

Efficient Remote Block-level I/O over an RDMA-capable NIC

Manolis Marazakis, Konstantinos Xinidis, Vassilis Papaefstathiou, and Angelos Bilas

Institute of Computer Science (ICS),
Foundation for Research and Technology-Hellas (FORTH),
Heraklion, Greece GR71110
e-mail: {maraz,xinidis,papaef,bilas}@ics.forth.gr

ABSTRACT

Modern storage systems are required to scale to large storage capacities and I/O throughput in a cost effective manner. For this reason, they are increasingly being built out of commodity components, mainly PCs equipped with large numbers of disks and interconnected of high-performance system area networks. A main issue in these efforts is to achieve high I/O throughput over commodity, low-cost system area networks and commodity operating systems.

In this work, we examine in detail the performance of remote block-level storage I/O over commodity, RDMA-capable network interfaces and networks. We examine the support that is required from the network interface for achieving high throughput. We also examine in detail the overheads associated in kernel-level protocols for networked storage access. We find that base system performance is limited by (a) interrupt cost, (b) request size, and (c) protocol message size. We examine the impact of techniques to alleviate these factors and find that our techniques combined can improve throughput by up to 100% over a simpler unoptimized configuration. Our current prototype is able to achieve a throughput of about 200 MBytes/s over a network that is capable of delivering about 500 MBytes/s. We identify major limiting factors, mostly at the I/O target-side.

Keywords

Networked storage, Block-level I/O, Remote DMA, Performance evaluation

1. INTRODUCTION

Currently, storage system architectures are undergoing a transition from directly- to network-attached. Traditional scalable storage systems employ high-end storage controllers that provide connectivity to large numbers of disks and are able to deliver high throughput. In addition, such controllers usually perform many (if not most) storage management functions, such as volume management, encryption, fault tolerance, and compression. These storage controllers have evolved over time to high-end storage area networks, such as fibre channel interconnects and systems built around them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS06, June 28-30, Cairns, Queensland, Australia.

Copyright C 2006 ACM 1-59593-282-8/06/0006 ...\$5.00.

Although today fibre channel systems offer high scalability to large numbers of disks and thus, on-line storage capacity, and high-throughput, they are costly. Furthermore, extending such systems with new functions is becoming more and more a significant problem due to (1) the increased needs for tailoring storage systems to application needs at short cycles, and thus at low cost, and (2) the effort associated with designing and implementing new features in custom storage controllers [37].

To deal with these issues, future storage systems will be based to a large extent on commodity system area networks, such as Infiniband [1], Myrinet [9], and PCI-Express/AS [30, 38, 26]. Such interconnects are the subject of research aiming at high scalability and performance of compute clusters. Being able to use the same interconnect for providing access to large amounts of online storage as well as high throughput will result in improved cost-efficiency.

Thus, there are currently efforts underway to examine the feasibility of attaching storage to large systems through system area networks. The proposed architectures usually attach 4-32 (low-end) SATA disks to storage controllers that are similar to today's PCs through (low-end) disk controllers. These *storage nodes* are then attached to a system area network accessible by application servers (Figure 1). Although variations of this architecture exist, overall most proposals follow the basic trend of building on commodity components.

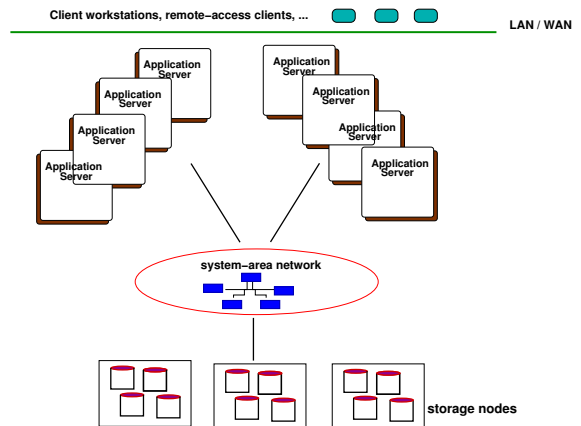


Figure 1: Overall System Organization.

However, this approach has two fundamental challenges. First, the architecture of the system changes from a *centralized* system where a high-end storage controller has global view of all I/O requests to a *distributed* architecture, where no single component has global knowledge. This shift in the architecture presents challenges: (a) providing scalable *shared* access to the available storage, and (b) extending the storage subsystem with functionality that

requires maintaining *metadata*.

Second, the performance of such systems lags significantly behind what is possible in directly attached high-end storage systems. Today, most storage systems based on system area networks use 1 GBit/s interconnects and are thus limited in the throughput they can provide from a single storage node. Besides the maximum capacity of the network links (and the overall network) used, another significant limitation is the I/O protocol stack at the edge of the system, both the application server (initiator) and the storage node (target). The proposed architectures rely not only on commodity hardware components to improve cost-efficiency, but on commodity software as well. Thus, the I/O protocol stack at the target and the initiator usually runs in the context of a commodity operating system and on the host CPU. Recent work has shown that these impose limitations in terms of the throughput they can provide [35, 36]. Most of this work has examined TCP/IP-type communication protocols [11], demonstrating significant limitations in today’s I/O architectures.

Finally, when designing the I/O protocol stack, a main issue is the division of features between the network interface and the software that runs on the initiator and target. Previous work has resulted in network interfaces that minimize copies by supporting virtual to physical address translation, minimize latency by allowing direct user access to the network interface, and minimize host CPU overhead by eliminating user-kernel context switches [15, 6, 39, 29, 34, 13]. However, these approaches have resulted in network interfaces that employ complex state machines and require large amounts of resources (both memory and CPU). Thus, there is not yet a clear understanding of the I/O performance that can be achieved on high-speed interconnects, the level of support required at the network interface level for storage I/O, and the limitations that may be imposed by initiator and target architectures on the performance of the I/O protocol stack as network speed increases.

We examine in detail the performance of the I/O path in commodity initiator and target architectures, and show limitations in achieving throughput similar to directly attached storage. We use a custom-build system area network that allows us to tune the support in the network interface. Our network is capable of about 500 MBytes/s throughput, using support for RDMA operations but no direct, user-level access. We believe that the features used in our network interface can be provided in most network interfaces at minimal cost.

We design and implement an I/O protocol stack in the Linux kernel that makes remote storage nodes appear as local disks. We analyze the overheads associated with the basic I/O path and identify system bottlenecks. We examine the effectiveness of three techniques in alleviating these bottlenecks and improving system throughput: interrupt silencing, cooperative batching of requests, and elimination of small messages. We find that each technique is able to improve throughput by up to 50% compared to the base system. When combined, the throughput is improved by up to 100% over a simpler configuration. However, we observe high CPU utilization levels, especially at the I/O target node. Moreover, we identify a further limiting factor, due to the high aggregate interrupt count (both storage controller and NIC related).

