Optimization and Bottleneck Analysis of Network Block I/O in Commodity Storage Systems

Manolis Marazakis, Vassilis Papaefstathiou, and Angelos Bilas
Institute of Computer Science,
Foundation for Research and Technology - Hellas (FORTH),
Heraklion, GR-71110, Greece
{maraz,papaef,bilas}@ics.forth.gr

ABSTRACT

Building commodity networked storage systems is an important architectural trend; Commodity servers hosting a moderate number of consumer-grade disks and interconnected with a high-performance network are an attractive option for improving storage system scalability and cost-efficiency. However, such systems incur significant overheads and are not able to deliver to applications the available throughput. We examine in detail the sources of overheads in such systems, using a working prototype to quantify the overheads associated with various parts of the I/O protocol. We optimize our base protocol to deal with small requests by batching them at the network level and without any I/O-specific knowledge. We also redesign our protocol stack to allow for asynchronous event processing, in-line, during send-path request processing. These techniques improve performance for a 8-disk SATA RAID0 array from 200 to 290 MBytes/s (45% improvement). Using a ramdisk, peak performance improves from 320 to 474 MBytes/s (48% improvement), which is 72% of the maximum possible throughput in our experimental setup. We also analyze the remaining system bottlenecks, and find that although commodity storage systems have potential for building high-performance I/O subsystems, traditional network and I/O protocols are not fully capable of delivering this potential.

Categories and Subject Descriptors

C.2 [**Computer System Organization**]: Performance of Systems; C.4 [**Computer System Organization**]: Computer System Implementation; D.4.2 [**Software**]: Operating Systems—*Storage Management*

General Terms

Performance, Measurement

Keywords

block-level I/O, I/O performance optimization, RDMA, commodity servers

1. INTRODUCTION

Increasing needs for storing and retrieving digital information in many application domains pose significant scalability requirements on modern storage systems. Moreover, such needs become more and more pressing in low-end application domains, such as entertainment, where cost-efficiency is important. To satisfy these needs, currently, storage system architectures are undergoing a transition from directly- to network-attached.

Previous work [19] has shown that building commodity networked storage systems is challenging and results in high network overheads. In particular, although commodity systems can provide adequate raw throughput in the I/O path from server disks to application memory, storage systems are not able to exploit it fully. The main reasons for this, as identified in [19] are: (a) Storage-specific network protocols require small messages for requests and completions; (b) Event completion is by nature asynchronous in modern architectures and consumes significant resources on both storage (target) and application (initiator) servers.

We expect that high-performance (e.g. 10 GBit/s) network interface controllers (NICs) that do not offer offloading characteristics will quickly become more affordable and therefore will be used extensively in commodity-grade servers. Currently, 10 GBit/s networking capability is supported by a number of commercial products, with a range of interconnect technologies. Specific examples include: (1) 10GbE offerings from Myricom [4] and Chelsio [2], (2) traditional cluster-interconnect technologies from Myricom [7] and Quadrics [24, 23], and (3) interconnects based on the Infiniband standards [1] from Mellanox [3], NetEffect [9], and other vendors. [15] presents a representative performance comparison of such interconnects. Remote RDMA is well understood and we expect it to be available in all such NICs. Ethernet-based NICs also provide such capabilities, e.g. based on the iWARP protocol [12] and there are ongoing efforts for interoperability between vendors.

In this work we investigate the sources of overheads. We use our custom-built NIC, since this option gives us control over most aspects of the I/O data path. We start from a base storage access protocol over the NIC, and then clearly identify the parts of the protocol that contribute to throughput bottlenecks. Our NIC is capable of 10 GBit/s (1.2 GBytes/s) throughput. However, due to PCI-X limitations, peak throughput in simple user-level, memory-to-memory, one-way transfers is about 626 MBytes/s (4 KByte messages in a two-node configuration connected back-to-back). The NIC and network provide reliable, in-order delivery without software protocol support. On top of this network, we develop an optimized remote storage protocol, extending the protocol designed in previous work [19]. As we are mostly concerned with network performance, we perform experiments with configurations that exercise parts of the full I/O data path, including a ramdisk block device, and in the end we also present results from real disks as well.

We consider how overheads from (a) small messages and (b) asynchronous event processing can be mitigated. For (a) we use a batching technique that operates at the network layer and transparently batches small messages (requests and completions) in the network queues both at the initiator and the target sides. The send-path in the I/O protocol stack, detects small messages in the network queue and issues a single remote DMA write request for a batch of messages, without interpreting their contents. The receive-path transparently treats this batch as multiple requests arriving concurrently, and services all requests and completions it includes. In all cases, data transfers (each of 4-KByte size) are serviced one by one. For (b) we modify the I/O protocol stack, both the send and receive path to allow for asynchronous event completion at any point during processing. Traditionally, asynchronous event completion occurs in interrupt handlers. Our base protocol uses asynchronous events only for remote messages. Local completions (e.g. for DMA transfers) are handled with a lightweight polling mechanism. Moreover, our base protocol performs batching of interrupts. In this work, we modify the protocol structure to allow execution of completion tasks not only in interrupt contexts (in the receive-path), but during the send path as well.

These optimizations improve end-to-end performance from 320 to 474 MBytes/s, when using a ramdisk. We see that our technique for processing completions during send-path processing is very effective in reducing (on average) the number of interrupts from one-every-8 requests to one-every-64 requests both at the target and the initiator. Both techniques are most effective when the workload consists of multiple concurrent threads. This results in more concurrent requests that provide more opportunity to process I/O completion events in the send-path.

When using eight SATA disks in a RAID0 array that offers a maximum throughput of about 450 MBytes/s, our base protocol achieves a maximum throughput of about 200 MBytes/s, whereas our enhanced protocol increases this by 45% to about 290 MBytes/s. At this point, the single PCI-X bus at the target becomes the limiting factor, as it has to carry the data traffic twice; For reads, the data is moved from disk to memory, and then moved again from memory to the network.

We then examine the remaining bottlenecks that limit end-to-end I/O throughput to 474 MBytes/s compared to the 626 Bytes/s achievable with at user-level. In our analysis we replace pieces of the protocol and create three "fake" configurations that exclude certain overheads. In this manner we are able to exactly pin-point limitations. We find that the throughput of the "fake" configurations is limited by CPU overheads (mostly at the target side).

Overall, we find that high performance I/O is possible over commodity components. However, the protocol used for remote storage access, needs to be designed specifically for dealing with limitations of commodity systems for small messages and for asynchronous event processing.

