Towards 100 Gbit/s Ethernet: Multicore-based Parallel Communication Protocol Design

Stavros Passas, Kostas Magoutis, and Angelos Bilas[†]
Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas (FORTH)
N. Plastira 100, Vassilika Vouton, Heraklion GR-70013, Greece
{stabat,magoutis,bilas}@ics.forth.gr

ABSTRACT

Ethernet line rates are projected to reach 100 Gbits/s by as soon as 2010. While in principle suitable for high performance clustered and parallel applications, Ethernet requires matching improvements in the system software stack. In this paper we address several sources of CPU and memory system overhead in the I/O path at line rates reaching 80 Gbits/s (bi-directional), using multiple 10 Gbit/s links per system node. Key contributions of our work are the design of a parallel high-performance communication protocol that uses *context-independent* page-remapping to (a) reduce packet processing overheads; (b) reduce thread management and synchronization overheads; and (c) address affinity issues in NUMA multicore CPUs. Our design result in the full 40 Gbits/s of available one-way Ethernet bandwidth and in 57.6 Gbits/s (72%) of the 80 Gbits/s maximum bi-directional throughput (limited only by the memory system), while leaving ample CPU cycles for application processing.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications

General Terms

Design, Performance

Keywords

Multicore CPUs, 100 Gbit/s Ethernet, Communication Protocol Design, Performance Evaluation

1. INTRODUCTION

Historically, high performance applications satisfy their communication needs through the use of specialized (and expensive) interconnection networks that offer high-bandwidth, low-latency communication [2, 11]. However, recent improvements in Ethernet technologies promise link rates in the range of 40-100 Gbits/s [10], matching those of traditional high-performance interconnects. While in principle suitable for the needs of clustered and parallel applications, Ethernet is typically associated with CPU and memory system overhead over protocols such as TCP/IP. Packet protocol processing, device interrupt handling, and memory copies for data movement are potential consumers of CPU and memory bandwidth, reducing the effective network bandwidth seen by applications.

A flurry of recent work on overhead reduction technologies that are applicable to Ethernet networks includes application programmer interface (API) and protocol improvements via remote direct memory access (RDMA) [18]; protocol offloading [15]; and new network interface card (NIC) designs [25]. While these approaches have been successful in demonstrating efficient data transfer over 1-10 Gbits/s data rates, improved capabilities of next-generation Ethernet networks demand new techniques that can leverage increasing network bandwidth while simultaneously freeing processor resources.

This paper explores the possibility of leveraging Ethernet as a cost-effective solution for high-performance communication in the 10-100 Gbits/s range. Our aim in expecting no special support from the network, either at the core (switches, routers, etc.) or at the edge (NICs), is to take advantage of economies of scale in standard Ethernet infrastructure. Given technology trends in network, CPU, and memory systems we believe that higher network speeds can be sustained at the application level by leveraging processing power in emerging multicore processors and by reducing memory bandwidth used for protocol processing.

This paper extends earlier work [12, 17] describing a non-parallel version of our Ethernet network transport protocol, named MultiEdge, which has the following characteristics:

- API support for RDMA.
- Support for in-order and out-of-order message delivery. The out-of-order mode suits RDMA transport semantics particularly well and is thus the common case in RDMA-intensive workloads.
- Lightweight flow-control optimized for intra-domain topologies.
- Transparent aggregation of multiple physical links as a single channel for data transmission.

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

[†]Also with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

The system we present in this paper implements the above characteristics, and additionally contributes:

- An efficient parallelization of the transport protocol over multiple CPUs and cores on modern processors;
- A novel copy reduction technique based on execution of VM page remapping operations outside the context of the target process; we refer to this technique as *context-independent* page remapping;
- A broader identification of challenges in parallel communication protocol design in non-uniform memory access (NUMA) multicore processors.

Our results show that a baseline non-parallel protocol that uses traditional (context-dependent) page-remapping achieves a maximum one-way throughput of 27.9 Gbits/s out of ideal 40 Gbits/s, whereas two-way throughput increases to 37.3 Gbits/s out of ideal 80 Gbit/s. Parallelization of the transport protocol and the use of context-independent remapping improves one-way aggregate bandwidth to 38.9 Gbits/s while leaving 62.5% of the total processor cycles (5 out of a total 8 cores) available for application processing. In terms of bidirectional throughput, our protocol achieves 57.6 Gbits/s (74% of ideal) at 50% of total CPU utilization.

The rest of this paper is organized as follows. Section 2 provides background on our Ethernet-based communication protocol and on standard copy avoidance mechanisms. Section 3 presents our networking protocol parallelization on multiple cores of a modern multiprocessor. Sections 4 and 5 present and discuss our experimental platform and results. We discuss related work in Section 6. Finally, we draw our conclusions in Section 7.