The rest of this paper is organized as follows. Sections 2 and 3 present our experimental platform and the design of the I/O path. Section 4 presents our performance evaluation. Section 5 surveys related work. Finally, Section 6 concludes the paper, summarizing the results obtained so far and outlining our plans for further work.

2. EXPERIMENTAL PLATFORM

In our work we rely on a custom, low-latency, high-throughput NIC that allows us to perform memory-to-memory transfers from

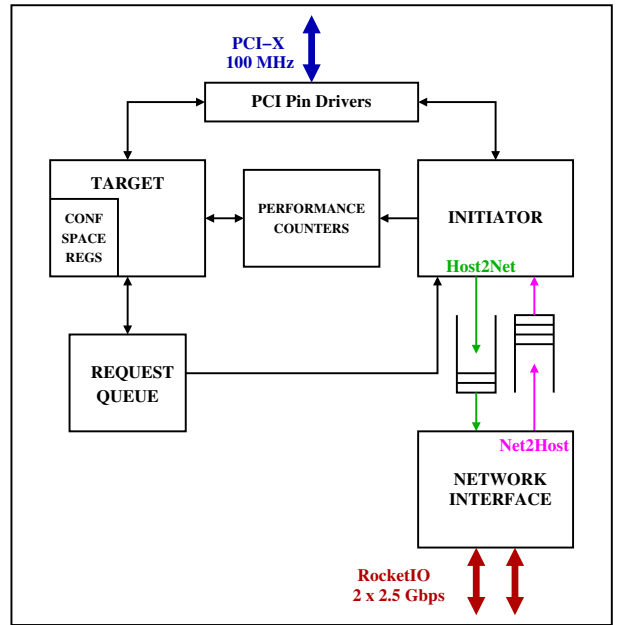


Figure 2: NIC architecture.

initiators to targets in a SAN environment. The NIC has been developed in-house to allow both detailed measurements of all aspects of communication in the I/O path as well as customization of communication primitives and operations. Next, we provide an overview of the NIC architecture and implementation [19]. The NIC is implemented as a 64-bit, 100-MHz PCI-X [27] peripheral, within a Xilinx VirtexII Pro FPGA. The design of the NIC is outlined in Figure 3. Figure ?? depicts the major structures (hardware as well as software) that concern us in the experiments presented in this paper.

At the physical layer, our NIC uses a pair of Rocket I/O serial links [3]. The serial links, each capable of 2.5 GBits/s (3.125 GBauds) full-duplex running at 78.125 MHz, are controlled by an FPGA-based transceiver controller. The two serial links are *bonded* by the NIC to appear as a single network link, as is the case in all high-end network interfaces today that use multiple physical links. Each packet is transferred over both links, multiplexing data at the byte level. The NIC applies a credit-based scheme for network flow control [22], to prevent overflowing the receiver’s NIC buffer, thus preventing loss of data in the event of heavy network load. The host-NIC DMA engine transfers 64-bit data words in bulk, to and from the host’s memory. A pair of FIFO queues (1024 entries, of 64-bit words) provides the interface between the DMA engine and the Rocket I/O transceivers. The NIC adds two 64-bit words to the payload: a header, containing the destination address, transfer size, and notification options, and a trailer, containing an error-detection checksum for the payload.

Host programs access the RDMA request queue, as well as a set of performance-related counters, via memory mappings (established with the cooperation of the NIC kernel module through the `mmap()` system call). Our current design does not support *protected* user-space access to NIC resources and is intended for use by the kernel. The main reason for this is that we are interested in exploring issues related to the I/O protocol stack, which because of transparency requirements in practice always involves the operating system kernel. However, user-space programs can still access the NIC in an unprotected manner for testing and benchmarking.

Currently the NIC supports only *RDMA write* operations. On

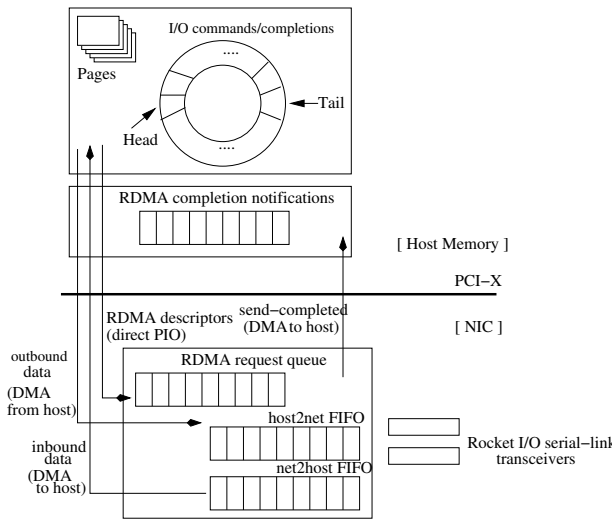


Figure 3: NIC-related structures.

the sender side, each remote write is specified through a transfer handle. The transfer handle specifies the local and remote physical addresses, the length of the transfer, and whether the transfer will generate (a) a local notification in the form of a 64-bit word written by the NIC into the sender’s memory when the transfer has finished, and (b) an interrupt at the receiver host when the last word of the message has been transferred to remote memory. Currently, the maximum message size is 4 KBytes; longer messages have to be segmented accordingly in a host library. On the receive path, data is directly transferred to the specified physical locations in memory, *without any receiver processor intervention*.

Posting a write RDMA operation requires posting the transfer descriptor to the NIC request queue, over the PCI-X target interface. In the current version of the NIC, posting a transfer descriptor requires four 32-bit PCI-X write operations in the NIC’s DMA queue: Two 32-bit words to specify the source physical address, one 32-bit word to specify the destination address; and a 32-bit word to specify the transfer size as a number of 64-bit words, various transfer flags, such as local and remote notification options, and the destination node identifier (a 7-bit *flow ID*). The NIC’s design is oriented for mostly kernel-space use, and allows for 64-bit addresses. As a provision for allowing some level of protection, the NIC can treat the 32-bit destination address word as a handle to be resolved based on pre-established bindings for remote memory regions. However, in the current version of the NIC this feature is not available; therefore, the destination word is treated as a physical address in the receiver’s memory.

