Exploiting Direct-Access Networking in Network Attached Storage Systems

A thesis presented

by

Konstantinos Magoutis

to

the Division of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

May, 2003

To my parents.

Abstract

In recent years network storage has emerged as an important systems research field, driven by the demand for scalable storage structures to satisfy the growing needs of Internet services. With the advent of high-speed networks, the efficiency of network storage systems is directly related to the host CPU overhead of communication protocols for a large class of applications. This thesis addresses the question of what system support is necessary to reduce the communication overhead of network file systems in common use, while interoperating with existing, widely deployed network infrastructure. It demonstrates that significant host CPU overhead reduction is possible with appropriate network interface support, and shows that two novel network mechanisms are equally effective in reducing overhead for large I/Os; one of them, however, has the additional advantage of reducing server overhead for small I/Os.

In addition to aiming for overhead reduction, an appropriately designed API is necessary to provide applications with control over I/O policy. This thesis addresses the question of whether a kernel implementation of such an API can enable high performance implementations of user-level services while retaining the benefit of protection and fault isolation provided by the user-kernel boundary. It shows that a kernel implementation of an I/O API has comparable overhead to a user-level implementation of the same API and that both enable high-performance network attached storage services.

Thesis Advisor: Margo Seltzer

Thesis Author: Konstantinos Magoutis

Title: Exploiting Direct-Access Networking in Network Attached Storage Systems

Acknowledgments

During my time at Harvard, I have been fortunate to enjoy the advice, support, and friendship of a number of extraordinary people. I want to thank each and every one of them.

I consider myself privileged to have been advised by Margo Seltzer. Margo has been a constant source of energy, inspiration, and encouragement throughout my graduate student career. I will never forget Margo's lively lectures in CS261 *Advanced Operating Systems*, where I learned the fundamental principles of systems research. Her guidance, advice, and support through my qualifying exam project, my exploration for a thesis topic, and finally, through the years of thesis research, has been invaluable. The high standards she instilled in me will always guide me in my work and in setting my future goals.

I am grateful to Eran Gabber and Michael Smith for serving in my thesis committee. Eran has been an amazing mentor and friend. His commitment, sense of responsibility, and many lessons on how to build robust systems, have been instrumental in my successfully completing the Ph.D. program. I am indebted to Mike for agreeing to join my thesis committee at an advanced stage and for his critical help in bringing my dissertation to completion.

I also want to acknowledge the influence of Jeff Chase's strong spirit and inspiring motivation in my work. My collaboration with Jeff in the 2002 Usenix paper and in later work has been instrumental in my becoming a better researcher.

There are many people that I have bonded with during the years I spent in the Ph.D. program.

Soundouss Bouhia and Dionisis Margetis spent innumerable hours with me, in and out of the DEAS and Harvard Square periphery. I wouldn't have made it through without Soundouss' inspiring company and without the late night movies and hanging out with Dionysis. I have also been lucky to enjoy the close friendship of Nikos Hardavellas and Konstantina Kakavouli. Besides our friendship, in Nikos I found the ideal roommate who never sleeps and enjoys talking about research (or anything else) until dawn. In Konstantina, I found the strongly argumentative spirit that I learned to appreciate and to look for in a friend. I wouldn't have made it through without their long friendship.

I want to thank my past and present officemates, Rocco Servedio, Catherine Zhang, Salimah Addetia, and Sasha Fedorova, as well as Shlomo Gortler and Omri Traub. I will always remember the indoors volleyball games with Rocco and Shlomo in the ESL office; the Thursday nights out with Shlomo to listen to latin jazz at Wally's; the long discussions with Catherine at Zoe; and the superb team we formed with Salimah and Sasha, who taught me the value and joy of working together in a group. I fondly remember the days of Summer 2001 that I spent debugging systems with Salimah, and the San Francisco weekend with Salimah and Omri. I consider myself fortunate to have worked with Sasha, particularly during the last year of my Ph.D. I couldn't have asked for a more supportive officemate and friend during that period.

During my stay at Bell Labs in June 1999-February 2000 and the summer of 2000, I was fortunate to meet many inspiring people, among them Payal Prabhu, Fabian Montrose, Josep Blanquer, and Liddy Shriver. I will never forget spending my 30th birthday with Payal and Fabian at a bistrot on 7th Avenue; the basketball games with Fabian and Josep; Josep's summer dinner parties with Catalan cooking; and the bike treks with Liddy.

I want to thank Maya Alexandresco for encouraging me to start the Ph.D. program and for periodically reminding me that I should finish it too. Agnès Fiamma for the swimming, the incredible cooking, and for sharing her car with me. Delphine Girard for

being a unique and extraordinary friend, and Elita Kazlas for her spirit and inexhaustible supply of friendship.

I also want to thank Rodrigo Rodrigues, Tiago Ribeiro and all our common friends, particularly Sara, Xana, and Francisca, for making my last year in the Ph.D. program unforgettable.

Finally, I would like to thank Network Appliance Inc. for their funding and support of this research.

Cambridge, Massachusetts

May 2003

Table of Contents

Chapter 1

Introduction

Today, magnetic disks are so inexpensive that users are finding new, previously unaffordable uses of on-line storage [42,119]. For example, banks, hospitals and other institutions are increasingly using digital images to store information such as business documents, check images, X-rays, magnetic resonance images, etc., on magnetic disks, replacing paper or film as the dominant information storage media. Fueled by the rapid decline of on-line storage cost, total installed storage capacity increases at a rate of about 80% each year [5]. In addition, the rapid growth of Internet e-commerce, with its bursty, unpredictable storage needs, and the shortage (and therefore, high cost) of storage management personnel, create the need for data-center outsourcing of on-line storage pools [42]. With this exploding demand for storage capacity, the traditional model of *direct-attached storage*, where magnetic disks are attached to the I/O backplane of application servers, becomes infeasible due to the scalability limits of traditional I/O buses. Instead, a *network storage* model, where the storage devices are separated from the application servers by a scalable networking infrastructure, as shown in Figure 1, emerges as the prominent alternative.

1.1 Network Storage

The network storage model offers many advantages over direct-attached storage [42]. First, it offers better scalability by avoiding the physical limitations of traditional I/O

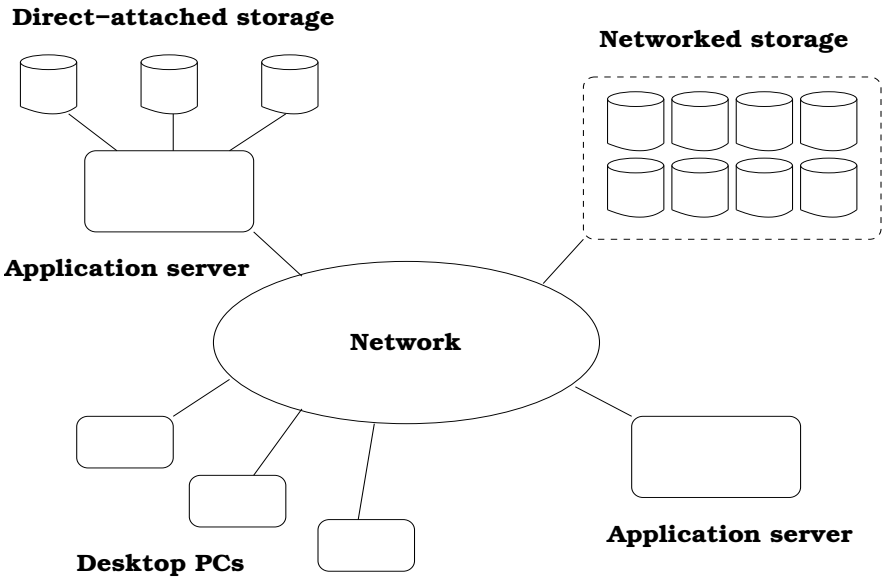

FIGURE 1. Network-attached versus direct-attached storage.

buses, such as SCSI. Second, network storage reduces wasted disk capacity by consolidating unused capacity previously spread over many directly attached storage servers. Third, it reduces the time to deploy new storage by hiding variations in disk configuration behind virtual storage pool abstractions. Fourth, it facilitates data maintenance operations, such as data backup, by performing the transfer from the on-line storage to tape without involving the application server. Finally, data sharing among clients is improved because all clients can access the same networked storage.

The network storage model, however, has several disadvantages compared to direct-attached storage. First, it has higher complexity due to the distributed nature of the network storage system. Second, transferring data over the network requires stronger security and safety guarantees than when transferring them on the system I/O bus. Finally, network protocol processing is more expensive than local hardware device access.

Network storage performance becomes increasingly limited by the end-system CPU and memory system overhead of data transfer, rather than by the traditional bottleneck of disk I/O. There are a number of reasons behind this trend: First, it is the fact that

network technology is improving at a rapid pace. Network link speed has scaled from 10Mb/s in 1994 to 2.5Gb/s today and soon to 10Gb/s [90], an improvement by a factor of 100-1000. Resource-intensive applications performing data streaming and on-line transaction processing can utilize this potential by performing I/O using aggressive read-ahead and write-behind policies and by distributing I/Os over a large number of disk spindles. In addition, the traditional disk I/O bottleneck is eased by the emergence of storage servers with large memory caches and new stable storage technologies, such as microelectromechanical systems [98]. Processor frequencies and system bus bandwidths, however, have only improved by a factor of about 20 since 1994 [90]. These technological trends point to the fact that sources of communication overhead such as memory copying and network protocol processing end up being the factors limiting I/O performance. Therefore, it becomes important to structure network storage systems aiming for reducing these sources of communication overhead.

Networked storage systems fall into two broad categories [42]: *Storage-area networks* (SAN), which typically offer a simple, untyped fixed-size (block), memory-like interface (such as *get block, set block*), and *network attached storage* (NAS) systems, which offer file system functionality to their clients and a richer, typed, variable-size (file) hierarchical interface to networked storage. NAS systems are typically accessed using a file-access protocol, such as the Network File System (NFS), by means of Remote Procedure Calls (RPC) over commodity Ethernet networks. SAN systems are typically accessed using the SCSI protocol over FibreChannel networks[1]. SAN systems provide network interface controller (NIC) support specifically for the block access protocol (e.g., SCSI), approaching the performance of direct-attached storage device access. NAS systems, on the other hand, currently rely on general-purpose RPC communication, which results to higher communication overhead than in SAN systems.

1. More recently, access to block storage over Ethernet networks has been proposed with the iSCSI protocol specification [72]. This approach, however, has not yet proven that it is a practical, viable alternative to FibreChannel-based SCSI block storage access.

1.2 Questions and Answers

The central problem addressed in this dissertation is that of reducing the host communication overhead of NAS systems. Today, most such systems use RPCs to transfer data. With conventional RPCs, the data payload is transferred in-line with the control information in the RPC message. In the absence of appropriate NIC support, this network I/O model induces memory copies in end-systems (client and server hosts). This is because the NIC cannot distinguish the data payload from the rest of the RPC message and has no information on the eventual destination of the data in host memory. It therefore places incoming RPCs in intermediate host memory buffers. Subsequently, the host copies the data payload one or more times until it reaches its final destination, as shown in Figure 2 (a). One way to avoid these memory copies is by using NIC support for a new I/O model, *Direct-Access Networking*, which is characterized by direct data transfers between the network and application space buffers. A more recent term for this I/O model, which will primarily be used in this dissertation instead of *direct-access networking*, is *Remote Direct Data Placement (RDDP)* [48]. With RDDP, the data payload is recognized and directly placed by the NIC to its final destination without any intermediate copies, as shown in Figure 2 (b). The *Remote* part in RDDP stands for the fact that the data destination buffer is targeted by the remote host. RDDP is an abstract protocol specification and does not imply a particular implementation. In this dissertation, I consider two different RDDP protocol implementations.

An existing RDDP technology, Remote Direct Memory Access (RDMA), enables a user-level process or the kernel to specifically target remote memory buffers in user-level or kernel space on a remote host. The semantics of RDMA are similar to the UNIX `memcpy` mechanism, only generalized for inter-process data transfer over the network. RDMA obviates the need for intermediate data staging by carrying the description of the destination memory buffer (e.g., host-id, process-id, virtual memory address, offset, length) with the rest of the control information accompanying the data transfer. The NIC can use the buffer description to DMA the trailing data payload directly to that buffer. With RDMA, the host CPUs are not involved in the actual data transfer, which is per-

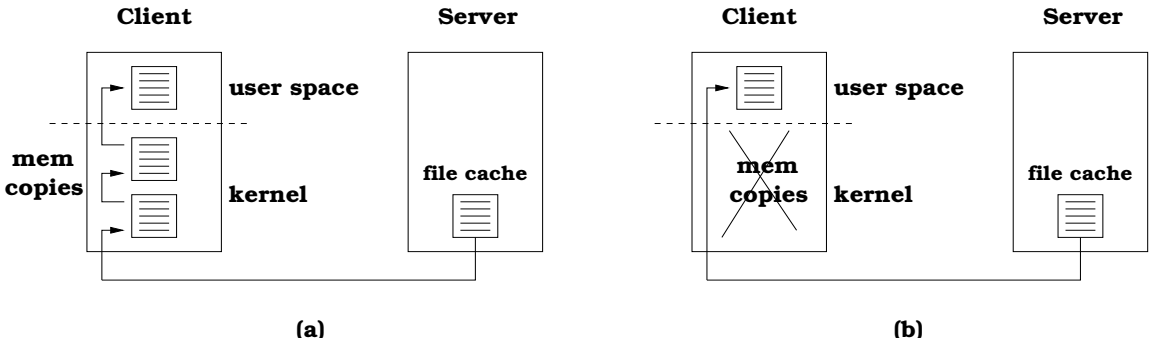

FIGURE 2. Comparison of Network I/O Models. Existing RPC implementations place the data payload in unaligned, intermediate memory buffers, requiring memory copying to move the data to their final destination (a). New RPC models using remote direct data placement (RDDP) or VM re-mapping can avoid memory copying (b).

formed entirely by the communicating NICs. RDMA is described in more detail in Section 2.3.2 on page 19.

This dissertation addresses three sets of questions, which are described in Sections 1.2.1, 1.2.2, and 1.2.3 below.

1.2.1 Comparison of RDMA Alternatives

This dissertation considers two prominent alternatives to RDMA:

- The first is a novel RDDP mechanism called RDDP-RPC. RDDP-RPC achieves direct data placement by enabling the NIC to separate the RPC data payload from the RPC control information (e.g., protocol headers), to identify the target host memory buffer, and to perform the DMA of the payload directly to that buffer. RDDP-RPC is described in more detail in Section 2.3.6.1 on page 22.

- The second is an alternative I/O model based on virtual memory (VM) re-mapping, described in Section 2.4.2 on page 29. In this model, the NIC is able to separate the RPC data payload from RPC control information and subsequently perform a DMA placing the data payload to an intermediate kernel buffer. The VM pages of the intermediate buffer are then re-mapped into the final destination buffer.

This dissertation addresses the following two questions:

- ***How does RDMA compare to the alternative RDDP-RPC technique?***

  Answer: Their performance is equivalent in the case of streaming workloads. The difference lies in their complexity of implementation and generality.

- ***How does RDMA compare to VM re-mapping?***

  Answer: RDMA has lower overhead than VM re-mapping. In my implementation, this is due to:

  (a) The transport protocol offload available in the case of the RDMA implementation. The VM re-mapping implementation uses host-based UDP/IP.

  (b) The cost of VM re-mapping, which is not incurred in the RDMA implementation.

RDDP requires that the transport protocol be offloaded to the NIC [73]. This offload should by itself achieve some overhead reduction, in addition to that of copy avoidance achievable with RDDP. The following is a related research question:

- ***What is the relative benefit of protocol offload vs. that of copy avoidance?***

  Answer: It is widely believed [73,96] that most of the benefit is due to copy avoidance. This dissertation supports this assessment, without, however, offering an experimental proof.

1.2.2 Reducing the Per-I/O Server CPU Overhead

Among the two RDDP alternatives considered in this dissertation, RDMA and RDDP-RPC, RDDP-RPC requires that both hosts be involved in setting up a data transfer. RDMA, however, does not necessarily require involvement of both sides in setting up the transfer. This can be avoided, for example, when the target NIC of an RDMA operation has all the information necessary to carry out the RDMA and does not need to interact

with the target host CPU. This property of RDMA can be used to reduce the per-I/O server CPU overhead by enabling client-initiated RDMAs, without wrapping them around RPCs, as shown in the last row of Table 1. The Optimistic RDMA (ORDMA) proposal introduced in this dissertation and described in Section 4.2 on page 56, is such a one-way RDMA primitive that addresses the challenges raised by such a design.

This dissertation addresses the following question:

- ***What are the benefits of Optimistic RDMA and under what conditions are they realized?***

  Answer: The benefits of Optimistic RDMA stem from the reduced server per-I/O CPU overhead. They are, (a) lower I/O response time, and (b) higher server throughput. These benefits are realized in multi-client workloads dominated by small I/Os, which stress the server CPU. The Optimistic RDMA model exhibits this fundamental advantage of RDMA over the alternative RDDP-RPC method. The applicability of ORDMA in widely-deployed commodity PC platforms, however, is currently restricted to read-only workloads.

Network I/O mechanism	**Steps in a read file I/O**	
	Client	**Server**
RDDP-RPC	RPC Request - -	- RPC Reply (in-line data) -
RDMA (with per-I/O RPC)	RPC Request - -	- RDMA Write to Client RPC Reply
Optimistic RDMA (without per-I/O RPC)	RDMA Read from Server -	- No involvement -

TABLE 1. Server CPU involvement in read I/O. No server involvement with RDMA (last row) means that the server-side of the data transfer is performed by the server NIC alone.

1.2.3 The Application Programmer's Interface

The RDDP I/O model enables direct data transfers between the network and application buffers. To achieve this, the application buffers used for I/O should be known to the NIC in advance, awaiting each and every incoming data payload. This requires an application programmer's interface (API) that enables *pre-posting* of receive buffers with the NIC, prior to the I/O taking place.

Pre-posting can be used for RDDP when data input takes place in response to explicit requests. This is the case, for example, when the communicating end-systems have a *client-server relationship*, as with the UNIX file API. When data input may take place at any time and without an explicit request, direct data placement in anonymous application buffers is possible by pre-posting a sufficient number of buffers to receive each and every incoming data payload. This works in the case of a *peer-to-peer relationship* between the communicating parties, such as in the case of interprocess communication using the UNIX sockets API. This method, however, cannot be used for direct targeting and usually requires data movement from the intermediate pre-posted buffers to the final data destination.

Besides the requirement to support pre-posting over an RDDP NIC, there are several questions on the nature of a good network and file API:

- ***What should the properties of such an API be?***

  Answer: A powerful and flexible API should have the following properties (Section 5.5 on page 79):

  (a) Support for event-driven design

  (b) Full control over I/O policy

  (c) Simple I/O cost model

  (d) Standard data passing semantics (*copy* or *share*)

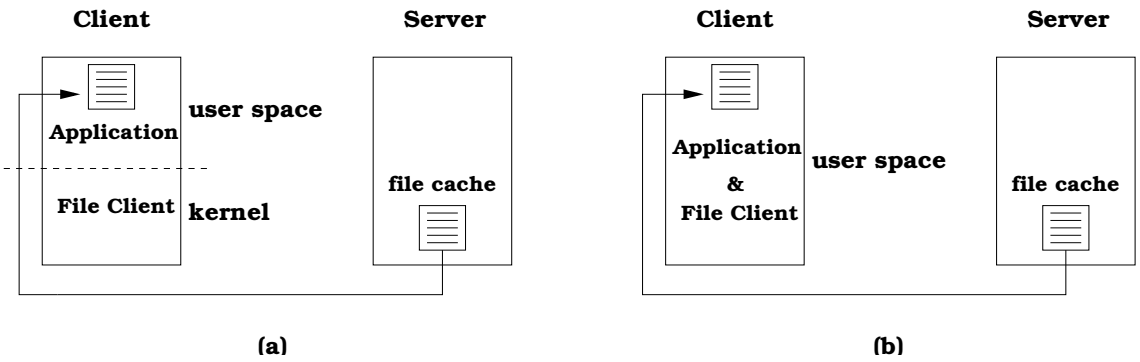

FIGURE 3. Kernel (a) versus user-level (b) implementation of a file API. The user-level structure assumes that device access is through a user-level host-device interface.

One API with these properties consists of a simple non-caching, non-blocking set of I/O operations: `read/write/select` (Section 5.5 on page 79)

The simplicity of this API facilitates efficient implementations with low complexity. By being a relatively low-level API, it enables implementations of any higher-level abstraction on top of it. This dissertation supports the argument that extensibility is primarily the result of a powerful and flexible API, obviating the need for new, radical extensible operating system structures.

- ***Is there a performance benefit in a user-level vs. a kernel implementation of this API?***

  Answer: The difference between these two choices lies only on the user-kernel protection boundary crossing. The cost of this boundary crossing is not high (500ns on the 1GHz Pentium III used in this dissertation and lower on more modern CPUs, such as current SPARCs). The choice also depends on how often the kernel boundary is crossed and how much work is performed on each crossing. This dissertation argues that the performance of network-attached storage applications (i.e., coarse-grain I/Os) depends primarily on an appropriate API, regardless of whether the API is implemented in user or in kernel address space.

One benefit of a kernel API is that it does not require a user-level host-NIC interface [113], reducing the complexity of the NIC implementation. Another benefit is the protection and isolation afforded by the user-kernel boundary. User-level APIs, however, are more portable as they do not depend on particular OS support besides a device driver.

- ***What type of kernel support is required for user-level implementations?***

  Answer: User-level APIs require kernel support for upcalls (e.g., signals) for I/O event notification. Upcalls provide an execution context to process events promptly. User-level I/O libraries, therefore, do not depend on the scheduling of user-level threads, a necessary requirement to guarantee I/O progress.

1.3 Dissertation Overview

In Chapter 2, I provide background to the main sources of communication overhead in operating systems and outline known approaches to overhead reduction. I make the distinction between per-byte and per-I/O sources of overhead and differentiate between application workloads performing large I/Os, which primarily depend on per-byte overhead reduction, and workloads performing small I/Os, which primarily depend on per-I/O overhead reduction. I argue that a remote direct data placement (RDDP) mechanism is necessary to reduce per-byte overhead. Existing RDDP mechanisms, however, require the use of RPC to set up the data transfer, which involves both communicating sides on a per-I/O basis. An RDDP mechanism that involves the host CPU only on the side that initiates the data transfer is required to further reduce the CPU overhead at the target of the I/O operation.

In Chapter 3, I focus on per-byte overhead reduction and compare two approaches to achieve RDDP. One way is by using RDMA. Another way is by using RDDP-RPC, which is possible with *tagged pre-posting of application buffers* (or pre-posting for short). I demonstrate that both mechanisms are equally effective in achieving high

performance for streaming workloads using large I/Os. The trade-off between these two mechanisms is in their ease of implementation and their generality.

In Chapter 4, I focus on per-I/O overhead reduction and introduce a new network I/O mechanism, Optimistic RDMA (ORDMA), which enables client-initiated RDMA and helps servers achieve lower per-I/O CPU overhead. I outline the design of the Optimistic Direct Access File System (ODAFS), an extension to DAFS that uses ORDMA for small I/Os. I demonstrate that ORDMA and ODAFS are effective in reducing I/O response time and improve server throughput in workloads dominated by small I/Os.

In Chapter 5, I revisit a number of commonly held assumptions about operating system overhead and argue that performance of kernel implementations can be as good as performance of user-level implementations, given a properly designed API. This evidence supports earlier results of the extensible operating systems research in the 1990's, namely that performance is primarily a result of the flexibility and expressive power of the API rather than due to flexibility offered by user-level implementations. I show that a kernel implementation of a file access API can be as efficient as a user-level implementation of the same API for streaming file access workloads, even for relatively small (4KB and 8KB) I/O sizes.

In Chapter 6, I outline my implementation of systems based on the new technologies proposed in this dissertation, as well as implementations of systems based on emerging technologies that originally motivated this research work.

In Chapter 7, I present related work, and finally, in Chapter 8, I discuss my conclusions and describe possible future research directions.

1.4 Contributions

This dissertation makes the following contributions:

- I demonstrate that end-system overhead reduction for NAS applications is possible with simple RDDP support on NICs offering transport protocol offload.

- I differentiate between *throughput-intensive workloads performing large I/Os*, which primarily depend on RDDP for copy avoidance, and *workloads performing small I/Os*, for which client-initiated RDMA is necessary to reduce server per-I/O overhead.

- I propose *Optimistic RDMA*, a new network I/O mechanism that enables client-initiated RDMA and benefits workloads performing small I/Os.

- I evaluate *Optimistic DAFS*, an extension to DAFS that uses ORDMA to improve server throughput and response time in workloads dominated by small I/Os.

- I show that the benefits of application-specific extensibility are primarily the result of a powerful and flexible API, regardless of whether the API is implemented in user or in kernel address space.

Chapter 2

Background

In this chapter, I provide background information about the architecture and implementation of protocols and systems that I use throughout this dissertation. First, I give a historical perspective of network storage systems research and discuss the sources of host CPU and memory system overhead in high-performance networking protocols and systems. Next, I present established and emerging techniques for reducing host overhead in network and file systems and discuss the effect of CPU overhead on standard performance metrics such as I/O throughput and response time. Another key to application performance, besides low overhead I/O protocols, is the flexibility to customize I/O policies to the needs of the application. In the last part of this chapter, I provide background information about systems designed to offer this flexibility as well as the role of the Application Programmer's Interface.

2.1 A Historical Perspective of Network Storage Systems Research

Until the mid-1990's, scalable storage systems were mainly distributed or network file systems, such as NFS [95], Sprite [83], AFS [46], Echo [14]. These systems have a clearly decomposed functionality between clients and servers and a communication interface between them that resembles a file I/O system call interface. The primary focus in these systems is scalability and availability, which they achieve by caching data and metadata in file system clients and by striping and replicating data in file system clients and serv-

ers. The main performance bottleneck at the time was synchronous disk I/O on file servers, a state reflected in benchmarks such as SPECsfs [92].

By the mid-1990's, a number of research projects started to contemplate different scalable storage system architectures. The *distributed virtual disk* [30,59] approach proposed servers exporting a block rather than a file interface as an alternative scalable storage paradigm, enabling multiple instances of a single-system file system to directly access file data as disk blocks. These research projects advocated that by separating the system cleanly into a block-level storage system and a file system, and by handling many of the distributed systems problems in the block-level storage system, the overall system is easier to model, design, implement, and tune [59].

Another alternative storage system paradigm was motivated by the fact that network file systems interpose a file server in the data path between the file system clients and the storage devices, introducing a bottleneck when streaming data through the I/O bus (e.g., PCI) on the file server [41]. The *Network Attached Secure Disks* [41,79] or NASD model enables a direct I/O data path between file clients and the storage devices but maintains the benefits of the network file system sharing model by using file servers for small I/O and metadata traffic. The NASD model is adopted by several emerging clustered storage systems, such as Slice [7], MPFS–HighRoad [34], GPFS [99] and Storage Tank [87].

At about the same time that these new scalable storage structures were proposed and evaluated, Ethernet or ATM networks were used to transport network I/O traffic at speeds up to 100-155 Mbps. CPU processing rates and main memory bandwidth, however, were substantially higher than network link or disk I/O transfer rates, making the network link or disk I/O the main performance bottleneck. As a result, the CPU and memory system overhead of a general-purpose communication abstraction, such as the RPC protocol, was still relatively low. Therefore, RPC was sufficient to implement either block-level or file-level network I/O in any scalable storage system.

By the late-1990's, two emerging trends posed new challenges to the scalable storage research community. First, the introduction of high-speed networks (622Mbps ATM

and 1Gbps switched Ethernet) and the projections of even faster networks, with network speed growing significantly faster than the CPU and memory bus bandwidth, introduced host CPU networking overhead as a serious performance bottleneck in network file systems. Second, the emergence of *Storage-area Networks* (SAN), exemplified by the FibreChannel [52] networking technology, opened the possibility of extending the SCSI protocol over high-speed switched networks, giving the distributed virtual disks approach a performance advantage over the network file system approach.

Strong interest in scalable storage by technology consumers in the early 2000's [42] prompted a re-categorization of the design space, into SAN-based systems, which use a block access protocol, such as FibreChannel or iSCSI [72], and which typically take advantage of special network hardware support to reduce networking host CPU overhead, and Network Attached Storage or NAS-based systems, which use a file access protocol, such as NFS, and which were still implemented over standard RPC. Besides the block-level vs. file-level interface difference between SAN and NAS systems, there has been a commonly-held belief that SAN is associated with storage-specific functionality at the network, from which NAS installations cannot benefit. This dissertation shows, however, that this is not a fundamental disadvantage of the NAS model; in other words, it is possible for NAS systems to achieve high performance given sufficient network interface support.

NAS-based systems have a number of advantages over SAN-based systems. In the NAS model, file servers handle sharing and synchronization. In addition, NAS storage volumes are under file system management and control. In SAN systems, however, single-system clients concurrently accessing a shared storage volume require synchronization mechanisms that are not present in current local file systems. Additionally, storage volumes accessed by user-level applications over a SAN are not under file system control and cannot be accessed using file system tools, which complicates data management.

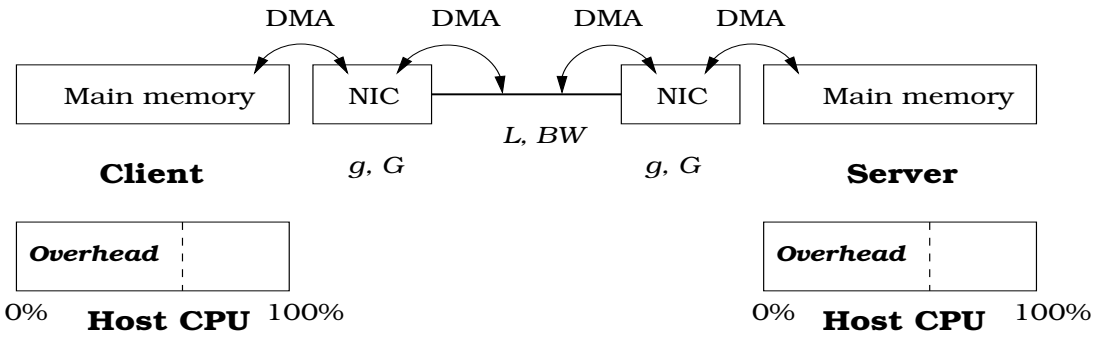

FIGURE 4. Model of a network storage system. NICs are characterized by the per-message (*g*) and per-byte gap (*G*). The network is characterized by the latency *(L*) and bandwidth (*BW)*, where $BW = G^{-1}$. Part of the host CPU is consumed by the overhead of communication.

2.2 High-Performance Networks

Performance of systems communicating over high-performance networks can be studied using the LogGP [4,28,66,67] model. LogGP consists of the following parameters, also shown in Figure 4:

Latency (L) of the communication link. This is the delay to transmit a small message on the wires and switches between two communicating network interfaces. Applications are usually insensitive to *L* for two reasons: First, modern switched network hardware achieves low link latencies, on the order of a microsecond, dwarfing communication latencies of other system components, such as the network interface or the host protocol stack. Second, applications are usually able to hide latency by pipelining the communication link, for example, by performing asynchronous network I/O operations.

