of Second USENIX Conference on File and Storage Technologies (FAST '03)*, march 2003.

[20] Dror Goldenberg and Michael Kagan and Ran Ravid and Michael S. Tsirkin. Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis. In *Hot Interconnects*, pages 128–137, 2005.

[21] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *SC '03*.

[22] B. Goglin and L. Prylli. Performance Analysis of Remote File System Access over a High-Speed Local Network. In *Workshop on Communication Architecture for Clusters, in Conjunction with International Parallel and Distributed Processing Symposium '04*, April 2004.

[23] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro '04*.

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

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

[26] M. Ko, M. Chadalapaka, , U. Elzur, H. Shah, P. Thaler, and J. Hufferd. iSCSI Extensions for RDMA Specification. http://www.ietf.org/internet-drafts/draft-ietf-ips-iser-05.txt.

[27] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proceedings of Second USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, March 2003.

[28] Marshall K. McKusick, Michael J. Karels and Keith Bostic. A Pageable Memory Based Filesystem. In *Summer 1990 USENIX Conference*, pages 137–143, June 1990.

[29] R. P. Martin and D. E. Culler. NFS Sentivity to High Performance Networks. In *ACM Sigmetrics*, 1999.

[30] 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.

[31] 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. In *Euro-Par*, September 2006.

[32] Myricom. Myrinet Software and Customer Support. http://www.myri.com/scs/GM/doc/, 2003.

[33] OpenFabrics Alliance. http://www.openib.org.

[34] P. Radkov. A Performance Comparision of NFS and iSCSI for IP-Networked Storage. In *FAST*, 2004.

[35] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer 1994*, pages 137–152, 1994.

[36] G. Peng, S. Sharma, and T. Chiueh. A case for network-centric buffer cache organization. In *Hot Interconnect 11*, August 2003.

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

[38] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[39] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02*, pages 231–244. USENIX, Jan. 2002.

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

[41] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. http://www.ietf.org/rfc/rfc1831.txt.

[42] R. Srinivasan. XDR: External Data Representation Standard. http://www.ietf.org/rfc/rfc1832.txt.

[43] Steven R. Soltis et al. The Global File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, 1996.

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

[45] T. Talpey et. al. NFS/RDMA ONC Transport. http://sourceforge.net/projects/nfs-rdma.

[46] 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.

[47] J. Wu, P. Wyckoff, and D. K. Panda. Pvfs over infiniband: Design and performance evaluation. In *ICPP*, pages 125–132, 2003.

[48] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the International Conference on Parallel Processing (ICPP'03)* , pages 125–132, 2003.

[49] Y. Xu and B. D. Fleisch. NFS-cc: Tuning NFS for Concurrent Read Sharing. *The International Journal of High Performance Computing and Networking (IJHPCN)*, 3, 2004.

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