3. SYSTEM SOFTWARE

The overall architecture of our I/O protocol stack is illustrated in Figure 4. The initiator and target modules that implement the remote access protocol initially establish their association over a TCP/IP socket, on a separate Gigabit Ethernet network. Then, the initiator is informed via a sequence of RDMA transfers of the actual device parameters, such as device size, block size, and sector size, and registers a block driver that mediates access to the remote device. All applications at the initiator can then access this block device as if it were a directly-attached physical block device, thus achieving *transparent* access to the remote storage available at the target. As a result, we can for example construct file systems on top of a remote block device. The initiator’s task is to forward I/O requests to the target, receive I/O completion notifications, and final-

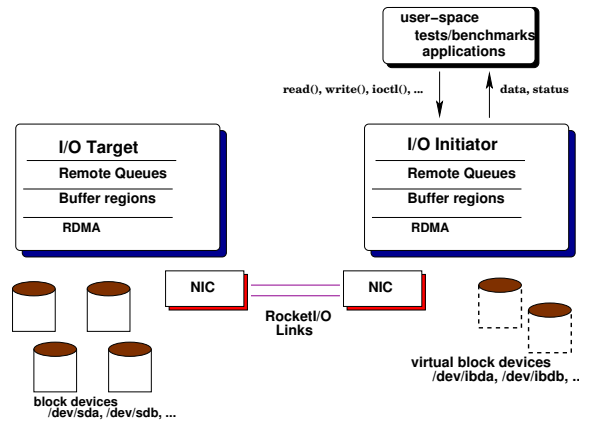


Figure 4: Overview of software architecture for remote block-level storage access.

ize the I/O requests by invoking the appropriate call-back function for each block request.

3.1 Base Protocol

The base remote block I/O protocol makes use of the RDMA operations to forward block read and write requests from the initiator to the target. To use RDMA operations the initiator and the target maintain a pair-wise command queue and a pool of data buffers. The command queue at the target stores I/O commands, produced by the initiator. The command queue at the initiator stores I/O completion notifications, forwarded by the target upon completing I/O commands that it has dequeued from its command queue. The command queue at both the initiator and the target is managed as a circular queue. The local host only consumes entries from the command queue whereas the remote host only adds entries to the command queue. To enqueue an entry in a remote command queue, each host uses two RDMA writes. The first transfers a fixed-size command structure and the second advances the queue tail pointer. To dequeue an entry, an RDMA write is issued to update the queue head pointer. Currently, the size of command queues is determined statically at initialization time.

3.1.1 Send Path

On the send path, upon completion of each out-bound RDMA transfer, the issuer needs to deallocate the local handle. Traditionally, NICs notify the host with an interrupt when the local RDMA operation has finished. However, interrupts incur high overhead. Instead, we require (and use) NIC support for *local notifications*: When a local DMA operation completes, the NIC writes back (via DMA) a notification (in the form of a 64-bit word) to a specified location in host memory. The issuer, instead of spinning on this notification word to eagerly free the transfer handle, checks lazily for free transfer handles, when more requests are posted later. This allows the sender to only incur the overhead of posting the transfer handle during issuing RDMA write operations.

3.1.2 Receive Path

On the receive path, messages are delivered directly to host memory without any receive processor intervention. However, when a message is marked with the interrupt flag, the NIC issues an interrupt to the receiving host CPU. The interrupt flag is required for messages that manipulate queue head/tail values, so that the I/O target and the I/O initiator consume command structures (I/O requests or completion notifications, respectively) from their respective queues.

3.1.3 I/O Buffer Management

Data blocks involved in read and write block requests are directly transferred to remote buffers with RDMA writes, without going through the command queue. Each command queue is associated with one or more separate sets of page-sized data buffers. Since there is a hard limit on the number of contiguous pages that can be allocated by the OS kernel, we allocate a fixed number of buffer regions, i.e. sets of contiguous pages. The number and size of buffer regions are currently fixed throughout the life-time of an initiator's association with a target, and, moreover, it is the same for all initiators regardless of their level of I/O activity. In the case of remote write I/O operations, the initiator selects one of the available reserved pages and transfers in that page the data to be written to block storage. The target uses this page for issuing the requested I/O operation. Upon completion, the target notifies the initiator of the outcome and the initiator marks the page as available for use in subsequent requests. In the case of remote read I/O operations, the initiator indicates the address of the page, as specified by the local OS kernel's block I/O framework, where the data from the remote block storage should be placed. The target reserves a local page to issue the requested I/O operation. Upon completion, the target transfers via remote DMA the data from its local page to the initiator's page.

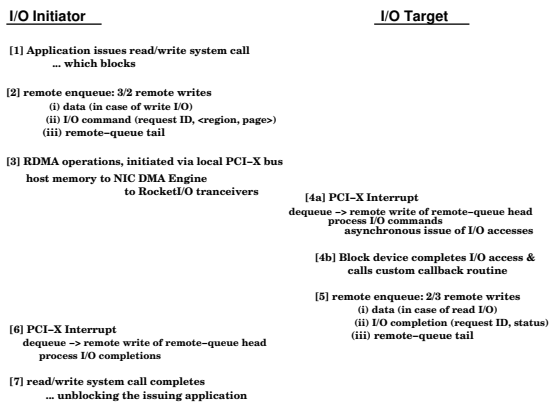


Figure 5: Actions and messages in the remote I/O protocol.

Thus, for each read request our base protocol design uses two RDMA writes from the initiator to the target to post the read command (command and tail pointer), two RDMA writes from the target to the initiator to post the reply (command and tail pointer), and one RDMA writes from the target to the initiator to return the actual data. Similarly five RDMA writes are generated for write requests (three from initiator to target and two from target to initiator). Each time an I/O request or completion message is dequeued for processing, an additional RDMA writes is issued to update the remote-queue's head-pointer. All RDMA writes generate a sender-side notification, so that the device driver can keep track of completed transfers and re-use the corresponding descriptor in the NIC's request queue (see Section 2 for details). Finally, each tail-pointer update is marked to generate an interrupt at the receiving side. This interrupt provides the means for the target to retrieve and process I/O requests, and for the initiator to retrieve and process I/O completions. Figure 5 shows the sequences of messages in our base remote I/O protocol.

3.2 Interrupt silencing

Previous work has shown that interrupt cost can be extremely high in high-performance networks [35]. Given that the I/O path in the kernel relies on interrupts for I/O request completion, it is

important to reduce the number of interrupts.

To minimize the number of interrupts asserted at each node we employ an *interrupt silencing* technique, as follows. Interrupt handlers are organized in two parts, a non-interruptible part that runs as soon as the interrupt is delivered and an interruptible one that may be scheduled for execution through the system scheduler. In the Linux kernel, these are the top and bottom-half handlers [10]. The top-half disables interrupt delivery from the NIC and schedules a bottom-half context (task) for performing the actual interrupt handling. In our design, we use the top-half handler to only schedule an interruptible bottom-half handler. The bottom-half handler, when scheduled, will process all requests present in the command queue. In the meantime, the NIC may deliver additional requests, which however, will not cause additional interrupts. When the bottom-half handler has finished processing all requests in the queue, it enables NIC interrupts. Since the bottom-half handler is interruptible, we ensure that there is no race condition between enabling interrupts and returning from the bottom-half handler by re-checking for new requests in the command queue as soon as interrupts are enabled. In this manner, at high loads no interrupts are delivered as long as a node is processing other requests.