Host CPU overhead (o) of the communication protocol, defined as the length of time that the host CPU is engaged in the transmission and reception of messages. Sources of host CPU overhead are mainly network protocol processing and data movement between the network and application buffers. The benefit from reducing host CPU overhead is prima-

rily for I/O-intensive applications whose processing saturates the CPU, as analytically shown in Section 3.1.

Gap (g) between consecutive small message transmissions or receptions. This parameter models the limit in throughput imposed by the NIC for transmission and reception of small messages. For message sizes above a certain threshold, this parameter is superseded by the *gap-per-byte* parameter, *G*.

Gap (G) between consecutive bytes for large message transmissions. This parameter reflects the fact that modern NICs have high-performance DMA engines transferring data between the network link and NIC memory buffers, and between NIC buffers and host buffers, matching the speed of the network link. The inverse of *G* is equal to the network link bandwidth.

P, the number of communicating nodes.

Modern high-performance network hardware, such as switches and network interfaces, feature low *L* and *G* parameters. High CPU overhead, however, can hide high-performance network characteristics from applications, due to the following reasons:

- Protocol processing on the outgoing data path, which consists of invoking the network service (e.g., via a system call), prepending protocol headers, ensuring data integrity (e.g., calculating checksums), and interacting with the network interface to initiate the data transfer.

- Network protocol processing in the incoming data path, which consists of scheduling a CPU context for protocol processing (e.g., an interrupt handler or a thread), parsing protocol headers, checking the integrity of data (e.g., calculating checksums), and delivering notification of the data arrival.

- Data movement between the network and application buffers, which typically requires memory copying to intermediate buffering systems.

Host CPU overhead consists of a per-byte component $o_{\text{per-byte}}$, which is the length of time that the CPU is engaged in data-touching operations such as memory copying or integrity checking, and a per-I/O component $o_{\text{per-I/O}}$, which is the length of time that the CPU is engaged in processing the I/O request incurred in network and file system protocol stacks. The per-packet component, due to message fragmentation and reassembly, disappears if the transport protocol is offloaded to the NIC. The following formula expresses the CPU overhead of file access in an I/O transferring m bytes:

$$o(m) = m \times o_{\text{per-byte}} + o_{\text{per-I/O}} \quad \textbf{(EQ 1)}$$

2.3 Network Protocol Support for Reducing CPU Overhead

A number of approaches have been proposed to reduce host CPU network protocol overhead. The four most prominent ones are (a) to offer a user-level instead of a kernel-based interface to the NIC (discussed in Section 2.3.1), (b) to enable a remote direct memory access (RDMA) mechanism (Section 2.3.2), (c) to offload the network transport protocol to the NIC, partly or fully (Section 2.3.3), and (d) to support a host-NIC interface designed for low overhead (Section 2.3.4).

Network storage systems can perform network data transfers using either storage-specific network support (discussed in Section 2.3.5) or Remote Procedure Calls (RPCs, Section 2.3.6). RPC-based network storage systems can be optimized as described in Sections 2.4.2 and 2.4.3.

2.3.1 User-level Networking

User-level networking [17,21,78,82,111,113,114] advocates offering applications direct, protected access to the network interface. This approach reduces the latency of small network I/Os by bypassing heavyweight host-based networking stacks and by avoiding the user-kernel interface. It also increases throughput for large I/Os by transferring data

directly in and out of a buffer pool in the application address space. User-level networking systems offer communication primitives to send and receive messages. Receives, in particular, are performed in anonymous buffers that are *pre-posted* by the application. This introduces the possibility of overflow if the application does not explicitly regulate data flow to ensure that receivers post buffers at least as fast as senders consume them. It also introduces the need for an additional memory copy in the receiving path if the communication buffer pool is not the final destination of the data. An overview of the history and present state of user-level networking as an interface to a direct-access NIC is described in Section 7.1.3.

2.3.2 Remote Direct Memory Access

Remote direct memory access (RDMA) systems [17,21,100,111,117] enable senders to target specific remote memory buffers, avoiding the need for data staging in intermediate buffers. This reduces memory copying, which is often necessary for data movement. In addition, sender-based control with RDMA avoids the possibility of buffer overflow at the receiver, since it does not require pre-posting of a memory buffer at the remote end. RDMA can be used either with a user-level or with a kernel interface to a NIC.

The term *Direct-access Networking* was introduced by the Direct-Access File System Collaborative (see Section 7.2) to characterize a class of transports offering RDMA capabilities. This term is used in a generalized sense in the title of this thesis to encompass any network mechanism that supports remote targeting of application buffers, independent of whether the targeting takes place by memory address, as in RDMA, or by another type of tag. Another, more descriptive term for the same I/O model, however, has recently become popular with the research community: *Remote Direct Data Placement (RDDP)* [48]. To be consistent with current practice, the term *RDDP* will be used for the remainder of ths thesis instead of *direct-access networking*.

2.3.3 Network Transport Protocol Offload to the NIC

Host CPU overhead can be reduced in traditional networking stacks by using standard techniques such as coalescing packet interrupts, using large maximal transfer units

(MTUs), and offloading data checksums to the NIC. These techniques are offered by commercially available high-speed network interfaces and supported by mainstream operating systems [23]. In addition to these simple approaches, offloading the entire transport protocol to the NIC has been considered and implemented either in firmware [20] or in silicon [3] on the NIC. Counterintuitively, the main benefit of offloading the network transport protocol to the NIC is not believed to be coming from the reduction of the network packet processing cost, which is not high even in complex protocols such as TCP [16,27]. Instead, the benefit is believed to be coming mostly out of the support for an RDDP mechanism [73]. Offloading the network transport to the NIC may even be counterproductive in some cases [96,101].

2.3.4 The Host-NIC Interface

There is currently a debate as to what is the right interface to an offloaded network transport protocol. Two alternatives, shown in Figure 5, are, the familiar UNIX sockets interface [112] and the Queue-Pair over IP (QP-IP) [20,111,113] interface, both compatible with the TCP/IP protocol suite. An advantage of the QP-IP interface over sockets is the ability to pre-post application buffers with the NIC, enabling direct data transfers between the

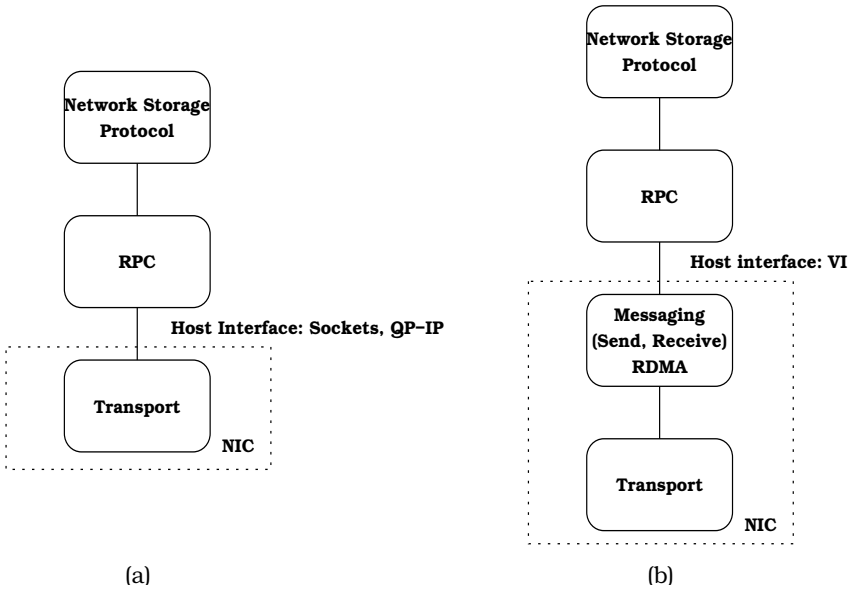

FIGURE 5. Host interface to network protocol offload to the NIC. The host interface can be UNIX sockets or QP-IP, as in (a), both compatible with the TCP/IP transport. Another possibility is VI, which introduces a new networking protocol above the transport, as in (b).

network and the pre-posted memory buffers. This helps avoid intermediate buffering and the data movement cost that this implies. QP-IP, however, does not allow senders to target specific pre-posted buffers at the receiver. In this dissertation, I introduce a new technique (referred to as RDDP-RPC, described in Section 2.3.6.1) that enables tagged pre-posting of application receive buffers. This technique augments the QP-IP interface by enabling sender-based management of receiver buffers. The Virtual Interface (VI) [111] protocol defines another possible interface to an offloaded transport protocol, supporting both buffer pre-posting and sender targeting of remote receive buffers via RDMA. VI, however, introduces an additional network protocol over TCP/IP.

2.3.5 Storage-area Networks

Storage-area networks preserve two important properties of direct-attached block I/O device interfaces, namely the ability to transfer data reliably without host involvement and the ability to place data directly in a user or kernel memory buffer without intermediate memory copies. SANs achieve these properties by offering NIC support for:

- Reliable data transfer, using an appropriate link- or transport-layer protocol.
- Directly supporting the block-level storage protocol, and therefore being able to identify the data payload, match it with its target host user or kernel memory buffer, and to place the payload directly to that buffer.

These capabilities come at the expense of requiring either appropriate network infrastructure that is not widely deployed, as in the case of FibreChannel [52], or NICs offering support for a block-level storage protocol, for example, is the case of FibreChannel and iSCSI [72] NICs.

2.3.6 RPC-based Approaches

Remote procedure call [15] is a general-purpose communication abstraction that can be supported over networks and NICs that provide simple unreliable, out-of-order link-layer data transfer semantics. Traditional NAS systems typically use RPCs for data trans-

fer. This is because such systems are often deployed over ubiquitous network infrastructure, such as Ethernet, which does not specifically support the NAS protocol. A drawback of using RPCs for file I/O data transfer is that, without special NIC and protocol support, this method requires staging of the data payload in intermediate host memory buffers and copying, to move the data to its final destination.

One way to solve the high overhead of RPC-based file I/O is by enabling direct data transfers between clients and storage nodes over a SAN for large I/Os, as proposed in the NASD model [41]. Alternatively, remote direct data placement mechanisms can be applied to RPC-based data transfers over IP networks without requiring a SAN. For example, DAFS [29,64] and NFS-RDMA [22] are two recently proposed NAS systems based on NFS, which use RDMA to avoid memory copying and to offload the transport protocol. These approaches promise to reduce communication overhead to levels comparable to that of block-level protocols.

2.3.6.1 Remote Direct Data Placement (RDDP)

Network storage systems can be implemented based on the interfaces and semantics of the network protocols shown in Figure 6. In particular, RPC can be implemented over a messaging layer, which can be offloaded to the NIC along with the transport protocol, as shown in Figure 5. The messaging layer can be accessed by the host via an interface that exports send and receive operations [20,111]. In addition, direct placement of upper-level protocol data payloads into their target host memory buffers can be achieved with RDDP [48], as shown in Figure 6(a,b) and described below.

A communication layer implementing RDDP must perform the following operations:

- Separate the protocol header from the data payload,
- Match the latter with its target buffer on the receiver, and
- Deposit it directly into its target buffer.

To be able to match the data payload with its target buffer on the receiver, the target buffer must be tagged and advertised prior to the I/O. Tag advertisement can be either implicit or explicit, as shown in Figure 6, depending on whether it is performed by the RPC protocol or explicitly by the NAS protocol. In either case, however, advertisement is performed by an RPC. The data payload can be in-lined in the RPC message or transferred separately, using remote direct memory access.

RDDP using RPC. One way to enhance RPC with RDDP is to associate the target buffer with an RPC-specific tag and advertise this tag to the remote host. The remote host must include the advertised tag in the RPC that carries the data payload. The receiving NIC must match the tag with the target buffer, separate the data payload from the protocol headers (*header splitting*), and deposit the data directly into its target buffer. I will refer to this method as RDDP-RPC.

RDDP using RDMA. Another way to implement RDDP is using RDMA, which is a network data transfer protocol [21,117]. The RDMA layer exports a remote memory read and

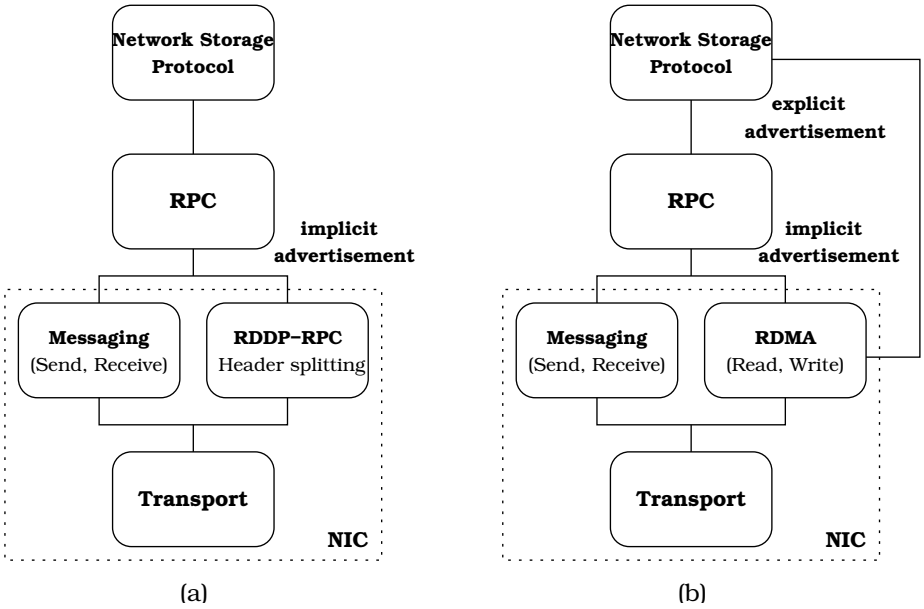

FIGURE 6. Protocol stack with the messaging and transport protocols offloaded to the NIC. RDDP is possible either by separating the data payload when in-lined in the RPC (a) or by transferring data separately with RDMA (b).

write interface. RDMA uses host virtual memory addresses as RDDP buffer tags. An RPC advertises the remote buffer and an RDMA moves the data to the target buffer. RDMA requires interaction with the upper-level protocol only to initiate the RDMA operation. It does not require interaction with the upper-level protocol at the target of the remote read or write operation. Only the RDMA initiator receives notification of completed events. I will refer to this method as RDMA.

User-level networking requires that RDMA use virtually addressed buffers. NICs with RDMA capabilities use a Translation and Protection Table (TPT), which is a device-specific page table, to translate virtual addresses carried on RDMA requests to physical addresses. To avoid limiting the size of the TPT, NICs can be designed to store the entire TPT in host memory, maintaining only a TLB on-board the NIC [78,117]. Systems using RDMA need to ensure that the NIC can find virtual to physical address translations of exported pages referenced in RDMA requests and that memory pages used for RDMA are kept resident in physical memory while the transfer takes place. Page registration through the OS is necessary in conventional NICs on the I/O bus, to ensure that address translations are available and that pages remain resident for the duration of the DMA.

Both RDDP-RPC and RDMA rely on the transport protocol offloaded to the NIC. They differ, however, in the complexity of implementation and in their generality. RDMA is a general-purpose data transfer mechanism: it is independent of any NAS protocol and exports a user-level API. NICs supporting RDDP-RPC are simpler to design and implement. They are customized, however, for particular NAS protocols and export a kernel API.

2.3.6.2 Implications of RDDP Tag Advertisement

Protocols using RDDP for direct data placement typically advertise buffer tags by an RPC on a per-I/O basis, as shown in Figure 7 (a). Advertisement of buffer tags on a per-I/O basis, however, means that both sides are involved in setting up each data transfer. In particular, a control transfer to the server is performed every time data transfer is needed, increasing the per-I/O cost of the data transfer. An alternative that avoids the need for per-I/O buffer advertisement is to cache advertisements in clients and carry file access

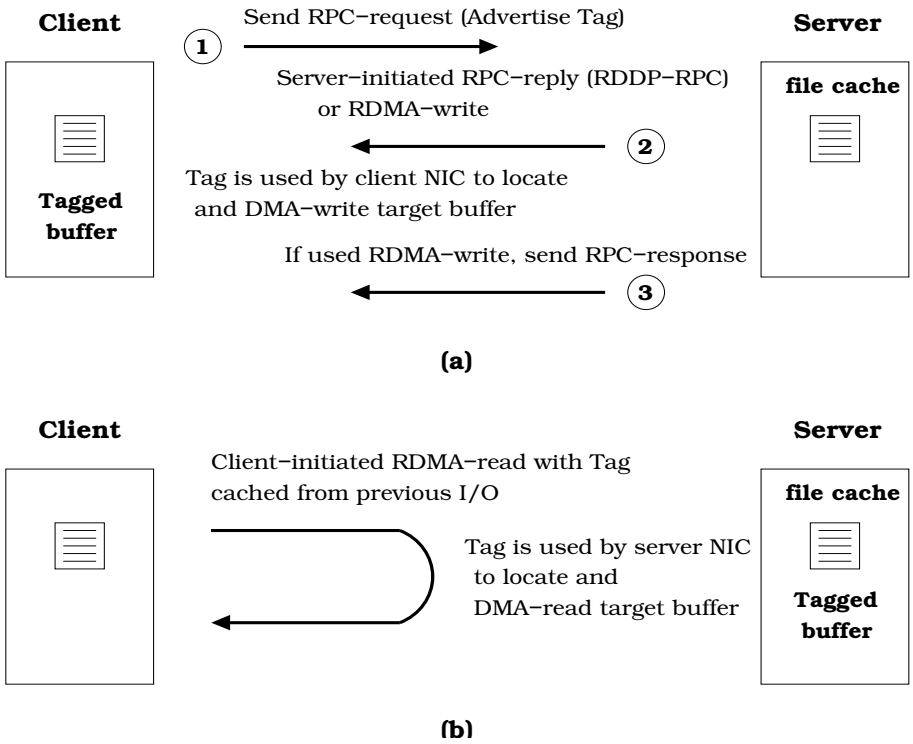

FIGURE 7. RDDP methods rely on *buffer tags* to enable NICs to identify target buffers. RDDP-RPC requires that tag advertisement take place on each I/O (a). Tag caching can be used with RDMA to avoid a per-I/O RPC (b). Figure shows steps in read I/O.

operations by RDMA only, as shown in Figure 7 (b). This enables the use of client-initiated RDMA without requiring buffer advertisement, thereby avoiding RPCs on each I/O.

2.3.6.3 Reducing Per-I/O Server CPU Overhead

The primary source of per-I/O CPU overhead is RPC processing. The main components of RPC are event notification, either by interrupt or polling, process scheduling, interaction with the NIC to start network operations or to register memory, and execution of the file protocol processing handlers. Part of the overhead of RPC is expected to improve with advances in core CPU technology. Other parts of the per-I/O overhead, however, such as interrupts and device control, are due to the interaction between the NIC and the host over the I/O bus and therefore not expected to improve as quickly as core CPU performance.

RDMA has fundamentally lower per-I/O overhead than RPC for remote memory transfers since it does not involve the target CPU. Reducing per-I/O overhead in file clients using RDMA is possible with techniques such as *batch I/O* in DAFS [29,64]. Using batch I/O, a single RPC is used to request a set of server-issued RDMA operations, amortizing the per-I/O cost of the RPC on the client. Reduction of per-I/O overhead is especially important on file servers, since servers receive I/O load from multiple clients.

Previous research examined the benefits of avoiding the control transfer inherent in the RPC mechanism when all that is needed is to perform an I/O operation. Thekkath and his colleagues [109] advocated the separation of data and control transfer in distributed systems. They proposed using RPC only when control transfer is necessary, otherwise use a pure network I/O mechanism, such as RDMA. To evaluate the benefits of this approach, Thekkath and his colleagues proposed a network file system structure based on client-initiated RDMA. Their RDMA model, however, makes the following simplifying assumptions:

- Virtual memory buffers are pinned in physical memory, at the client and the server.

- Remote memory address mapping is based on a hash-based scheme, enabling clients to compute the remote memory location of data and metadata, avoiding the need for buffer advertisement by RPC.

Pinned virtual memory buffers is not a realistic assumption, particularly in the case of a file server buffer cache. In addition, hash-based remote memory mapping methods require significant network file system re-design. In this dissertation, I propose extensions to the RDMA mechanism and to the DAFS protocol, both described in Chapter 4, that do not require these simplifying assumptions. These extensions can be easily incorporated into existing RDMA and network file system implementations.

2.3.6.4 Messaging and Transport Layers

The main purpose of the RDDP-RPC and RDMA protocols described in Section 2.3.6.1 and shown in Figure 6, is to support tagged remote direct data placement. RDMA provides for data transfer and requires the use of a separate RPC for control transfer. RDDP-RPC, however, combines data transfer with control transfer. Control transfer requires sending and receiving messages through a messaging protocol layer. Besides message transmission and delivery, a messaging layer provides event notification but leaves event handling to upper-level protocols such as RPC. An example of a protocol providing user-level messaging and RDMA is the Virtual Interface (VI) architecture [111].

Underlying all the protocols mentioned above is a need for a transport protocol layer that exports a reliable, in-order stream abstraction, similar to the TCP sockets interface. In addition, transport protocol support for framing, such as in SCTP [105], is required by RDDP in order to preserve upper-level protocol header and data payload boundaries.

2.4 High-Performance Network Storage Systems

Reduction of communication overhead is one of the primary goals in high-performance network storage system design today. In this section, I will discuss the applicability of several overhead reduction mechanisms in network storage systems.

2.4.1 Overhead Reduction Techniques

The primary source of per-byte overhead in NAS systems is memory copying, which is sometimes required due to duplicate buffering/caching in different software layers and in different address spaces. Ideally, the I/O payload should be transferred directly from its source to the destination buffer without unnecessary copying in between. In addition, sharing data between applications and file caches should be possible without memory copying. Extensive research on the subject of avoiding memory copies in operating systems, primarily to optimize inter-process communication, has resulted in copy avoidance mechanisms, with Mach's *Copy-on-Write* (COW) [1] as a prime example. With COW, two

communicating address spaces (either two user-level processes or a user-level process and the kernel) can share mappings of the same physical memory region until the memory region is modified, at which time a physical memory copying takes place. An example of such sharing is the BSD `fork` interface [70]. Data sharing in a similar way but without a lazy copy is also possible, as is for example the case in the BSD `mmap` interface [70]. Data movement by sharing VM mappings is usually necessary when both communicating address spaces wish to keep accessing the same data, for example between a consumer of data and a cache. However, when the originator of the data does not wish to maintain a handle to the data buffer, such as for data movement between buffering systems, another way to achieve data movement is by trading VM mappings to physical memory regions. These possibilities based on VM re-mapping are summarized in Table 2.

Overhead Reduction Techniques		**Sharing**	**Use**
VM re-mapping	Sharing Mappings	Explicitly	UNIX `mmap`
		Implicitly	COW (UNIX `fork`)
	Trading Mappings	No	Move data between buffering systems
RDDP		No	Direct device-application I/O

TABLE 2. Summary of overhead reduction techniques.

Copy avoidance in network I/O is relatively easy because data caching is usually not necessary in the network I/O path. Avoiding memory copies can be done using VM page re-mapping and NIC support for gather/scatter sends, and by depositing incoming data either in a page-aligned location or directly at the final destination. In the latter case, two ways to achieve direct data placement in host memory are either within the context of RPC (RDDP-RPC) or in combination with RDMA. Caching in the I/O data path, which is a standard requirement in file systems, complicates things as data needs to be shared between the data cache and the application. An NFS client using VM page-remapping [64] is evaluated in Chapter 3.

2.4.1.1 Semantics of Data Movement

The choice of copy avoidance method is influenced by the semantics of the network or file I/O API. The API semantics determines whether the data originator retains access to its I/O buffers after issuing the I/O operation (as in *copy* and *share* semantics) and whether modifying the I/O buffer may result in corruption of the buffer data.

- **Copy**. Originator may keep accessing a data buffer after initiating I/O, without fear of corrupting the transfer. An example of an interface with copy semantics is the UNIX `read/write` API.

- **Share**. Originator may keep accessing a data buffer after initiating I/O, but will corrupt the transfer if the buffer is modified prior to I/O completion. An example of an interface with share semantics is the UNIX `mmap` API.

- **Move**. Originator loses handle to data buffer. Corruption is not possible.

Copy semantics are the most restrictive but can be emulated using an interface with share semantics by write protecting the originating buffer for the duration of the I/O [18,33]. For the remainder of this chapter, I will focus on overhead reduction techniques in conjunction with an API with copy or share semantics.

2.4.2 Systems Based on VM Re-mapping

Data movement using VM mechanisms can be achieved by sharing VM mappings either for the duration of the IPC or for longer, or by trading VM mappings.

2.4.2.1 Sharing VM Mappings

Copy avoidance by sharing VM page mappings has been proposed in networking systems. One way is by using the *fbufs* abstraction [33,84]: with *fbufs*, two communicating entities share a buffer by mapping the same physical memory in both address spaces. Sharing with fbufs is safe and secure by raising the protection of the VM pages in the originating address space. An alternative, similar way to use sharing of VM mappings is

using Brustoloni's *Transient Copy-on-Write* [18] mechanism to avoid copying in the outgoing data path. Copy avoidance techniques based on temporal sharing of VM mappings do not naturally extend to file I/O [19]. The reason is that the buffer cache needs to maintain a valid a copy of the data, which requires a physical copy if the application modifies the data buffer. However, this is not a problem if file I/O is performed using the UNIX `mmap` interface.

2.4.2.2 Trading VM Mappings

Copy avoidance by trading VM mappings is a practical method for data movement when sharing between the communicating buffering systems is not required, as for example is the case between network buffers and the buffer cache in the kernel. Copy avoidance mechanisms based on trading of VM mappings do not interoperate with the UNIX read/write file I/O interface. Trading VM mappings between the buffer cache and the application is still possible but should be followed by invalidation of the transferred buffer cache blocks.

2.4.3 Systems Based on RDDP

Another way to avoid memory copies is to use NIC support for RDDP. This method, however, does not facilitate data sharing, which is possible with VM re-mapping techniques. The emergence of commercially available NICs offering user-level networking and RDMA capabilities by means of the VI networking protocol, motivated a number of new system designs taking advantage of these capabilities.

Block Storage. The benefits of using the VI protocol to perform block-level storage server access has been examined by Zhou and her colleagues [122]. They found that user-level storage access, avoidance of interrupts by use of polling, and reduction of the locking/synchronization cost contribute to high TPC-C transaction rates with a VI-based storage server. However, they do not compare their system to other alternatives such as iSCSI, FibreChannel, or optimized RPC-based block-level or file-level implementations.

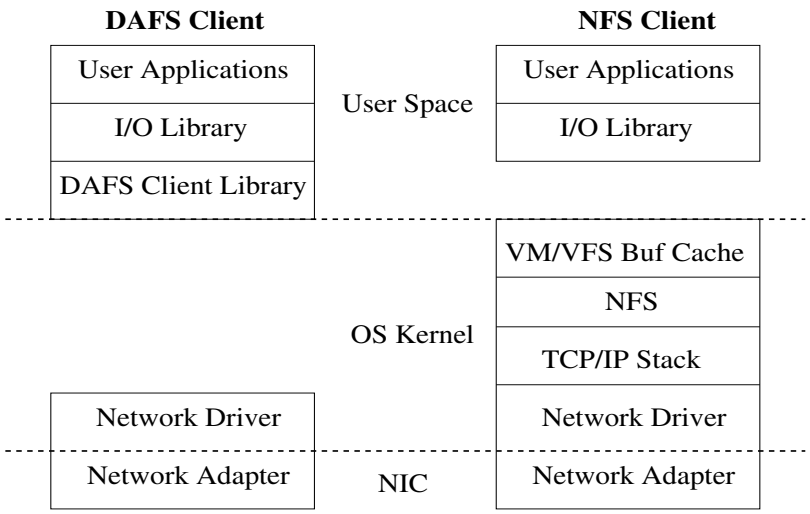

FIGURE 8. User-level vs. kernel-based client file system structure. DAFS takes advantage of user-level networking technology to enable a user-level file client. This differs from typical NFS implementations, which are kernel-based.

File Storage. The benefits of using VI for file-level storage have been targeted by systems such as DAFS [29,64] and NFS-RDMA [22]. The DAFS specification is based on NFS version 4, but departs from it in that network I/O data transfer is done by RDMA instead of being in-lined in the RPC for transfer sizes above a certain threshold. DAFS requires that clients decide whether RDMA is used for data transfer on a per-I/O basis. Clients also provide the remote memory pointers to the client application I/O buffers. Other advantages of the DAFS file API is the support for asynchronous I/O and for RDDP. DAFS is targeted towards user-level client implementations as shown in Figure 8. This is possible by taking advantage of the kernel bypass enabled by the user-level networking technology described in Section 2.3.1. The primary benefit of this approach is increased portability for client implementations and thus reduced dependence on particular OS vendors offering more powerful kernel APIs than others. My implementation of a DAFS server is described in detail in Section 6.3.3.

Just like DAFS, NFS-RDMA uses RDMA for long data transfers, but it performs the RDMA transfer within the RPC protocol, transparently to the NFS implementation. The advantage of this approach is that this RDMA-optimized RPC layer can be transparently usable by any RPC service besides NFS.

RDDP-RPC has never before been used in a network storage system. It has, however, been implemented in an in-kernel global shared memory system [23]. This dissertation offers the first evaluation of a network attached storage system using RDDP-RPC, in Chapter 3.

2.5 I/O Throughput and Response Time

Throughput and response time are standard I/O metrics used to assess performance in NAS systems. In this section I describe how CPU overhead affects these metrics.

Throughput is important for applications that can sustain several simultaneously outstanding transfers, either by having some knowledge of future accesses, or by involving a number of simultaneous synchronous activities, such as concurrent transactions in OLTP.

From Equation 1 on page 18 and with the per-byte component of overhead associated with memory copying eliminated using RDDP, overhead is dominated by its per-I/O component. In addition to host CPU overhead, the performance of network storage applications may also depend on the network link latency (L) and bandwidth ($BW_{network}$), and the NIC transfer rate (G^{-1}). Modern NIC architectures using DMA engines for transfers between the network link and host memory ensure that the NIC is not the bandwidth bottleneck for messages larger than a certain threshold, i.e., $G^{-1} > BW_{network}$.

The I/O throughput achievable with a stream of I/O requests, each of size m, can be limited either by the network or by the (client or server) CPU:

$$\text{Throughput}(m) = min\left\{BW_{network}, \frac{m}{o_{per\text{-}I/O}}\right\} \quad \textbf{(EQ 2)}$$

For large I/O blocks, even a low I/O request rate can saturate the network, and the throughput is determined by $BW_{network}$. For small I/O blocks, however, the CPU is more likely to become the resource limiting throughput. This is because the CPU is saturated processing RPCs at lower I/O rates than necessary to keep the NIC data transfer

engine fully utilized. It is therefore important to reduce the per-I/O overhead for small file accesses. A previous study found that file server throughput in NFS workloads modeled by SPECsfs is most sensitive to host CPU overhead [67].

Besides throughput, response time is also important in transactional-style network storage applications that perform short transfers and cannot hide network latency using read-ahead prefetching or write-behind policies. Such applications usually have unpredictable access patterns involving small file blocks or file attributes. Response time is the time needed to complete a remote file I/O request and comprises the transmission round-trip time on the network link, the NIC latencies, control and data transfer costs on the host I/O buses, and interrupt and scheduling costs in the case of remote procedure call-based I/O [110]. For a heavily loaded server, response time increases by the amount of queueing delays [67].