2. BACKGROUND

This work extends MultiEdge [12, 17], a high-performance communication protocol geared towards high-speed Ethernet-based local-area networks. MultiEdge offers reliable transfer semantics using window-based flow control with positive and negative acknowledgments, and packet retransmits in case of packet loss, similar to TCP/IP [21]. MultiEdge additionally supports framing, with a choice of in-order or out-of-order delivery, and the simultaneous utilization of multiple physical links, features found in more advanced transport protocols such as SCTP [19]. MultiEdge is an end-to-end protocol that does not require any support from the network core and runs over standard Ethernet protocol.

In a departure from traditional socket (send/receive) based APIs over Ethernet, MultiEdge presents applications with a remote read/write memory API [18]. The initiator of the operation identifies the remote buffer using either a remote virtual memory address or a buffer id and an offset within this buffer. Both should be registered explicitly by the application through a system call. Once registered a buffer may be used in multiple communication operations. It is important to note that the API of MultiEdge does not require any hardware support for RDMA and is implemented over standard Ethernet NIC hardware, without scatter-gather capabilities.

We assume the reader is familiar with the function and operation of the network I/O path in traditional implementations of kernel-level communication protocols [21]. Such implementations typically require a number of memory copies during protocol processing and when crossing the user-kernel boundary in the send and receive paths. Several techniques to mitigate the cost of copying have been proposed in the past, a prominent one being virtual memory (VM) page remapping [3, 6]. Based on this technique, data movement between two buffers can be achieved through the use of virtual to physical address translation, page pinning, and page remapping in the operating system VM structures. In our current prototype we use specially-adapted VM page remapping and associated techniques to eliminate memory copies in sending and receiving data and discuss the implementation of each direction separately.

A key challenge in the send path is transferring data directly from application buffers. Using a user-level buffer for communication requires that the buffer be pinned in physical memory throughout the duration of the I/O operation. Previous approaches have implemented such pinning as part of a copy-on-write operation during each write system call [3]. To reduce the per-I/O overhead associated with this mechanism we chose to instead expose the pinning operation through a system call and perform it either by the application as part of an initialization procedure or through an I/O library. Our mechanism works for both synchronous and asynchronous I/O semantics. In the former case, the system call does not return before the I/O is complete. In the latter case, the semantics of asynchronous I/O already deal with application accesses to user buffer before I/O completion.

In the receive path, the goal of any copy avoidance mechanism is to achieve direct deposit of the incoming data to its destination user buffer without intermediate copies. Previous research on programmable NICs [2, 14, 18] showed that enabling application pre-posting of receive buffers directly with the NIC offers such a mechanism. However this capability requires extensive NIC support and is expensive. Our mechanism, which does not require special NIC support, relies on initially appropriately depositing incoming packets to kernel buffers and subsequently using page remapping to trade the physical pages underlying the kernel buffer with those of the targeted user-level buffer.

3. PROTOCOL PARALLELIZATION

The control flow in our communication protocol, which is designed to offer reliable transfer semantics, window-based flow control with positive and negative acknowledgments, and packet retransmits in case of loss, is shown in Figure 1. The protocol send path is parallelizable in a straightforward manner as user contexts (threads or processes) that initiate communication operations can be mapped to different CPUs, cores, and NICs for the duration of the operation. Parallelization of the receive path is more challenging as incoming packets arrive at an arbitrary NIC and must compete for shared resources during protocol processing. In more detail, the receive path involves the following steps:

- Incoming packets are deposited in host memory buffers managed by (per-NIC) Ethernet rings. In general, each ring contains packets destined for different connections.
- Packets are moved from the Ethernet ring to another ring associated with the target connection.

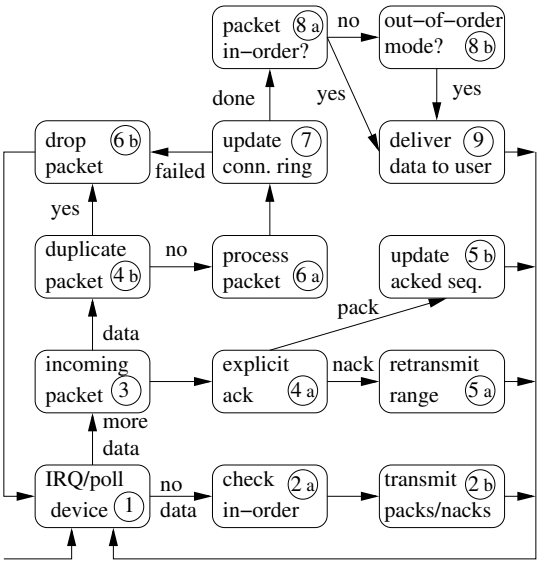


Figure 1: Protocol control flow diagram.