3.3 Request batching

In many systems, at the kernel level, I/O requests arrive at the block level, as page-size requests, regardless of the application request size. These requests are issued to the disk queue; then, a kernel task queue is scheduled for execution [10], forcing the underlying storage devices to process any pending I/O requests. In performing remote I/O at the block level, although the application at the initiator side may have specified a large number of I/Os, e.g. when using a large request size, the target sees individual small I/O requests. Thus, the target may either invoke the disk scheduler too early if it is eager or too late if it waits for more requests to arrive, affecting disk seek time and efficiency. We use a simple technique where each I/O request from the initiator carries information about whether more requests are pending in the initiator and will follow. The target uses this information to either invoke the disk scheduler when no additional requests are expected, or delays invocation until more requests have arrived. In our evaluation we explore the impact of varying the number of pending I/O requests before invoking the disk scheduler.

3.4 Implementation

We implement our system under RedHat Linux 9, updated to run with version 2.4.30 of the Linux kernel. We divide the remote I/O path framework into two major parts, corresponding to the two sides in an I/O operation, initiator and target. Our protocol is implemented as block-level driver module for the initiator and target, which is dynamically loaded and linked with the kernel. The Linux 2.4 kernel series [10] forces I/O accesses to be issued as block-sized transfers, regardless of the request size specified by the user-space application that is going to consume the data. The block size is in most cases 1024 bytes. The block I/O framework in the kernel allows for merging of I/O requests that reference contiguous blocks, up to a limit of one page (4 KBytes). Moreover, the block I/O framework can group together I/O requests, provided that the block device has registered a request queue with the kernel's block I/O framework. Taking advantage of these kernel features, remote block-level requests reach our remote I/O framework as chains of one or more consecutive requests, each at the granularity of a 4-KBytes page.

The initiator module reserves one major device number, and handles all I/O accesses for this major number, regardless of the minor number specified by the application. The target module is regis-

tered as a character device at its host, and exports a set of bindings for block devices. Each binding is defined by a pair of 8-bit integers (major, minor) that uniquely identify the block device to its host. The target module reserves one major device number and listens (on a TCP/IP socket) for bind requests from initiators. Each of the associated minor numbers (currently up to a limit of 8) can be mapped to a different physical block device. A bind request from an initiator includes a minor number to be interpreted by the target, allowing the initiators to bind to different physical block devices.

4. PERFORMANCE EVALUATION

Next, we present performance measurements from our prototype. We first examine base communication performance, and then look into remote block I/O overheads and optimizations.

4.1 Experimental Setup

The prototype we use in our experiments consists of two Dell 1600SC servers, each with a single Intel Xeon CPU, running at 2.4 GHz, 512 MBytes of main memory, and two 64-bit PCI-X slots running at 100 MHz. One of the nodes serves as I/O target, with 8 directly-attached SATA Western Digital disks (WD-800), connected to a BroadCom RAIDcore controller. Total capacity is 614.1 GBytes. On the I/O target node, the storage controller and the NIC occupy slots on the same (only available) 100 MHz PCI-X bus. The two nodes have a dedicated IDE system disk, and two types of interconnects: a Gigabit Ethernet adapter for system administration and monitoring, and our custom NIC for all data transfers.

In our block-storage experiments we use a RAID-0 volume for all disks at the storage node. We build this volume using the Linux multi-disk (MD) driver with the stripe size set to 128 KBytes. The initiator binds to the storage node and its single (RAID) volume through our I/O path. The remote volume appears locally as a regular block device, indistinguishable from local devices.

In our evaluation we use mostly the xdd benchmark [18] with the `-dio` option to bypass the initiator’s buffer cache. We vary the request size from 4 KBytes up to 1 MBytes for both read and write accesses. Each experiment transfers a total of 4 GBytes between the initiator and the target. We also present measurements from filesystem-based experiments.

To examine the impact of our optimizations, we present results for different system configurations:

- *LOCAL*: Directly-attached disks
- *BASE*: Remotely-attached disks, base protocol
- *BASE1*: + Interrupt silencing
- *BASE4*: + I/O batch-factor of 4
- *BASE16*: + I/O batch-factor of 16
- *BASEreqsz*: + I/O batch-factor of request size

4.2 Base communication

The first set of measurements characterizes the capabilities of the communication network we use and its basic primitives.

The theoretical maximum throughput of the host-NIC DMA engine is one 64-bit word at every 100-MHz PCI-X clock cycle or 762.9 MBytes/s, assuming zero bus arbitration and protocol overheads. The theoretical maximum throughput for the pair of Rocket I/O links is one 64-bit word at every 78.125-MHz Rocket-I/O clock cycle or 596 MBytes/s. Therefore, the maximum theoretical end-to-end throughput is limited to that of the network links. Figure 6 summarizes the results from two simple user-space benchmarks

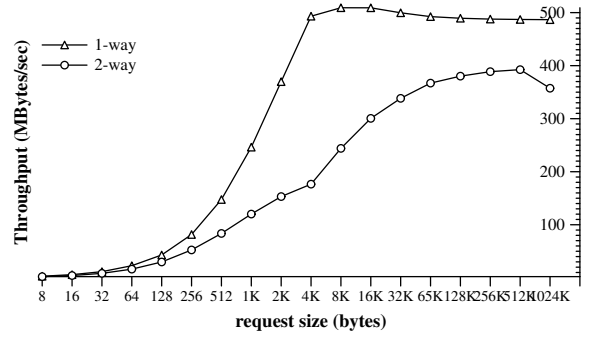


Figure 6: Base communication performance. Results represent the average over 10000 transfers.

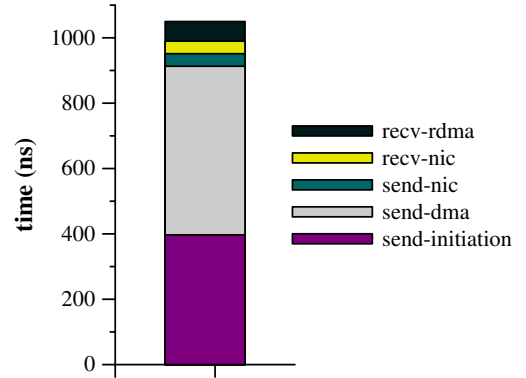


Figure 7: Breakdown of end-to-end latency for an 8-byte message.

that measure the maximum achievable throughput. One of the benchmarks initiates one-way transfers without waiting for a response from the receiver, whereas the second produces a two-way, ping-pong traffic pattern. In all cases, all messages sent are of the same size. As mentioned above, when message size exceeds 4 KBytes the host library breaks the message to 4-KByte segments.