2.6 Effect of the OS Structure

Extensive research on system support to enable I/O-intensive applications to achieve performance close to the limits imposed by the hardware, suggests two key areas: Low overhead I/O protocols and the flexibility to customize I/O policies to the needs of applications. One way to achieve both is by supporting user-level access to I/O devices enabling user-level implementations of I/O protocols. *User-level networking* is an example of this approach. In this dissertation, I argue that the real key to high performance in I/O-intensive applications is *user-level file caching and network buffering*, both of which can be achieved without user-level access to NICs.

2.6.1 Flexibility and the Role of the API

Since the early 1980's, it was understood that a fixed set of OS abstractions with a fixed API as envisioned in the original UNIX system design [91] could not accommodate the needs of all applications [106]. Instead, applications need the flexibility to customize or add new OS abstractions and to specify the I/O policies that best suit their resource use. Microkernels such as Mach [1], component-based systems such as Pebble [39], extensible operating systems such as SPIN [13] and VINO [97], and more radical architectures such

as the Exokernel [53], aim at providing applications with control over policies. Despite their different approaches to extensibility, a common challenge these systems faced was the design of appropriate APIs. The primary goal in designing an API was to offer the necessary degree of flexibility required by I/O-intensive applications. Examples of particular interest are file system and virtual memory prefetching and replacement policies, as well as customizations of the network protocol stack. SPIN and VINO enable such optimizations by inserting extension code in policy points in the kernel. The Exokernel approach proposed safe multiplexing of hardware resources such as CPU cycles and TLB entries, disk blocks, and network packets, to user processes, which are expected to implement all OS abstractions in libraries. In this dissertation, I argue that the cost that all these systems paid to support flexible APIs is too high in terms of system complexity. A single API that offers the appropriate degree of flexibility is possible and is a superior solution.

A secondary goal of extensible systems is to enable rapid deployment of new operating system services or services that are too application-specific to be included in mainstream operating systems. One example is a transactional memory service, implemented as an extension to Mach [36] and SPIN [94]. Once developed and tested, however, such services can be included in the kernel in the form of dynamically loadable modules, a kernel extension mechanism already available and in use in a number of mainstream kernels [70]. A similar argument against the use of extensible operating systems, except for kernel debugging and development, was made by Druschel and his colleagues [32], who proposed focusing on designing appropriate kernel-level APIs. To support this argument, they designed IO-Lite, a unified buffering and caching system implemented in the kernel and exporting an API with move semantics [84]. IO-Lite, however, makes no special provisions for application-specific I/O policies and removes control of the data layout from applications.

Any API to a kernel abstraction should offer both *proactive* and *reactive* control over I/O policies. For example, in the case of a kernel file cache, a hint-based API such as `madvise` can inform the cache in advance (proactively) about prefetching and replacement policies. An additional (reactive) mechanism is required to propagate cache events

that require eager handling by the application, such as cache write-back activity. One such mechanism applicable to mainstream operating systems, is kernel upcalls to user-level handlers [26]. Small and Seltzer [102], however, found that upcalls may not be a cost-effective solution in many platforms. User-level caching of file and network data and a low-level kernel I/O API providing access to file and network socket abstractions is a solution that avoids these problems.

2.6.2 Support for Efficient Servers

User-level networking removes the kernel from the critical communication path and enables user-level implementations of low overhead networking protocols, such as Active Messages [114], as well as application-specific customization of popular protocols, such as TCP/IP. Integration with the network protocol yields significant performance benefits in the case of Cheetah [40], a Web server implemented in user space and supported by the Xok Exokernel. Another benefit of user-level networking is lower overhead and lower latency in responding to HTTP events. User-level networking is possible with user-level network interfaces [21,111,113]. Another way to achieve these benefits is by implementing the server in the kernel, as in the case of AFPA [51]. The disadvantage in both cases is losing the benefit of protection and fault isolation that is possible when the server and the network protocol are in separate address spaces.

Good Use of Existing APIs. User-level services such as the Flash Web server [85], which use existing operating system APIs, rely on the memory-mapped interface to files, non-blocking interfaces, and user-level caching of several structures to avoid inefficiencies when involving the OS. They cannot, however, customize the TCP/IP stack, except through parametrized socket interfaces.

TCP/IP offload. Another approach to networking overhead reduction is to offload the TCP/IP stack to the NIC. The host interface to a TCP-offload NIC can be the *queue pair* (QP) abstraction [20,111], which can be used to implement lightweight communication primitives, such as Active Messages [114]. A TCP-offload NIC, in addition, enables a net-

work I/O model that allows direct transfers between the network and application buffers, referred to as remote direct data placement (RDDP) [48]. A disadvantage of this approach, however, is that it makes it harder to customize the TCP/IP protocol. For example, application-specific optimizations such as *knowledge-based packet merging* [40] require interaction with the TCP-offload NIC.

Copy Avoidance. One of the major sources of host overhead is memory copying for data movement between software interfaces. Previous research proposed a combination of VM re-mapping techniques and NIC support to avoid memory copies. I/O-Lite [84] is a unified buffering and caching system that requires a new API with *move* semantics. Genie [18] tries to avoid copies in the network I/O path without changing the UNIX (POSIX) interface. Genie is interoperable with the `mmap` file API [19], but not with the `read/write` file API. Copy avoidance is also possible with RDDP support, either through remote direct memory access (RDMA) protocol, or with the RDDP-RPC mechanism discussed in Section 2.3.6.1. DAFS [29,64] is a network attached storage system optimized for user-level networking and RDMA. Another approach to avoid copies in the disk-network path is to use complex APIs such as `sendfile`. In this dissertation, I argue that a simple read/write API can be equally effective, despite the additional user-kernel boundary crossing. It also reduces overall system complexity. This is somewhat similar to the RISC vs. CISC argument [44].

Control over Caching Policy. The existing POSIX APIs do not provide full support for controlling all caching policies. Ideally, an I/O API should offer control over read-ahead, write-back and page replacement. The `mmap` interface implementations in common use hide caching policies from applications; some control, however, is possible with OS support, through *prefetch* and *release* system calls [76]. Another way to achieve prefetching with `mmap`, is to use `mincore` and memory touching by helper threads to avoid blocking the main process on page faults, a method used by the Flash Web server [85]. However, there is no mmap interface to control write-back. This is a serious limitation for applica-

Network I/O Mechanism	NAS System	Uses RDMA	Per-I/O Tag Advertisement
Header Splitting	NFS re-mapping	No	No
RDDP-RPC	NFS pre-posting	No	Yes
RDMA	NFS hybrid, DAFS	Yes	Yes
Optimistic RDMA	Optimistic DAFS	Yes	No

TABLE 3. Network I/O mechanisms and NAS systems evaluated in this dissertation. RDDP mechanisms target per-byte overhead. Optimistic RDMA combines RDDP and per-I/O overhead reduction.

tions such as Berkeley DB [81], which require control over write-back to implement write-ahead logging (WAL). To implement WAL, Berkeley DB has to resort to user-level caching and to use the read/write I/O interface. A problem with this approach is duplicate caching in the user and kernel file caches, as well as the lack of control over the kernel I/O policies. An alternative way to achieve user-level caching is with DAFS [64], a new network attached storage system that uses RDDP to bypass the OS. The benefits of using DAFS with Berkeley DB are exhibited in Chapter 3 and in a recent study [64]. DAFS is intended to run over user-level NICs. In this dissertation, however, I demonstrate that user-level caching is possible and equally effective without user-level NICs.

2.7 Summary of Systems Evaluated in this Dissertation

Table 3 summarizes the system prototypes evaluated and compared in this dissertation. In Chapter 3, I compare five network file systems. These are NFS, NFS pre-posting, NFS hybrid, DAFS, and NFS re-mapping. All NFS variants derive from the same implementation of NFS Version 3 [86]. In Chapter 4, I evaluate Optimistic DAFS and compare it to DAFS.

Chapter 3

Application Performance

This chapter focuses on the performance of applications that perform network file access using large I/Os. This is an important class of applications that includes streaming multimedia and certain database workloads, for example, those involving complex query processing (e.g., join operations). The performance of such applications depends on the balance between their computational and communication requirements, as analytically shown in Section 3.1. Improving the performance of applications that perform large I/Os depends critically on the per-byte overhead reduction achievable through copy avoidance with the methods described in Section 3.2. The potential for overhead reduction with each method is illustrated in the experimental evaluation of Section 3.3.

3.1 Effect of Communication Overhead on Application Performance

For the purpose of this study, I model an application as a filter that processes an incoming stream of records. In this model, the application is reading records from a file server over the network using large I/Os. The I/O block size is independent of the record size. I assume that either the application or the file system performs sufficient prefetching of I/O blocks to ensure that either the network link or the client CPU (or both) are saturated at peak performance. The server is practically never a bottleneck in such applications due to the large I/O size used. Performance is measured in terms of the throughput achieved (records/s or MB/s).

Parameter	Description
a	Application consumption of client CPU per MB of data (s/MB). The inverse of a is the application record processing rate.
G	Gap per byte (s/MB). The inverse of G is the peak network bandwidth for large messages.
o_{low}	Network file system overhead when using some form of an overhead reduction mechanism (e.g., RDDP)
o_{high}	Network file system overhead when incurring the cost of memory copying for data movement.

TABLE 4. Summary of model parameters.

The application is modeled by a single parameter, its CPU consumption per MB of data a s/MB, which is assumed to be constant over time. The CPU overhead of the network file system is o s/MB and the network can transfer data at a peak rate of $1/G$ MB/s[1] [4,28], as per the network model of Section 2.2. The application performance can be expressed as

$$min\left(\frac{1}{G}, \frac{1}{a+o}\right) \quad \text{MB/s} \qquad \textbf{(EQ 3)}$$

depending on whether the network link ($1/G$) or the client CPU ($1/(a+o)$) turns out to be the limiting resource.

In the analytical calculations that follow, I refer to two network file system setups with different overhead parameters o_{low} and o_{high} ($o_{low} < o_{high}$). One of the systems (o_{low}) is able to reduce its per-byte overhead using one of the mechanisms described in Chapter 2, whereas the other (o_{high}) incurs the overhead of memory copying for data movement. Figure 9 shows the client throughput, calculated analytically using Equation 3, as a function of the network file system overhead for three characteristic values of the application processing rate a: A heavily CPU-bound client (very large a, Figure 9(A)) achieves low throughput, practically independent of the network file system overhead. An I/O-bound

1. The discussion in this section assumes that the application performs I/O using large block sizes, for which the network can reach its peak asymptotic bandwidth. For the Myrinet network used in this dissertation, this is achievable for 4KB or larger blocks, as can be experimentally verified using a low overhead, raw communication benchmark.

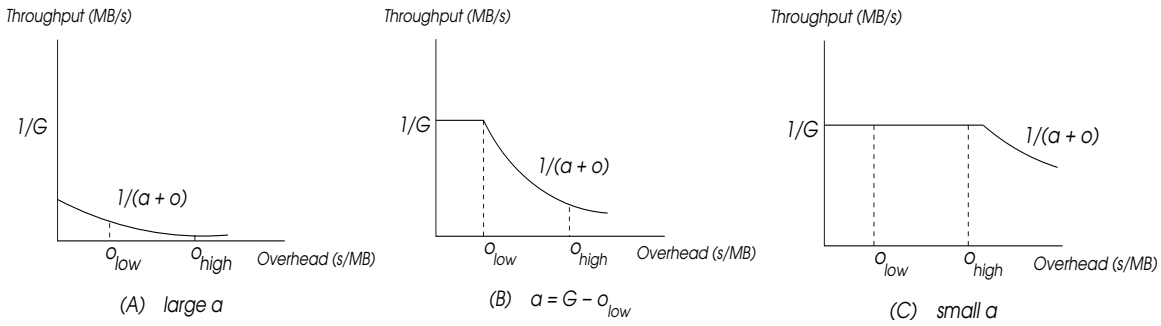

FIGURE 9. Effect of overhead under different application characteristics. (A) CPU-bound, (B) balanced CPU and I/O usage (a = G - o_{low}), and (C) I/O-bound.

client (small a, Figure 9(C)) achieves throughput close to the peak ($1/G$) offered by the network link for a range of (small to moderate) values of overhead. The effect of overhead becomes *maximal* for a client with balanced CPU and I/O usage (Figure 9(B)), as I will analytically show in the remainder of this section.

Using Equation 3, I obtain a qualitative result on the benefit of lowering the network file system overhead (o) for a particular application (a). I am interested in determining under which conditions the benefit of lower overhead becomes maximal. This benefit can be expressed as

$$\frac{\text{Throughput}_{low}}{\text{Throughput}_{high}} = \frac{min\left(\frac{1}{G}, \frac{1}{a + o_{low}}\right)}{min\left(\frac{1}{G}, \frac{1}{a + o_{high}}\right)} \quad \textbf{(EQ 4)}$$

The maximal benefit is achieved for a value of a that maximizes Throughput_{low} and simultaneously minimizes Throughput_{high}. From Equation 4, this value is a = G - o_{low}. This is the largest a (bounded above by the capacity of the CPU) for which the network link is saturated. The maximal such value is clearly the one that saturates the client CPU. Consequently, the maximal benefit is obtained when both CPU and network link resources are simultaneously saturated. For this value of a, the throughput ratio becomes

$$\left.\frac{\text{Throughput}_{low}}{\text{Throughput}_{high}}\right|_{a = G - o_{low}} = 1 + \frac{1}{G} \cdot (o_{high} \dot{-} o_{low}) \quad \textbf{(EQ 5)}$$

The maximal relative performance improvement is therefore linearly dependent on the amount of overhead reduction.

3.2 Low Overhead File I/O

File I/O in traditional operating systems is staged in the file system buffer cache, and memory copies are usually necessary to move data between network buffers, the file system cache, and application buffers. One way to avoid copies in network file systems is using *direct transfer file I/O*. This differs from what is commonly referred to as *direct file I/O* and associated with the **`O_DIRECT`** flag of the POSIX open system call. While direct file I/O implies a disabled file cache, which does not necessarily reduce memory copying, direct transfer file I/O additionally implies copy-free data movement between the storage device and user-space buffers.

In the following sections, I outline the use of two prominent network I/O models in reducing the overhead of network file I/O. These are (a) direct data transfer between the NIC and application space buffers using remote direct data placement (RDDP) (Section 2.3.6.1), and (b) copy avoidance using VM re-mapping (Section 2.4.2.2). Both network I/O models require new support at the NIC. Both can be implemented within a kernel-based file client. While a kernel implementation does not reduce system-call costs, there is no need to modify existing applications or even to re-link them if the kernel API is preserved. However, it does require new kernel support, which is a barrier to fast and reliable deployment. One of these mechanisms, remote direct memory access (RDMA), enables a user-level file client structure, as for example in the case of the direct access file system (DAFS) described in Section 2.4.3.

Direct transfer file I/O is easily achievable in direct-attached or SAN-based storage systems by programming the disk controller to DMA the requested data blocks directly to application buffers. Direct transfer file I/O in network file systems is more challenging, as general-purpose NICs are not aware of upper-level transport protocol packet formats and their semantics, and thus cannot usually be programmed to DMA the data payload directly into application buffers.

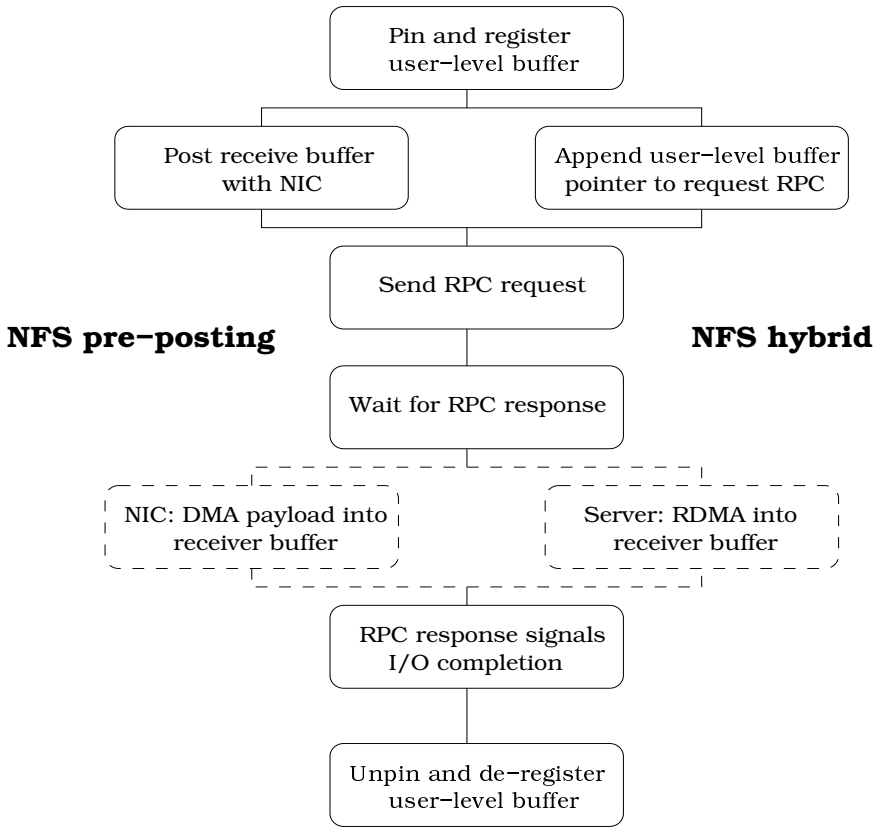

FIGURE 10. NFS client actions for a read request with either RDMA or RDDP-RPC.

3.2.1 Direct Transfer File I/O Using RDDP

One way to achieve direct transfer file I/O is with NIC support for RDMA or RDDP-RPC. File system clients must be modified so that their I/O operations bypass the buffer cache and propagate memory buffer information to the NIC, as shown Figure 10. A drawback of using an RDDP mechanism is the need to register and pin user-level buffers. In the case of kernel file clients, registration has to be performed by the kernel client, possibly on-the-fly and for each I/O, to be transparent to user-level applications. One problem with this requirement is the possibility that the kernel may be unable due to per-process resource limits to pin the user-level buffers required for the transfer. Besides introducing additional failure modes, the need for on-the-fly memory registration and de-registration introduces a performance penalty in the data transfer path.

3.2.1.1 Direct Transfer File I/O Using RDMA

In this method, tag advertisement is performed using RPC but data transfer is performed using RDMA. RDMA imposes no buffer size or alignment restrictions. Direct transfer file I/O based on RDMA is used in the recently proposed DAFS [29,64] and NFS-RDMA [22] systems. DAFS is a file access protocol that performs data transfers using server-initiated RDMA read and write operations, after explicitly advertising buffer addresses using RPC. In Sun's NFS-RDMA, buffer addresses are implicitly advertised by the RPC protocol. NFS-RDMA uses client- or server-initiated RDMA read operations issued from within the RPC protocol to pull data from remote buffers.

RDMA requires registration and pinning memory buffers on both the client and the file server. This is a disadvantage not found in RDDP-RPC, which requires registration and de-registration only on the receiving side (e.g., the client in the case of reads). An advantage of RDMA, however, is that the frequency of host interaction with the NIC can be reduced by caching registrations at the client and the server. With RDDP-RPC, NIC interaction is required on each I/O to pre-post application receive buffers.

In Section 3.3, I evaluate the performance of a kernel-based NFS-derivative system that performs data transfers using server-initiated RDMA. I refer to this system as *NFS hybrid* because I consider it to be a hybrid between NFS-RDMA and DAFS. The reason is that NFS hybrid performs data transfers using RDMA under the POSIX API, just like NFS-RDMA. However, just like DAFS, it modifies the NFS protocol to pass client buffer pointers to the server over the wire. My implementation of NFS hybrid for FreeBSD 4.6 over the Myrinet LANai is described in detail in Section 6.3.2.

3.2.1.2 Direct Transfer File I/O Using RDDP-RPC

The RDDP-RPC protocol enables the NIC to identify and separate NAS and RPC headers from the data payload and deposit the latter directly into the target buffer on the host using DMA. This dissertation presents the first evaluation of an NFS system using RDDP-RPC. In Section 3.3, I refer to this system as *NFS pre-posting*. My implementation of

RDDP-RPC for the Myrinet LANai and NFS pre-posting for FreeBSD 4.6 is described in detail in Section 6.2.

3.2.2 Direct Transfer File I/O using VM Re-mapping

Using an RDDP mechanism is one of several alternatives for improving access performance of network storage. In this section I consider a prominent competing approach as a basis for the empirical comparisons in Section 3.3. This approach uses VM re-mapping, which is described in Section 2.4.2, to reduce the overhead of data movement in a kernel-based NFS client. Like systems depending on RDDP, meaningful NFS enhancements of this sort also rely on new support in the NIC. Section 6.1 describes an implementation of a FreeBSD NFS client that uses VM re-mapping. This implementation relies on appropriate Myrinet LANai NIC firmware modifications for header splitting. The structure of this NFS client, which I refer to as *NFS re-mapping* in the experiments of Section 3.3, is shown in Figure 11.

In the remainder of this chapter, I present an experimental performance evaluation of (a) three systems based on RDDP mechanisms (DAFS, NFS hybrid, NFS pre-post-

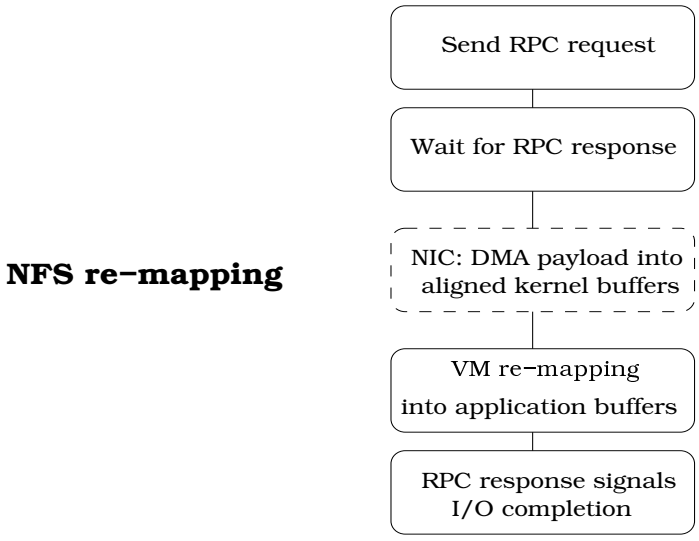

FIGURE 11. NFS client actions for a read request with VM re-mapping. The NIC splits the protocol headers before performing the DMA of the payload into aligned kernel buffers.

ing), (b) a system based on VM re-mapping (NFS re-mapping), and (c) standard NFS. All systems were evaluated on a high-speed networking infrastucture using commodity PCs.

3.3 Experimental Results

The experimental setup consists of a cluster of four PCs, each with a 1GHz Pentium III processor, 2GB SDRAM and the ServerWorks LE chipset. The PCs are connected via a 2Gb/s Myrinet switch over full-duplex ports. Each NIC has a 200MHz LANai9.2 network processor with 2MB of on-board SRAM in 66MHz/64-bit PCI slots. PCI bus throughput is measured at 450MB/s. All PCs run FreeBSD 4.6. Myrinet offers native support for the GM communication library [78]. GM is a user-level, event-driven networking library that supports messaging and remote memory read and write operations. The LANai drivers and firmware are based on the GM-2.0 alpha1 release featuring support for remote direct memory access get and put primitives. The VI library is based on the Myricom VI-GM 1.0 release. This is a host-based user-level library mapping VI operations to GM operations and used by the user-level DAFS client [64]. A kernel port of the VI library supports the DAFS server [65]. Ethernet emulation is implemented in the standard LANai GM-2.0 firmware and drivers and supports UDP and IP checksum offloading and interrupt coalescing. The Ethernet packet MTU is 9000 bytes. GM data transfers, however, are fragmented and reassembled by the LANai using a 4KB MTU.

The GM driver and firmware are modified as described in Section 3.2 for RDDP-RPC. NFS pre-posting and NFS hybrid are implemented by modifying the FreeBSD 4.6 kernel, as shown in Figure 2. NFS pre-posting uses the RDDP-RPC device interface. NFS hybrid uses GM put to perform server-initiated RDMA writes to client memory buffers. NFS re-mapping is implemented by modifying the FreeBSD 4.6 kernel as shown in Figure 11. The LANai firmware is modified to perform header-splitting and to DMA the payload in page-aligned 8KB network memory buffers.

Given the very low transmission error rates of Myrinet, I use UDP as a transport protocol to avoid the higher overhead of TCP. This configuration approximates the benefits of offloading TCP if it were supported by the NIC. Table 5 reports baseline network performance of the protocols used over the Myrinet network. These numbers are col-

lected using the `gm_allsize`, `pingpong` and `netperf` programs for GM, VI-GM and UDP/IP protocols, respectively.

Protocol	Roundtrip (us)	Bandwidth (MB/s)
GM	23	244
VI	23 (poll)	244
	53 (block)	244
UDP/Ethernet	80	166

TABLE 5. Baseline Myrinet Performance. One-byte roundtrip time.

3.3.1 Client Overhead

In this section, I present measurements of read throughput with a simple client and application performance with the Berkeley DB database [81].

3.3.1.1 Client Read Throughput

This experiment measures file read throughput with a simple client performing asynchronous read-ahead without any data processing. I compare DAFS to standard NFS and to three optimized NFS implementations: NFS pre-posting, NFS hybrid, and NFS remapping. The client reads data sequentially, using a varying block size, from a 1.5GB file warm in the server file cache. Read-ahead prefetching at the application level is done via the DAFS and POSIX aio APIs. NFS is mounted with the read-ahead parameter set to zero in all cases to avoid kernel prefetching using the NFS I/O daemons (nfsiod). UDP/IP is modified so that the NFS transfer size can match the application block size up to 512KB.

Figure 12 shows that for block sizes larger than 32KB DAFS can sustain read throughput of about 230 MB/s. As shown in Figure 13, it achieves this throughput consuming less than 15% of the client CPU for 64KB or larger blocks by offloading the transport to the NIC and by being able to avoid all memory copies. Per-I/O overhead is progressively better amortized, since the unit of data movement always matches the application block size. For small block sizes, DAFS achieves low per-I/O overhead by

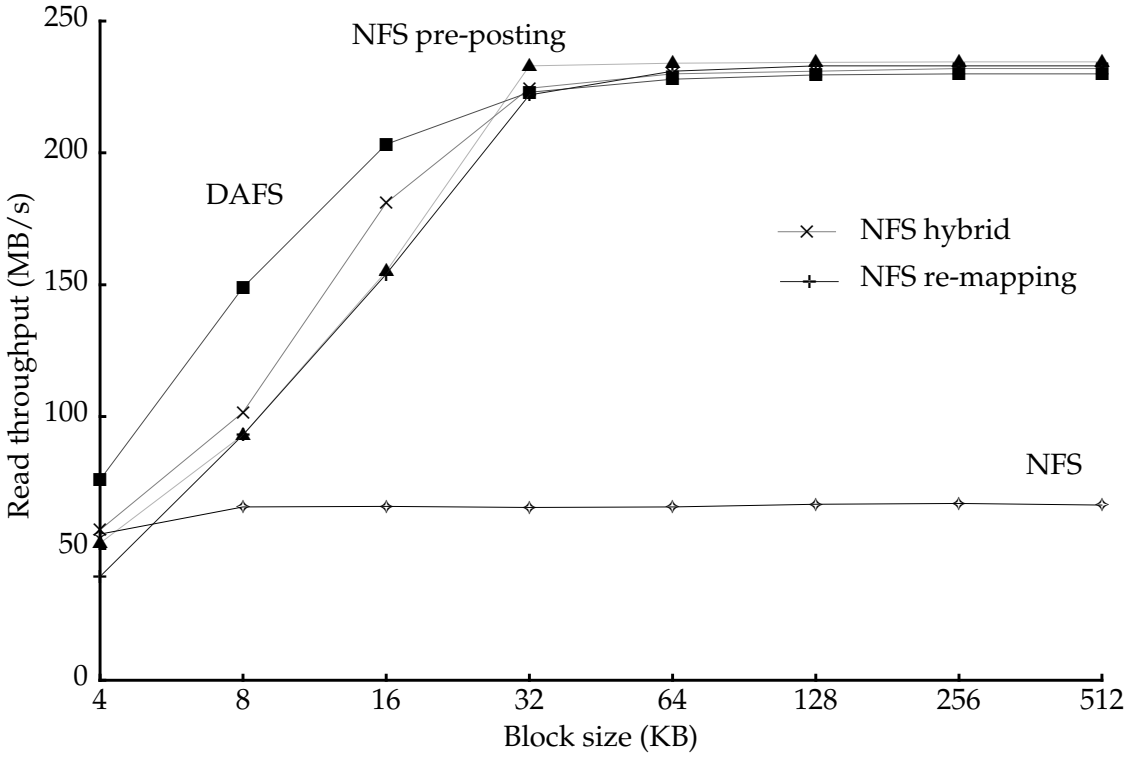

FIGURE 12. Client bandwidth performing read-ahead with variable application I/O block size.

using polling instead of interrupts. Similarly to DAFS, NFS hybrid sustains 230 MB/s for block sizes of 32KB or larger with CPU utilization dropping rapidly with increasing block size. However, even though both DAFS and NFS hybrid use RDMA, NFS hybrid uses more of the client CPU due to its higher per-RPC overhead. This is because NFS hybrid incurs the overhead of per-RPC interrupts whereas DAFS is able to avoid it by user-level polling. In addition, DAFS benefits from its integrated design that leads to shorter code paths compared to the general-purpose NFS implementation. Both DAFS and the NFS hybrid clients avoid registering application buffers with the NIC on each I/O by caching registrations.