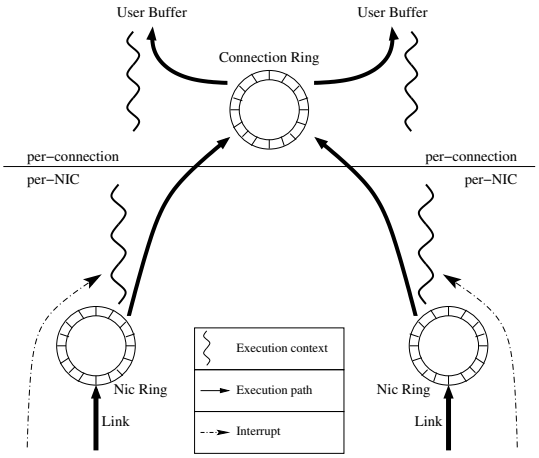


Figure 2: Flow diagram using per-connection threads.

- Depending on delivery order semantics, there are two options:

  (a) With *out-of-order* message delivery, packets are processed immediately upon arrival. Processing involves performing all protocol bookkeeping and remapping message buffers to the application address space.

  (b) With *in-order* message delivery, the receiver ensures that all preceding packets have been received prior to processing the currently arrived packet.

- If the processed packet is the last segment of a message carrying a notification, a signal is sent to the application process.

- Occasionally (based on delivery semantics and system thresholds) an acknowledgment is sent back to the sender.

- Following packet handling, the processing context should poll the NICs for newly arrived packets and start over.

In typical implementations of the above described steps (Figure 2), moving packets from Ethernet to connection rings occurs in per-NIC threads that are woken up by NIC interrupts at packet arrival. Higher-level (connection-specific) packet processing occurs in per-connection threads that have access to application virtual memory structures for page-remapping purposes. For scalability, the system needs to employ multiple per-NIC as well as per-connection protocol threads, which must synchronize for packet movement between Ethernet and connection rings. The above requirements result in (a) large number of threads; (b) synchronization for packet processing in the common path; and (c) complex affinity characteristics between thread scheduling and protocol metadata placement.

Our parallel protocol design aims to address the above issues. A key challenge is to perform page-remapping directly by the network threads: These threads have no default knowledge of application VM structures and essentially need to assume a different "personality" for each packet. With this ability, however, packet processing can be performed end-to-end by network threads alone, eliminating per-connection process threads. This, in turn, eliminates several synchronization points in out-of-order delivery, which is the common case for RDMA-intensive workloads, and simplifies affinity issues to some extent.

The main structures and operations of our parallel protocol are shown in Figure 3. Solid lines correspond to out-of-order processing mode and dashed lines correspond to in-order processing. In out-of-order mode, network threads remove packets from Ethernet rings independently, directly remap them to user buffers, and mark the packet as delivered in the connection ring. In-order delivery requires first placing packets in the connection ring and then processing packets of single messages concurrently with other threads in the ring.

In the following sections we discuss our handling of synchronization issues in concurrent access to shared protocol state by network threads, our use of context-independent VM page remapping to reduce data movement overhead, and finally, our handling of memory alignment issues.

3.1 Thread Synchronization

Concurrent threads executing our communication protocol need to synchronize over access to shared state at specific points in the control flow diagram of Figure 1. We employ wait-free synchronization in all cases using either *trylocks* before entering a critical section, or atomic instructions to perform state variable updates. In the former case, a thread manages to enter the critical section while all others continue with executing other protocol paths, such as servicing new incoming packets. In the latter case, updates to shared state such as protocol counters are implemented with atomic instructions that are typically available in modern processors.

The synchronization points in our parallel communication protocol are summarized in Table 1 and described below:

- (State 1) For each transmitted packet, a lock ensures that a single transmission completion event is signaled prior to the packet's buffer being reclaimed for reuse.

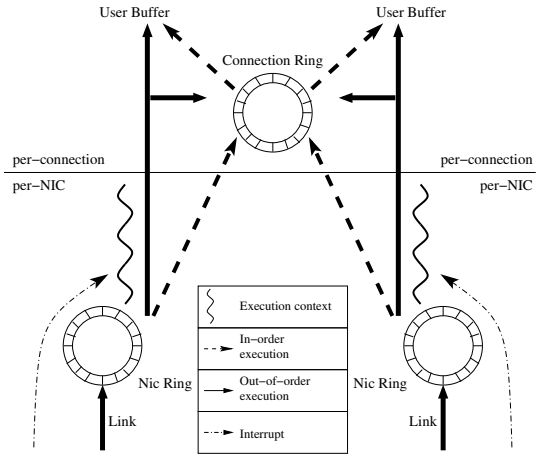


Figure 3: Flow diagram using per-NIC threads.