We use hardware counters at the NIC level to examine the behavior of the host-NIC DMA engine. Each 32-bit CPU-to-NIC write operation requires on the order of 10 PCI-X cycles. Thus, initiating a single RDMA write operation requires about 40 PCI-X cycles or about 400ns. Moreover, it takes on the order of 50 PCI-X cycles (500 ns) for any DMA operation from the host’s local memory to begin transferring data (PCI-X read). These delays dominate the latency for small transfer sizes, as shown in Figure 6. After the initial delay, the DMA engine is capable of reading one 64-bit word per PCI-X cycle and placing it in the transceiver’s out-bound FIFO. If there were no further disruptions, this would result in the maximum transfer rate of 762.9 MBytes/s. For a DMA transfer of 4 KBytes (i.e. 512 64-bit words) from a host’s memory, we measure a delay of 577 PCI-X cycles, of which 512 cycles actually transfer data words (89% utilization of PCI-X cycles). Of the remaining 65 cycles, 8 cycles are attributed to bus arbitration and PCI-X protocol phases and 57 cycles are consumed until we receive the first data word. This last time interval is the duration for a *PCI-X split transaction* to complete and we have found it to be almost constant, in the range 45-60 cycles, regardless of the DMA transfer size. Figure 7 shows a breakdown of the one-way latency for a small, 8-byte message (payload). The overhead is divided in the following components: send-initiation, send-DMA, send-NIC, recv-NIC, recv-DMA. The *send-initiation* component includes the

PCI-X overhead during posting the transfer descriptor. The *send-DMA*, *recv-DMA* components include all PCI-X overhead related to the data transfer itself. Finally, *send-NIC* and *recv-NIC* is the time spent in the send and receive NICs. We measure these components using the corresponding cycle counters on the NIC boards. We see that the *send-path* is the most expensive part for this type of short transfer. The *send-initiation* component accounts for 40% of the overall latency, while the *send-dma* component accounts for 51.6%. Using the on-board cycle counters, we find that approximately 46 PCI-X cycles out of the *send-dma* component are PCI-X related (PCI-X *split* duration). This initial delay occurs before transferring any data from the sender host’s memory, and can only be amortized with larger transfer sizes.

4.3 Remote block I/O

Our second set of measurements focuses on the performance of remote *block I/O* accesses. In our setup, each of the 8 SATA disks attached to our storage node is capable of a sequential I/O rate in the order of 60 MBytes/s for both read and write accesses. Figures 8(a,b,c,d) present throughput measurements for four different access patterns: sequential read, sequential write, sequential access with 70:30 read-write, and random access with a 70:30 read-write ratio. Figures 9(a,b,c,d) present latency measurements.

For sequential I/O, the remote I/O configurations achieve a peak I/O throughput of 162 MBytes/s for read accesses, and 184 MBytes/s for write accesses, approximately one-third and one-half of the directly-attached storage configuration, respectively. With directly-attached storage, the corresponding peak rates are 480 MBytes/s (reads) and 360 MBytes/s (writes). We observe that the latency per I/O operation is 2-3 times higher in the remote I/O configurations.

We also examine CPU utilization levels, at both the initiator and the target. At the initiator, for the peak read I/O throughput, we observe utilization levels of 20-25%, whereas for the peak write I/O throughput we find that utilization reaches 50%. At the target, CPU utilization reaches or exceeds 90%. This suggests that target CPU utilization may be a significant limitation to remote I/O performance at even higher network link speeds.

4.4 Interrupt silencing

Interrupt silencing aims to prevent the NIC from asserting an interrupt for each and every incoming message that has been marked to generate an interrupt at the receiver host. Figure 8 shows that employing interrupt silencing results in up to three and four times better throughput for sequential reads and writes respectively, for large block sizes (BASE vs BASE1 configurations).

4.5 Request batching

Setting the *batch-factor* parameter to a value greater than one allows for interleaving I/O request completion messages with new I/O requests, thus taking better advantage of available NIC cycles. We observe that by varying the *batch-factor* from 1 up to 16 the read I/O throughput scales from 95 MBytes/s to 162 MBytes/s, whereas the write I/O throughput scales from 128 MBytes/s to 184 MBytes/s. Latency per I/O operation is not negatively affected, however CPU utilization at the I/O initiator increases by up to 18%.

For the sequential read-write mix, remote I/O configurations achieve 70-80% of the I/O throughput for the directly-attached storage configuration with up to 10% higher latency per I/O operation. Again, CPU utilization levels are considerably higher (up to 17%), but the difference is not as pronounced as in the sequential I/O experiments.

For the random read-write mix, the remote configurations achieve up to 85% of the I/O throughput for the directly-attached storage configuration, with up to 20% higher latency, and CPU utilization

workload	throughput (MB/sec)	latency per I/O (msec)
Seq. read	171.6 (8.3%)	6.1 (7.5%)
Seq. write	199.9 (10.1%)	5.3 (8.6%)
Seq.70R-30W	96.2 (1.9%)	10.9 (1.8%)
Rand.70R-30W	48.9 (0.4%)	21.5 (0)

Table 1: I/O throughput and latency measurements using *xdd*, after short message elimination.

configuration	storage controller	NIC	Total
LOCAL/READ	24047	-	24047
BASE16/READ	27223	50193	77426
LOCAL/WRITE	33658	-	33658
BASE16/WRITE	49712	193788	243430

Table 2: Interrupt handler activations, at the I/O target node, for sequential read and write accesses (request size: 1 MB, transfer volume: 4 GB).

up to 11%.

In comparing the remote I/O configurations, we find no performance advantage in setting the *batch-factor* parameter to values higher than 16. For the random read-write mix, increasing the *batch-factor* parameter negatively affects I/O throughput and latency per I/O operation.

4.6 Small Protocol Messages

By using a user-space benchmark that emulates the communication pattern of our remote block-level I/O protocol, we have come to the conclusion that a limiting factor for the achievable remote I/O throughput is the presence of short RDMA transfers, needed in our protocol for managing queues. Thus, an important direction for optimization is to examine how these messages can be minimized, or even eliminated.

Specifically, we adapted the I/O protocol so that queue head and tail values are not explicitly updated, via RDMA transfers. Instead, command and completion messages incorporate a *pending* flag, so that the recipient can identify the messages that need to be processed; effectively, the recipient determines on its own how to update its queue tail. Moreover, each message includes the queue head value at its originator, so that the initiator and the target keep track of each other’s progress in processing the messages in their corresponding queues. A new command or completion will only be enqueued if the sender can determine that the recipient’s queue is not full. Table 1 shows representative results from applying this optimization, for the BASE16 remote I/O configuration under the four workloads used in the evaluation. The request size is set to 1MB, and the data transfer volume is 4 GB. The impact of this optimization is shown as a percentage relative to the throughput and latency measurements of Figures 8 and 9. For sequential reads and writes, and with varying request sizes, the improvement in throughput is on the order of 10%, with a corresponding improvement in the latency per I/O operation.

4.7 Aggregate Interrupt Counts

The performance of the I/O target node is directly affected by the combined count of interrupt handler activations for both the NIC and the storage controller. Representative data points are shown in Table 2, for sequential I/O workloads (generated using *xdd*) and the request size set to 1 MB, with a total transfer volume of 4 GB. The counts of interrupt handler activations due to the storage controller were obtained from `/proc/interrupts`.