NFS pre-posting sustains 235 MB/s for block sizes 32KB or larger, with 8KB IP fragments. It slightly outperforms systems using RDMA because the size of Ethernet packets (8KB) is twice the size of the 4KB GM fragments. The decline in the client CPU utilization is eventually limited for large block sizes as the total number of IP fragments is independent of the block size. NFS pre-posting performs transport-level processing

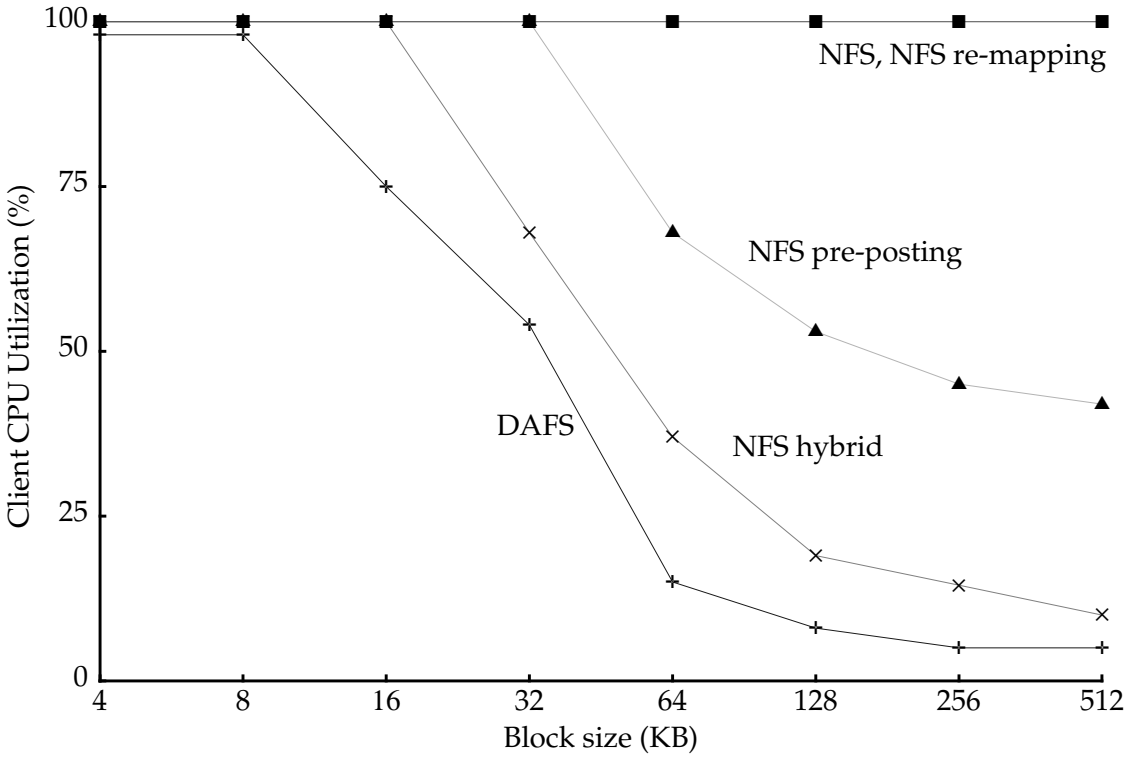

FIGURE 13. Client CPU utilization performing read-ahead with variable application block size. One way to look at these curves is as corresponding to *per-byte* (NFS), *per-page* (NFS re-mapping) and *per-I/O* (DAFS, NFS pre-posting, NFS hybrid) costs.

only on the last IP fragment of a UDP datagram. However, the LANai still notifies the host on each incoming IP fragment. Even with interrupt coalescing, this incurs a non-negligible host overhead due to IP fragmentation and reassembly. In addition, the NFS pre-posting client interacts with the NIC for pre-posting application receive buffers on each I/O.

NFS re-mapping sustains 235 MB/s for block sizes 32KB or larger, showing that VM re-mapping is an effective overhead reduction technique. Just like NFS pre-posting, data transfer with NFS re-mapping is performed in 8KB IP fragments. Unlike NFS pre-posting, however, NFS re-mapping performs transport-level processing of all IP fragments. The additional overhead of NFS re-mapping, compared to NFS pre-posting, is the UDP/IP processing of IP fragments and VM re-mapping costs. This additional overhead is reflected in the CPU utilization graph, showing a saturated client CPU for all NFS I/O block sizes[2]. An exact break-down of the NFS re-mapping overhead between transport

(UDP/IP) and VM re-mapping costs would require non-intrusive instruction profiling, which is not available on the Pentium platform. Statistical kernel profiling [69], however, shows that, contrary to earlier claims [108], VM re-mapping costs (invalidating TLB entries, adding new TLB entries) account for only about 10-20% of the client CPU utilization. The VM re-mapping overhead, however, could be higher in a symmetric multiprocessor architecture due to the higher cost of TLB shoot-down, which has to be broadcasted on the system bus.

Standard NFS achieves a maximum throughput of 65 MB/s, limited primarily by memory copying, which saturates the client CPU.

3.3.1.2 Berkeley DB Performing Asynchronous I/O

In this experiment, we[3] use Berkeley DB to show the effect of client CPU overhead in application performance. Berkeley DB [81] (DB) is an embedded database management system that provides recoverable, transaction-protected access to databases of key/data pairs. It is linked into the application address space and maintains its own user-level cache of recently accessed database pages. DB is modified to asynchronously prefetch database pages when it is possible to pre-compute a set of required pages.

In this experiment, an application uses DB to compute a simple equality join with 60KB records. The result of the join is a large list of keys, retrieved from the database file located on the server. DB pre-computes the list of required pages and performs read-ahead by maintaining a window of outstanding I/Os. To vary the computational requirements of the application, we increase the amount of data copied from the DB cache into the application buffer for each record, from one byte to 60KB, and report the application throughput in Figure 14. The throughput sustained by the application when there is little memory copying is close to the wire throughput for all systems except NFS re-mapping

2. I attempted to use the Alpha processor's low overhead hardware support for profiling, in a manner similar to its use in Chase et al. [23], with limited succcess. The problem was that, under FreeBSD, traps into PALcode, used for context switching, incorrectly appear to account for a large number of cycles, skewing results.

3. This experiment was performed in collaboration with Sasha Fedorova who was responsible for setting up Berkeley DB in each case.

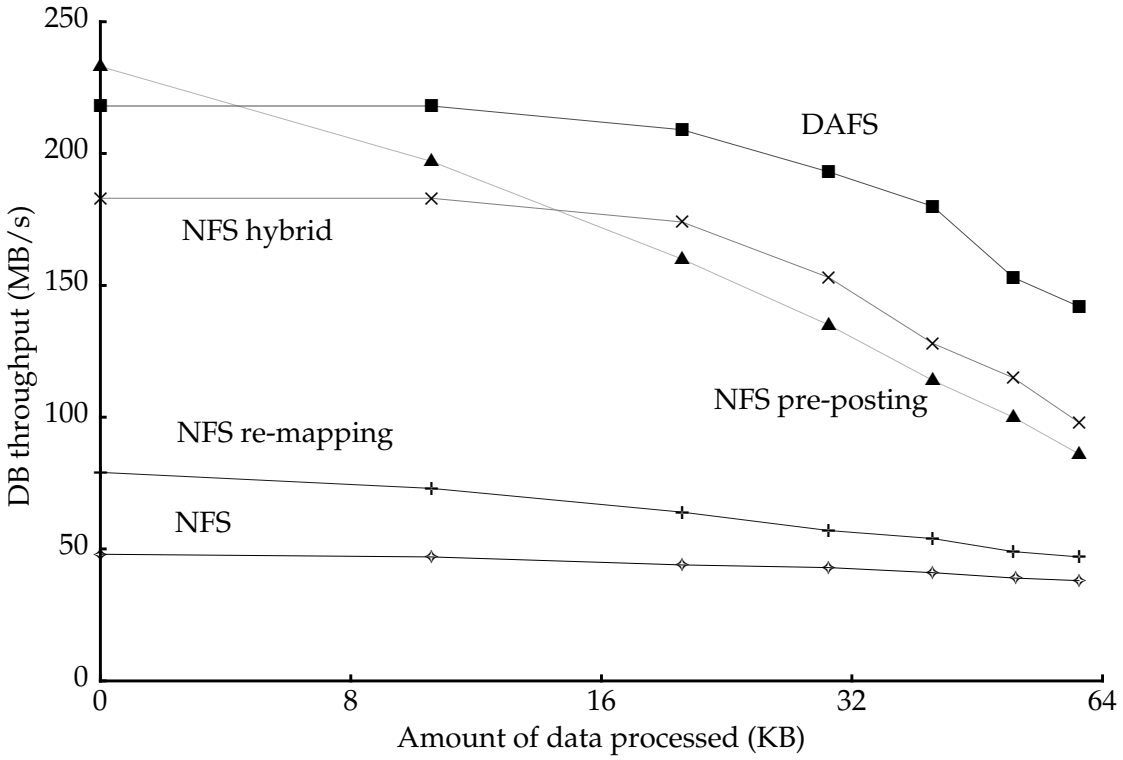

FIGURE 14. Berkeley DB performing asynchronous I/O using 64KB I/O blocks.

and standard NFS. NFS pre-posting performs slightly better than the other systems, as is also the case in Figure 12. As the amount of copying increases, performance becomes limited by the client CPU. Relative system performance is inversely proportional to each system's client CPU overhead for 64 KB network I/O transfers (Figure 13), as was analytically predicted in Equation 3.

An important point from Figure 14 is that the relative benefit of low overhead implementations diminishes when the application is compute-bound (right end of Figure 14). As application processing time per block decreases (from right to left in Figure 14), reducing I/O overhead yields progressively higher returns because the I/O overhead is a progressively larger share of total CPU time.

3.4 Summary

In this chapter, I focused on end-system overhead reduction in NAS applications. I showed that three network I/O mechanisms, RDMA, RDDP-RPC (pre-posting), and header-splitting with VM re-mapping, enable file access throughput that saturates a

2Gb/s network link when performing large I/Os on relatively slow, commodity PCs. In addition to raw file access performance, measurements of a throughput-intensive Berkeley DB workload show that, as predicted in Section 3.1, low-overhead implementations deliver the strongest benefit for balanced workloads, in which application processing saturates the CPU when I/O occurs at network speed.

Chapter 4

Server Performance

High-performance network attached storage servers are typically dedicated systems serving I/O requests from multiple clients. By not having to share the CPU with applications, NAS servers have more resources available for serving client I/O load. This usually means that it is either the client CPU or the network link that limits performance in the case of a single or small number of clients. For workloads involving many clients and small I/Os (e.g., 4KB), however, such as in departmental office and engineering applications and in remote memory paging [38], performance can be limited by the server CPU, due to the per-I/O control transfer and processing overhead of RPC. Even when RDDP is used for data transfer, existing systems use RPCs for buffer tag advertisement on a per-I/O basis, as described in Section 2.3.6.2. These RPCs contribute to per-I/O CPU overhead, reducing server throughput and increasing response time in workloads dominated by small I/Os. One way to address these problems is to use client-initiated RDMA, without wrapping the RDMA in an RPC to prepare the server on a per-I/O basis.

Client-initiated RDMA without per-I/O RPC, however, introduces a number of challenges: First, by not involving the server CPU on a per-I/O basis, the standard file access control mechanisms are bypassed. Since clients are, in general, expected to be mutually untrusted, new security mechanisms are necessary to enforce access control without RPCs. Second, since the remote memory buffers are not prepared prior to the

RDMA, there is a possibility of remote page faults or other exceptions induced by RDMA access. Third, using an RPC for each I/O enables file-level locking for the duration of the I/O. Client-initiated RDMA, however, changes the atomicity of file access, as memory hardware can only guarantee file access atomicity at the level of a memory word or cache line.

In Section 4.1, I discuss the security issues underlying the RDMA model, as well as the unique challenges of client-initiated RDMA without RPCs on a per-I/O basis. In Section 4.2, I introduce Optimistic RDMA, my extension to the RDMA model that addresses the transport-specific challenges described above. The remaining challenges are addressed in the design of Optimistic DAFS, my extension to DAFS described in Section 4.3.

4.1 RDMA Security

The three main concerns in RDMA are access control, authentication, and encryption, in order of increasing safety guarantees. I define these safety and security issues in more detail below:

- **Access Control**. Remote memory accesses are permitted only if these remote memory segments accessed have been explicitly exported by the remote node.
- **Authentication**. Data originates from a client whose identity can be verified.
- **Encryption**. Data cannot be observed in cleartext while in-transit on the network.

The authentication and encryption issues can be addressed in the transport protocol underlying the RDMA protocol. For example, IPsec [56] is one way of doing it when RDMA is layered over IP.

4.1.1 Access Control

Existing protocols have different ways of achieving access control. For example, the VI protocol associates exported segments and VI endpoints with a Process Tag (PTag) identi-

fier. Access is possible only if the Ptag of the memory region accessed matches the Ptag of the VI over which the access is attempted. This is equivalent to enabling an application to access all memory that is mapped on the address space of the application's process. With this mechanism, an RDMA operation succeeds if

- The memory region has been exported, which can be verified at the NIC, and
- The access is over the VI connection that has been associated with this memory region.

However, just like applications need to take additional care (e.g., use type-safe languages) to avoid wild pointer dereferences, this VI mechanism allows the possibility for accidental access to exported memory regions. The reason is that the VI protection model is too coarse, enabling access uniformly to all memory exported over a particular VI.

One way to enforce a more fine-grain model is to associate a *protection key* [21,49] (PKey) with *each* exported memory region. A PKey should be a large (128-160 bit) number drawn from a sparse key space in a way that avoids collisions and prevents exhaustive search. One possibility is to use message authentication codes (MAC) [57], computed over a quantity identifying the memory region, such as the tuple *(address, length)*. Although per-region PKeys are an improvement over per-VI PTags, neither PKeys nor PTags ensure authentication.

4.1.2 Authentication

Authentication involves proving that a client attempting an RDMA to a remote memory region was indeed granted access to that memory region. Associating a single public key PKey with a memory region does not guarantee authentication, since an adversary gaining access to this key can successfully access the remote memory region. Authentication can be achieved in two ways:

- **Authenticating all packets communicated over a network connection**. This should be at the network or transport layer and be transparent to the RDMA pro-

tocol. A secret key should be shared between client and server, just like in the case of IPsec AH-based authentication. Authenticating all packets is expected to have a higher cost than authenticating RDMA operations only.

- **Authenticating RDMA operations only**. A secret key (PKeyS) should be associated with each memory region and shared between the server and the client. These keys should be given to the client over a secure channel. The per-region key PKeyS is used to compute a keyed-hashed MAC (HMAC) over the tuple (*address, length, timestamp*); the timestamp guards against replays of RDMA operations. The HMAC is computed at the initiator of the RDMA operation, it is sent alongside with rest of the RDMA request, and re-computed and validated at the target of the RDMA operation. In addition to authentication, PKeyS provides per-region access control and thus supersedes PKey. The difference between PKeyS and the PKey is that PKeyS is secret whereas PKey is public and transferred on the wire. A complication of this mechanism is the need for a secure channel to hand out per-region PKeyS keys.

4.1.3 Encryption

Data encryption can be enforced via a secret key cryptography algorithm such as DES or triple DES (3DES), as in IPsec EPS-based encryption.

4.2 Optimistic RDMA

In this section, I introduce Optimistic RDMA (ORDMA), a novel network I/O mechanism that enables client-initiated RDMA without the need for an RPC on a per-I/O basis. The following design challenges must be addressed in an ORDMA mechanism:

Security. To avoid accidental corruption or malicious buffer access by mutually untrusted clients, ORDMA uses the cryptographic methods described in Sections 4.1.1, 4.1.2 and 4.1.3. Since the server is allowed to revoke access privileges to an exported memory segment, for example, when protecting or invalidating VM page translations, there is a need

to provide a mechanism to handle remote memory access faults. The next paragraph examines two possibilities for such a mechanism and describes the one adopted by ORDMA in more detail.

Handling remote memory access faults. Client-initiated RDMA may be faced with a number of exception conditions at the target NIC. For example, some of the targeted VM pages may no longer be resident in physical memory. In addition, targeted pages may be locked or protected. In the case of non-resident pages, one option is to enable the NIC to trigger a page-in disk I/O. However, this solution significantly increases the complexity of the NIC design and most importantly, it may not be supported by the OS. The ORDMA model enables clients to initiate RDMA that is guaranteed to succeed only if the target buffer is valid and exported by the server and is neither locked nor protected. If any of these conditions are not met, a recoverable access fault is signaled to the client by a network exception. After catching an ORDMA exception, a client handler may recover by retrying the access using an alternate access method, such as RPC.

Two important design choices in any ORDMA-based system are: (a) how a client finds references to server memory buffers, and (b) how a client handles exceptions due to failed ORDMAs. Section 4.3 describes the choices I have made in the Optimistic Direct Access File System. In the following section, I discuss the safety requirements for RDMA, existing approaches to ensure these requirements, and how ORDMA improves on them.

My implementation of Optimistic RDMA for FreeBSD 4.6 and the Myrinet NIC are described in detail in Section 6.4.1.

4.3 Optimistic DAFS

The Optimistic Direct Access File System is an extension of the DAFS [29,64] protocol. Just like DAFS, ODAFS can use RPCs for all file requests. In addition to RPC requests, ODAFS clients may issue ORDMAs to directly access exported data and metadata buffers in the server file cache.

ODAFS is based on the following key principles:

- Clients maintain a directory or cache of remote references to server memory. These directories can be built either eagerly when clients ask the server for memory references, or lazily when the server piggybacks memory references with each RPC response.

- Directory entries need not be eagerly invalidated when the server invalidates VM mappings for exported references[1]. Instead, invalid ORDMAs are caught at the server NIC, which throws exceptions reported to clients[2]. An important advantage of this consistency mechanism is that the server does not need to keep track of clients caching memory references.

- Clients are always prepared to catch an exception for each ORDMA operation. In such a case, the client issues an RPC to access the data.

Other important considerations for ODAFS clients are determining the size of the ORDMA directory, particularly in relation to the memory requirements for file data and attribute caching, and the replacement policies appropriate for maintaining the ORDMA directory. For the purposes of this study, I assume that the size of the ORDMA directory is small compared to the size of the data cache, and use the LRU replacement algorithm for ORDMA references. However, since ORDMA accesses are expected to be issued in response to client cache misses, a more appropriate strategy would be similar to the Multi-Queue algorithm for storage server caches [122].

My implementation of Optimistic DAFS for FreeBSD 4.6 and the Myrinet NIC is described in detail in Section 6.4.2.

1. Piggybacking invalidations in RPC responses may reduce ORDMA exceptions without requiring explicit invalidation messages. This method, however, will require that the server maintains state about which clients cache what remote memory references.

2. The cost of ORDMA exceptions, however, should be masked by the much higher latency of the expected disk I/O in the server.

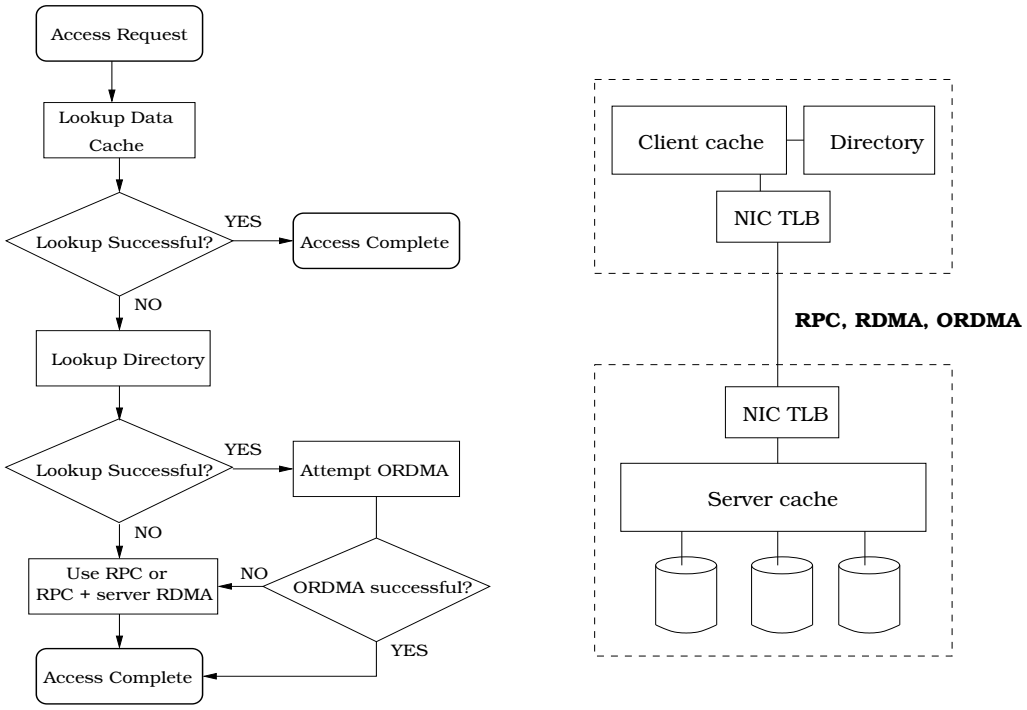

FIGURE 15. Optimistic DAFS client action flowchart (left) and system structure (right). A *directory* in the client is caching remote memory references to the server cache. Remote memory access is performed either via RPC, or via RPC followed by server-initiated RDMA, or via client-initiated ORDMA. Caching takes place at different layers: Client data cache, directory, and NIC TLB; Server data cache and NIC TLB.

4.3.1 Benefits and Limitations

ODAFS is targeted for workloads performing small I/Os. ODAFS is most beneficial with significant memory-to-memory I/O traffic, such as that caused by small files and attribute accesses, and high server cache hit rates. The benefit comes mainly from the low server CPU overhead of the ORDMA mechanism. However, there are a number of workload characteristics that limit the applicability of ORDMA, and consequently the effectiveness of ODAFS. These are:

- **Few remote memory accesses**, e.g., when client caching is effective in locally satisfying most file requests [77]. Note that this factor reduces the usefulness of any remote file access protocol.

- **Low ORDMA success rate**, i.e., low server cache hit rates. If many ORDMAs result in failure, ODAFS performance is similar to that of DAFS as the cost of ORDMA exceptions and subsequent RPCs is masked by the high latency of server disk I/O.

- **Many file accesses that cannot be satisfied via ORDMA**. This could be because the remote memory location of the target data may not be exportable. Examples are directory name lookups, which require significant processing on the server besides the actual data transfer.

- **Small read–write ratio**. Writes require the update of associated file state, such as time of last modification and file block status on the server, besides the actual data transfer. Append-mode writes are harder as they further require allocating disk blocks on the server, checking resource limits, and potentially serializing over concurrent appending accesses.

- **Low NIC TLB hit rates**. Satisfying TLB misses for a NIC on the I/O bus can be significantly more expensive than for a CPU TLB. In addition, network storage working sets can be very large and access patterns may not have enough locality to render NIC TLBs effective.

Mixing ORDMA- and RPC-based file access has implications on the atomicity of file I/O [112]. RPC-based file access guarantees that the entire I/O operation is atomic by locking the entire file for the duration of the I/O. However, ORDMA-based file access guarantees that at most one memory word is read or written atomically. By using both access methods, ODAFS effectively offers ORDMA's atomicity semantics. For UNIX file I/O semantics, client applications should explicitly lock files for the duration of I/O. In practice, however, this is not a problem for high-performance applications, such as databases, which explicitly lock files for the duration of their I/O operations.

4.4 Effect of Caching at Various System Levels

The efficiency and applicability of ODAFS on a workload depends on the function of various caching layers depicted in Figure 15. Caching efficiency is directly related to the access pattern and to the sharing of the particular workload as well as the size of the cache. High-performance application servers have caches of the order of 16-64GB. Standard clients, such as desktop and laptop PCs, have smaller memory caches (512MB-2GB). Storage server caches are usually of the order of 16-64GB, with a total storage capacity in the tens of TBs, as shown in Table 6.

System	Cache	Disk Space
Network Appliance FAS900c	12GB	48TB
EMC Symmetrix 8830	64GB	70TB
IBM ESS Shark 800T	64GB	27TB

TABLE 6. Characteristics of three high-end network storage systems (as of 2002/2003). The Network Appliance FAS900c is an NFS/DAFS file server. The EMC and IBM systems are block storage servers accessed using SCSI over FibreChannel.

Caches in various system layers are used to provide the buffers for staging data when performing read-ahead and write-behind, and to take advantage of data re-use, either due to locality in a client's reference stream, or due to sharing between multiple clients. Cache misses can be classified as *capacity* misses, due to being unable to fit the entire application working set, or *consistency* misses, due to invalidations triggered by conflicting access to shared data by other clients. Application access patterns can broadly be categorized as random or sequential. Random access patterns are common in on-line transaction processing (OLTP) and database workloads. In addition to uniformly distributed random patterns in such workloads, patterns with skewed frequency distributions due to locality of reference are common in applications such as Web servers. These patterns can be modeled by the Zipf distribution [123]. Sequential access patterns are common in scientific and decision-support applications. In case of sequential access patterns, caches typically initiate read-ahead.

Client cache. Limited client caching capacity and the possibility of sharing in multi-client workloads are expected to cause *capacity* and *consistency* misses and thus increase the frequency of remote access to server memory. In multi-client workloads with write activity, file updates necessitate invalidations of cached file state and force frequent remote memory access to data blocks and metadata (e.g., file attributes). Consistency callbacks invoked by the file server, trigger the following actions by clients:

- Invalidate and flush cached file state when recalling caching rights.
- Reload the cache in subsequent accesses. If caching rights cannot be re-acquired, this will lead to frequent remote I/O.
- Go to the server to open a file when lacking caching rights.

ODAFS can improve the first two actions by optimistically flushing and reloading the client cache. The third action, however, may dominate performance with many small files, unless directory lookups on the server are very efficient.

Server Cache. The server cache acts as an extension of the client cache. The network access time of the server cache is typically many orders of magnitude faster than disk access. Significant benefits are possible due to the increased cache capacity and the avoidance of disk I/O. Wong and Wilkes [120] and Zhou, Philbin and Li [122] considered alternative server cache replacement policies. They reported improved server cache hit rates with their proposed server cache management algorithms for a variety of workloads and trace-based simulations. High server cache hit rates make t_{server} the dominating factor in the mean I/O response time

$$t_{\text{mean}} = h_{\text{c}} \times t_{\text{c}} + h_{\text{server}} \times t_{\text{server}}$$

where h_{c} and h_{server} are the client and server cache hit rates, respectively, and t_{c} is the client cache access time, which is negligible compared to t_{server}. As I show in Section 4.5.1.2, the relative performance between DAFS and ODAFS in a latency-sensitive workload is

determined by the relative performance of their remote memory access primitives (RDMA vs. ORDMA), independent of the client cache hit ratio.

High cache server hit rates also put a strain on the server CPU, eventually limiting throughput. In Section 4.5.1.3, I show that server throughput depends on the overhead of the remote memory access mechanism in a throughput-sensitive workload dominated by small I/Os.

Client Directory. The remote memory reference directory is used in case of client data cache misses and, as such, it should be managed similarly to the server cache. This is a read-only cache.

NIC TLB. The server TLB should also be managed similarly to the directory cache. The Myrinet NIC TLB is currently managed using an LRU policy.

4.5 Experimental Results

In this section, I report experimental results using the setup described in Section 3.3. Table 5 on page 47 reports baseline network performance for the protocols used over the Myrinet network. In addition to the experimental setup described in Section 3.3, the GM driver and firmware were modified as described in Section 6.4.1 for ORDMA, except for protection keys, which are not yet supported in our implementation. In addition, the Optimistic DAFS client and kernel server were developed as described in Section 6.4.2.

4.5.1 Server I/O Throughput and Response time

In this section I present microbenchmark and PostMark results highlighting the properties of ORDMA and the upper bounds for performance improvements in ODAFS applications. In all cases, a file cache based on DAFS open delegations [2] is interposed between the application and the DAFS/ODAFS API. To avoid introducing platform-specific parameters, such as the cost of NIC memory registration and TLB misses, I ensure that RDMA is issued on pre-registered buffers and always hits in the NIC TLB. The cost

of a NIC TLB miss is about 9µs for ORDMA in my prototype. This penalty can be reduced in NICs that are integrated on the memory bus, or share a TLB with the host CPU [10].

4.5.1.1 Microbenchmarks

In this section I present measurements of I/O response time in reading a 4KB block from server memory using (a) in-line RPC read, that is, the data payload in-lined with the RPC response, (b) direct RPC read, that is, the data payload transferred by server-initiated RDMA write, and (c) client-initiated ORDMA read. The file cache is configured with a small number of data blocks but with a large number of headers that can retain remote memory references. In this microbenchmark, a simple application sequentially reads twice a 1GB file that was preloaded in the server cache, in increments of 4KB. The client cache is configured with a 4KB block size and did not contain any portion of the file prior to starting the experiment.

I/O mechanism	**Response Time (µs)**	
	in mem.	**in cache**
RPC in-line read	128	153
RPC direct read	144	144
ORDMA read	92	92

TABLE 7. I/O response time with 4KB block size. The measurements in the column labeled "in mem." extend up to the point where the server response touches in client memory. The column labeled "in cache" includes, in addition, the data copy into the client file cache. The RPC direct and ORDMA cases transfer data directly into the client file cache.

During the first pass, all I/O requests miss in the client cache, which, in response, initiates remote file accesses using either in-line or direct RPC. RPC responses carry remote memory references to file blocks on the server cache. During the second pass, I/Os still miss in the client cache. However, this time remote I/O may also be performed by ORDMA since the client cache managed to map the entire file on the server after having accessed it once during the first pass. Table 7 shows the I/O response time during the second pass using different network I/O mechanisms. RPC in-line involves a memory copy

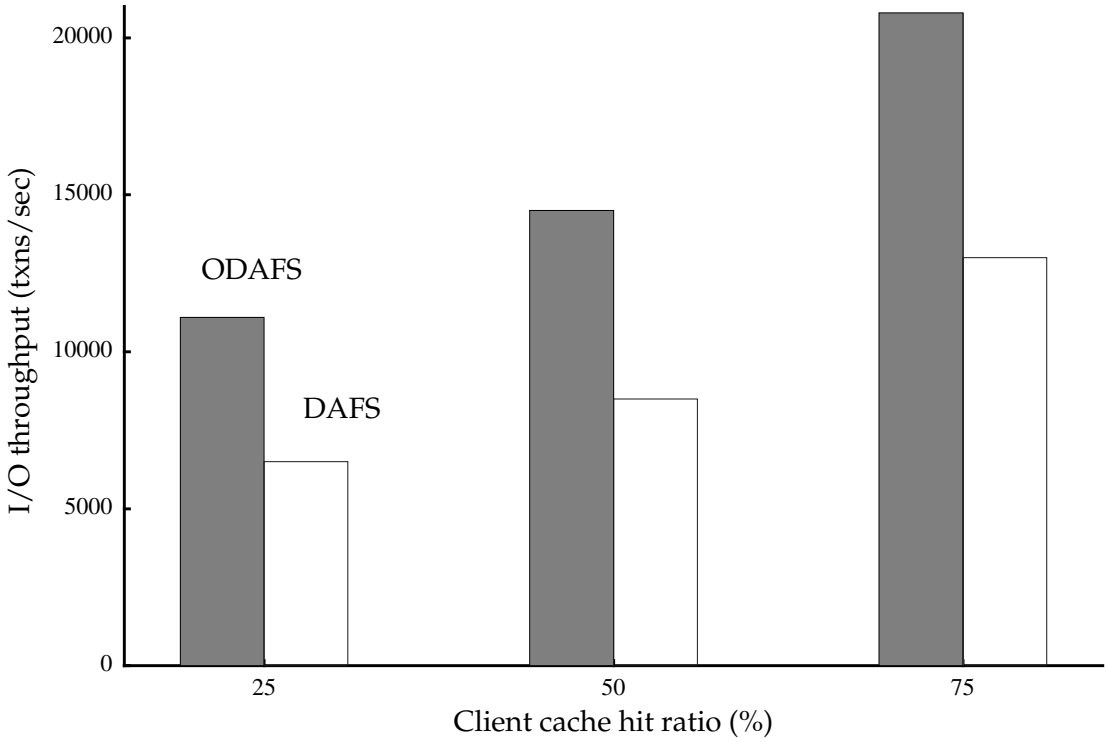

FIGURE 16. PostMark I/O throughput with ODAFS (dark bar) and DAFS (light bar). Single client with variable cache hit ratio.

in the client from the communication buffers to the file cache. ORDMA yields about 36% lower response time than direct RPC.