The rest of this paper is organized as follows. Section 2 presents the necessary background from previous work, including an overview of the RDMA-capable NIC used. Section 3 presents our protocol optimizations for batching and asynchronous event processing. Section 4 presents our performance evaluation and analysis of remaining bottlenecks. Section 5 presents related work. Finally, Section 6 draws our conclusions.

2. BACKGROUND

Current research efforts 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 equipped with low-end disk controllers. These *storage nodes* are then attached to a system area network accessible by application servers. Although variations of this architecture exist, overall most proposals follow the basic trend of building on commodity components.

In this work, we start from the remote block-level storage system described in [19, 18]. This system relies on a high-performance network interface card (NIC) capable of performing remote DMA (RDMA) write operations, and delivering transfer completion notifications. The original version of the NIC used two such links; in this work we use an enhanced version that combines four Rocket I/O [5] links to achieve 10 GBit/s data rate. The NIC is a 64-bit PCI-X [21] peripheral, running at 100 MHz.

Such systems require an I/O protocol for accessing storage remotely, through the interconnect. In our previous work [19], we designed a block-level I/O protocol stack, consisting of kernel modules for the initiator and target sides of the I/O path. The initiator and target modules provide remote storage access at the block-level in the kernel. All applications at the initiator can access the remote block device, as if it were a directly-attached physical block device. This guarantees *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. Overall, the initiator's primary task is to forward I/O requests to the target, receive notification of completion, and finalize the I/O requests by invoking the appropriate call-back function for each application block request. Our base block-level I/O protocol encapsulates all performance-critical I/O commands and their resulting data and completion notifications into RDMA operations.

Our previous work [19] presents the design of the I/O protocol and a number of optimizations for (1) reducing protocol messages, (2) dynamically coalescing interrupts, and (3) statically batching I/O requests. We find that, despite these optimizations, the system achieves a maximum of 320 MBytes/s when using a ramdisk and a maximum of 200 MBytes/s when using real disks (8 SATA disks in a RAID0 configuration capable of about 450 MBytes/s).

In this paper, we extend our previous work, as follows:

- We present a more elaborate technique for eliminating interrupts related to both I/O commands at the target and I/O completions at the initiator. This technique relies on a hybrid polling scheme, that enables to process I/O completions while still in a send-path context and, symmetrically, to recover and issue I/O commands at the target while still in a receive-path context.
- We explore the performance characteristics of a range of "fake" configurations that constitute aprts of the real I/O data-path between the initiator and the target. This approach allows to highlight the performance limits along the I/O data-path.
- We consider in our experimental evaluation the relationship between interrupt counts and CPU utilization, at both the initiator and the target, with the throughput levels achieved, for a varying number of I/O issuing threads.

For our purposes, we define a `commodity-grade` server as a host machine that incorporates mostly mainstream hardware components, selected based on their cost/performance ratio rather than their native performance and/or reliability potential. Specifically, we focus on machines running general-purpose OS kernels, and hosting only a moderate number of low-cost disks, without use of specialized storage device controllers (such as FibreChannel [14]). Moreover, we assume that such a server is dedicated to a sole I/O-related role (either initiator or target). Our current experimental system relies on a PCI-X bus-based NIC; we expect that NICs based on the PCI-Express [6] serial-lane standard will become more affordable, allowing commodity-grade servers to take advantage of its robust flow-control mechanisms and its performance potential [20, 16].

3. PROTOCOL DESIGN

In this section we outline the base protocol for remote block-level I/O, and then describe optimizations aimed to improve its efficiency.

3.1 Base Protocol

The base remote block I/O protocol [19] makes use of RDMA operations to forward block read and write requests from the initiator to the target. Both the target and the initiator maintain circular queues for I/O commands and completions, respectively, as well as a pool of page-sized data buffers. At both sides, the local host only consumes entries from its queue, whereas the remote host only adds entries to this queue. Data blocks involved in read and write block-level I/O requests are directly transferred to remote buffers with RDMA writes, without going through the queues.

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. The base protocol incorporates three optimizations:

- It reduces the number of messages required for managing the request queues by eliminating explicit updates to queue head/tail pointers. Instead, it uses special markers in each I/O request to differentiate valid from invalid requests.

- To reduce the number of interrupts, the protocol statically batches interrupts when I/O requests are prepared and issued at the initiator side. The protocol marks only the last 4-KByte request of a longer sequence of requests for generating an interrupt at the target. This triggers interrupts at the target side only every few requests.

- To further minimize the number of interrupts asserted at each node the base protocol employs 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. 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.

3.2 Asynchronous event processing

Our previous work has shown that although these optimizations are effective, overall system throughput is still lagging significantly compared to raw hardware performance. We see that interrupts counts are still high, despite the base protocol optimizations. For this reason we introduce the following extensions: static target-side interrupt batching and send-path asynchronous event processing.

First, we extend the simple batching technique used at the initiator to also apply to the target side. Responses prepared by the target are marked appropriately so that longer sequences generate interrupts every few responses. This statically reduces the number of interrupt service routine activations at the initiator, thus, lowering its overall CPU utilization. Target-side interrupt batching is somewhat different in each design and implementation as the target does not have access at once to all responses that correspond to a single user-initiated I/O operation. Responses are generated only as each 4-KByte request is being completed by the physical-disk driver at the target, thus requiring additional state. To effect batching of interrupts at the initiator, the target counts the number of completions that correspond to I/O commands that had been setup to trigger an interrupt at the target. We have experimentally determined that a good value for batching interrupts at the initiator is 8, whereas at the target the batching-factor is 2.

Then we examine how asynchronous events can be further reduced by processing them inline during send-path processing in the I/O stack. The basic idea is that I/O completions are especially harmful at high I/O request rates. However, when there is a large number of I/O requests, the system tends to spend more time in the send path of the I/O stack. Thus, there is opportunity to process in-line completions of previous requests that have already arrived at the initiator. Doing so can reduce significantly the need for interrupts and context switches.

To achieve in-line processing of request completions (and all asynchronous events) we need to ensure proper synchronization between the interrupt context and the send-path. The send-path at the initiator checks for I/O completions as soon as it finishes posting all pending I/O requests. Checking is done by polling the I/O completion queue for new arrivals, while keeping interrupt delivery from the NIC disabled.