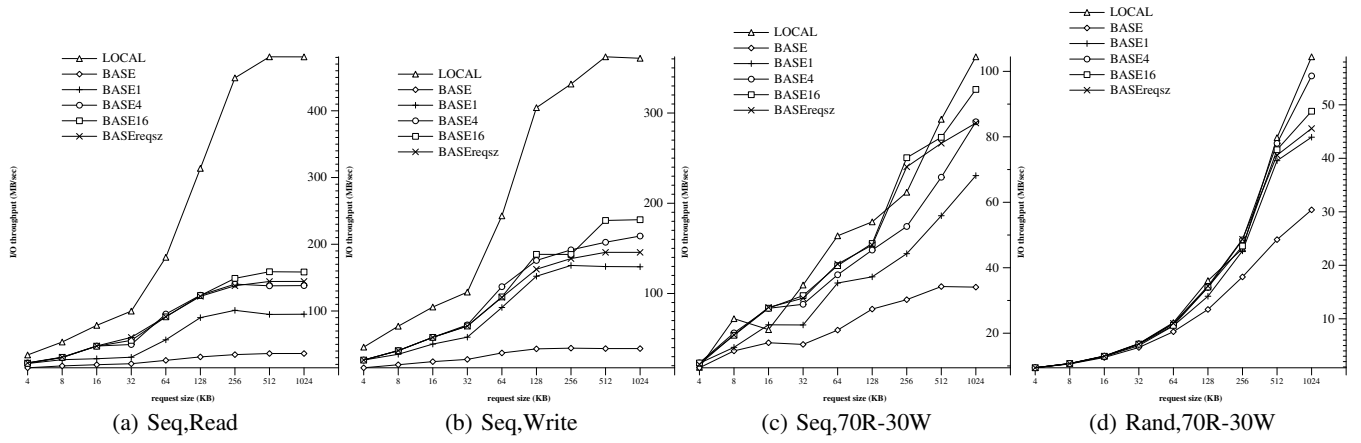


Figure 8: I/O throughput measurements using xdd.

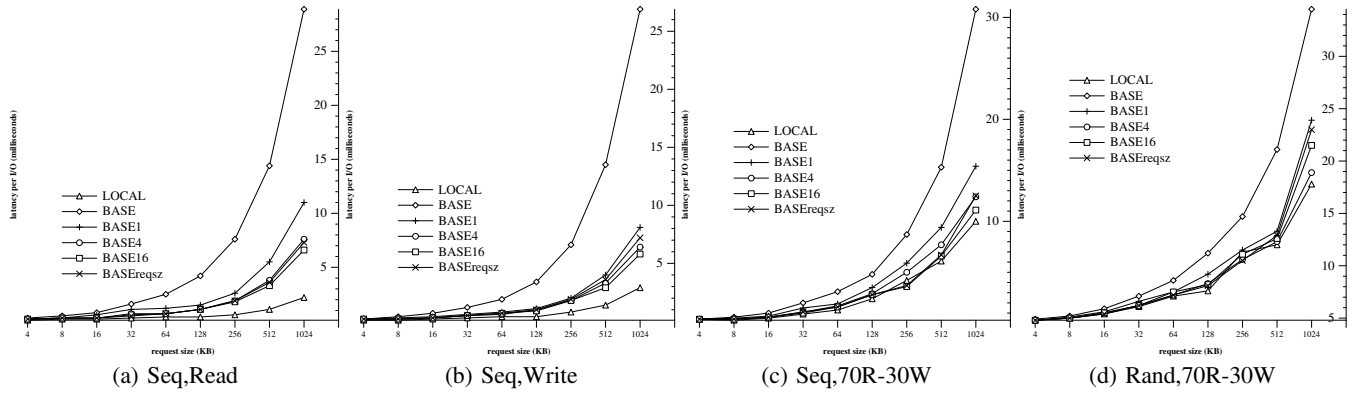


Figure 9: I/O response time measurements using xdd.

In the remote block I/O configuration, the I/O target not only performs additional work (in response to interrupts from the NIC), but also suffers from a significant increase in the number of interrupts from the storage controller (+13.2% for reads, +47.6% for writes). This behaviour implies that the NIC’s interrupts interfere with some form of “batching” optimization applied by the storage controller, leading to a reduced I/O throughput level. Moreover, the I/O target node’s CPU utilization reaches or exceeds 90% in these experiments, placing a hard limit to the achievable I/O throughput.

4.8 Filesystem I/O results

Table 3 presents results from a sequence of filesystem operations. The filesystem is built on top of a remote block device. The table shows the overall running time for each step in the following script, for the *LOCAL* and *BASE16* system configurations. It also shows (in parentheses) the system time for each step.

- `mkfs`: Create filesystem on top of the block storage volume, using the `mkfs` command-line utility. We build a filesystem of `reiserfs` [2], with block-size equal to 4 KBytes.
- `mount`: Mount the filesystem using the `mount` command-line utility.
- `copy`: Copy a compressed `tar` archive file from the dedicated IDE system disk to the filesystem. Specifically, we copy the source tree of the Linux kernel version 2.4.30, a `tar` file compressed using `bzip2`. The file size is 29.7 MBytes.

- `extract`: Extract the files from the compressed `tar` file. This step produces a 184.9 MBytes directory/file tree.
- `archive`: Create a `tar` archive file that contains the directories and files extracted in the previous step.
- `compress`: Compress the `tar` archive file produced in the previous step, using the `bzip2` command-line utility.
- `remove`: Recursively remove all directories and files in the filesystem.
- `umount`: Un-mount the filesystem, by executing the `umount` command-line utility.

An important difference from the block I/O experiments is that the filesystem is accessed through the kernel’s VFS layer [10] and block I/O operations are mediated by the buffer cache. Thus, issuing read/write operations only results in RDMA transfers in cases of misses in the buffer cache. Moreover, the kernel’s VFS layer caches metadata for the filesystem, thus helping the I/O initiator node to avoid RDMA transfers.

We observe that for certain operations the runtime for remote I/O configuration is actually *less* than that of the directly-attached storage configuration. The `archive` operation is particularly expensive under the remote I/O configuration, as it involves not only reading and processing data blocks, but also reading filesystem metadata for all files and directories. Therefore, the subsequent `compress` operation is able to retrieve a large number of data blocks from the

step	LOCAL	BASE16
mkfs	20.386 (0.270)	10.971 (0.330)
mount	0.303 (0.050)	16.323 (0.070)
copy	0.612 (0.220)	0.667 (0.260)
extract	29.610 (4.630)	30.950 (5.060)
archive	14.829 (2.550)	94.374 (2.160)
compress	83.710 (0.630)	26.667 (0.370)
remove	2.708 (1.480)	2.708 (1.410)
umount	0.172 (0.110)	0.275 (0.120)

Table 3: Filesystem test results for archive file (containing a hierarchy of directories and short files). Each column shows running time and (in parentheses) system time. All times are in seconds.

step	LOCAL	BASE16
copy	0.157 (0.270)	0.163 (0.160)
uncompress	43.932 (3.560)	45.337 (3.370)
scan	81.524 (0.640)	83.784 (2.210)
compress	496.400 (1.160)	501.013 (2.640)

Table 4: Filesystem test results for trace-file (a single large file, that is processed in a sequential scan). Each column shows running time and (in parentheses) system time. All times are in seconds.

buffer cache at the I/O initiator node, rather than having to issue RDMA transfers.