4.5.1.2 Effect of Client Caching

In this experiment, I model a file client accessing a set of small files synchronously over DAFS and ODAFS. The file set size exceeds the client cache size in all cases. I model such a latency-sensitive workload by configuring the PostMark [54] benchmark for read-only transactions without file creations or deletions. Each read I/O is preceded by a file open and followed by a file close operation. After the first open of a file, which grants the client an open delegation, each subsequent open or close for that file is satisfied locally. I use a 4KB average file size and configure the client cache with a 4KB block size. The client cache hit ratio determines the frequency of remote memory access. By varying the size of the client cache and keeping the file set size constant I progressively increase its hit ratio from 25% to 50% to 75%. I find that in all cases ODAFS yields about 34% higher through-

put than DAFS (Figure 16), reflecting the difference in response time between ORDMA and direct RPC. This is because, despite the benefit of client caching, overall performance is sensitive to the cost of remote memory accesses. The DAFS server CPU utilization drops from 30% to 25% to 20% as the client cache hit ratio improves. However, ODAFS uses no server CPU after it manages to collect remote memory references for the entire server cache, which occurs after the client has accessed each file at least once.

4.5.1.3 Server Throughput

In this experiment, I show the effect of per-I/O overhead on server throughput. I model a multi-client, throughput-intensive workload dominated by small I/Os by configuring two clients to sequentially read twice a 1GB file that was preloaded into the server cache. For reads larger than the cache block size, the cache starts internal read-ahead up to the size of the application request. To vary the unit of network I/O, I progressively increase the cache block size from 4KB to 64KB and measure server throughput for each cache block size during the second pass, as shown in Figure 18. With ODAFS, the two clients are able to saturate the server network link for all cache block sizes (except for 64KB due to a performance bug in GM *get*) without using the server CPU. DAFS yields lower server throughput for small I/O blocks, saturating the server CPU due to processing direct RPCs. For the smallest cache block size of 4KB for which the difference between DAFS and ODAFS is maximal, the DAFS server is primarily constrained by network interrupts. Switching to polling for all network events, DAFS throughput improves to about 170 MB/s reducing the performance improvement attainable from ODAFS to 32%.

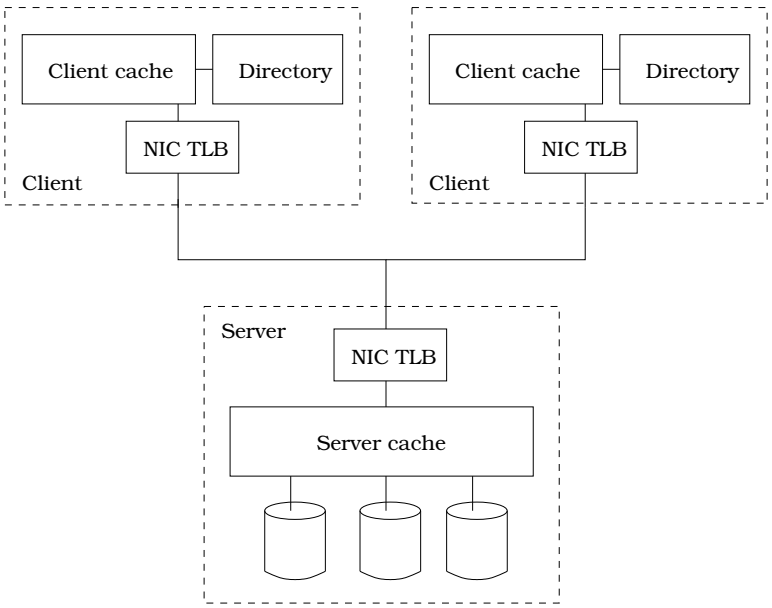

FIGURE 17. Server throughput experimental setup. Two clients sequentially reading a file preloaded on the server cache, twice. The application block size is a multiple of the cache block size, to trigger read-ahead by the cache. The measurements are taken during the second pass over the file. The client cache, server cache, and NIC TLB hit ratio is 0%, 100%, and 100%, respectively.

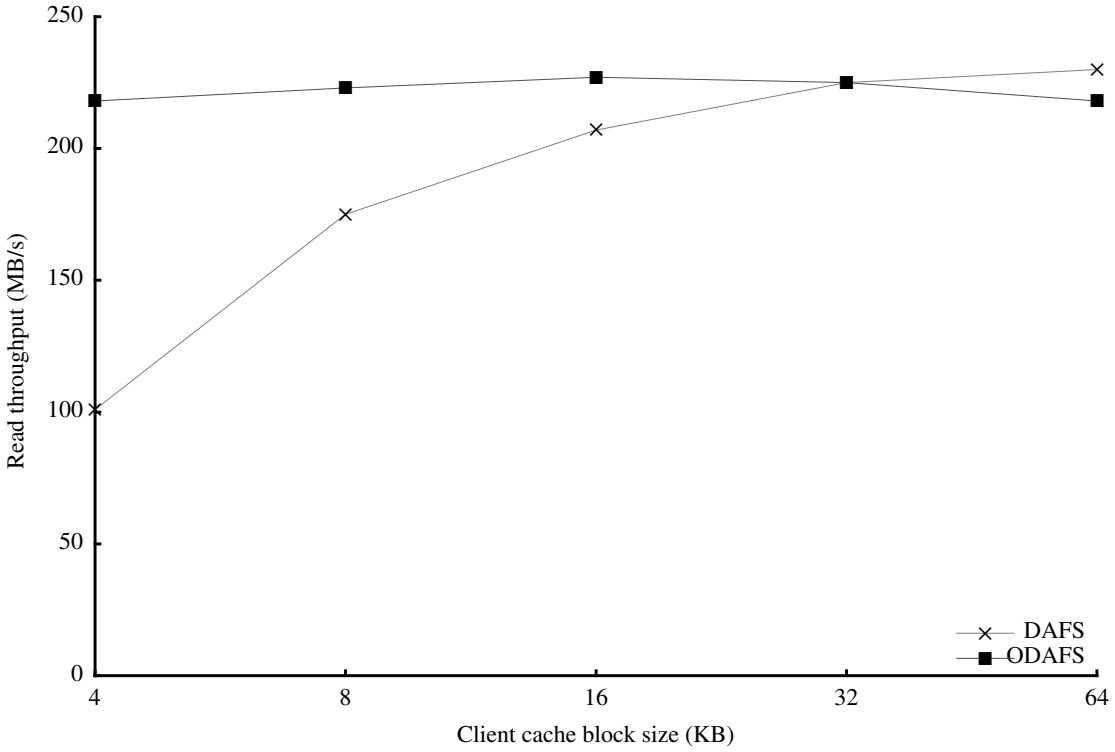

FIGURE 18. Server throughput. Two clients reading a large file using a large block size.

4.6 Summary

In this chapter, I focused on file access workloads involving multiple clients and small I/Os. Such workloads stress the file server and are sensitive to per-I/O CPU overhead. To ease the server CPU bottleneck, I propose a new network I/O mechanism, Optimistic RDMA, that aims to improve server throughput and response time, and Optimistic DAFS, my extension to the DAFS protocol that uses ORDMA. I evaluated ORDMA and ODAFS on a prototype implementation on state-of-the-art network hardware. Performance improvements in server throughput and response time with ORDMA/ODAFS range up to 32% and 36%, respectively, for small I/O transfers in my prototype.

Chapter 5

Effect of Operating System Structure

Extensive research on system support for enabling I/O-intensive applications to achieve performance close to the limits imposed by the hardware suggests two key areas: low overhead I/O protocols and the flexibility to customize I/O policies to the needs of applications. One way to achieve both is by supporting user-level access to I/O devices, which enables user-level implementations of I/O protocols. One example of this approach specific to network interface controllers (NICs) is *user-level networking*, which is described in Section 2.3.1 and Section 7.1.3. In this chapter, I argue that the real key to flexibility and high-performance in I/O-intensive applications is *user-level file caching and network buffering*, both of which can be achieved without user-level access to NICs.

Avoiding the need to support user-level networking carries important benefits for overall system design: First, a NIC exporting a privileged kernel interface is significantly simpler to design and implement than one exporting a user-level interface. Second, the kernel is re-instated as a global system resource controller and arbitrator. I describe a kernel API and NIC support that can be used to implement high-performance servers and demonstrate that its performance is comparable to the best of fully user-level and in-kernel servers.

Since the early 1980's it was understood that a fixed set of operating system abstractions and a single API cannot be sufficient for the needs of all applications [106].

The flexibility to be able to set application-specific OS policies and for a way to extend APIs is necessary. *Extensible operating systems* such as VINO [97] and SPIN [13] and radical OS architectures such as Exokernel [53] aimed for maximum flexibility in specifying application-specific policies. Besides performance, rapid deployment of new functionality is another reason that motivates the need for an extensible API.

5.1 Fallacies

Modular, component-based operating systems [1,39] with protection enforced by separate hardware address spaces have good software engineering properties, such as well-defined APIs between subsystem components and fault-containment within a component. Besides the goal of structuring an OS with good software engineering properties in mind, another goal is to achieve separation between policies and mechanisms [1], offering applications the flexibility to specify their own policies using the mechanisms exported by the operating system. Previous research in operating system structure resulted in two commonly-held beliefs:

- *Co-locating application servers and the operating system in the same address space is required to achieve the necessary degree of flexibility.* Flexibility is believed to be primarily the result of integration between software layers, as was shown by user-level Exokernel-based [53] or kernel-based servers [51].

- *Single address space implementations are more efficient because they avoid the cost of protection domain crossings*. Modular design and protection with hardware address spaces is still believed to be costly, although research in systems such as L4 [61] and Pebble [39] showed that the overhead of switching protection domains could be as low as 150 machine cycles. Besides the cost of the IPC to cross protection domains, modular systems tend to have longer memory footprints and thus, bad cache behavior [24]. Monolithic operating systems [112] are also believed to suffer from the cost of crossing the user-kernel system-call interface.

In this chapter, I show that:

- User space implementations of high-performance servers over a kernel API can be as efficient as fully user-level implementations, given sufficient network and OS support for RDDP, network protocol offload, and efficient event notification and handling. Their performance difference lies only on the cost of crossing the system-call protection boundary, which is rarely a dominant cost in I/O intensive applications.

- High performance can be achieved with a kernel I/O API offering just the I/O mechanisms, enabling applications to specify their own I/O policies.

5.2 Introduction

The need to reduce networking overhead in system-area networks in the early 1990's motivated a flurry of research on user-level networking protocols. Projects such as SHRIMP [17], Hamlyn [21], and U-Net [113] proposed user-level access to a network interface controller (NIC) as an approach that offered two primary benefits: First, it enabled host-based implementations of new, lightweight networking protocols with lower overhead compared to kernel-based TCP/IP protocol stacks. Second, for applications requiring use of the TCP/IP protocol stack, there is a potential for application-specific customization of user-level libraries. In recent years, the need for scalable and more manageable services, such as HTTP servers and network storage systems, leads to more I/O going through the network, linking overall I/O performance to the efficiency of the network subsystem and making network interface controllers (NICs) a key system I/O device. The large-scale deployment of high-speed (1 Gb/s and soon 10 Gb/s) Ethernet networks stimulated interest in the design of systems that offload TCP/IP to a new generation of NICs and can transfer data directly between the network and application buffers. Many such TCP-offload NICs are currently being designed to export user-level interfaces to host applications [73].

In this chapter, I show that a user-level interface to TCP-offload NICs is not necessary. A kernel host interface to the NIC in combination with an appropriately designed network and file I/O application programming interface (API) can provide the perfor-

mance and degree of flexibility needed by I/O-intensive applications. I show that this is possible with a simple asynchronous `read/write` network and file API, whose key features are direct data transfers between the I/O device and application buffers and efficient user-level event notification and handling. An important use of this API is to support user-level file caching and network buffering. User-level caching offers *full control over I/O policies*, an essential requirement for resource-intensive applications, such as databases [106]. Kernel caching through the standard `read/write` or `mmap` APIs is preferable *only* when efficient inter-process sharing through a file cache is needed, as shown in the comparison of Table 8. When the NIC support described in this dissertation is used for network data transfers to and from an in-kernel cache, it does not offer applications any additional control over their I/O policies.

User-level File Caching Mechanism	**Inter-process Sharing**	**Control over I/O Policy**	**Level of I/O Atomicity**	**Visibility of Updates**
`mmap`	Yes	Prefetching, Replacement	Memory word	Immediately (Shared Mappings)
`read/write` with RDDP	No	Full	Memory word *or* I/O operation, when wrapped in RPC	On read I/O

TABLE 8. Comparison between two user-level file caching mechanisms.

The network and file API described above enables a new operating system (OS) structure that blends ideas from traditional and novel/radical OS architectures: In accordance with standard OS principles [70], resource abstractions, such as files and network sockets, and global system policies, are implemented in the kernel. However, similarly in spirit to more radical system designs, such as the Exokernel [53], full control over application-specific I/O policies is possible with caching implemented in user space. This caching, however, is performed over the file or network socket abstractions rather than the raw hardware. The key advantages of this approach over Exokernel's are improved

security and safety, due to a larger common kernel code-base, support for global resource policies, and improved portability. To avoid conflicts between the user-level file and kernel VM policies in systems with a unified file/VM management, I propose that applications use a combination of memory locking and dynamic adjustment of cache size to the physical memory available to each application.

In summary, the main arguments presented in this chapter are: First, NICs exporting a user-level host interface pose significant implementation challenges. In contrast, NICs exporting a kernel host interface are simpler to implement. Second, with the transport protocol offloaded to the NIC, the key to lowering networking overhead is remote direct data placement (RDDP). A kernel interface to a NIC enables a simple implementation of RDDP based on *packet filters* and *tagged buffer pre-posting*. Unlike existing RDDP mechanisms which require new protocols, such as remote direct memory access (RDMA), my implementation interoperates with existing transport protocols. Third, this NIC interface can be used to implement a simple, kernel-based zero-copy I/O API. Applications over this API are as efficient as applications using a user-level implementation of the same API. The performance difference lies only in the user-kernel boundary crossing, which is not a dominant cost. In addition, applications using this kernel-based I/O API can be competitive with full in-kernel services. Fourth, kernel involvement is necessary in order to enforce global policies. Systems that advocate implementing all system policies in user-space libraries [53] do not offer a solution to the problem of implementing global policies without involving a privileged server. In addition, security and safety are other benefits of my proposed design: For example, common kernel-based networking code may be more robust against attacks than a multitude of user-level networking stacks. A benefit of user-level networking is better portability due to bypassing the OS. However, a kernel interface that is simple to implement should be rapidly incorporated into most mainstream operating systems.

The layout of this chapter is as follows: In Section 5.3, I compare the benefits of user-level networking to those of a kernel interface to a NIC offering TCP-offload and RDDP. In Section 5.4, I present simple kernel support that enables applications to dynam-

ically adjust to the per-process available physical memory. In Section 5.5, I outline the design of a file I/O API for flexibility. In Section 5.6, I propose mechanisms for efficient implementation of that API. In Section 5.7, I examine the role of the kernel in enforcing global policy and facilitating inter-process sharing through a file cache, and in Section 5.8, the portability and specialization benefits of user-level implementations.

In Section 5.9, I extend the analytical model developed in Section 3.1 in order to study the effect of the user-kernel protection boundary crossing in network-attached storage application performance. My motivation to use an analytical model backed with experimental measurements of key parameters, instead of a full experimental system comparison, is based on the fact that the latter depends significantly on the quality of the implementations and focuses on a particular point of the design space. An analytical model enables the examination of the entire design space, provides significant qualitative results, and points to the key parameters affecting system performance. Earlier studies of operating system structure, such as the classical work by Lauer and Needham [58], have also used analytical modeling to reason about system performance.

Finally, in Section 5.10, I present experimental results supporting the arguments outlined in this chapter.

5.3 User-level vs. Kernel Interface to a NIC

Traditional NICs are designed to be directly accessible by a single trusted entity, typically the kernel. User-level applications perform I/O by interacting with higher-level abstractions, such as files and network sockets. When these abstractions are implemented in the kernel, the latter ensures that NICs and other devices can be safely shared by multiple processes.

Devices exporting a user-level host interface, as shown in Figure 19 (a), cannot rely on the kernel for safe multiplexing between user-level processes. For this reason, a user-level NIC has to implement and export higher-level abstractions, such as virtual connection endpoints, and to maintain per-process and per-connection state. Such a NIC should be provisioned with sufficient resources to manage a large number of connec-

tions. In addition, the NIC should maintain a table to store translations of user-space virtual addresses to use in memory transactions on the system bus.

With switched Ethernet increasingly used as a low-latency, high-bandwidth (1-10Gb/s) system-area network, there is renewed interest in low overhead implementations of the TCP/IP protocol. One widely considered option is to offload TCP/IP to the NIC and enable direct transfers between the network and application buffers, as shown in Figure 19 (b). I argue that since the TCP/IP protocol is offloaded to the NIC, and thus, hard to customize, a user-level interface to such NICs is not necessary. A kernel interface can match two important benefits of a user-level networking protocol:

- **Copy avoidance** through direct transfers between the network and application address space buffers, possible with a remote direct data placement protocol [48].
- **Control over I/O policy**, which can be achieved by performing file caching and network buffering in application address space.

It also offers two other benefits of user-level networking:

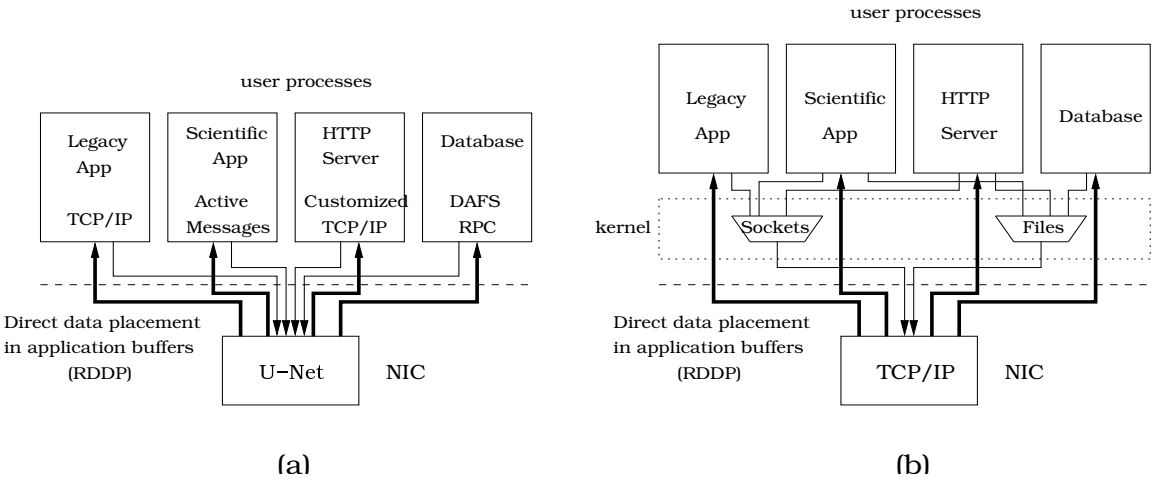

FIGURE 19. (a) User-level networking vs. (b) kernel API to a TCP-offload NIC. Both models enable RDDP, i.e., direct data transfers between the network and application address space. In user-level networking, it is the NIC that multiplexes the hardware between multiple processes.

- **Low per-I/O latency of interaction with the NIC**. A user-level access to the NIC bypasses the kernel. Crossing the user-kernel interface, however, need not be an expensive operation, as shown in Section 5.10.1.

- **Customization of the networking protocol**, which is achievable through a parametrized interface, e.g., socket options.

In practice, the only performance difference between a user-level and a kernel interface to a TCP-offload NIC is the user-kernel protection boundary crossing inherent with a kernel-based system-call API to higher-level services. In Section 5.9, I show that this performance difference does not significantly affect application performance in most network attached storage applications.

Direct data transfers between the network and application-space buffers through either a user-level or a kernel interface require that the applications buffers are *registered* with the NIC, i.e., pinned in physical memory and their VM translations known to the NIC for the duration of the I/O operation. Registration involves the kernel and can be performed either per-I/O or less frequently, by caching registrations. Pre-registering large amounts of memory is sometimes possible but results in underutilization of physical memory in multiprogramming workloads.

5.4 A Hybrid OS Structure

Mainstream operating systems, such as UNIX, implement abstractions such as files (FS), address spaces (VM), and network connections (Net) in the kernel, as shown in Figure 20 (a). Applications in these systems have limited control over I/O policies and often experience high communication overhead because of the need for data movement between software layers. Some extensible operating systems, such as VINO and SPIN, preserve this structure and address these issues by enabling safe execution of kernel extensions. An alternative to extensible operating systems, the Exokernel, structured as shown in Figure 20 (c), enables construction of all OS abstractions in user space.

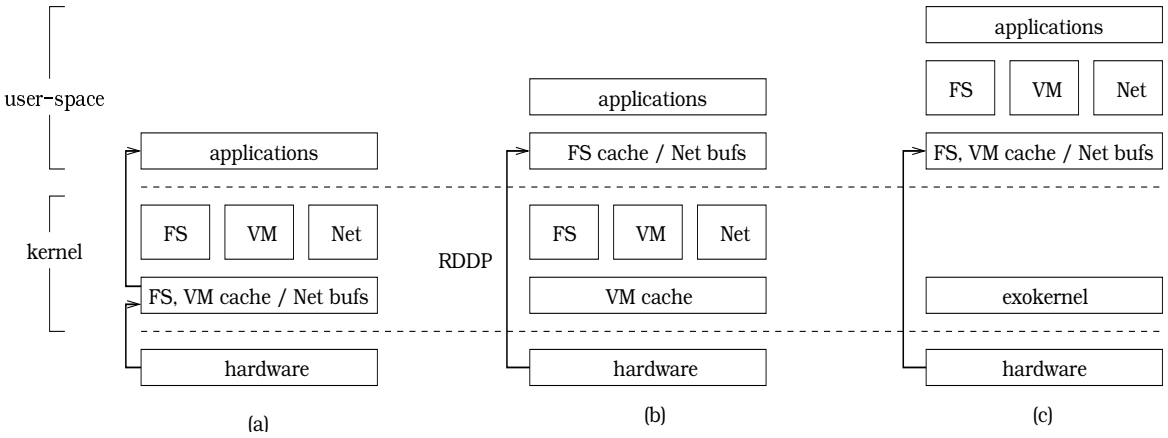

FIGURE 20. OS structures. (a) Mainstream (UNIX) and extensible systems (SPIN, VINO), (b) Hybrid (described in Section 5.4): File caching in the application address space, (c) Exokernel.

This dissertation describes a new OS structure, Hybrid OS, which is depicted in Figure 20 (b). Hybrid OS combines features from (a) and (c). Like mainstream OS structures, Hybrid OS places all OS abstractions in the kernel. Following trends towards scalable network storage, it assumes that applications access storage using file access over a network attached storage protocol, such as NFS. Hybrid OS exposes all file I/O policies to user-space applications by moving file caching to user space using RDDP for file data transfers. Network transfers are also performed using direct transfers between the network and user-space buffers. In this model, the kernel is still responsible for VM management, CPU scheduling, and most importantly, global resource allocation decisions.

The need for memory registration with the NIC requires involvement of the kernel, as described in Section 5.3. This is true for Hybrid OS, as well as for user-level networking systems. Some amount of registration caching is desirable to reduce the frequency of interaction with the NIC and to avoid VM page faults on the network data path. This implies that some fraction of the system physical memory would be pinned on behalf of the application and be unavailable to other system tasks. To ensure that there is enough unpinned physical memory to be shared by new system tasks, some amount of per-I/O registration is necessary. This is a cost that was not taken into account by user-level networking research [113], which assumed a limited buffer pool. Per-I/O registration opens up the possibility that user-level file accesses result in page faults in the

absence of any coordination between the application and the VM system. Some form of application control over virtual memory management over the region used by the user-level cache or adaptation to changing memory demands is necessary to avoid or reduce such page faults.

5.4.1 Kernel Support for User-level File Caching

A problem with user-level file caching on a pageable memory object is that kernel VM policies applied on the memory object may conflict with the file I/O policies used by the application. For example, the VM page replacement policy may differ from the file cache replacement policy, causing page faults. In addition, since the VM system is unaware that the user-level file cache is actually backed by a remote file, paging activity results in duplicate I/O. Some form of cooperation between the user-level cache, the kernel file system, and the VM module is necessary to avoid these problems. A solution offered by microkernels is to delegate responsibility for page replacement, page-in and page-out I/O to user-level processes [1,71]. Extensible systems, similarly, offer applications control over VM policies. In mainstream kernels, however, which are the focus of my work, applications normally cannot fully control the memory backing their user-level cache.

My goal is to enable the cache to use all available physical memory if there is no competition from other processes or fall back and use a smaller but fixed amount in case of memory pressure. The application should not be involved in each VM action since that would require support available only in microkernels and extensible systems. My scheme distinguishes between two system states: (a) a single application, and (b) multiple applications competing for memory. In the first case, the user-level cache is allowed to use all available physical memory. Part of the memory is pinned, with the rest registered with the NIC on a per-I/O basis. In the second case, the user-level cache shrinks to the part of physical memory that is pinned and cannot be reclaimed by the kernel. The kernel provides feedback to the application as to which is the case, prompting adjustment of the cache size. This ensures that the probability of a user memory access to the file cache resulting in a VM page fault is kept small. Previous research has explored the benefits of

kernel support that informs applications of their resource state [53,75], or even applications inferring that state without special support, for example, by probing the kernel [11].

5.5 Designing an I/O API for Flexibility

Based on the experience of implementing high-performance server applications [40,51,85] over prototype research operating systems [53,84], it is now well understood that a good I/O API should have the following characteristics:

Incorporate support for event-driven server structures. Event-driven servers have a number of advantages [83,118] over thread- or process-based servers and require non-blocking I/O APIs. Such APIs separate the issue of I/O operations from the notification and handling of their completion. Most implementations of the `read/write` I/O API, however, are blocking. The `mmap` API performs I/O by means of VM page faults, which cannot be performed in a non-blocking manner by most operating systems.

Expose the cost of I/O operations. The cost of I/O operations can be exposed by providing applications with information about system caching activity at various memory objects. Such information can be obtained, for example, with the `mmap/ mincore/mlock` interfaces [70].

Offer control over I/O policy. Control over I/O policies, such as file read-ahead, write-back, and page replacement, is important for resource-intensive applications, such as databases [106]. Some degree of control over I/O policies can be achieved in existing systems with APIs such as `madvise` or, in some cases, with compiler and OS support [76].

Support traditional data passing semantics. Data passing between the application and the kernel address space is traditionally performed via copy or share semantics [18]. With copy semantics, the application is free to modify a buffer after issuing an I/O on it without fear of corrupting the transfer. With share semantics, the application should not

modify the buffer until the I/O completes or otherwise risk corrupting the transfer. An alternative data passing method, however, uses move semantics [18], where buffers are exchanged between the application and the kernel. Move semantics is a natural way to avoid memory copying in the user-kernel boundary, but deprives applications of control of the allocation and layout of the I/O buffers.

An additional desirable property of an API is simplicity, so that it can be easily implemented. This dissertation proposes a non-blocking, non-caching `read/write` I/O API, incorporated with a versatile event notification mechanism. The basic components of this API are:

- File and network I/O operations: `read`, `write`
- Event handling: `poll`, `select`, `notify`. The blocking versions of `poll` and `select`, as well as `notify`, *require* kernel involvement. `Notify` is described in Section 5.6.2.

This API offers the desirable properties for the following reasons: First, being a non-blocking API, it can support event-driven services. Second, it exposes the cost of I/O by always involving interaction with the I/O device. However, this assumes that other hidden costs, such as VM page faults in user-level memory access, are avoided, as described in Section 5.4.1. Third, it offers full control over the I/O policy, for example when the I/O is staged in a buffer pool in the application's address space. Fourth, it supports standard share semantics. In addition to these properties, such an API is easy to implement and can be used to compose more complex operations. For example, the equivalent of a `sendfile`[1] interface can be composed by a file read and a network write. This API can easily be added to existing operating systems.

1. The `sendfile` API, which is offered by many UNIX variants such as Linux, FreeBSD, and Solaris, as well as Microsoft Windows, enables an application, such as a Web server, to transfer a disk file to a network socket without reading it into user space. Using scatter/gather NIC capabilities, it is possible to transfer data off the kernel buffer or VM cache without any memory copying. Properly implementing `sendfile`, however, can be tricky due to the interaction between many OS subsystems (file system, VM, network).

System		Host-NIC Interface	Protocol Requirements	Alignment Constraints
RDDP	RDMA	User/Kernel	RDMA	No
	Tagged pre-posting	Kernel	Framing	No
Payload alignment & VM re-mapping		Kernel	Framing	Yes

TABLE 9. Zero-copy input mechanisms.

5.6 Implementing the I/O API

An asynchronous, non-caching `read/write` I/O API requires NIC support for zero-copy data transfer, discussed in Section 5.6.1, and OS support for efficient event notification and handling, which is discussed in Section 5.6.2.

5.6.1 Zero-copy I/O

A key to reducing the host overhead of communication is the ability for zero-copy data transfers between the network and the application address space[2]. One challenge with zero-copy I/O is that it should be applicable to any data transfer protocol, such as HTTP, NFS, etc. Zero-copy I/O can be achieved with appropriate NIC and OS support, as described below for the input and output data paths:

Input. Zero-copy receives are possible for both *solicited* and *unsolicited* payloads.

For solicited payloads, such as read responses, zero-copy transfer is possible with NIC support for remote direct data placement. RDDP requires that either the application buffers are explicitly targeted by senders using packet header fields in a special remote memory access protocol (for example, RDMA), or that the application provides buffer

2. The RDDP mechanisms which will be described in Section 5.6.1.1 (RDDP-RPC) and Section 5.6.1.2 (RDMA) have been introduced in previous Chapters. They are briefly described again here for completeness.

location information to the receiving NIC prior to the I/O. In the latter case, a framing protocol such as SCTP [105], is necessary, so that the NIC can detect the boundaries of upper-level protocol headers and the data payload.

For unsolicited payloads, such as read requests and writes, zero-copy transfer is possible with NIC support for data payload alignment and subsequent data movement to application buffers, if necessary. The latter can be performed either by copying for small amounts of data, or by VM page re-mapping. In the latter case, either the system or the application must properly align their buffers to make sure VM re-mapping is possible. These zero-copy input mechanisms are summarized in Table 9.

Output. Zero-copy sends are possible by setting up device transfers directly from application buffers, using OS support to ensure that memory buffers are memory resident for the duration of I/O [18].

For the remainder of this chapter I assume TCP/IP offload on the NIC and focus on HTTP and NFS RPC as two examples of higher-level I/O protocols.

5.6.1.1 Tagged Pre-posting of Application Buffers

Pre-posting is the act of providing input buffers to the NIC prior to the I/O taking place. It is the I/O method used in the queue pair (QP) abstraction [20,111]. In untagged pre-posting, buffers are not associated with incoming messages. The NIC places an incoming message to any of the posted receive buffers. In tagged pre-posting, each receive buffer is associated with a particular incoming message. The tagged pre-posting method described here does not require a new RDDP protocol. It relies, however, on a framing protocol to mark the boundaries of protocol headers and the payload on the TCP/IP byte stream. The host interacts with the NIC to inform it where to place incoming data. The NIC is able to (a) identify higher-level protocol data formats, and (b) match a payload with its target buffer in application space and transfer data directly between the device and that buffer. It can achieve (a) and (b) with a combination of downloadable protocol-

specific packet filters and by preparing the NIC for RDDP prior to packet arrival by tagged pre-posting of receive buffers.