Although our initial approach was to stop polling for completions as soon as the arrival queue was found empty, it turns out that a different, simple heuristic is very effective: As long as at least one I/O completion is found in the arrival queue, the protocol keeps polling the queue until it retrieves an I/O completion that has the interrupt bit set. This heuristic combines static interrupt batching information at the target with inline processing at the initiator and results in significant reduction of interrupts.

After posting pending I/O commands, if the send-path finds no I/O completions pending, it will finish its execution, leaving interrupt delivery from the NIC enabled; later arrival of an I/O completion will be able to trigger an interrupt that will lead to I/O completions being processed.

This coupling of the send-path with the receive-path processing is also applied at the target side. The code that posts I/O completions checks for new I/O commands before completing its execution. In this manner, we strive to issue new I/O commands as soon as possible, without incurring the overhead of having to schedule an interrupt-handling context.

We expect this optimization to be most effective for the smaller I/O request sizes, in particular in the case of a single I/O issuing thread. With more threads, and especially with large request sizes, we expect to find several pending-to-be-processed I/O commands and completions, at the target and at the initiator respectively. Moreover, the CPU utilization of the initiator is expected to be higher, since the send-path code spends some of its time polling.

3.3 Small RDMA-request batching

Due to the high initiation cost over the PCI-X bus and the overhead associated with PCI-X arbitration, small messages have a significant negative impact on performance. To reduce the overhead from posting RDMA descriptors for small RDMA transfer requests, such as those encapsulating I/O commands and completions, we introduce batching at the network layer in the I/O stack.

The initiator's block I/O layer in the send-path fills-in the network request queue as before. However, the network layer, instead of issuing an RDMA request for each separate request in the queue, it issues an aggregate transfer request. Currently, each aggregate request consists has a size of 4 KBytes (page size). For read requests the send-path code just sends the aggregate request to the target, where the separate commands are placed in consecutive positions of the I/O command queue. For write requests, the send-path also posts RDMA transfer descriptors for the page-sized data blocks to be written by the target to the disks.

This batching technique is also applied at the target for the I/O completion messages. Thus, under a sequential I/O workload both sides take steps to reduce the overhead of posting RDMA descriptors. Since I/O commands and completions consume 64 bytes in our current implementation of the remote I/O protocol, we can fit up to 64 such message in a page-sized buffer, which is currently the maximum RDMA transfer size.

3.4 I/O Path

Figure 1 illustrates the data path that needs to be traversed for each remote I/O operation: I/O request processing starts at the I/O initiator node, after the data consuming application (data producing application in the case of writes) issues I/O system calls to the local OS kernel. I/O requests reach the block-device driver that implements the initiator's side of the remote block-level I/O protocol (marked IBD in Figure 1). At this point, I/O commands are encapsulated in messages that are transmitted by issuing RDMA operations. This step entails posting RDMA transfer descriptors to be consumed by the NIC. Posting the RDMA descriptors involves PCI-X writes. Actual transfers from the initiator's host memory involve PCI-X reads, from a "pinned" memory region reserved for I/O commands, without the need for a data copy by the NIC driver. Messages are serialized and encoded for transmission over the set of RocketIO links. The NIC at the target collects incoming messages (deserialization and decoding), and directly places them into the target host's memory. After placement of an I/O command, the NIC can trigger an interrupt to notify the target's side of the I/O protocol (marked TBD in Figure 1) of the new arrivals. The target issues the I/O commands to its locally-attached block devices. This step is handled asynchronously, as the target will be notified of I/O completions via a local interrupt raised by the storage device controller. I/O issue and transfer of commands and data by the target to/from the storage devices involves DMA operations over the target's PCI-X bus. Once the target is notified of local I/O completion, it transmits the corresponding data to the initiator together with an I/O completion message, by posting RDMA descriptors. The I/O completion is set to trigger an interrupt at the initiator, so that the IBD driver can locally complete the corresponding I/O request. Writes are handled in a corresponding manner, again by traversing the data-path between initiator and target.

4. PERFORMANCE EVALUATION

In this section we first present our experimental platform. Then we present a performance evaluation of our optimized protocol and finally, we discuss the remaining system bottlenecks.

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 BC4852 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. The results presented in this section were obtained with these two endpoints, connected back-to-back, with 4 RocketIO serial links (i.e. without an intervening switch).

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. 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, with an average seek latency of 14 milliseconds.

In our evaluation we use the xdd benchmark [13] with the "direct I/O" (`-dio`) option to bypass the initiator's buffer cache. We generate sequential I/O workloads, in order to achieve close to peak I/O performance for the various system components. We vary the request size from 64 up to 512 KBytes. Each experiment transfers a total of 4 GBytes (a total volume of 1048576 4-KByte data blocks) between the initiator and the target. The metrics that we report in our experiments are summarized in Table 1. We vary the number of I/O issuing threads (q) from 1 to 4.

4.2 Optimized Protocol

Figure 2 shows the I/O throughput using the base I/O protocol for a remotely-accessed ramdisk block device. We achieve up to 320 MBytes/s, but as we we will show in Section 4.3.3 it is possible to achieve up to 48% higher throughput with the optimized protocol. The improvement is derived mostly from eliminating more interrupts at both initiator and target.

Table 2 shows the number of interrupts triggered at the initiator and at the target, under the hybrid polling scheme described in Section 4.2, for sequential reads and writes. We observe that a significant degree of interrupt coalescing is achieved. The I/O workload for each of these runs consists of 1048576 4-KByte requests, therefore our (static) policy for setting up an I/O command to trigger an interrupt at the target could trigger up to 131072 interrupts, 1 every 8 I/O requests. Likewise, our policy for setting up an I/O completion to trigger an interrupt at the initiator could trigger up to 65536 interrupts, 1 every 16 I/O completions.

Now we examine the effectiveness of our optimizations. Asynchronous event processing in the send-path (both initiator and target) exhibits a trade-off between the degree of polling, CPU utilization, and achieved throughput. Although polling may help avoid several interrupt activations, this is not necessarily a guarantee for higher I/O throughput. To illustrate this point, we present results for three alternative schemes for the polling scheme, for sequential

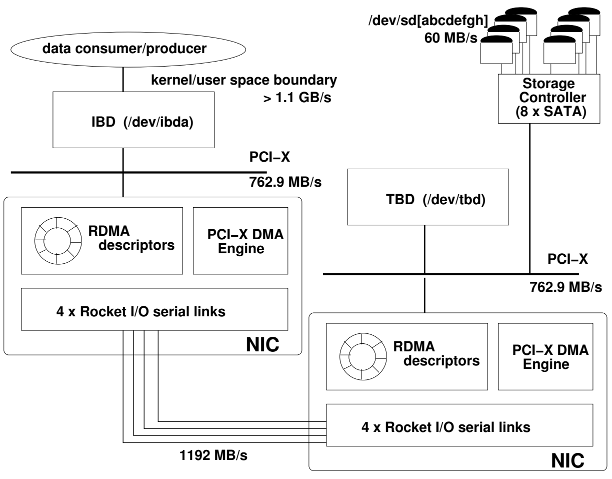


Figure 1: Data path for remote block-level I/O.