Table 4 reports results from a sequence of filesystem operations that manipulate a 515.4-MByte trace-file. Unlike the previous experiment, where we create and manipulate a hierarchy of directories containing short files, in this experiment the filesystem has to host a single large file. Another important difference is that in this experiment we process the trace-file in a sequential scan, resulting in mostly sequential I/O accesses. The sequence of operations in this experiment is as follows:

- `copy`: Copy the compressed trace-file from the dedicated IDE system disk to the filesystem.
- `uncompress`: Uncompress the trace-file, using the `bunzip2` command-line utility.
- `scan`: Scan the trace-file, one-byte-at-a-time, using the `wc` command-line utility. The trace-file is in ASCII format and contains 2,169,400 lines, each consisting of approximately 250 characters.
- `compress`: Compress the trace-file produced in the previous step using the `bzip2` command-line utility. This step produces a file of size 23.16 MBytes.

Table 3 reports the running-time, as well as the system time (in parentheses), for each operation.

For all steps in this test script, the running time for the remote I/O configuration is within 1-4% of the directly-attached storage configuration. This test involves sequential read/write I/O, especially for the `scan` step. However, by having to actually *consume* (process) the data, the I/O throughput advantage of the directly-attached configuration does not become as pronounced as in the previous test.

Finally, Table 5 shows results from a sequence of filesystem operations that result from the execution of the `Postmark` benchmark [20]. `PostMark` creates a set of files with random sizes within

	LOCAL	BASE16
transactions per second	9.0	7.0
read xput (MBy/s)	26.48	22.32
write xput (MBy/s)	37.64	31.71
runtime (system time)	590.89 (189.14)	669.01 (220.30)

Table 5: Results from the Postmark benchmark (1000 files, 5000 transactions). All times are in seconds.

a configurable range, and then performs a number of transactions. Each transaction consists of a randomly-chosen pairing of file creation or deletion with file read or append. We setup `Postmark` to perform 5000 transactions over a set of 1000 files, file size varying from 10 KBytes to 10 MBytes. There is no bias toward read/write transactions, and the read/write block size is set to 4 KBytes. With these settings, `Postmark` reads 15.2 GBytes of data, and writes 21.6 GBytes. Overall running time is higher by 13% for the remote I/O configuration. System time is about 16.5% higher, as the remote I/O configuration requires significantly more kernel-space processing than the directly-attached storage configuration. Correspondingly, read and write throughput is approximately 18% lower.

4.9 Summary

We see that I/O performance at the block level, although exceeds what has been demonstrated over 1 GBit/s interconnects, does not match the performance potential of the network link. Our results show that both dealing with interrupts, batching requests, and eliminating small messages is important, particularly for block-level I/O workloads. Finally, we show that I/O performance at the filesystem level is similar to the local configuration.

5. RELATED WORK

With the advent of clusters and their extensive use as a computational platform there has been a lot of research on scalable communication subsystems in clusters. For instance, there is a long history of previous work in improving base communication performance by enabling user-level communication, eliminating copies of data, and reducing host overheads and context switches [15, 6, 39, 29, 34, 13]. Similarly there has been work on network interface architectures and support for high-performance cluster communication [8, 7, 9, 17, 16, 1, 32]. Finally, there has been extensive work in evaluating their performance of low-latency, high-speed interconnects in various contexts [4, 23]. Our work differs from this efforts and builds on previous work in two important ways: (a) We are interested in examining the *minimum* level of hardware support that can enable high-throughput for networked storage, and (b) We focus on OS kernel-level communication, as opposed to user-level communication since storage applications require a high-degree of transparency and we analyze and optimize the storage I/O path in the kernel, which has received far less attention.

Regarding (a), previous work has proposed mechanisms for protected, direct user-level access and virtual memory translation. The first usually involves per-user virtualized request queues [13, 25] whereas the second uses a TLB-based address translation technique [12, 5]. Our work shows that an efficient queue structure that allows asynchronous posting and completions of requests is the main mechanism required. In addition, we use the ability to interrupt a remote node and to send local acks in the form local memory DMA writes from the NIC to host memory.

Regarding (b), interconnects designed for user-level access may be used by kernel-level applications as well, the increased complexity is an important consideration for application development.

Moreover, there has been efforts to provide transparent user-level access to storage [41, 42, 23, 40]. However, to benefit from aggressive NIC features requires extensive changes to either the application code itself or the supporting runtime system; In practice, we believe this is a significant concern for portability and robustness, both very important in storage applications. In contrast, the user-kernel interface is well-defined and supporting results in higher transparency. In our work, we examine in detail overheads associated with the kernel-level I/O path.

The authors in [28] use a 2 GBit/s Myrinet network as the transport in an experimental evaluation of a remote disk driver. However, they evaluate a small-scale setup with a single disk. Moreover, they do not examine the kernel-level overheads associated with accessing remote storage over system area networks. The authors in [21] present remote disk drivers over a 1.28 GBit/s Gigaset interconnect [16]. They find that their approach is more efficient than using TCP/IP as the transport layer and they are able to achieve a maximum throughput of about 30 MBytes/s. In contrast, our work targets more aggressive, next-generation architectures and systems.

In terms of the specific techniques we use, RDMA [33, 1, 14] has become a core capability for low-latency, high-throughput interconnects. The authors in [23] evaluate the performance characteristics of 3 types of RDMA-capable interconnects: Myrinet [9], Quadrics [32, 31], and Infiniband [1]. The evaluation in [23] also explores the implications of completion notification and address translation capabilities in the NIC. The evaluation in [24] shows the performance advantages of implementing an Infiniband NIC over a serial, point-to-point interface (as in PCI Express), over the more common local I/O bus architecture (as in PCI-X). This evaluation focuses mostly on MPI workloads.

Interrupt silencing has been used in the past in lower speed interconnects [42], where interrupt cost is not as important. In our work we design and implement this technique on a faster interconnect and also evaluate in detail its impact on system performance. Request batching has been used in various contexts. Our approach does not delay messages, but rather notifies the receiver that “more will follow” so it may wait before taking specific actions. Unlike previous work, we examine the effectiveness of this technique with respect to remote disk scheduling. Finally, although previous work has presented protocols for access to remote storage [42], our work quantifies the effect of various aspects in modern, serial-link based interconnects.

6. CONCLUSIONS

Networked storage systems are a main trend in providing scalable, cost-effective storage. Such systems rely increasingly on commodity nodes equipped with multiple disks and interconnected with commodity system area networks. As the throughput of system area network increases, it becomes important to examine the overheads associated with remote storage access.