Packet Filtering. Packet filters based on the Berkeley Packet Filter [68] (BPF) can be installed by privileged users or by the kernel and are in use in network monitoring utilities such as `tcpdump`, in modular or extensible systems such as x-kernel [47], Exokernel [53], SPIN [13], VINO [97], and in systems such as Genie [18] and IO-Lite [84], which can benefit from early demultiplexing of network traffic. Packet filters are based on a simple language whose programs can easily be verified for safety. For example, packet filters describing the HTTP and NFS RPC packet formats can be simply composed and efficiently applied on an I/O data stream.

Tagged Pre-posting of Application Receive Buffers. Tagged pre-posting of receive buffers can be used to associate higher-level protocol data units, such as headers and data payloads, with specific buffers in the application address space [63]. The tag can be a protocol-specific identifier, such as the RPC transaction number in the case of NFS. Even when such an association is not necessary, for example when an application wants to receive protocol data units (e.g., HTTP requests) in anonymous buffers, a pre-posting API yields zero-copy implementations.

Pre-posting helps both in small message exchanges, such as requests for data in HTTP or NFS (untagged), and in large message transfers (tagged). The benefit in small message exchanges is in reducing the host overhead associated with transforming the untyped byte stream to the HTTP or NFS RPC record stream. Large message transfers benefit from the reduction in the data movement host overhead.

5.6.1.2 Remote Direct Memory Access

Using an RDMA protocol, the incoming messages carry the virtual memory addresses of the target application buffers. This requires a prior message exchange to communicate these addresses to the RDMA initiator. RDMA, however, does not require pre-posting of buffers by the receiver. RDMA can be used for user-level or kernel access to the NIC. The

latter is much easier to implement as there is no need for an address translation table on the NIC. This is because physical memory addresses can be used to target host buffers.

Hardware Event Notification	Event Handling
User-level (polling)	Running user-level thread
Kernel-based (interrupt or polling)	Awaken user-level thread (*poll*/*select*)
	Asynchronous upcall (UNIX signal)

TABLE 10. Implementation of user-level event processing. Notification by polling in the kernel and handling by user-level upcall is expected to yield the lower overhead/latency.

5.6.2 Efficient Event Notification and Handling

One of the keys to building efficient servers such as Cheetah [40,53], which is based on the Exokernel, or kernel servers such as those based on the Adaptive Fast Path Architecture [51] (AFPA), is a suitable event notification and handling mechanism. Hardware event notification can take place either by interrupts or by polling. Event handling can be performed either by a separate thread or by an upcall to a user-level procedure. In what follows, I focus on user-level event handling over a kernel I/O API.

5.6.2.1 Polling vs. Interrupts

Polling is the act of checking the status of I/O, typically by examining a memory location in kernel address space. Polling, however, is also possible in some systems by checking the device status mapped in application address space [21,111,113]. Polling has low overhead as it entails only a device interaction over the bus. In the case of a cache-coherent I/O bus, this may just be a cached memory access. A drawback of polling is that there may be a long delay between the time the event happens and the time it is handled after successful polling. An advantage, however, is that by having full control over when and at which points polling takes place, it is easy to account for event handling on behalf of the process to which the events are delivered.

Interrupts are an alternative mechanism for hardware event notification. An interrupt has its own stack and priority level. The priority of hardware interrupts is usually

higher than that of user-level processes. Interrupts typically result in low latency in handling events. They require, however, saving and restoring the context of the executing process, which increases the event notification overhead. They can occur at any time the system priority permits. Therefore, care is needed to account for the interrupt processing overhead on behalf of the process to which the events are delivered, rather than to the currently executing activity.

5.6.2.2 Threads vs. User-level Upcalls

Event handling in user-space can be performed either in the context of a thread or in a kernel upcall to a user-level procedure. Thread-based handling can be performed with existing APIs such as `select()` or `poll()`, which enable a user-level thread to check the status of an I/O or sleep waiting for an I/O condition. Upcall-based handling is modeled on the hardware interrupt paradigm and has two key advantages over threads: First, handling takes place with low latency: the delay between the occurrence of the event and its handling is limited by the interval until the next scheduling decision. Second, handling is event-driven and ensures progress by providing an execution context to process events. Upcalls obviate the need for application or user-level library threads otherwise needed for servicing I/O. User-level upcalls in mainstream UNIX-based kernels are currently used only for a small set of signals [70] communicating process error conditions and changes to process state. Kernel support for more general user-level upcalls enabling handling of I/O events is needed, as described in Section 5.6.2.3.

5.6.2.3 Kernel Support for Event-Driven Servers

Previous experience with high-performance server architectures suggests that the following two factors are key to building efficient user-level servers over a kernel I/O API:

- **Polling in the kernel for low overhead.** An adaptive scheme that switches to interrupts should be used when the presence of compute-bound processes does not allow for frequent polling.

- **Kernel upcalls to user-level handlers** to reduce latency and to ensure progress.

One way to achieve both requirements is to modify the kernel scheduler to poll for I/O events and schedule upcalls on a per-process basis. In this way, upcalls can be processed next time the process is scheduled, similarly to the signal delivery mechanism in UNIX systems. The Xok exokernel [53] uses a similar technique to combine efficient polling with low latency delivery and handling. Implementation on a mainstream kernel requires an API that enables a process to register upcall procedures to handle kernel-exported I/O events. The kernel support for user-level upcalls can be used to implement the VI `notify` API [111], which is not currently offered by any operating system. It can also be used with existing abstractions such as sockets and QP-IP [20]. For example, in the case of QP-IP, an upcall can be registered at the time a QP or CQ is created. An upcall is scheduled each time a new message is received or transmitted on that QP.

To further reduce the overhead and latency of event handling, it is possible to enable upcalls to user-level application handlers synchronously with event notification. This mechanism is similar to the one used in *scheduler activations* [9] and requires the use of schedulable kernel threads migrating into user-space and executing user-level procedures.

5.7 Benefits of the Kernel

Two key benefits of the kernel are global policy and secure, efficient data sharing.

5.7.1 Global Policy

The role of the kernel has traditionally been to safely and securely multiplex system hardware resources between competing processes. An overriding concern is to ensure fairness in resource use. Operating system research has long recognized the importance of separating policy from mechanism and looking into ways to enable applications to specify the I/O policies that best suit their needs. One important issue is how to combine per-process policies into the global system policy, particularly when applications have

conflicting needs. For example, many systems use global LRU for system-wide memory management.

Extensible systems offered different solutions in controlling global policy. Systems such as SPIN [13] and VINO [97] offered applications control over resource policy but retained global policy in the kernel. Other systems that advocated eliminating all OS abstractions and reducing the kernel to task of safely multiplexing hardware resources, such as the Exokernel [53], or pushing abstractions into intelligent devices [113] do not offer a solution on how to enforce global policies without going through the kernel.

5.7.2 Efficient Data Sharing

Inter-process data sharing is significantly simplified with the kernel on the I/O path. For example, the kernel cache, which can be mapped in the kernel address space, can be used for sharing. The existence of two alternative I/O paths, one that includes the cache and one that bypasses it, ensures that only processes that are willing to share files do so. Systems that do not use a kernel I/O API and always bypass the kernel have to rely on shared memory segments and pair-wise shared mappings to implement some form of sharing. An example of this approach is Maeda's user-level service framework [62], which is extended from user-level networking protocols to file systems. In addition to data sharing, code sharing is beneficial as well: For example, a common kernel-based networking code may be more robust against attacks than a multitude of user-level networking stacks.

5.8 Benefits of User-level Implementations

Two key benefits of user-level implementations are improved portability and the opportunity for specialization.

5.8.1 Portability

User-level implementations of OS services that only rely on device support (e.g., for user-level networking) are fundamentally more portable than user-level implementations that

rely on specific kernel support, such as an appropriate API[3]. This is a significant benefit for application developers who do not depend on features provided by certain operating systems and not offered by others. This observation draws from my experience with the user-level DAFS file client, which is easily portable between the FreeBSD, Linux and Solaris platforms with no modifications. It does, however, require porting of the NIC device driver.

5.8.2 Specialization

User-level implementations of OS services can usually be deployed in the form of user-level libraries that are linked with applications. This approach enables the possibility for specializing implementations of services and for placing these implementations in separate libraries. Applications can choose the appropriate library to use according to their specific needs. Specialized services are simpler to design, implement, debug and maintain and can be more tightly integrated than general-purpose services, leading to shorter code paths with improved performance. Previous research demonstrated that specialization can, in some cases, be used to produce in-kernel implementations of operating system functionality with performance comparable to user-level implementations [74,75,88,89].

These observations draw from my experience with user-level DAFS and kernel-based NFS implementations of file clients with similar functionality. The DAFS client offers a simple API customized for high-performance resource-intensive applications and does not offer a caching layer. The NFS client, however, is designed as a general-purpose service to suit the needs of a wide variety of applications. It offers a POSIX API and supports caching, asynchronous I/O, and optionally, direct data placement. Although the NFS file client can be used to provide the functionality for which the DAFS client is specialized (e.g., asynchronous I/O with direct data placement, as shown in Chapter 3), this comes at the cost of increased layering due to the higher code complexity, leading to

3. Needless to say, they are also more portable than kernel-based implementations of OS services.

longer code paths and high per-I/O overhead. The DAFS client implementation, on the other hand, has a more integrated design. It achieves compact, short code paths, offering the desired service with lower per-I/O overhead, as was shown in Section 3.3.1.

5.9 Analytical Performance Modeling

In this section, I examine the effect of the protection domain crossings inherent in a kernel-based file system implementation with a system-call API, on application performance. This study is based on the analytical model developed in Section 3.1 and assumes that (a) applications saturate the host CPU when performing I/O using relatively small block sizes (e.g., 4KB-64KB), and (b) the NIC supports transport protocol offload and an RDDP mechanism, eliminating the network protocol, memory copy and checksum overheads listed in Table 11. I assume that the NIC itself never becomes a performance bottleneck[4], except for adding a latency component in the I/O data path.

Source of Host Overhead	Type of Overhead without Offload/RDDP	Type of Overhead with Offload/RDDP
System Call	Per-I/O	Per-I/O
File System	Per-I/O	Per-I/O
Interrupts	Per-Packet	Per-I/O
Device Interaction	Per-Packet	Per-I/O
Network Protocol	Per-Packet	-
Memory Copying	Per-Byte	-
Checksums	Per-Byte	-

TABLE 11. Sources of file system client overhead and their type. Most of the sources of overhead are eliminated with a NIC offering transport protocol offload and RDDP.

With the assumption of transport protocol offload and the availability of an RDDP mechanism, the only host overheads remaining are the system-call, file system, device interaction, and interrupt processing costs, all incurred on a per-I/O basis. As will be shown later, the real difference between a user-level and a kernel-based implementation

4. This assumption does not always hold. For example, the NIC can limit I/O throughput when its processing speed significantly lags behind the host CPU [101,96] or when using very small message sizes.

of a file client is the cost of the system-call protection domain crossing. I will next examine the effect of this cost on I/O throughput and latency, based on analysis of the operations to issue and complete I/Os (Table 12).

API	Per-I/O Overhead		Total Per-I/O Overhead	
	Issue I/O	**Complete I/O**	**Best Case**	**Worst Case**
User-level	Interaction with the NIC	User-level NIC polling, *or* System call + *(possible)* interrupt	Two NIC interactions	System call + two NIC interactions + interrupt
Kernel-based	System call + interaction with the NIC	System-call + NIC polling + *(possible)* interrupt	Two system calls + two NIC interactions	Two system calls + two NIC interactions + interrupt

TABLE 12. Estimating the total per-I/O overhead for a user-level and for a kernel-based API. The file system overhead is the same in all cases and not shown here. The kernel-based API implementation incurs an additional overhead for implementing global policies (Section 5.7.1), which is not shown here. The "Best Case" is based on the assumption that polling for I/O completion is always successful. The NIC supports transport protocol offload and RDDP in both cases.

5.9.1 Throughput

For the purpose of this section, I am focusing on throughput-intensive applications that perform I/O using asynchronous operations at the speed of the host CPU. The reason that I focus on this domain is because the effect of additional overhead due to the system calls is expected to be maximal when the CPU is saturated. Table 12 estimates the per-I/O overhead, assuming that each I/O involves two operations, one to issue the I/O and another to complete it. The difference between the "Best Case" and "Worst Case" columns is that the latter assumes that the check for I/O completion is unsuccessful and the application has to block waiting on the I/O. Blocking on I/O typically involves a system call and an interrupt for notification.

Interrupts can be avoided in most cases in throughput-intensive workloads. This is possible either because polling in user or kernel space is always successful, as is the

case when small blocks are used on a fast network and with a sufficient amount of asynchronous I/Os, or because the cost of per-I/O interrupts is amortized, as is the case when large blocks are used. Looking at the "Best Case" column in Table 12, it becomes evident that the only difference between the user-level and the kernel-based API, assuming the same file system implementation, is the overhead of two system calls in each I/O.

Next, I will estimate the degradation in application performance caused by the system-call overhead in a kernel-based API, compared to the performance achievable with the same implementation and a user-level API. Using Equation 3 on page 40 and dropping the assumption of large block sizes, the throughput (in terms of number of I/O operations per second) achievable for a particular block size b can be expressed as follows:

$$\text{Throughput (b)} = min\left(\frac{1}{G}, \frac{1}{a + o(b)}\right) \qquad \frac{\text{I/O}}{\text{s}} \qquad \textbf{(EQ 6)}$$

For the purpose of this section, I consider block sizes b for which the host can achieve its peak throughput *1/G* when performing raw communication[5]. Note that the network throughput for very small blocks (e.g., of the order of one to a few tens of bytes) may be limited by the NIC to less than the asymptotic bandwidth $1/G$ of the network[6].

The application processing a and overhead *o(b)* are expressed in units of time-per-I/O operation. The overhead incurred by the host CPU per unit of time increases with decreasing block sizes. In the case of a kernel-based API, the overhead of the two system calls is an additional factor that contributes to the load of the host CPU. Next, I focus on the following two questions:

5. In the Myrinet (LANai9.2) network used in this dissertation, this is possible for 4KB or larger blocks as can be shown with a simple benchmark performing raw communication using Myrinet's native GM communication library.

6. This limit stems from the latency incurred in setting up the NIC DMA engines for sending or receiving very small messages. This limitation is captured in the *gap* (g) parameter of the LogGP model [4], described in Section 2.2.

- Under what conditions does the performance difference between the user-level and the kernel-based API become noticeable in throughput-intensive applications that have small computational requirements (i.e., small a)?

- What is the effect of the system-call overhead when the application involves significant data processing (i.e., large a) and is therefore heavily CPU-bound?

The performance degradation using a kernel-based API compared to a user-level API can be expressed as:

$$\text{Performance Degradation (\%)} = \frac{\text{Throughput}_{user} - \text{Thoughput}_{kernel}}{\text{Throughput}_{user}} \quad \textbf{(EQ 7)}$$

$$= \frac{min\left(\frac{1}{G}, \frac{1}{a + o(b)_{user}}\right) - min\left(\frac{1}{G}, \frac{1}{a + o(b)_{kernel}}\right)}{min\left(\frac{1}{G}, \frac{1}{a + o(b)_{user}}\right)}$$

$$= \frac{\frac{1}{a + o(b)_{user}} - \frac{1}{a + o(b)_{kernel}}}{\frac{1}{a + o(b)_{user}}}$$

$$= \frac{o(b)_{kernel} - o(b)_{user}}{a + o(b)_{kernel}}$$

$$= \frac{\delta}{a + o(b)_{kernel}}$$

Where the overhead parameter δ is defined as:

$$\delta = \text{User-kernel Crossing} + \text{Parameter Checking} + \text{Global Policy} \quad \textbf{(EQ 8)}$$

This parameter captures the fact that besides the overhead of the user-kernel crossing, a kernel-based API incurs the overhead of parameter checking, which may

involve the cost of traversing the page table in case of validating memory addresses, and that of implementing global policies (Section 5.7.1).

The effect of the system-call overhead becomes maximal when the host CPU is close to its saturation point. Equation 7 therefore becomes most interesting in the region of I/O block sizes for which the host is CPU-bound. This excludes block sizes for which performance may be limited (a) by the NIC, or (b) by the network link[7]. A close look at Equation 7 points to answers to the questions posed earlier:

- For throughput-intensive applications with small computational requirements (i.e., small a), the degradation depends on the ratio:

$$\frac{\delta}{\text{File System Overhead} + \text{Device Interaction Overhead} + \delta} \quad \textbf{(EQ 9)}$$

  The decomposition of the kernel API overhead to file system, device interaction, and user-kernel crossing costs, is based on the assumptions about NIC support outlined earlier, i.e., transport protocol offload and RDDP. It is also based on the fact that cost of interrupts can be made negligible by successful polling. With a 0.5µs system-call cost on the Pentium III and with the file system and device interaction overhead in the tens of µs, this ratio is expected to be roughly somewhere between 0-5%. In practice, the 5% upper bound should be a conservative estimate if one accounts for the additional non-negligible overhead of application-level I/O libraries such as Berkeley DB [81].

- For applications performing significant data processing (i.e., large a), there is practically negligible performance degradation since Equation 7 is no longer sensitive to the user-kernel crossing cost.

7. For the Myrinet network, the interesting range is between 4KB and 32KB. The upper limit is derived from experiments with most of the file system implementations described in this dissertation.

5.9.2 Response Time

The effect of the system-call interface in I/O latency is minimal. For example, the additional latency imposed by two system calls, one to issue the I/O and one to complete it, is about one microsecond on the 1GHz Pentium III platform used in my experiments. System-call measurements are outlined in Section 5.10.1. This is less than 1% of the latency to fetch a 4KB file block from server memory, as measured in Table 7 on page 64.

5.10 Experimental Results

In this section, I report experimental results using the setup described in Section 3.3. Table 5 on page 47 reports baseline network performance for the protocols used over the Myrinet network.

5.10.1 System Call Cost

To quantify the cost of crossing the user-kernel boundary, I measured the system-call cost on a Pentium III CPU clocked at 1GHz with FreeBSD 4.6. I found that a call to the `getpid()` system service takes about 500 cycles (500ns). This result is corrected for counter manipulation overhead. Earlier work measuring system-call invocation overhead under different operating systems on the Pentium found that low overhead access to system functionality is possible with mainstream operating systems such as BSD UNIX and Windows [24].

5.10.2 Cost of Memory Registration

In this section, I present measurements of the cost of memory registration and compare it to the system-call measurement of the previous section. Table 13 presents the results for registering a variable number of pages. For these measurements, I used the Giganet cLAN NIC instead of the Myrinet LANai NIC. The advantage of the cLAN over the LANai in this particular experiment is that the former features a fully hardware-based implementation of the VI protocol [111]. In the LANai implementation of the VI protocol, the VI Translation and Protection Table (TPT) is maintained by the host. Thus, the LANai

VI memory registration/de-registration operations are more expensive than they would be if the TPT was maintained by the NIC. For this reason, the cLAN offers lower cost of interaction with the NIC for registering memory over a commodity PCI bus, and is more representative of real implementations. Besides the interaction with the NIC, registration requires kernel involvement to wire page translations in physical memory. However, the cost of these operations does not depend on the NIC technology used. The observations in Table 13 agree with a previous performance study of the performance of VI-based user-level networking systems [104].

Number of Pages	Time (us)
1	15
2	17
3	19
10	28
100	165

TABLE 13. Cost of memory registration on the Giganet cLAN NIC with FreeBSD 4.6.

Section 5.4.1 argues that good utilization of physical memory requires some amount of per-I/O registration in any system offering remote direct data placement, independently of whether the NIC is accessible through a user-level interface or through a kernel-based API. Comparing the measurements of Table 13 with the system-call measurements of Section 5.10.1, it becomes clear that the cost of the user-kernel protection boundary crossing is a small fraction of the cost of registering memory with the NIC. Thus, a user-level networking system that does frequent per-I/O registrations is expected to perform similarly to a kernel-based system, despite the fact that the latter incurs the additional overhead of the system-call API.

5.10.3 File Access Performance

In this section, I show that a network file system client can be implemented equally efficiently in user or kernel address space. I compare a user-level DAFS client (which I refer to as *DAFS*) to an implementation emulating a kernel-based DAFS client (*kDAFS*).

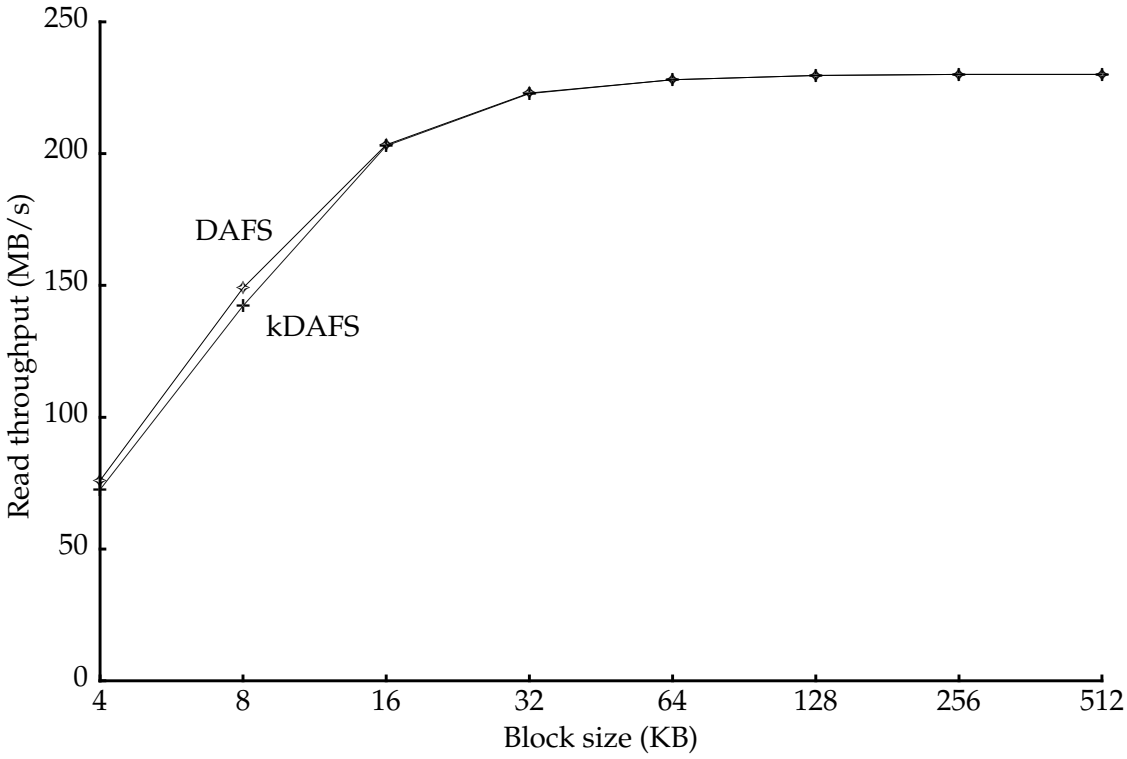

FIGURE 21. User-level versus emulated kernel-based DAFS client performance in streaming file access. The kernel structure is about 5% slower for small (4KB and 8KB) blocks and performs similar to the user-level structure for larger blocks.

The latter is derived by modifying the user-level DAFS client to perform a null system call at each function entry. A true kernel-based DAFS client is expected to be faster than kDAFS for two reasons: First, the only difference between a true kernel-based DAFS client and my approximation is that the former would execute code and access data in the kernel instead of a user-level address space. This, however, should not impede performance in any way. Actual kernel-based performance should be better due to privileges enjoyed by the kernel address space, such as wired page mappings. Second, the user-level approximation may in fact perform more than one system-call per I/O if there is a need to block waiting for I/O completion.

Figure 21 shows that DAFS and kDAFS are equally effective in reducing communication overhead in a simple streaming workload. The only performance difference occurs for small (4KB and 8KB) blocks, where the effect of the system-call overhead is maximal. Even in this case, however, kDAFS is within 5% of the performance of the fully

user-level client. Note that this experiment focuses on communication using I/O block sizes common in network storage applications. Distributed scientific and other applications typically use smaller I/O units and may find an advantage in the lower latency offered by user-level networking.

5.11 Summary

In this chapter, I have shown that an appropriately designed API can provide the flexibility required by I/O-intensive applications. Full control over I/O policies is possible with this API through the use of application-level file caching and network buffering. Even more importantly, this flexibility can be achieved without user-level access to NICs, significantly simplifying NIC design and implementation. There is a commonly-held belief, however, that kernel APIs hurt performance, due to the overhead of protection domain crossing. In this chapter, I show that this overhead is rarely a problem for I/O-intensive network attached storage applications. To demonstrate this, I use an analytical model backed by experimental measurements of key parameters. I show that performance in such applications is dominated by other overheads, such as those of the file system and the host interaction with the NIC, incurred on a per-I/O basis. Use of the kernel has other benefits, besides the protection and fault-isolation provided by the system-call boundary. The kernel is trusted by all applications, and thus, its involvement required to enforce global policies. In addition, involvement of the kernel on the I/O is key to efficient, secure data and code sharing between applications. User-level implementations, on the other hand, are more portable and can be better specialized to the needs of I/O-intensive applications.

Chapter 6

System Design and Implementation

This chapter describes the design and implementation of the systems evaluated in this dissertation and outlined in Table 14.

NAS System	Network I/O Mechanism	Uses RDMA	Per-I/O Tag Advertisement
NFS re-mapping	Header Splitting	No	No
NFS pre-posting	RDDP-RPC	No	Yes
NFS hybrid, DAFS	RDMA	Yes	Yes
Optimistic DAFS	Optimistic RDMA	Yes	No

TABLE 14. NAS systems and network I/O mechanisms evaluated in this dissertation. RDDP mechanisms target per-byte overhead. Optimistic RDMA combines RDDP and per-I/O overhead reduction.

All systems were implemented by modifying or extending the FreeBSD 4.6 operating system. FreeBSD is a derivative of the Berkeley Software Distribution (BSD) release 4.4 [70], a variant of the UNIX [91] operating system. FreeBSD 4.6 features a unified VM and file system cache. All NFS variants described in Sections 6.1 (NFS re-mapping), 6.2 (NFS pre-posting) and 6.3.2 (NFS hybrid), were implemented by modifying the standard FreeBSD NFS client and server implementation. The DAFS kernel server is a new kernel module that initially required changes in the FreeBSD buffer cache subsystem, as described in Section 6.3.3.2. It was later modified to directly use the VM system interface,

as described in Section 6.3.3.3, avoiding the need for any core kernel modifications. The only kernel support required by user-level code, such as the DAFS and ODAFS clients, is a driver for the Myrinet network interface controller.

6.1 NFS Re-mapping

This section describes the NFS re-mapping server design and implementation. This implementation differs from the one presented in the 2002 Usenix study [64] in that it is based on the Myrinet NIC instead of the Alteon Tigon II. The advantage of this implementation is that it allows a more fair comparison between the VM re-mapping and RDMA I/O models on the same network hardware. In addition, the Myrinet network and LANai9.2 NIC remove many of the shortcomings of the older Alteon/Tigon-II hardware, such as its higher NIC latency (132µs vs. 80µs one-byte UDP packet roundtrip time) and lower link speed (120MB/s vs. 244MB/s).

The most general form of copy avoidance for file services uses *header splitting* and *page flipping*, variants of which have been used with TCP/IP protocols for more than a decade [19,23,108]. To illustrate, we[1] briefly describe FreeBSD enhancements to extend copy avoidance to *read* and *write* operations in NFS. Most NFS implementations send data directly from the kernel file cache without making a copy, so a client initiating a *write* and a server responding to a *read* can avoid copies. We focus on the case of a client receiving a *read* response containing a block of data to be placed in the file cache. The key challenge is to arrange for the NIC to deposit the data payload---the file block---page-aligned in one or more physical page frames. These pages may then be inserted into the file cache by reference, rather than by copying. It is then straightforward to deliver the data to a user process by remapping pages rather than by a physical copy, but only if the user's buffers are page-grained and suitably aligned. This also assumes that the file system block size is an integral multiple of the page size.

1. Part of the text describing the NFS nocopy VM re-mapping techniques (Section 6.1) is adapted from the Usenix 2002 study [64]. The text describing the NFS-nocopy system in that paper was contributed by Jeff Chase.

To do this, the NIC first strips off any transport headers and the NFS header from each message and places the data in a separate page-aligned buffer (*header splitting*). Note that if the network MTU is smaller than the hardware page size, then the transfer of a page of data is spread across multiple packets, which can arrive at the receiver out-of-order and/or interspersed with packets from other flows. In order to pack the data contiguously into pages, the NIC must do significant protocol processing for NFS and its transport to decode the incoming packets. NFS complicates this processing with variable-length headers.

I modified the part of the Myrinet LANai adapter firmware emulating Ethernet communication to perform header splitting for NFS *read response* messages. This is sufficient to implement a zero-copy NFS client. Our modifications apply only when the transport is UDP/IP and the network is configured for Jumbo Frames, which allow NFS to exchange data in units of pages. To allow larger block sizes, we altered IP fragmentation code in the kernel to avoid splitting page buffers across fragments of large UDP packets. The RPC layer was modified to trade VM mappings of *mbuf* pages with their destination buffer in the file cache. Finally, buffer cache VM pages are re-mapped copy-on-write (COW) into application buffers[2]. This allows zero-copy data exchange with NFS block-transfer sizes up to 32KB. Large NFS transfer sizes can reduce overhead for bulk data transfer by limiting the number of trips through the NFS protocol stack; this also reduces transport overheads on networks that allow large packets.

While this is not a general solution, it allows an assessment of the performance potential of optimizing a kernel-based file system rather than adopting a direct-access user-level file system architecture like DAFS. It also approximates the performance achievable with a kernel-based DAFS client, or an NFS implementation over VI or some other RDMA-capable network interface. As a practical matter, the RDMA approach embraced in DAFS is a more promising alternative to low-overhead NFS. Note that page flipping NFS is much more difficult over TCP, because the NIC must buffer and reassem-

2. Another possibility would be to trade VM mappings between file cache and application buffers, but that would require invalidating the buffer cache page mappings.

ble the TCP stream to locate NFS headers appearing at arbitrary offsets in the stream. This is possible in NICs implementing a TCP offload engine, but impractical in simpler NICs such as the Myrinet LANai.

Header splitting without NIC support is in some cases possible using a software technique called *header patching* [19]. This technique, however, is specific to the UDP protocol, requires extra protocol negotiation between the endpoints, and requires agreement of a fixed header size for all protocols that benefit.

6.2 NFS Pre-posting

This section describes the NFS client implementation that uses RDDP-RPC supported by the Myrinet LANai and FreeBSD 4.6. The client was modified as shown in Figure 10 on page 43. In this implementation, I use the RPC transaction numbers as buffer tags. A tag is associated with an application buffer at the time when the latter is pre-posted by the receiving host, prior to sending the RPC request. Buffer tags are implicitly advertised in the context of the RPC protocol message exchange. RDDP-RPC imposes no buffer size or alignment restrictions on application buffers. Pre-posting of receive buffers (or pre-posting, for short) has previously been used in a kernel-resident RPC-based global shared memory service [6].