Table 1: Evaluation metrics.

MB/s	I/O throughput, expressed in MBytes/sec
R-IRQs-I	interrupt count, at initiator, with sequential read workload
R-IRQs-T	interrupt count, at target, with sequential read workload
R-CPU-I	CPU utilization (%), at initiator, with sequential read workload
W-IRQs-I	interrupt count, at initiator, with sequential write workload
W-IRQs-T	interrupt count, at target, with sequential write workload
W-CPU-I	CPU utilization (%), at initiator, with sequential write workload

reads issued by two concurrent threads:

- *Anticipatory* polling: The polling loop completes when the send-path code at the initiator finds an I/O completion (correspondingly, the receive-path code finds an I/O command at the target).
- *Aggressive* polling: Polling ends when at least one interrupt-triggering I/O completion is found at the initiator (correspondingly, an I/O command at the target). In other words, with this setting we spend more time waiting for an arrival, by polling even if there are currently no I/O commands or completions found.
- *Minimal* polling: After posting all pending I/O completions at the initiator, the send-path code checks only once for any arrivals of I/O completions, without explicitly checking for arrivals that would trigger an interrupt (likewise at the target).

Table 3 shows the results for each polling scheme. All schemes achieve comparable throughput levels, since in all runs the I/O target is completely saturated. We observe that with the *aggressive* scheme we greatly reduce the number of interrupts at the initiator, at the cost of significantly higher CPU utilization. The *minimal* scheme leads to almost the same behavior as the *aggressive* scheme, showing that most of the time there are pending I/O completions at the initiator (I/O commands at the target). Therefore, it makes sense from a performance point of view to anticipate their arrival and process them in-line rather than having to schedule an interrupt-processing context to handle them. The *anticipatory* setting leads to lower CPU utilization at the initiator, therefore it is our preferred setting for all the experiments reported in this paper.

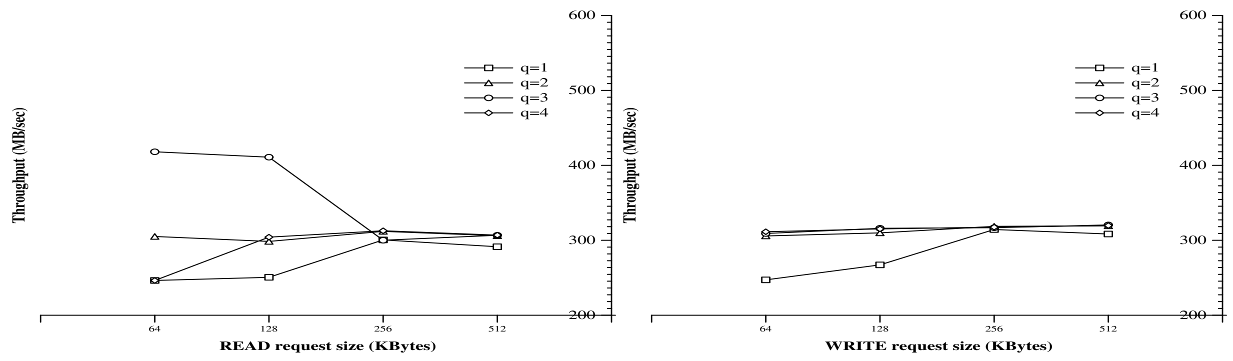
4.3 Analysis of Remaining Bottlenecks

We now examine the remaining system bottlenecks. 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. Practically, the maximum achievable PCI-X bus throughput is less than the theoretical peak. Using a hardware-based benchmark, we measured DMA transfer rates up to 659.2 MBytes/s from the host memory to the NIC, and up to 678.1 MBytes/s in the opposite direction. The theoretical maximum throughput when using four Rocket I/O links is two 64-bit word at every 78.125-MHz Rocket-I/O clock cycle, i.e. 1192 MBytes/s. Therefore, the maximum theoretical end-to-end throughput is limited to that of the PCI-X bus.

Figure 3(a) summarizes the results from a simple user-space benchmark that measures the maximum achievable throughput. This benchmark initiates one-way transfers, i.e. without waiting for any response from the receiver, by posting RDMA descriptors in the NIC-resident RDMA request queue. After each descriptor has been posted, this benchmark checks the local notification word written by the NIC in host memory to obtain the head and tail values for the RDMA request queue, so that it can ensure that there is sufficient space for posting the next RDMA descriptor. PCI-X write-combining is enabled for the transmitting endpoint. This feature allows more efficient posting of RDMA descriptors from the host to the NIC-resident RDMA request queue. We achieve up to 626 MBytes/s, which is 95% of the practical limit for the PCI-X bus at the initiator.

To reveal the remaining system bottlenecks that are responsible for not achieving the same throughput when performing I/O we examine three "fake" configurations, each completing requests prematurely, and at a different point in the I/O path:

- *FAKE(I):* I/O operations are encapsulated into protocol messages and transmitted to the I/O target, which however is disabled. The I/O initiator unilaterally completes I/O operations, without waiting for the I/O target.
- *FAKE(I+T):* I/O operations are completed by the I/O target immediately upon reception, without actually issuing I/O re-



(a) Reads (b) Writes

Figure 2: I/O Throughput using the base I/O protocol on a remote ramdisk, for sequential reads and writes.

Table 2: Interrupt counts (initiator, target in 1000s) and % CPU utilization (initiator) for the *REMOTE(RAMDISK)* configuration.