This work investigates the overheads associated with performing remote I/O over commodity system area networks and operating system kernels and the type of support required to support efficient, kernel-level access to remote storage. We design a protocol for remote storage access and explore techniques for improving interrupt cost and disk request size. In our work we use a custom-built NIC that offers the following core capabilities: RDMA write, sender-side notification of RDMA write completion, and receiver-side interrupt generation. We believe that this minimal set of features should be able to support high-performance remote storage access in commodity systems.

Our current results show that while dealing with interrupts and disk request size are important, remote storage access throughput is also limited by small protocol messages. Although our network

is capable of delivering about 500 MBytes/s and the disks about 450 MBytes/s, we are currently able to achieve only an end-to-end throughput of about 200 MBytes/s (best-case results). The I/O target node exhibits extremely high CPU utilization levels. Our current understanding of this issue is this is a direct result of the overwhelming aggregate count of interrupts from both the storage controller and the NIC. Our work quantifies overheads that are not as pronounced when using lower speed interconnects, and shows that future work should aim at reducing transfer initiation costs, and reducing the aggregate number of interrupts to be processed.

Overall, we believe that presenting performance results from a real system contributes significantly to understanding the issues in achieving high throughput in commodity networked storage systems.

7. ACKNOWLEDGMENTS

We would like to thank the members of the CARV laboratory at ICS-FORTH and especially the hardware group for the useful discussions. We are indebted to George Kalokairinos, Aggelos Ioannou, and Prof. Manolis Katevenis for enabling this work by providing a working network interface. We thankfully acknowledge the support of the European FP6-IST program through the SIVSS (STREP 002075) and UNISIX (MC EXT 509595) projects, and the HiPEAC Network of Excellence (NoE 004408).

8. REFERENCES

- [1] An infiniband technology overview. Infiniband Trade Association, <http://www.infinibandta.org/ibta>.
- [2] Reiserfs. Namesys Inc, <http://www.namesys.com>.
- [3] Rocket i/o user guide. Xilinx Inc, <http://www.xilinx.com/bvdocs/userguides/ug024.pdf>.
- [4] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proc. of The 1998 Supercomputing Conference on High Performance Networking and Computing (SC97)*, Orlando, Florida, Nov. 1998.
- [5] R. Azimi and A. Bilas. mini: Reducing network interface memory requirements with dynamic handle lookup. In *Proc. of the 17th ACM International Conference on Supercomputing (ICS03)*, June 2003.
- [6] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP15)*, December 1995.
- [7] M. Blumrich, C. Dubnick, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *Proc. of The 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA2)*, Feb. 1996.
- [8] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proc. of the 21st International Symposium on Computer Architecture (ISCA21)*, pages 142–153, Apr. 1994.
- [9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovicm, and W. Su. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, 1995.
- [10] D. Bovet and D. Cesati. *Understanding the Linux Kernel*. O’ Reilly Media, 2002. 2nd Edition.
- [11] J. Chase, A. Gallatin, and K. Yocum. End-system optimizations for high-speed tcp. *IEEE Communications*, 39(4):68–74, 2001. Special issue on TCP Performance in Future Networking Environments.
- [12] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A mechanism for address translation on network interfaces. In *Proc. of The 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS8)*, pages 193–203, San Jose, CA, Oct. 1998.
- [13] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*. Stanford, CA, USA., Aug. 1997.
- [14] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The virtual interface

- architecture. *IEEE Micro*, 18(2):66–76, 1998.
- [15] T. Eicken, D. Culler, S. Goldstein, and K. Schauer. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture (ISCA19)*, pages 256–266, May 1992.
- [16] Giganet. Giganet cLAN family of products. <http://www.emulex.com/products.html>, 2001.
- [17] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proc. of the IEEE Spring COMPCON '96*, Feb. 1996.
- [18] I/O Performance Inc. The xdd i/o benchmark. <http://www.ioperformance.com>.
- [19] G. Kalokairinos, V. Papaefstathiou, A. Ioannou, D. Simos, M. Papamichail, G. Mihelogiannakis, M. Marazakis, D. Pnevmatikatos, and M. Katevenis. Design and implementation of a multi-gigabit nic anda scalable buffered crossbar switch. Technical Report TR376-04-2006, FORTH-ICS, 2006.
- [20] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., 1997.
- [21] K. Kim, J. Kim, and S. Jung. Gnbnd/via: A network block device over virtual interface architecture in linux. In *Proceedings of the 16th Annual IEEE International Parallel and Distributed Processing Symposium*, 2002.
- [22] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Proceedings of the ACM SIGCOMM Conference*, 1994.
- [23] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, and D. Panda. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, 2004.
- [24] J. Liu, A. Mamidala, A. Vishnu, and D. Panda. Performance evaluation of infiniband with pci express. *IEEE Micro*, 25(1):20–29, 2005.
- [25] A. M. Mainwaring and D. E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proc. of The 1999 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP99)*, pages 119–130, May 1999.
- [26] D. Mayhew and V. Krishnan. Pci express and advanced switching: Evolutionary path to building next-generation interconnects. In *Proceedings of the 11th IEEE Symposium on High Performance Interconnects*, 2003.
- [27] I. Mindshare and T. Shanley. *PCI-X System Architecture*. Addison-Wesley Professional, 2001.
- [28] V. Olaru and W. Tichy. On the design and performance of remote disk drivers for clusters of pcs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2004.
- [29] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): efficient, portable communication for workstatin clusters and massively-parallel processors. *IEEE Concurrency*, 1997.
- [30] PCI-SIG. Pci express. <http://www.pcisig.com>.
- [31] F. Petrini, F. E., and A. Hoisie. Performance evaluation of the quadrics interconnection network. *Journal of Cluster Computing*, 6(2):125–142, 2003.
- [32] F. Petrini, W. Feng, A. Hoisie, S. Coll, and F. E. The quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.
- [33] J. Pinkerton. The case for rdma, 2002. RDMA Consortium, http://www.rdmaconsortium.org/home/The_Case_for_RDMA-02053.pdf.
- [34] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In *PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA, March 30 – April 3 1998*.
- [35] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.
- [36] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. Eta: Experience with an intel xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, 2004.
- [37] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: Enterprise storage systems on a shoestring. In *Proc. of the ASPLOS 2004*, Oct. 2004.
- [38] A. S. I. SIG. Asi technical overview. <http://www.asi-sig.org>.
- [39] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.
- [40] J. Wu and D. K. Panda. Mpi/io on dafs over via: Implementation and performance evaluation. In *IPDPS*, 2002.
- [41] W. Yu, S. Liang, and D. K. Panda. High performance support of parallel virtual file system (pvfs2) over quadrics. In *ICS*, pages 323–331, 2005.
- [42] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proc. of the 29th International Symposium on Computer Architecture (ISCA29)*, May 2002.