The implementation of an RDDP-RPC-based kernel client offering direct transfer file I/O requires a device interface that communicates the following information to the NIC:

- A description of the user memory buffer, including the physical address pointing to the buffer, where data coming from the network is to be directly placed.

- A description of the request including the RPC transaction number and the type of request, enabling the NIC to recognize the data payload in the RPC response.

This scheme requires simple modifications in the vnode layer of existing network file clients to avoid the user/kernel copy, pin the user-level buffer in physical memory and give the NIC the description of the user-level buffer rather than a pointer to an inter-

mediate buffer cache location. Both synchronous and asynchronous file I/O over an NFS client offering such support enjoys zero-copy, uncached data transfer.

One drawback of this scheme is that the NIC needs to be able to parse transport and application-level headers to understand RPC responses, which raises security and safety issues. These issues can be addressed by requiring supervisor privileges to program the NIC. Another drawback is that by bypassing the buffer cache, which abstracts the device layer, the file client is no longer part of the device-independent part of the kernel. Since not all NICs are expected to support an RDDP-RPC API, the file client depends on the availability of a device-specific API. However, making NIC-assisted direct transfer file I/O an option of the `mount` command is expected to work well in practice.

6.3 Systems using RDMA

This section describes my DAFS server and NFS-hybrid client and server implementations [63,64,65] for the FreeBSD 4.6 kernel. My DAFS server implementation has been deployed and extensively used in a variety of academic and industrial environments. Duke University, Rutgers University, Network Appliance Inc., and EMC Inc., are some of the primary external users of this software.

6.3.1 NIC Requirements and Support for RDMA

This section describes the basic characteristics of the Virtual Interface [111] (VI) protocol, used in the implementation of the DAFS server. These are:

- *Direct application or kernel access to the NIC.* This is possible by mapping parts of the I/O bus space belonging to the NIC onto the process' or the kernel's address space. Memory accesses to this space by the kernel or the process are translated by the VM hardware into I/O bus transactions that are subsequently claimed by the NIC.

- *Memory registration with the NIC.* The NIC includes a memory management unit to enable it to translate virtual memory addresses to physical bus addresses to use

when setting up DMAs transfers. The host is responsible for registering VM translations with the NIC prior to their use.

- *Efficient, asynchronous event notification and delivery mechanism.* VI supports the *completion group* (CG) abstraction that simplifies the task of simultaneously polling a large set of connections by aggregating their event notification and delivery into a single structure. The CG is similar to the UNIX `select` or `poll` mechanism, only more efficient by virtue of being directly supported by the hardware rather than implemented in the kernel.

6.3.2 NFS Hybrid

This section describes the NFS client and server implementation that use RDMA over the LANai NIC and FreeBSD 4.6. The NFS client API is unchanged, but the NFS client is modified as shown in Figure 10 on page 43. The NFS wire protocol is extended to support new RPC procedures that enable remote memory pointer exchange between client and server. The server is modified to issue RDMAs transferring data between the server buffer cache and client application buffers in response to appropriate client RPC requests. Both the NFS client and the server perform caching of buffer registrations to reduce the frequency of interaction with the NIC.

6.3.3 DAFS

In this section I describe the DAFS client and server evaluated in this dissertation. The DAFS server is described in more detail as I personally designed and implemented this system.

6.3.3.1 Client

The prototype user-level DAFS client used in this dissertation was designed by Jeff Chase and Richard Kisley. It was originally implemented by Kisley and subsequently jointly maintained and extended by the Duke and Harvard teams. This DAFS client was used in the 2002 Usenix study [64].

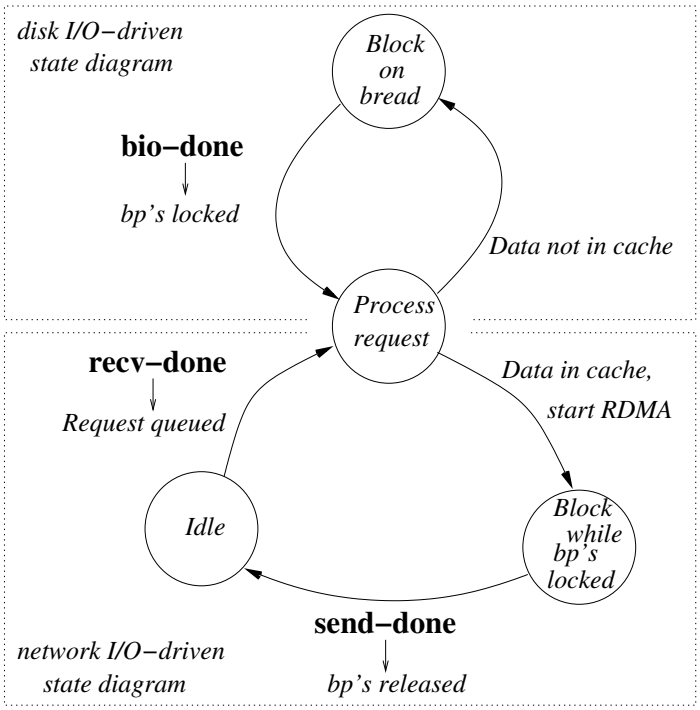

FIGURE 22. Event-Driven DAFS Server. Blocking is possible with existing interfaces.

6.3.3.2 Original Server Design and Implementation

This section describes the original DAFS server design and implementation using existing FreeBSD kernel interfaces with minor kernel modifications. The prototype DAFS kernel server follows the event-driven state transition diagram of Figure 22. Events are shown in boldface letters. The arrow under an event points to the action taken when the event occurs. The main events triggering state transitions are *recv-done* (a client-initiated transfer is done), *send-done* (a server-initiated transfer is done) and *bio-done* (a block I/O request from disk is done). Important design characteristics of the DAFS server in the current implementation are:

- The server uses the buffer cache interface to perform disk I/O (i.e. `bread()`, `bwrite()`, etc.). This is a zero-copy interface that can be used to lock buffers (pages and their mappings) for the duration of an RDMA transfer. RDMA transfers take place directly to or from the buffer cache.

- RDMA transfers are initiated in the context of RPC handlers but proceed asynchronously. It is possible that an RDMA completes long after the RPC that initiated it has exited. Buffers involved in RDMA need to remain locked for the duration of the transfer. RDMA completion event handlers unlock those buffers and send an RPC reply if needed.

- The kernel buffer cache manager is modified to register/de-register buffer mappings with the NIC on-the-fly, as physical pages are added or removed from buffers. This ensures that the NIC never takes translation miss faults and pages are wired only for the duration of the RDMA.

Each of the network and disk events has a corresponding event handler that executes in the context of a kernel thread.

- *recv-done* is raised by the NIC and triggers processing of an incoming RPC request. For example, in the case of read or write operations, the handler may initiate block I/O with the file system using `bread()`. After data is locked in buffers (hereafter referred to as *bp*'s) in the buffer cache, RDMA is initiated and the *bp*'s remain locked for the duration of the transfer.

- *send-done* is raised by the NIC to notify completion of a server-initiated (read or write) RDMA operation. The handler releases locks (using `brelse()`) on *bp*'s involved in the transfer and sends an RPC response.

- *bio-done* is raised by the disk controller and wakes up a thread that was blocking on disk I/O previously initiated by `bread()`. This event is currently handled by the kernel buffer cache manager in `biodone()`.

The server forks multiple threads to allow concurrent processing in order to deal with blocking conditions. Kernel threads are created using an internal `rfork()` operation. One of the threads is responsible for listening for new transport connections whereas the rest are workers involved in data transfer. All transport connections are bound to the

same completion group. Message arrivals on any transport connection generate *recv-done* interrupts that are routed to a single interrupt handler associated with the completion group. When the handler is invoked, it queues the incoming RPC request, notes the transport that was the source of the interrupt, and wakes up a worker thread to start processing. After parsing the request, a thread locks the necessary file pages in the buffer cache using `bread()`, prepares the RDMA descriptors and issues the RDMA operations. The RPC does not wait for RDMA completion. A later *send-done* interrupt (or successful poll) on a completed RDMA transfer descriptor starts clean up and release of resources that the transfer was utilizing (e.g. *bp* locks held on file buffers for the duration of the transfer), and sends out the RPC response. Threads blocking on those resources are awakened.

Event-driven design requires that event handlers be quick and not block between events. My original server design deviates from this requirement due to the possibility of blocking under certain conditions:

- The need to wait on disk I/O initiated by `bread()`. It is possible to avoid using the blocking `bread()` interface by initiating asynchronous I/O with the disk using `getblk()` followed by `strategy()`. I opted against this solution in the early design since disk event delivery is currently disjoint from network event delivery, complicating event handling. Integrating network and disk I/O event delivery is possible with appropriate kernel support [65].

- Locking a *bp* twice by the same kernel thread or releasing a *bp* from a thread other than the lock owner causes a kernel panic. Solutions are (a) to defer any request processing by a thread while past transfers it issued are still in progress, to ensure that a *bp* is always released by the lock owner and a thread never locks the same *bp* twice, or (b) to modify the buffer cache so that these conditions no longer cause a kernel panic. To avoid wider kernel changes in the original implementation, I chose to implement (a).

An important concern when designing an RDMA-based server is to minimize response latency for short transfers and maximize throughput for long transfers. In the current

design, notification of incoming messages is done via interrupts and notification of server-initiated transfer completions via polling. Short transfers using RDMA are expected to complete within the context of their RPC request. In this way, the RPC response immediately follows RDMA completion, minimizing latency. Throughput is maximized for longer transfers by pipelining them as their RDMA operations can be concurrently progressing. The DAFS kernel server presently runs over the Emulex cLAN and Myrinet VI-GM-1.0/GM-2.0-alpha1 transports.

6.3.3.3 Evolution of the Server Design

The original DAFS server design was constrained by the use of the file system interface (`bread()`, `brelse()`, etc.) in many ways. Besides the locking issues mentioned in the previous section, the use of *bp*'s, which are mapped on the buffer cache virtual address space, raises two other issues: First, the core kernel (VFS layer) has to be modified to register buffer cache mappings with the NIC. This makes the core (upper-part) of the FreeBSD kernel device-dependent. Second, the limited virtual address space allocated to the buffer cache in 32-bit CPUs, such as the Intel x86, implies that buffer cache mappings will be frequently invalidated, causing unnecessary and costly de-registrations and re-registrations with the NIC.

The solution chosen for the second (and current) generation of the DAFS server design was to access file data using the VM `getpages()` and `putpages`() interface instead. The benefits of this approach for overall system design are the following:

- The DAFS server (rather than the VFS layer) controls registration and de-registration with the NIC. The rest of the kernel need not be modified to accommodate the server.

- File data can be mapped on a larger virtual address space than allowed by the buffer cache. This is particularly useful in systems with large physical memory configurations. In this way, VM pages that cache file state are guaranteed to

remain registered with the NIC, for so long as these pages are resident in physical memory.

6.4 Optimistic RDMA and Optimistic DAFS

This section describes the implementation of the Optimistic RDMA (ORDMA) mechanism and the Optimistic DAFS (ODAFS) system, which were described and evaluated in Chapter 4.

6.4.1 Implementing ORDMA

The two main ORDMA implementation issues are (a) how to synchronize between the NIC and the host CPU when accessing VM pages, and (b) how to report NIC–to–NIC network exceptions in case of remote memory access faults.

6.4.1.1 NIC–host CPU synchronization

Synchronization is necessary because the NIC is allowed to set up DMA transfers between the network and main memory independently of the CPU. The kind of NIC–host CPU synchronization depends critically on OS support for multiple processors. An ORDMA-capable NIC in a multiprocessor OS can fully participate in the VM system by pinning/unpinning and locking/unlocking VM pages in response to network events. This is because a multiprocessor OS offers the necessary synchronization structures for the NIC to appear indistinguishable from an additional CPU to the OS, except for its performance. On the other hand, a NIC in a uniprocessor OS may not be able to pin pages from interrupt handlers if, for example, the OS is non-preemptive. In this case, synchronization via the host memory resident TPT is necessary.

The NIC should ensure that the following two conditions hold for the duration of DMA: First, pages involved in DMA have to remain resident in physical memory. Second, conflicting accesses by another CPU or NIC should not be allowed. I chose to satisfy both requirements by treating VM pages with translations loaded in the NIC TLB as both pinned and locked. The alternative of locking pages only for the duration of an I/O requires frequent NIC–host CPU interaction and was deemed too expensive in the case of

a NIC on the I/O bus. All pages in the TPT, except those with translations loaded on the NIC TLB, may be locked and invalidated by the host. The NIC updates the state of TPT entries by interrupting on each TLB miss. These interrupts increase CPU overhead but have the side-effect of speeding up the loading of TPT entries into the NIC, which is now done via a host-initiated programmed I/O operation, instead of (possibly several) NIC-initiated DMA on the PCI bus.

A drawback of having to synchronize via a device-specific page table is that the OS has to be aware of and adapt to the idiosyncrasies of the NIC. For example, it should always check with the NIC TPT before reclaiming a page and account for the fact that attempts to reclaim a physical page may fail until the page is evicted from the NIC TLB. To avoid starvation, the OS must increase its minimum free page threshold by the maximum amount of physical memory with page translations loaded on the NIC TLB. The OS must also be able to limit the effective size of the NIC TLB to avoid excessive pinning by the NIC.

6.4.1.2 NIC–to–NIC exceptions

ORDMAs may fail due to a variety of conditions, such as invalid address translation, protection violation, or failure to lock page(s). I decided to support such exceptions by extending the VI protocol with recoverable RDMA failure semantics. Since VI is a layer on top of Myrinet's GM in our prototype, I first modified the Myrinet GM Control Program to report such conditions as exceptions in low-level `get` (i.e., RDMA read) and `put` (i.e., RDMA write) operations. These exceptions are reported as "soft" or recoverable transport errors in the VI descriptor status flags, and can be appropriately handled by higher-level software, such as the DAFS client and the ODAFS user-level cache.

6.4.2 Implementing ODAFS

My implementation of a prototype ODAFS client and server extends the following existing DAFS components: a user-level DAFS file cache [2], a user-level DAFS API implementation [64] and a DAFS kernel server [65]. The ODAFS prototype relies on the ORDMA support for Myrinet described in Section 6.4.1.

The ODAFS server piggybacks remote memory references to data blocks in its kernel file cache onto RPC responses to the client. The ODAFS client stores these references in cache block headers. As data blocks are reclaimed by the client cache, memory references are allowed to live in "empty" headers. The client cache is configured with many more empty headers than data blocks. Ideally, it should have enough buffer headers to be able to map the entire server physical memory available for file caching.

I also modified the DAFS API to allow passing of ORDMA references, and the DAFS client implementation to include ORDMA operations in its event loop. On ORDMA exceptions, the DAFS client retries the operation using RPC in order to guarantee success. At RPC completion, the fresh piggybacked reference to the server buffers is passed to the ODAFS client.

The ODAFS server maps file blocks on a private 64-bit virtual address map. This is to ensure that there is always enough virtual address space to map large amounts of physical memory for long periods of time. Thus, I ensure that NIC TLB invalidations are due to the OS reclaiming a VM page due to memory pressure and never due to having to share a small virtual address space. This 64-bit address space is addressable only by the NIC and never by the CPU. It is therefore independent of whether the CPU has a 32- or 64-bit architecture. The current implementation of this 64-bit address space on the 32-bit Intel Pentium architecture is based on a paged segmentation scheme, a technique that has been used in the past to account for a limited machine-supported address space. For example, it is similar to the *overlays* used in the case of the 16-bit PDP-11 architecture.

Ideally, the replacement algorithm used in the server NIC TLB should be the same as the algorithm used in the client ORDMA directory.

6.5 Summary

This chapter described my implementation of the systems evaluated in this dissertation. All software is freely available from `http://www.eecs.harvard.edu/dafs`.

Chapter 7

Related Work

This chapter expands on the related work described in Chapter 3. In Section 7.1, I introduce direct-access networking and relate Remote Direct Memory Access (RDMA), which is the prominent implementation of direct-access networking, to the new mechanisms that I propose in this dissertation. In Section 7.2, I describe related previous work on direct-access network storage systems. In Section 7.3, I relate direct-access network storage systems to alternative designs based on virtual memory (VM) techniques.

7.1 Direct-access Networking

The term *direct-access networking* was introduced by the Direct-Access File System Collaborative (see Section 7.2) to characterize a class of transports offering RDMA capabilities. This dissertation generalizes this term to encompass any network mechanism that supports remote targeting of application buffers, independent of whether the targeting takes place by memory address, as in RDMA, or by another type of tag. An example of an alternative to RDMA for direct-access networking is the *RPC with tagged buffer pre-posting* mechanism, which is related to previous work in Section 7.1.2.

Besides the issue of direct data transfers between the network and application buffers, another issue that characterizes direct-access networks is the degree of involvement of the end-system hosts in setting up the data transfer. RDMA does not strictly require a handshake between the communicating hosts prior to the data transfer. Such a

handshake is not necessary, for example, when the initiating side already possesses a memory reference to a remote buffer and the latter is known to be prepared for the RDMA (e.g., registered with the NIC). Previous work in RDMA-based network file systems relied on impractical assumptions to safely use RDMA in this way [109]. In practice, however, existing systems always wrap RDMA in an RPC on a per-I/O basis to set up the transfer at both ends prior to the RDMA [29]. This method avoids remote memory access faults during the data transfer at the expense of incurring the cost of an RPC on each I/O. *Optimistic RDMA* is an extension to RDMA that avoids this cost by offering a practical mechanism to handle and recover from possible remote memory access faults.

RDMA can be used either with a user-level or with a kernel interface to a NIC. An overview of the history and present state of user-level networking as an interface to a direct-access NIC is described in Section 7.1.3.

7.1.1 Remote Direct Memory Access

Spector [103] was the first to propose a communication model based on *remote references*, simple inter-process communication primitives that allowed efficient implementation of remote memory access. Spector's implementation, however, was constrained by the simplicity of his experimental platform, which was composed of Xerox Alto workstations and 2.94 Mbit/s Ethernet. Thekkath and his colleagues [109] extended Spector's remote reference model to a true remote memory access model, incorporating virtual memory, protected access, and time-slicing on workstations. Their remote memory access model is composed of three operations (READ, WRITE, and Compare-and-Swap) and implemented as MIPS machine instructions using unused opcodes from the R3000 instruction set. A number of later projects explored RDMA implementations with appropriate NIC support. The SHRIMP [17] project explored implementing a global shared memory abstraction over specially-designed or commodity [78] network hardware. The original SHRIMP prototype consisted of specialized hardware that mapped parts of a process' address space to remote physical memory. Memory access by the application on a remotely mapped region was transparently claimed by the SHRIMP hardware, by snooping on the memory bus, and transformed into an RDMA operation. Remote memory

access with SHRIMP involved sending physical memory addresses to the remote node. Exported virtual memory regions were permanently pinned in physical memory. Research in the context of the Hamlyn project [21] proposed sender-based management of receiver buffers through the use of RDMA as a way to eliminate buffer overruns. All these early RDMA systems made similar choices in terms of how to keep exported memory buffers resident for the duration of an I/O. They all assumed that a host CPU is either the initiator of the RDMA or aware that an RDMA was in progress, and thus prepared for it. The Optimistic RDMA mechanism removes this assumption and enables a practical RDMA model where an RDMA can safely take place without the target host CPU being aware or prepared for it.

RDMA also relates to bulk transfer primitives available in super-computers since the 1980's, such as the Thinking Machines CM-5 [60] *scopy* (functionally similar to a remote memory write operation) and the IBM SP-2 [107] *remote memory copy*.

7.1.1.1 Address Translation Mechanisms

The primary requirement of RDMA is that the NIC holds physical memory addresses for the directly-accessed application buffers and the application I/O buffers remain resident in physical memory, at least for the duration of the I/O. To account for the address translation needs of the RDMA mechanism, the NIC is associated with a device-specific page table. Due to physical memory size limitations of the NIC, the latter keeps only a cache of the page table entries (i.e., a TLB), with the rest of the page table stored in host memory. NIC TLB misses can be resolved by looking up the NIC-specific page table. Misses in the NIC page table are possible when the RDMA is initiated by a remote node without a previous RPC to prepare the target memory buffers. Figure 23 shows the different possibilities for address translation for the needs of the NIC and for keeping VM pages resident in physical memory for the duration of RDMA.

The issues with NIC address translation for RDMA were examined by Welsh and his colleagues [117] and by Schoinas and Hill [100]. Welsh and his colleagues proposed

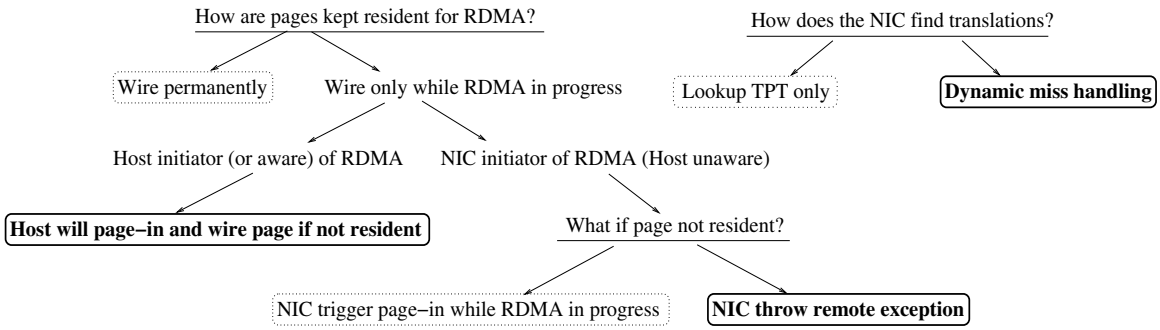

FIGURE 23. Design Decisions for RDMA Page Access and Address Translation. Optimistic RDMA choices are in bold.

that page translations be pinned in physical memory while their entries are loaded on the NIC TLB. Lookups into the NIC TLB are performed by the NIC itself. Misses, however, are handled by the host after the NIC throws an interrupt. In case of a NIC TLB miss, the host is expected to lookup the NIC page table, issue a page fault to transfer the data from the disk if the page is not resident in physical memory, and re-load the missing entry in the NIC TLB. They did not, however, address the practical details of how well such a design would work in a real implementation. In particular, this design requires that the OS supports page-fault handling triggered by a device other than the CPU, which is usually not possible in uni-processor operating systems. It also requires that the faulting RDMA operation be suspended for a significant amount of time, which may require significant NIC-host collaboration due to the limited resources of the NIC. Shoinas and Hill examined the NIC design space more thoroughly and considered the cases of (a) the NIC performing both the lookup and the miss handling, which requires significant OS support to treat the NIC as a processor, and (b) the host CPU performing both the lookup and the miss handling, which is possible when the host CPU has direct, protected access to the NIC TLB. Miss handling by the host CPU is a practical choice with a NIC attached on the I/O bus. Miss handling by the NIC is possible with NICs that are better integrated with the host memory system (e.g., attached on the system bus) or that share a TLB with the host CPU [10].

My Optimistic RDMA model differs from previous RDMA models in the way access-fault handling is performed. With ORDMA, a miss in the NIC page table does not trigger action by the host or by the NIC to establish a valid mapping and complete the RDMA. Instead, an access fault results in reporting an exception to the initiator of the ORDMA, signaling failure to perform the remote access.

7.1.1.2 Cost of Registration and De-registration

The cost of memory registration and de-registration can be avoided by pre-registering all I/O buffers at once, if a limited buffer pool is used for communication. Resource-intensive applications, however, such as databases, require large numbers of I/O buffers. In such cases, a limited buffer pool introduces the need for memory copying to move data received in intermediate user-level buffers to their final destination. The alternative of registering application buffers on a per-I/O basis avoids memory copying, at the expense of incurring registration costs. Zhou and her colleagues [121] proposed batching de-registrations to reduce the average cost of de-registering memory. The DAFS server evaluated in this dissertation reduces de-registrations by (a) implementing a large virtual address space to match the physical memory configuration of the server machine, which often exceeds the extent of the per-process virtual address space available in 32-bit architectures, and (b) by batching de-registrations.

7.1.2 Other Approaches to Direct-access Networking

Anderson and his colleagues [6] used a form of tagged pre-posting to achieve direct data placement in RPC-based data transfer. This work was in the context of an in-kernel RPC service that they used to implement a Global Memory Service (GMS) [37] layer. Even though their work was not directly related to a network storage system, the GMS network memory layer can be transparently used by the local file and VM systems, achieving close to peak network bandwidth using a memory-mapped I/O interface. The RPC with tagged pre-posting mechanism used in this dissertation is similar to the approach described here in that it uses an RPC-specific tag (e.g., the RPC transaction number) to identify anticipated incoming data payloads. It differs, however, in that it interoperates

with the explicit read/write I/O interface (in addition to the mmap interface), enables direct transfers to application buffers, and is integrated with the NFS protocol.

7.1.3 User-level Networking

User-level access to NICs was originally proposed, developed and used in distributed memory super-computers such as the Thinking Machines CM-5 [60], the IBM SP-2 [107], and the Meiko CS-2 [45], since the 1980's. These systems provided support for low-latency communication protocols, such as Active Messages [114]. The potential benefits of user-level networking with commodity NICs were demonstrated by the U-Net [113] research project, which focused on the low communication latency and higher flexibility possible with user-level access to ATM network interface controllers. A user-level networking protocol has been recently standardized with the Virtual Interface (VI) architecture [111], a specification supported by Intel, Compaq, and IBM. This protocol facilitated the commercialization of user-level networking technology and its introduction in commodity NICs.

A number of other research projects explored the potential of user-level networking in distributed applications [17,21,78,82]. The SHRIMP [17] and Hamlyn [21] projects also combined user-level networking with RDMA. The Mitsubishi DART [82] was another user-level NIC offering very low latency over ATM networks. The Myrinet network offered a programmable NIC platform [78] over which most of the user-level networking research has been conducted. All systems proposed and evaluated in this dissertation were implemented on a switched Myrinet network with the LANai 9.2 NIC, which is the most advanced implementation of the LANai architecture at the time of this writing.

Besides reducing the latency of interaction with the NIC, user-level networking also reduces the overhead of event handling. In particular, polling from user-space offers much lower latency than the alternative of dispatching a thread to provide a process context. Another way to reduce this overhead is by using Optimistic RPC (ORPC) [115]. With ORPC, handling of the RPC takes place within the context of the original hardware interrupt handler, avoiding the overhead associated with dispatching a thread. Hardware

interrupt handlers, however, are non-schedulable entities, and thus, cannot block. The ORPC model is optimistic in the sense that it is hoped that event handling can be completed without blocking. If, however, blocking is necessary, a process or thread context is spawned to carry on with event handling. ORPC is a mechanism to reduce the per-I/O overhead of network data transfer, and thus, competitive to ORDMA. The latter, however, involves no interaction with the target host, resulting in greater reduction of per-I/O overhead. ORPC still involves hardware interrupts and host-device interaction over the I/O bus, operations whose performance is not expected to improve as quickly as core CPU technology.

Besides user-level networking, user-level file systems were proposed in the context of DAFS, as will be described in the following section. Earlier work arguing against user-level file systems [116] assumed some form of kernel mediation in the I/O data path, such as when using the UNIX `mmap` interface.

7.2 Direct-access Network Storage

In the fall of 2000, a new network attached storage protocol, the Direct Access File System (DAFS), was proposed by Network Appliance Inc. [29]. DAFS relies on user-level networking to enable a user-level file client structure and RDMA for network protocol offload and memory copy avoidance [64]. Network Appliance and Intel jointly formed and led the DAFS Collaborative, an academic and industrial consortium with the goal to standardize the DAFS protocol specification [29].

DAFS was designed with a number of assumptions in mind: First, the focus is placed on the file client, where the application is running. With DAFS, the file server is assumed to be overprovised and to always have sufficient resources. This assumption is reflected in many design decisions taken in the DAFS specification. For example, when there is a choice of performing an operation, such as issuing an RDMA, adapting endianness, or computing a checksum on the client or on the server, DAFS chooses to do it on the server. This assumption was challenged in this dissertation, which looked at file access workloads involving multiple clients and small I/Os. Such workloads are expected to stress the file server and require new mechanisms that lower the server over-

head. One such mechanism is Optimistic RDMA, described in Section 4.2. Second, the DAFS designers assumed that the server is trusted to directly access client buffers. This is another reason why RDMA operations are always initiated by the server. When RDMA can be initiated by (mutually untrusted) clients, such as with Optimistic DAFS, stronger safety guarantees than currently offered by RDMA implementations are necessary to ensure that clients access server buffers for which they have been granted access rights (Section 4.1).

Prior to DAFS, Thekkath and his colleagues [109] suggested using direct-access networking primitives, such as remote memory read and write operations, for network file access. Thekkath advocated the separation of data and control transfer in distributed systems and proposed a new network file system structure based on client-initiated RDMA as a means to lower I/O latency. They proposed using RPC only when control transfer is necessary, otherwise use a pure network I/O mechanism, such as RDMA. One limitation of their work, however, is the fact that their RDMA model makes the following simplifying assumptions:

- Virtual memory buffers are pinned in physical memory, at the client and the server.

- A hash-based remote memory address mapping scheme, enabling clients to compute the remote memory location of data and metadata, avoiding the need for buffer advertisement by RPC.

Pinned virtual memory buffers is not a realistic assumption, particularly in the case of a file server buffer cache. In addition, hash-based remote memory mapping methods require significant network file system re-design. The advent of the VI protocol specification and the commercialization of the VI technology made practical new designs possible, along the lines proposed by Thekkath. The Optimistic RDMA network I/O mechanism and the Optimistic DAFS are two examples of such designs. ORDMA removes the first assumption made by Thekkath, namely, that the VM buffers need to be

pinned in physical memory. ODAFS avoids the need for a hash-based mapping scheme, enabling a much simpler system design.

Besides NAS systems, direct-access networking has been used in block storage access. One such study was performed by Zhou and her colleagues [122]. They found that user-level storage access, avoidance of interrupts by use of polling, and reduction of the locking/synchronization cost contribute to high TPC-C transaction rates with a VI-based storage server. However, they do not compare their system to other alternatives such as iSCSI, FibreChannel, or optimized RPC-based block-level or file-level implementations.

Doyle and his colleagues [31] used DAFS to implement model-based resource provisioning in a Web service utility. In their prototype, a user-level Web server (Dash) incorporates a DAFS user-level file system client, which enables user-level resource management and full control over file caching and data movement.

Buonadonna and Culler [20] proposed a new networking API, Queue Pair over IP (QP-IP) as an appropriate interface to NICs supporting transport protocol offload. QP-IP is an alternative to sockets that is interoperable with IP protocols. QP-IP requires pre-posting of communication buffers with the NIC. In addition to basic networking benchmarking, Buonadonna and Culler evaluated a block-based network storage service, the Network Block Device (NBD), and found significant throughput improvement and CPU overhead reduction compared to the socket interface with a host-based TCP/IP stack. The RPC with tagged pre-posting mechanism contributed in this dissertation can be thought of as an extension to the QP-IP interface.