# threads	1				2				3				4			
I/O size	64	128	256	512	64	128	256	512	64	128	256	512	64	128	256	512
R-IRQs-I	65.5	65.5	65.5	57.3	62.2	38.1	49.1	41.7	65.5	39.9	49.1	41.5	65.5	40.3	49.1	41.5
R-IRQs-T	65.5	32.7	16.3	8.1	32.7	20.4	15.1	8.1	27.7	19.1	15.1	8.1	30.4	19.2	15.2	8.1
R-CPU-I	21.4	25.2	37.0	28.5	32.8	41.3	40.2	32.1	48.4	38.6	38.5	35.2	45.5	39.8	32.1	36.4
W-IRQs-I	65.5	65.5	65.5	58.0	33.1	32.7	32.7	32.3	62.8	32.9	32.7	32.6	63.4	32.9	32.7	32.6
W-IRQs-T	65.2	63.7	64.0	63.2	65.1	64.7	64.4	64.3	63.3	64.4	64.5	64.4	64.6	64.5	64.5	64.4
W-CPU-I	23.6	33.9	39.5	39.7	47.4	59.8	57.2	55.5	67.2	64.1	56.8	56.1	67.2	58.9	55.8	56.6

quests to its directly-attached storage devices. The I/O initiator waits for I/O completion messages from the I/O target.

- *REMOTE(RAMDISK):* The I/O target fully implements the remote block-level I/O protocol; however, it issues I/O requests to a local block device that implements a `ramdisk`.

4.3.1 FAKE(I)

Figure 3(b) shows the throughput achievable with the *FAKE(I)* configuration for a sequential I/O workload consisting of write operations with sizes in the range 64-512 KBytes. This configuration serves to investigate one-way data transfer throughput, therefore it can be compared with the results show in Figure 3(a).

We observe no significant variation to the achievable throughput as the number of threads varies, when request size is equal or larger than 128 KBytes. Even with only one thread, CPU utilization at the initiator reaches almost 100%, leaving no available CPU cycles for issuing more requests. We achieve about 560 MBytes/s, or around 90% of the maximum one-way throughput of the user-level test, as shown in Figure 3(a). This corresponds to 85% of the practical limit of the PCI-X bus at the initiator.

4.3.2 FAKE(I+T)

Figure 4 shows the throughput achievable with the *FAKE(I+T)* configuration, for sequential read and write requests, with up to 4 threads. This configuration achieves up to 550 MBytes/s or 98% of the throughput achieved with *FAKE(I)*. This corresponds to 83% of the practical limit of the PCI-X bus at the initiator.

We observe a marked increase of the achievable throughput when the number of threads changes from 1 to 2, reaching the throughput achievable with *FAKE(I)*; with 3 or 4 threads, no further throughput increase is observed. With two threads, the initiator can overlap send-path processing from one of the threads with new I/O requests from user-space, effectively increasing the number of I/O requests in-flight. With more more threads this benefit is diminished due to context-switching overheads.

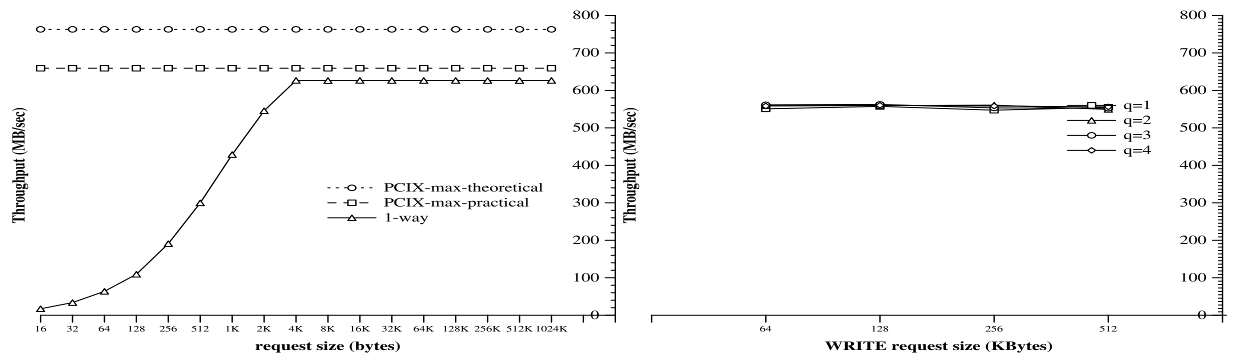
CPU utilization at the I/O initiator for sequential read I/O requests (Table 4) reaches up to 52%; for write requests, CPU utilization at the initiator reaches up to 78%. However, at the target side, CPU utilization peaks at almost 100%, for both read and write requests. In the case of a read I/O request, the I/O initiator has to initiate RDMA transfers for (short) I/O command messages and then to handle I/O completion message arrivals. The data from the read requests are directly placed in the initiator's host memory without burdening the CPU with any processing. However, the CPU has to handle interrupts that signal the arrival of I/O completions. In the case of write I/O requests, the initiator has to perform more work in the send-path, including the initiation of an extra RDMA operation for transferring each data page to the I/O target.

4.3.3 REMOTE(RAMDISK)

Figure 5 shows the throughput achieved with a local ramdisk, i.e. without using our remote block-level I/O protocol. The major limitation is the overhead of copying data between kernel- and user-space, as part of handling `read/write()` system calls. This local configuration does not entail interrupt processing at the target originating from the ramdisk block device.

Figure 6 shows the throughput of *REMOTE(RAMDISK)* for sequential I/O requests with up to 4 threads. Table 2 shows that CPU utilization for read requests reaches up to 48%; for write requests CPU utilization at the initiator reaches up to 67%. In this configuration the initiator performs the same amount of work as in the FAKE(I+T) configuration but the target has to perform memory-copy operations when processing read/write requests from/to the ramdisk block device. However, in this configuration the target still does not have to handle interrupts triggered from the block device. For most runs, the initiator's CPU utilization is slightly lower (up to 10% for reads, up to 17% for writes) in comparison with FAKE(I+T). Overall, we achieve up to 484 MBytes/s, which is 88% of the throughput achieved with *FAKE(I+T)*.

In the next set of experiments, we evaluate the impact of interrupt processing at the I/O target, by experimenting with a RAID-0 array



(a) User-level Communication (b) FAKE(I)

Figure 3: Throughput for base, user-level NIC communication and for the *FAKE(I)* configuration. Both of these experiments exhibit one-way data flow only.

Table 3: Comparison of three alternative settings (anticipatory, aggressive, minimal) of the polling scheme. The workload consists of sequential reads with two I/O issuing threads: Results show the number of interrupts (initiator, target) and CPU utilization (initiator) for the *REMOTE(RAMDISK)* configuration (I/O size in KBytes).

	MB/s			R-IRQs-I			R-IRQs-T			R-CPU-I		
I/O size	Ant	Aggr	Min	Ant	Aggr	Min	Ant	Aggr	Min	Ant	Aggr	Min
64	359.3	483.1	483.2	62280	11	7	32771	50136	56831	32.8	100.0	100.0
128	467.9	478.2	477.0	38171	16383	16381	20444	16374	16374	41.3	41.3	49.1
256	471.2	476.2	475.5	49144	16381	16386	15148	15384	15797	40.2	40.2	49.5
512	473.3	475.2	475.4	41700	20423	20432	8182	4187	4174	32.1	32.1	40.5

consisting of 8 SATA disks. We expect the additional load at the I/O target to affect the overall I/O throughput.

4.4 Real-Disks Configuration

Figure 8 shows the throughput achieved when using the 8-disk RAID0 device, for sequential I/O requests with up to 4 threads. For comparison, the corresponding throughput results with the RAID0 array locally-attached are shown in Figure 7. We see that the maximum throughput in the locally-attached case is around 450 MBytes/s.

The highest I/O throughput achieved with the actual disks and including our optimizations is about 290 MBytes/s for reads and 283 MBytes/s for writes. The main reason for the reduced performance compared to the ramdisk configuration is the existence of a single PCI-X bus at the target. path between initiator and target. In particular, the target has to handle not only interrupts for I/O commands emanating from the NIC, but also interrupts triggered by the SATA storage controller to signal completion of previously posted I/O operations. Moreover, a major limitation is that both the NIC and the disk controller at the target reside on the same PCI-X bus. This bus has to be traversed twice for each I/O command: first from the storage device controller to the target's host memory, and then from the target's host memory to the NIC for transmission back to the initiator. As mentioned above, the maximum achievable throughput of our PCI-X is about 660 MBytes/s resulting in a peak end-to-end I/O throughput of about 330 MBytes/s.

5. RELATED WORK

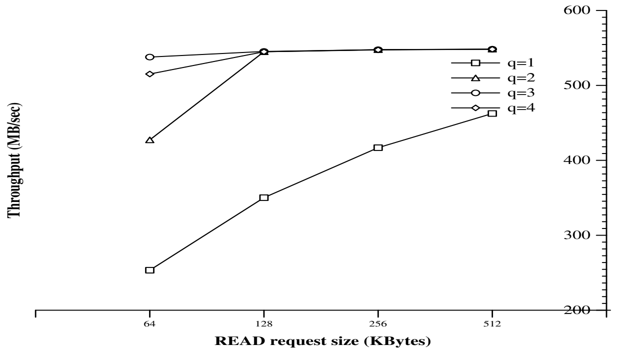
As mentioned earlier, the NIC prototype we use and the base I/O protocol have been presented in our previous work [19, 18]. That previous work established the baseline performance for our system. In this paper, we present and evaluate optimizations for asynchronous event processing and request batching that significantly improve I/O throughput. We also analyze the remaining system bottlenecks by instrumenting our working prototype.

RDMA [25, 1, 11] has become a core capability for low-latency, high-throughput interconnects. The authors in [15] evaluate the performance characteristics of 3 types of RDMA-capable interconnects: Myrinet [7], Quadrics [24, 23], and Infiniband [1]. The evaluation in [15] also explores the implications of completion notification and address translation capabilities in the NIC. The evaluation in [16] 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.

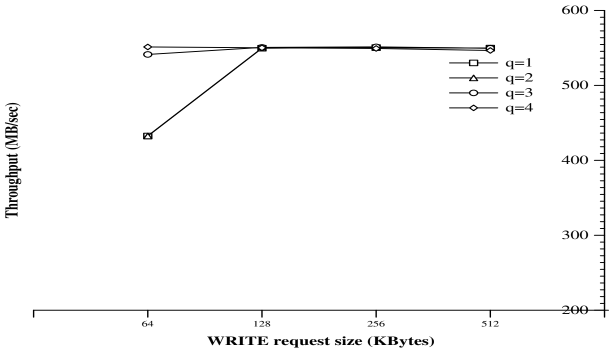
Previous work has shown that interrupt cost can be extremely high in high-performance networks [26] and that it is important to reduce the number of interrupts. Interrupt silencing has been used in the past in lower speed interconnects [30], where interrupt cost is not as important. In our work we design 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 [30], our work quantifies the effect of various aspects in modern, serial-link based interconnects.

Previous work has examined the benefits from using RDMA-capability for improving the performance of storage systems. RDMA-assisted iSCSI [17] uses Infiniband to reduce host overheads. PVFS2-over-Quadrics [29] uses Quadrics to improve the performance of the PVFS2 parallel filesystem. Our work focuses on low-level overheads such as interrupt processing, and also the overlap of send-path and receive-path processing due to the asynchronous nature of the remote I/O protocol interactions.

The authors of [28] present a method to reduce interrupts in the NIC based on a constant-period polling scheme. The authors



(a) Reads



(b) Writes

Figure 4: I/O Throughput for the *FAKE(I+T)* configuration.

Table 4: Initiator CPU utilization for the *FAKE(I+T)* configuration.

# threads	1				2				3				4			
I/O size (KB)	64	128	256	512	64	128	256	512	64	128	256	512	64	128	256	512
R-CPU-I (%)	18.4	24.1	26.4	31.3	37.5	50.2	44.6	42.5	52.2	48.1	44.9	42.8	50.2	46.9	42.8	43.1
W-CPU-I (%)	18.4	24.1	26.4	31.3	51.1	74.1	76.6	65.1	82.3	72.1	71.9	66.7	82.6	78.7	71.6	68.5

Table 5: Summary of I/O throughput scores for various configurations.

configuration	MB/s	reference point
FAKE(I)	**560**	626 (one-way PCI-X transfers from memory)
FAKE(I+T)	**550**	560 (*FAKE(I)*)
REMOTE(RAMDISK)	**474**	550 (*FAKE(I+T)*)
REMOTE(8-SATA-RAID0)	**290**	474 (*REMOTE(RAMDISK)*), 450 (*local RAID0*)

in [10] presents an adaptive polling scheme that reduces the number of interrupts in the NIC at high network loads. The polling period is determined based on measurements of packet inter-arrival times. Similarly, [22] presents a polling mechanism to be used at high network loads instead of triggering an interrupt per packet arrival. Each activation of the polling thread is assigned a packet quota for fairness purposes. In our work, due to the statically-enforced batching of interrupts, by both initiator and target, not all of the incoming messages are set to trigger an interrupt. Moreover, messages are set to trigger interrupts in the NIC as defined by a remote I/O protocol (rather than a "flat" stream of network packets), which enables us to apply a polling scheme based on the idea of anticipating the arrival of interrupt-triggering protocol messages while still in the initiator's send-path, after posting commands, or in the target's receive path, after posting completions.