7.3 Reducing Overhead in Network File Systems using VM Techniques

The overhead of memory copying on the I/O path through the host CPU and memory system can be reduced by performing logical rather than physical memory copying. One way to achieve this is by using virtual memory (VM) re-mapping and copy-on-write (COW) [1] techniques. *Fbufs* [33] is an example of an inter-process communication (IPC) mechanism that supports sharing of I/O buffers between multiple protection domains with a combination of VM page re-mapping and shared memory. The *fbufs* model has

been used to implement a zero-copy network protocol stack in the context of the Solaris operating system [108] and a unified caching and buffering system called IO-Lite [84]. Both systems introduce a network and file API with move rather than copy semantics. The drawback of move semantics is that the application does not control the data layout since *fbufs* are allocated by the system. Brustoloni [18] introduced a number of VM techniques that can be used to support efficient data movement on the network I/O path with copy semantics. In addition, he showed that these techniques interoperate with the UNIX `mmap` interface but not with the explicit read/write I/O interface [19]. Magoutis and his colleagues [64] implemented and evaluated an NFS implementation that uses VM page re-mapping to avoid memory copying in the incoming I/O data path.

All the VM techniques mentioned in this section require some support from the network interface controller (NIC) to achieve early demultiplexing and/or aligned placement in host memory buffers. *Fbufs*, for example, require the NIC to examine the incoming data packet, associate it with a particular process, and place the data in a buffer from the pool belonging to that process. Brustoloni's *input with early demultiplexing* [18] technique assumes that the NIC aligns the data payload of incoming packets to the preferred alignment of system buffers. VM page-remapping can be subsequently performed to application-aligned buffers. Magoutis and his colleagues [64] implemented a header-splitting NIC that separates the NFS data payload from all headers and DMAs the payload to page-aligned system buffers. This system is described in detail in Section 6.1.

Chapter 8

Conclusions and Future Work

8.1 Conclusions and Wider Implications

The current exploding demand for storage capacity, driven by the rapid growth of Internet e-commerce and by the rapid decline of the cost of on-line storage, makes the traditional model of direct-attached storage infeasible due to the scalability limits of traditional I/O buses. Instead, a *network storage* model, where the storage devices are separated from the application servers by a scalable network infrastructure, becomes the prominent alternative. The emergence of high-speed networks and the ability of applications and storage systems to transfer data at high rates, using aggressive I/O policies and a high degree of parallelism in storage devices, put a stress on the part of the I/O data path through the host CPU and memory system. The current mismatch in the technological advances in network and memory bandwidth predicts that this performance bottleneck will only become worse in the future. These trends suggest that network storage system designers should focus on optimizations that reduce the overhead of network communication.

Network storage systems based on a block abstraction, or Storage-area Networks (SAN), currently enjoy support provided by storage-specific network infrastructures to reduce their communication overhead to the levels of direct-attached storage. Network attached Storage (NAS) systems, however, which are based on a file abstraction, pres-

ently lack such support, and thus, result in higher communication overhead than SAN. This dissertation shows that this is not a fundamental disadvantage of the NAS model. In other words, it is possible for NAS systems to achieve high performance given appropriate network interface support.

In this dissertation, I make the distinction between two types of overhead: Per-byte (or data touching overhead) and per-I/O. One important point in the NAS system design space that I consider is that of I/O-intensive file access workloads that perform large I/Os. This is a class of workloads sensitive to the per-byte overhead of memory copying, which is required for data movement due to the need for staging file data on the I/O path between the network and application memory buffers. Two ways to avoid memory copying, both relying on some network interface support currently not offered by networking adapters, are to:

(a) Transfer data directly between the network and application buffers, bypassing the kernel; or

(b) Move data by virtual memory page re-mapping rather than by physical memory copying.

I consider two different mechanisms to achieve (a) and one mechanism to achieve (b). The mechanisms to achieve (a) are remote direct memory access (RDMA) and RPC with buffer pre-posting. I show that all three mechanisms enable file access throughput that saturates a 2Gb/s network link when performing large I/Os on relatively slow, commodity PCs. There are, however, differences in the overhead characteristics of these mechanisms. RDMA has the lowest overhead when the host has previously registered all VM pages used for communication with the NIC and there is no longer a need for per-I/O registration. Buffer pre-posting and page re-mapping have higher overhead compared to RDMA. This additional overhead stems from the need for registering buffers with the NIC on a per-I/O basis with the former mechanism and from the cost of VM re-mapping with the latter. Using an analytical model of network attached storage application performance and experimental measurements of a Berkeley DB workload, I show that the

strongest benefit from reducing communication overhead is for balanced workloads, in which application processing saturates the CPU when I/O occurs at network speed.

Another important design point examined in this dissertation is that of workloads involving multiple clients and small I/Os. These workloads are most sensitive to the per-I/O CPU overhead on the server. To ease that bottleneck, I propose *Optimistic RDMA* (ORDMA), a new network I/O mechanism that improves server throughput and response time, and *Optimistic DAFS* (ODAFS), my extension to the DAFS protocol that uses ORDMA. Performance improvements in server throughput and response time with my implementation of ORDMA and ODAFS on a high-speed network infrastructure range up to 32% and 36%, respectively, for small I/O transfers.

Designing appropriate network I/O mechanisms, such as those described in this dissertation, is one of the keys to improving the performance of I/O-intensive NAS applications in high-speed networks. These mechanisms, however, make up only one side of the equation. Another important issue is the effect of the OS structure in network attached storage application performance. In particular, a number of research projects in the past have argued that:

(a) Co-locating the application server and the operating system in the same address space, as proposed by the Exokernel architecture, is key to achieving high performance due to the high degree of flexibility that they offer (e.g., by eliminating all fixed OS abstractions [35]), and due to the avoidance of the protection domain crossing inherent in a system-call API.

(b) Extensible operating systems, such as SPIN and VINO, are another prominent alternative structure that offers the high degree of flexibility required by resource-intensive applications. These systems, however, have been challenging to design and implement due to the complex safety and security issues involved.

In this dissertation, I show that flexibility can be achieved with an appropriately designed API that exposes the I/O mechanisms and enables applications to specify and fully control their I/O policies. This is possible, for example, by implementing file caching and network buffering policies in user space. Such an API can be simple to imple-

ment and therefore expected to facilitate its rapid deployment in mainstream operating systems. In addition to flexibility, I show that the cost of the protection domain crossing in a kernel API to the file system can be a small fraction of the total I/O overhead, even in systems that have already reduced that overhead by using the network I/O mechanisms proposed in this dissertation. An important benefit of involving the kernel on the I/O path, besides the enforcement of global policies and the safe and secure sharing of code and data between multiple applications, is that there is no need for a user-level interface to network adapters, simplifying their implementations.

The conclusions of this dissertation have wider implications. They are evidence to the convergence of network and I/O interconnects, as well as to the convergence of the SAN and NAS models, as described in the following sections.

8.1.1 Network-I/O Convergence

Traditionally, storage and general network data traffic has been routed over separate hardware data paths, which are exemplified by parallel or serial SCSI channels and Ethernet networks, respectively. This separation led to different hardware and software support for each type of I/O. The emergence of the network storage model as practically the only feasible solution to the explosive demand for on-line storage, and the ubiquity, scalability and low cost of Ethernet networks, point to their convergence as the next goal. New support for efficient block and file access over Ethernet networks, such as provided by the iSCSI [72] protocol and the RPC with buffer pre-posting mechanism proposed in this dissertation and applicable to NFS, proves that storage I/O traffic over widely-deployed network infrastructure can be as efficient as with locally-attached storage I/O channels. In addition, the emergence of *direct-access networks* offers a *unifying technology* that can be used to build general-purpose networking adapters that efficiently support both block and file network storage traffic, in addition to the data traffic produced by other, more general forms of inter-process network communication.

8.1.2 NAS-SAN Convergence

New network support for NAS and SAN systems, compatible with the widely-deployed Ethernet infrastructure, lays the road to the anticipated convergence between the NAS and SAN models. This convergence is in terms of the degree of specialization these systems require from the network infrastructure. For example, SAN systems, traditionaly more specialized than NAS, are in the future expected to be deployed over Ethernet networks using the iSCSI protocol. In contrast, NAS systems, traditionally less specialized than SAN, are now expected to be offered new services over Ethernet networks, in the form of direct-access networking techniques such as RDMA and RPC with buffer preposting mechanism (RDDP-RPC).

Network Storage Architecture	Network -aware Client	File Sharing	Virtualization and Security	Scalability Limit
NAS	Yes	Server	Server	Server I/O subsystem throughput
SAN	No[a]	Clients	Server	Server I/O subsystem throughput
NASD	Yes	Server	Server/Storage Devices	Aggregate storage device throughput

TABLE 15. Comparison between the NAS, SAN, and NASD models.

a. This holds for a single client only. Multiple distributed clients require a synchronization mechanism.

The significance of the convergence between the NAS and SAN models lies on the fact that the choice between the two for deployment in a particular network environment no longer depends on the network infrastructure offered by that environment. The choice between the two systems can instead be made based on their structural differences summarized in Table 15. Comparing NAS to SAN, NAS is preferable in a multi-client setup due to its better file sharing properties. For a single client, however, a SAN is preferrable if standard SCSI block access semantics is all that is required by the application. Both SAN and NAS are subject to the same scalability limitations due to the involvement of

the file or block storage server on the I/O data path between the clients and the storage devices.

NASD, which was described in Section 2.1, is a bridging model, offering server-managed file sharing while avoiding the storage server I/O bottleneck. NASD, however, requires strong security support at the storage devices. It also requires the deployment of network-aware clients, just like with NAS. Wide adoption of the NASD model relies on storage device manufacturers offering the necessary security support. In practical terms, improved scalability options in the server I/O subsystem available with new technologies such as Infiniband and PCI-X/PCI-Express may give the NAS model a competitive advantage over NASD, particularly given the significant requirements of the latter to ensure safe and secure direct client access to storage devices.

8.2 Future Work

An open question that is likely to stimulate further research in the field of direct-access network attached storage in the near future is whether strong security in the network or transport layer can be combined with RDMA without altering its benefits. This question is particularly intriguing given the technological and time-to-market constraints limiting the processing power of NICs [101]. A security protocol widely considered to be a standard on the Internet today, and therefore, a prominent candidate for deploying on direct-access network adapters, is IPsec [56]. The question that arises is whether IPsec can be offloaded to the NIC without affecting RDMA performance, particularly at projected multi-gigabit per second network speeds.

Another possible direction of future research is an extension of the work presented in Chapter 5 to experimentaly compare the performance of a user-level implementation to a kernel implementation of the proposed API in a variety of workloads. Such a comparison is expected to validate the analytical modeling approach of Chapter 5, taking into account second-order effects such as cache, TLB, and page table behavior. It can be performed using the Pentium processor architecture, possibly an additional alternative architecture, or a CPU simulator. The latter can be used to study the effect of the CPU architecture on system performance [93].

References

[1]. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, “Mach: A New Kernel Foundation for UNIX Development”, in *Proceedings of Summer 1986 USENIX Conference*, pp. 93-112, Altanta, GA, 1986.

[2]. S. Addetia, “User-level Client-side Caching for DAFS”, Harvard University TR-14-01, March 2002.

[3]. Alacritech, Inc., TCP/IP acceleration based on SLIC Technology, `http://www.alacritech.com`.

[4]. A. Alexandrov, M. Ionescu, K. E. Schauser, C. Scheiman, "LogGP: Incorporating Long Messages into the LogP model - One Step Closer Towards a Realistic Model for Parallel Computation", in *Proceedings of 7th Annual Symposium on Parallel Algorithms and Architecture (SPAA'95)*, pp. 95-105, Santa Barbara, CA, July 1995.

[5]. D. Anderson, J. Dykes, E. Riedel, “More than an Interface - SCSI vs. ATA”, in *Proceedings of Second USENIX File and Storage Symposium*, pp. 245-256, San Francisco, CA, March 2003.

[6]. D. Anderson, J. S. Chase, S. Gadde, A. Gallatin, K. Yocum, “Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet”, in *Proceedings of 1998 USENIX Annual Technical Conference*, pp. 143-154, New Orleans, LA, June 1998.

[7]. D. Anderson, J. S. Chase, A. Vahdat, "Interposed Request Routing for Scalable Network Storage", in *Proceedings of 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 259-272, San Diego, CA, October 2000.

[8]. T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang, “Serverless Network File Systems”, in *Proceedings of 15th ACM Symposium on Operating Systems*

Principles (SOSP-15), pp. 109-126, Copper Mountain Resort, Colorado, December 1995.

[9]. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", In *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP-13)*, pp. 95-109, Pacific Grove, CA, October 1991.

[10]. B. Ang, D. Chiu, L. Rudolph, Arvind, "Message Passing Support on StarT-Voyager", CSG Memo 387, MIT Laboratory for Computer Science, July 1996.

[11]. A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, "Information and Control in Gray-Box Systems", in P*roceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-18)*, pp. 43-56, Banff, Canada, October 2001.

[12]. M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J. Ousterhout, "Measurements of a Distributed Filesystem", in *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP-13)*, pp. 198-212, Pacific Grove, CA, October 1991.

[13]. B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain, CO, December 1995.

[14]. A. Birrell, A. Hisgen, C. Jerian, T. Mann, G. Swart, "The Echo Distributed File System", Technical Report #111, DEC SRC, Palo Alto, CA, September 1993.

[15]. A. Birrell, B. Nelson, "Implementing Remote Procedure Calls", in *ACM Transactions on Computer Systems*, (2)1:29-59, February 1984.

[16]. T. Blackwell, "Speeding up Protocols for Small Messages", in *Proceedings of ACM SIGCOMM* '96, pp. 85-95, Stanford, CA, August 1996.

[17]. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer", in *Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA)*, pp. 142-153, Chicago, IL, April 1994.

[18]. J. Brustoloni, P. Steenkiste, “Effects of Buffering Semantics on I/O Performance”, in P*roceedings of Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pp. 277-291, Seattle, WA, October 1996.

[19]. J. Brustoloni, “Interoperation of Copy Avoidance in Network and File I/O”, in *Proceedings of IEEE INFOCOM'99 Conference*, pp. 534-542, New York, NY, March 1999.

[20]. P. Buonadonna, D. Culler, “Queue-Pair IP: A Hybrid Architecture for System Area Networks", in *Proceedings of 29th International Symposium on Computer Architecture (ISCA)*, Anchorage, AK, May 2002.

[21]. G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, J. Wilkes, “An Implementation of the Hamlyn Sender-Managed Interface Architecture”, in *Proceedings of 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 245-259, Seattle, WA, October 1996.

[22]. B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, O. Asad, “NFS over RDMA”, in *Proceedings of Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, Karlsruhe, Germany, August 2003.

[23]. J. S. Chase, A. Gallatin, K. Yocum, “End System Optimizations for High-Speed TCP”, in *IEEE Communications*, (39)4:68-74, April 2001.

[24]. J. B. Chen, B. N. Bershad, “The Impact of Operating System Structure on Memory System Performance”, in *Proceedings of 14th ACM Symposium on Operating System Principles (SOSP-14)*, Asheville, NC. December 1993.

[25]. N. Christenson, T. Bosserman, D Beckermeyer, “A Highly Scalable Electronic Mail Service using Open Systems”, in *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[26]. D. Clark, ”The Structuring of Systems Using Upcalls”, in *Proceedings of 10th ACM Symposium on Operating Systems Principles (SOSP-10)*, pp. 171-180, Orcas Island, WA, December 1985.

[27]. D. Clarke, V. Jacobson, J. Romkey, H. Salwen, “An Analysis of TCP Processing Overhead”, in *IEEE Communications Magazine*, 27(6):23--29, June 1989.

[28]. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", in *Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1-12, San Diego, CA, May 1993.

[29]. M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, M. Whittle, "The Direct Access File System", in *Proceedings of Second USENIX File and Storage Technologies Conference*, pp. 175-188, San Francisco, CA, March 2003.

[30]. M. Devarakonda, A. Mohindra, J. Simoneaux, W. Tetzlaff, "Evaluation of Design Alternatives for a Cluster File System", in *Proceedings of 1995 USENIX Annual Technical conference*, pp. 35-46, New Orleans, LA, January 1995.

[31]. R. Doyle, J. Chase, O. Asad, W. Jin, A. Vahdat, "Model-Based Resource Provisioning in a Web Service Utility", in *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[32]. P. Druschel, V. Pai, W. Zwaenepoel , "Extensible Kernels are Leading OS Research Astray", in *Proceedings of 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 1997.

[33]. P. Druschel, L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, Asheville, NC, ecember 1993.

[34]. EMC Cellera HighRoad, White Paper, `http://www.emc.com/pdf/products/celerra_file_server/HighRoad_wp.pdf`, January 2002.

[35]. D. Engler, M. F. Kaashoek, "Exterminate All Operating Systems Abstractions", in P*roceedings of 5th Workshop on Hot Topics in Operating Systems (HotOS V)*, Orcas Island, WA, May 1995.

[36]. J. Eppinger, L. Mummert, A. Spector. "Camelot and Avalon". Morgan Kaufmann, San Francisco, CA, 1991.

[37]. M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, C. Thekkath, "Implementing Global Memory Management in a Workstation Cluster", in *Proceedings of 15th*

ACM Symposium on Operating Systems Principles (SOSP-15), Cooper Mountain Resort, CO, December 1995.

[38]. E. Felten, J. Zahorjan. "Issues in the Implementation of a Remote Memory Paging System", CS TR 91-03-09, University of Washington, March 1991.

[39]. E. Gabber, C. Small, J. Bruno, J. Brustoloni, A. Silberschatz, "The Pebble Component-Based Operating System", in *Proceedings of the 1999 USENIX Annual Technical Conference*, pp. 267-282, Monterey, CA, June 1999.

[40]. G. Ganger, D. Engler, F. Kaashoek, H, Briceno, R. Hunt, T. Pickney, "Fast and Flexible Application-Level Networking on Exokernel Systems", in *ACM Transactions on Computer Systems*, 20(1):49-83, February 2002.

[41]. G. Gibson, D. Nagle, K. Amiri, J. Buttler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka, "A Cost-Effective, High-Bandwidth Storage Architecture", in *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pp. 92-103, San Jose, CA, October 1998.

[42]. G. Gibson, R. Van Meter, "Network Attached Storage Architecture", in *Communications of the ACM*, 43(11): 37-45, 2000.

[43]. J. Hartman, J. Ousterhout, "The Zebra Striped Network File System", in *ACM Transactions on Computer Systems*, 13(3):274--310, August 1995.

[44]. J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kauffman, 1992.

[45]. M. Homewood, M. McLaren, "Meiko CS-2 Interconnect Elan-Elite Design", in *Proceedings of Hot Interconnects VI*, Stanford, CA, August 1993.

[46]. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, "Scale and Performance in a Distributed File System", in *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.

[47]. N. Hutchinson, L. Peterson, " The x-kernel: An Architecture for Implementing Network Protocols", in *IEEE Transactions on Software Engineering*, 17(1):64-76, January 1991.

[48]. IETF Remote Direct Data Placement (RDDP) Working Group, `http://www.ietf.org/`

[49]. InfiniBand Trade Association, InfiniBand Architecture Specification Volume 1, Release 1.0, October 24, 2000.

[50]. A. Iyengar, J. Challenger, D. Dias and P. Dantzig, "High-Performance Web Site Design Techniques", in *IEEE Internet Computing*, 4(2):17-26, March 2000.

[51]. P. Joubert, R. King, R. Neves, M. Russinovich, J. Tracey, "High-Performance Memory-based Web Servers: Kernel and User-space Performance", in *Proceedings of 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[52]. C. Jurgens, "FibreChannel: A Connection to the Future", in *IEEE Computer*, 28(8):88-90, August 1995.

[53]. M. F. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, K. Mackenzie, "Application Performance and Flexibility in Exokernel Systems", in *Proceedings of 16th ACM Symposium on Operating System Principles (SOSP-16)*, St. Malo, France, October 1997.

[54]. J. Katcher, "PostMark: A New File System Benchmark", Network Appliance TR-3022, October 1997.

[55]. M. Kazar, B. Leverett, O. Anderson, V. Apostolides, B. Bottos, S. Chutani, C. Everhart, W. Mason, S. Tu, E. Zayas, "Decorum File System Architectural Overview", in *Proceedings of Summer 1990 USENIX Technical Conference*, June 1990.

[56]. S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, `http://www.ietf.org/rfc/rfc2401.txt`, November 1998.

[57]. H. Krawczyk, M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", IETF, Network Working Group, RFC 2104, February 1997.

[58]. H. C. Lauer, R. M. Needham, "On the Duality of Operating System Structures", in *Operating Systems Review*, 13(2):3-19 (1979).

[59]. E. Lee, C. Thekkath, "Petal: Distributed Virtual Disks", in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pp. 84-92, Cambridge, MA, 1996.

[60]. C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. Daniel Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, R. Zak, "The Network Architecture of the Connection Machine CM-5 (Extended Abstract)", in *Proceedings of Symposium on Parallel Algorithms and Architectures (SPAA-92)*, pp. 272-285, San Diego, CA, June 1992.

[61]. J. Liedtke, "On Micro-Kernel Construction", in *Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pp. 109-126, Copper Mountain Resort, Colorado, December 1995.

[62]. C. Maeda, "Service Decomposition: A Structuring Principle for Flexible, High-Performance Operating Systems", PhD Thesis, Carnegie Mellon, December 1997.

[63]. K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, "Making the Most out of Direct-Access Network Attached Storage", in *Proceedings of Second USENIX File Access and Storage Symposium*, pp. 189-202, San Francisco, CA, March 2003.

[64]. K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, E. Gabber, "Structure and Performance of the Direct Access File System", in *Proceedings of 2002 USENIX Annual Technical Conference*, Monterey, CA, pp. 1-14, June 2002.

[65]. K. Magoutis, "Design and Implementation of a Direct Access File System Kernel Server for FreeBSD", in *Proceedings of the USENIX BSDCon 2002 Conference*, pp. 65-76, San Franscisco, CA, February 2002; also appears in UNIX Magazine (in Japanese), March 2003, ASCII Publications, Japan.

[66]. R. Martin, A. Vahdat, D. Culler, T. Anderson, "Effects of Communication Latency, Overhead and Bandwidth in a Cluster Architecture", in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pp. 85-97, Denver, Colorado, June 1997.

[67]. R. Martin and D. Culler, "NFS Sensitivity to High-Performance Networks", in *Proceedings of SIGMETRICS '99/ PERFORMANCE '99 Joint International Conf. on Measurement and Modeling of Computer Sys*tems, pp. 71-82, Atlanta, GA, May 1999.

[68]. S. McCanne, V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", In *Proceedings of the Winter 1993 USENIX Conference*, pp. 259-269, San Diego, CA, January 1993.

[69]. S. McCanne, C. Torek, A "Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling", in *Proceedings of the Winter 1993 USENIX Conference*, pp. 387-394, San Diego, CA, 1993.

[70]. K. McKusick, K. Bostic, M. Karels, J. Quarterman, "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley, 1996.

[71]. D. McNamee, K. Armstrong, "Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies", in *Proceedings of the Mach Usenix Workshop*, pp. 31-43, Burlington, VE, October 1991.

[72]. K. Meth, J. Satran, "Design of the iSCSI Protocol", in *Proceedings of the 20th IEEE/ 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 7-10, 2003, Paradise Point Resort, San Diego, CA.

[73]. J. Mogul, "TCP Offload is a Dumb Idea Whose Time Has Come", in *Proceedings of Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 2003.

[74]. D. Mosberger, L. Peterson, P. Bridges, S. O'Malley, "Analysis of Techniques to Improve Protocol Processing Latency", in *Proceedings of SIGCOMM'96*, Stanford, CA, August 1996.

[75]. D. Mosberger, L. L. Peterson, "Making Paths Explicit in the Scout Operating System", in *Proceedings of the Second USENIX symposium on Operating systems Design and Implementation (OSDI'96)*, Seattle, WA, October 1996.

[76]. T. Mowry, A. Demke, O. Krieger, Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications, in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implmenetation (OSDI'96)*, Seattle, WA, October 1996.

[77]. D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems (your cache ain't nuthin' but trash), In *Proceedings of 1992 USENIX Annual Technical Conference*, pp. 305-314, San Antonio, TX, January 1992.

[78]. Myricom LANai9.2 and GM communication library, Myricom Inc., http://www.myri.com

[79]. D. Nagle, G. Ganger, J. Butler, G. Goodson, C. Sabol, "Network Support for Network-Attached Storage", in *Proceedings of Hot Interconnects VI*, Stanford, CA, August 1999.

[80]. M. Nelson, B. Welch, J. Ousterhout, "Caching in the Sprite Network File System", in *ACM Transactions on Computer Systems*, 6(1):134-154. February 1988.

[81]. M.Olson, K. Bostic, M. Seltzer, "Berkeley DB", in *Proceedings of USENIX Annual Technical Conference (FREENIX Track)*, pp. 183-192, Monterey, CA, June 1999.

[82]. R. Osborne, Q. Zheng, J. Howard, R. Casley, D. Hahn. "DART - A Low Overhead ATM Network Interface Chip. In *Proceedings of Hot Interconnects* IV, Stanford, CA, August 1996.

[83]. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, B. Welch, "The Sprite Network Operating System", in *IEEE Computer Group Magazine* 21(2), 1988.

[84]. V. Pai, P. Druschel, W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System", in *Proceedings of Third USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pp. 15-28, New Orleans, LA, February 1999.

[85]. V. Pai, P. Druschel, W. Zwaenepoel, "Flash: An Efficient and Portable Web Server", In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[86]. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, D. Hitz, "NFS Version 3 Design and Implementation", in *Proceedings of 1994 USENIX Annual Technical Conference*, Boston, MA, June 1994.

[87]. D. Pease, "IBM Storage Tank", *Work in progress presented at 1st USENIX File and Storage Technologies Conference*, Monterey, CA, January 2002.

[88]. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System", in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Copper Mountain Resort, Colorado, December 1995.

[89]. C. Pu, H. Massalin, and J. Ioannidis. "The Synthesis Kernel", in *Computing Systems*, 1(1):11-32, 1988.

[90]. R. Recio, "Server I/O Networks: Past, Present, and Future", in *Proceedings of Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, Karlsruhe, Germany, August 2003.

[91]. D. M. Richie, K. Thompson, The UNIX timesharing system. Bell Systems Technical Journal 57, 6 (July-Aug 1978), 1905-1929.

[92]. D. Robinson, "The Advancement of NFS Benchmarking: SFS 2.0", in *Proceedings of XIII USENIX LISA Conference*, Seattle, WA, November 1999.

[93]. M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, A. Gupta, "The Impact of Architectural Trends on Operating System Performance", in *Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[94]. Y. Saito and B. Bershad, "A Transactional Memory Service in an Extensible Operating System", in *Proceedings of 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[95]. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", in *Proceedings of 1995 USENIX Annual Technical Conference, Portland, OR*, pp. 119-130, June 1985.

[96]. P. Sarkar, S. Uttamchandani, K. Voruganti, "Storage Over IP: When Does Hardware Support Help?", in *Proceedings of Second USENIX File and Storage Symposium (FAST'02)*, pp. 231-244, San Francisco, CA, March 2003.

[97]. M. Seltzer and Y. Endo and C. Small and K. Smith, "Dealing with Disaster: Surviving Misbehaved Kernel Extensions", in *Proceedings of 1996 USENIX Symposium on Operating System Design and Implementation*, Seattle, WA, October 1996.

[98]. Schlosser, S., Griffin, J., Nagle, D., Ganger, G., "Designing Computer Systems with MEMS-based Storage". In P*roceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Cambridge, MA, November 2000.

[99]. F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", in *Proceedings of First USENIX Conference on File and Storage Technologies (FAST'01)*, Monterey, CA, January 2002.

[100]. I. Schoinas, M. D. Hill, "Address Translation Mechanisms in Network Interfaces", in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NE, February 1998.

[101]. P. Shivam, J. S. Chase, "On the Elusive Benefits of Protocol Offload", in *Proceedings of Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, Karlsruhe, Germany, August 2003.

[102]. C. Small, M. I. Seltzer, "A Comparison of OS Extension Technologies", in *Proceedings of 1996 USENIX Annual Technical Conference*, pp. 41-54, San Diego, CA, 1996.

[103]. A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network", in *Communications of the ACM*, pp. 246-260, April 1982.

[104]. E. Speight, H. Abdel-Shafi, J. Bennett, "Realizing the Performance Potential of the Virtual Interface Architecture", in *Proceedings of 13th International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[105]. R. Steward, C. Metz, "SCTP: New Transport Protocol for TCP/IP", in *IEEE Internet Computing*, pp. 64-69, November 2001.

[106]. M. Stonebraker, Operating System Support for Database Management, in *Communications of the ACM*, 24(7):412-418, July 1981.

[107]. C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, P. R. Varker, "The SP-2 High-Performance Switch", in *IBM Systems Journal* 34(2):185-204, 1995.

[108]. M. Thadani, Y. Khalidi, “An Efficient Zero-copy I/O Framework for UNIX”, SMLI TR95-39, Sun Microsystems Lab, Inc., May 1995.

[109]. C. Thekkath, H. Levy and E. Lazowska, “Separating Data and Control Transfer in Distributed Operating Systems”, in *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 2-11, San Jose, CA, October 1994.

[110]. C. Thekkath and H. Levy, “Limits to Low-Latency Communication on High-Speed Networks“, in *ACM Trans. on Computer Systems*, 11(2):179-203, 1993.

[111]. Virtual Interface Architecture Specification, Version 1.0, `http://www.viarch.org`, December 1997.

[112]. U. Valhalia, “UNIX Internals: The New Frontiers”, Prentice Hall, NJ, 1996.

[113]. T. von Eicken, A. Basu, V. Buch and W. Vogels, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing”, in *Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pp. 40-53, Copper Mountain Resort, CO, December 1995.

[114]. T. von Eicken, D. Culler, S. Goldstein, K. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation”, in *Proceedings of 19th Annual Symposium on Computer Architecture (ISCA)*, pp. 256-266, Gold Coast, Australia, May 1992.

[115]. S. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, W. Weihl, "Optimistic Active Messages: A Mechanism for Scheduling Communication and Computation", in *Proceedings of 5th Symposium on Principles and Practices of Parallel Programming*, pp. 217-226, April 1995.

[116]. B. Welch, “The File System Belongs in the Kernel”, in *Proceedings of the 2nd USENIX Mach Symposium*, pp. 233-250, November 1991.

[117]. M. Welsh, A. Basu and T. von Eicken, “Incorporating Memory Management into User-Level Network Interfaces”, in *Proceedings of Hot Interconnects V*, pp. 27-36, Stanford, CA, August 1997.

[118]. M. Welsh, D. Culler, E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[119]. J. Wilkes, Keynote talk at USENIX Conference on File and Storage Technologies, March 31, 2003, San Francisco, CA.

[120]. T. Wong, J. Wilkes, "My Cache or Yours? Making Storage More Exclusive", in *Proceedings of 2002 USENIX Annual Technical Conference*, pp.161-175, Monterey, CA, June 2002.

[121]. Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, K. Li, "Experiences with VI Communication for Database Storage", in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pp. 257-268, May 2002, Anchorage, AK.

[122]. Y. Zhou, J. Philbin, K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches", in *Proceedings of 2001 USENIX Annual Technical Conference*, pp. 91-104, Boston, MA, June 2001.

[123]. G. Zipf, "Human Behavior and Principle of Least Effort", Addison-Wesley Press, Cambridge, MA, 1949.