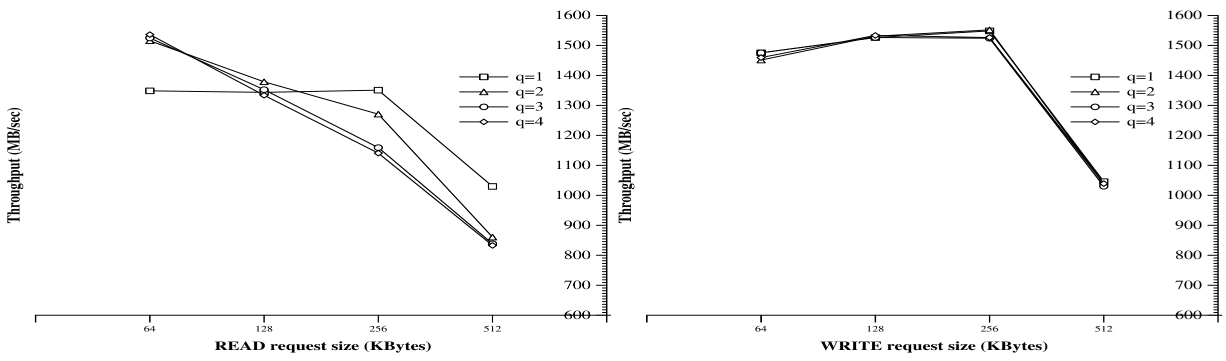
Performance studies of sequential I/O workloads are presented in [27, 8]. These studies include measurements throughout the I/O data path, for a stand-alone I/O server. An important finding from these studies is that although the sequential I/O throughput of disks and controllers has been increasing, the performance bottleneck has shifted to the system backplane (PCI bus for these studies), which has not scaled up. In our work, we focus on a networked storage system, and include in our study performance effects from the NIC and its interaction with the storage subsystem.

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. Our previous work [19] has examined the overheads associated with remote block-level I/O. Our platform uses a NIC that provides support for remote DMA operations and a base I/O protocol stack that has already been optimized for remote I/O. This work has shown that, although today all hardware components in the I/O data-path are capable of high throughput, commodity networked storage systems are not able to fully deliver this level of performance to storage applications.

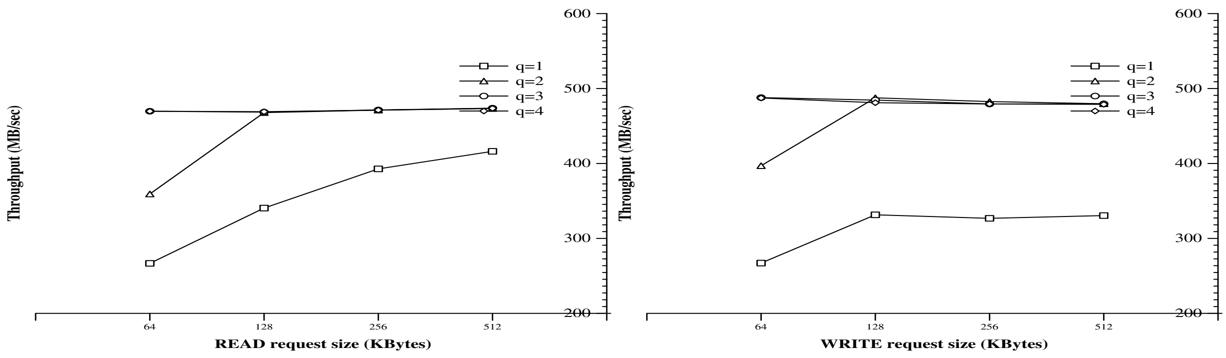
We present optimizations for reducing asynchronous event processing overheads and for batching small requests dynamically at the network layer in the I/O stack. Table 5 summarizes the throughput levels achieved with the various configurations discussed in this Section. For each configuration, this table also shows the corresponding reference point. Our optimized protocol achieves about 474 MBytes/s in a remote ramdisk configuration, compared to the 320 MBytes/s of the base protocol. We then examine the remaining bottlenecks in detail. We use our working prototype and a ramdisk device to create "fake" protocol configurations that expose parts of the protocol that are responsible for significant overheads. By studying a remote ramdisk configuration, We find that up to the point where the target has to generate I/O completions to be sent back to the initiator (together with data blocks in the case of reads) we can achieve about 474 MBytes/s. However, at this point the I/O target's CPU is saturated. Finally, we examine a configuration with real disks, and we find that throughput increases from 200 to 290, by about 45%. In this configuration, the bottleneck for further improving throughput is the single PCI-X bus at the target side.

Overall, we find that although commodity storage systems have potential for building high-performance I/O subsystems, traditional network and I/O protocols are not adequate. We show that redesigning the I/O protocol layers around mitigating the effects of small messages and asynchronous event processing on modern com-



(a) Reads (b) Writes

Figure 5: I/O Throughput for the local *ramdisk* configuration, as measured at the I/O target.



(a) Reads (b) Writes

Figure 6: I/O Throughput for the *REMOTE(RAMDISK)* configuration.

modity architectures and interconnects improves performance within 28% of the hardware limits. Finally, as CPU cycles are an important resource in application servers (initiators), we believe that future work should concentrate on achieving similar levels of performance but at lower CPU utilization.

7. ACKNOWLEDGMENTS

We would like to thank the members of the CARV laboratory at ICS-FORTH for the useful discussions. We are indebted to George Kalokairinos, Aggelos Ioannou, and 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 UNIsIX (MC EXT 509595) project, and the HiPEAC Network of Excellence (NoE 004408).

8. REFERENCES

[1] An Infiniband Technology Overview. Infiniband Trade Association, http://www.infinibandta.org/ibta.

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

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

[4] Myri-10G Overview. http://www.myricom.com/Myri-10G/overview/.

[5] Rocket I/O User Guide. Xilinx Inc, http://www.xilinx.com/bvdocs-/userguides/ug024.pdf.

[6] D. Anderson, R. Budruk, and T. Shanley. *PCI Express System Architecture*. Addison-Wesley Professional, 2003.

[7] 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.

[8] L. Chung, J. Gray, B. Worthington, and R. Horst. Windows 2000 Disk I/O Performance. Technical Report MS-TR-2000-55, Microsoft Research, 2000.

[9] D. Dalessandro, P. Wyckoff, and G. Montry. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *Proceedings of the RAIT Workshop (in conjunction with the IEEE International Conference on Cluster Computing)*, 2006.

[10] C. Dovrolis, B. Thayer, and P. Ramanathan. HIP: Hybrid Interrupt-Polling for the Network Interface. *ACM OS Review*, 35(4), 2001.

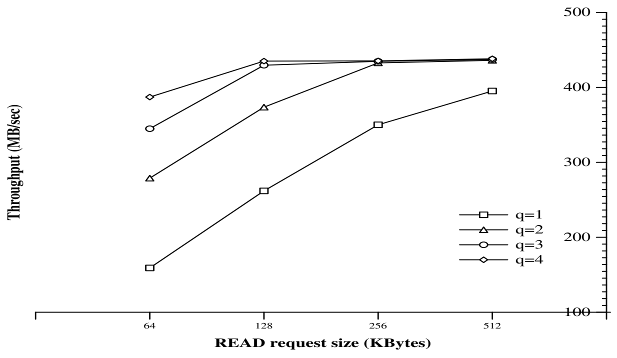
[11] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[12] B. Hausauer. iWARP: Reducing Ethernet Overhead in Data Center Designs. *CommsDesign*, Feb. 2004. http://www.commsdesign.com-/article/printableArticle.jhtml?articleID=51202855.

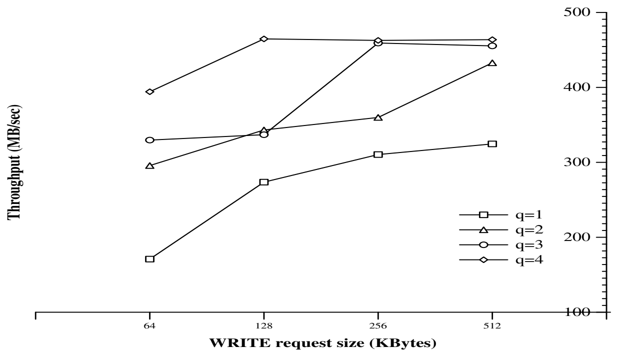
[13] I/O Performance Inc. The xdd I/O Benchmark. http://www.ioperformance.com.

[14] C. Jurgens. Fibre Channel: A Connection to the Future. *IEEE Computer*, 28(8), 1995.

[15] 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.

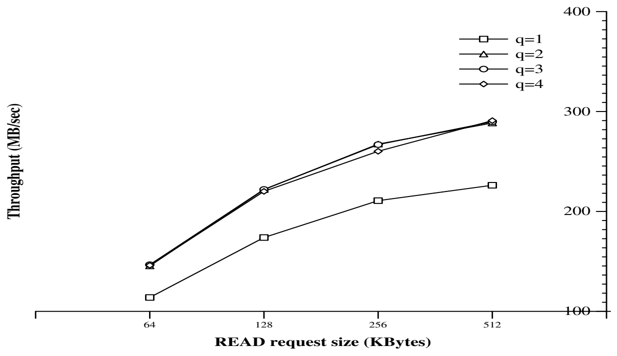


(a) Reads

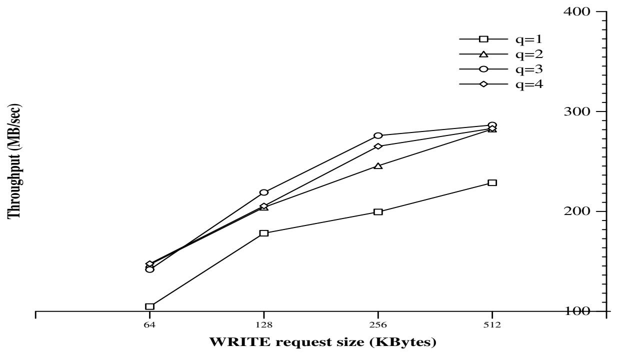


(b) Writes

Figure 7: I/O Throughput for the 8-SATA-RAID0, locally-attached configuration.



(a) Reads



(b) Writes

Figure 8: I/O Throughput for the *REMOTE(8-SATA-RAID0)* configuration.

[16] J. Liu, A. Mamidala, A. Vishnu, and D. Panda. Performance Evaluation of Infiniband with PCI Express. *IEEE Micro*, 25(1):20–29, 2005.

[17] J. Liu, D. Panda, and M. Banikazem. Evaluating the Impact of RDMA on Storage I/O over InfiniBand. In *Proceedings of the SAN Workshop (in conjunction with HPCA Conference)*, 2004.

[18] M. Marazakis, V. Papaefstathiou, G. Kalokairinos, and A. Bilas. Experiences from Debugging a PCI-X-based RDMA-capable NIC. In *Proceedings of the RAIT Workshop (in conjunction with IEEE International Conference on Cluster Computing*, 2006.

[19] M. Marazakis, K. Xinidis, V. Papaefstathiou, and A. B. s. Efficient Remote Block-level I/O over an RDMA-capable NIC. June 2006.

[20] 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.

[21] Mindshare, Inc. and T. Shanley. *PCI-X System Architecture*. Addison-Wesley Professional, 2001.

[22] J. Mogul, , and K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, 7(2), 1997.

[23] F. Petrini, F. E., and A. Hoisie. Performance Evaluation of the Quadrics Interconnection Network. *Journal of Cluster Computing*, 6(2):125–142, 2003.

[24] 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.

[25] J. Pinkerton. The case for rdma, 2002. RDMA Consortium, http://www.rdmaconsortium.org/home/The_Case_for_RDMA-02053.pdf.

[26] 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.

[27] R. Riedel, C. van Ingen, and J. Gray. A Performance Study of Sequential I/O on Windows NT4. In *Proceedings of the 2nd USENIX Windows NT Symposium*, 1998.

[28] J. Smith, , and C. Traw. Operating System Support for End-to-end GBps Networking. *IEEE Network*, 7(2), 1993.

[29] W. Yu, S. Liang, and D. Panda. High performance support of parallel virtual file system (pvfs2) over quadrics. In *Proceedings of the 19th ACM International Conference on Supercomputing*, 2005.

[30] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with VI Communication for Database Storage. In *Proceedings of the 29th Int'l Symposium on Computer Architecture*, May 2002.