State	Description
1	Signaling transmission completions
2a	Scanning the rings in-order
2b	Transmission of explicit acks
5a	Retransmissions due to nacks
7	Buffer entry accesses
9	Memory page accesses

Table 1: Description of synchronization points. State numbers correspond to control flow states in Figure 1.

- (State 2a) In the case of in-order delivery, a lock ensures that packets are processed in order by a single thread while the remaining threads are free to process other pending work.
- (State 2b) Positive or negative (nack) acknowledgment processing and transmission is protected by a per-connection lock.
- (State 5a) Retransmission of a range of packets in response to receipt of a nack is protected by a lock per entry (i.e., per packet) on the connection transmit ring.
- (State 7) To avoid concurrent processing of duplicate packets, a lock per entry protects each entry on the receive ring.
- (State 9) To avoid concurrent remapping of pages belonging to overlapping memory buffers a lock protects each page of the registered to the protocol memory ranges.
- Finally, concurrent scanning of the ring is made possible using an atomic addition operation.

Synchronization costs are expected to be minimal in the common case scenario of out-of-order, lossless data transfer in RDMA applications over high-performance Ethernet interconnects.

3.2 Context-Independent Page Remapping

VM page remapping is a procedure by which data movement between two buffers (in our case, a kernel and an application buffer) takes place through exchange of the physical pages backing the buffers. Traditional implementations of page remapping are implemented as follows: For simplicity and without loss of generality we assume that both buffers are page-sized and page-aligned. The procedure typically starts by performing a walk through the target process page table (PT) and identifying the entries that describe the application buffer. This is typically performed in the context of the target process, thus we qualify this procedure as *context-dependent* page remapping. Following the PT walk-through, the physical pages of the target buffer are traded with those of the source buffer and the TLB entries for the application buffer pages are flushed. Finally, if the application pages were pinned in physical memory, the new pages are pinned as well. If the source buffer is part of a buffer pool used for communication, its physical pages are returned to that pool.

In our parallel communication protocol design we introduce a novel implementation of VM page remapping where the context of execution of the remapping actions can be a kernel thread unrelated to the context of the target application process. We call this technique *context-independent* page remapping. To enable kernel threads involved in network processing to manipulate the virtual address space of the target applications, at the time of NIC initialization by the application we store a pointer to the kernel memory manager structure associated with the application. During packet processing we restore the corresponding memory manager when the need arises to access a process's user address space. For x86/x86_64 architectures it suffices to point the `cr3` register to the PT of the process we need to access and modify certain system variables to bring the system to a consistent state.

As a further optimization, each time the application explicitly registers a buffer we cache the corresponding kernel PT structure (including different levels of PT entries if necessary). During VM page remapping we use the cached PT structures to manually locate the PT entry that points to the old page and replace it with the new page. Finally, after every page remapping we update our cached PT structures to keep our data consistent with the kernel PTs.

3.3 Memory Alignment

The use of VM page remapping in a network communication protocol requires special attention to memory alignment issues, particularly when considering the restrictions imposed by standard NIC interfaces. In our work we have addressed the following issues:

Dealing with packet headers. VM page remapping requires that memory buffers be aligned in page boundaries. This poses a challenge for incoming packet processing as it requires depositing the payload (i.e., packet contents after striping all protocol headers) from the NIC right into a page-aligned kernel buffer. This requirement cannot be enforced in a straightforward manner as packet headers are not identifiable by commodity Ethernet NICs and cannot be separated from the payload to two page-aligned memory buffers by separate DMA operations. The solution we implement, which is similar in spirit to the technique proposed in [3], works as follows:

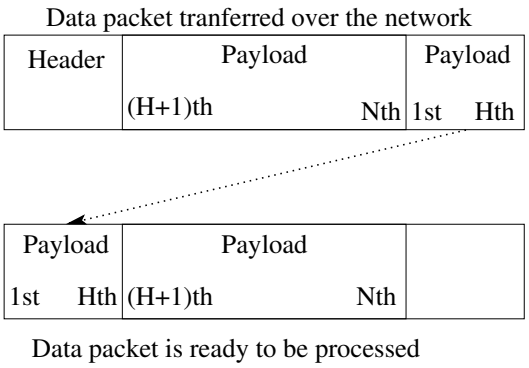


Figure 4: Header and data placement. H, N are the header and payload sizes respectively.

Assuming header size H bytes and payload size N bytes for a given message, the sender transmits the header bytes first (H), followed by payload bytes H+1 to N, followed by payload bytes 1 through H, as shown in Figure 4. The receiver places the entire message (header plus payload) page-aligned to a number of page-sized buffers (4 KBytes in our case). The header bytes are subsequently copied to a separate small buffer while payload bytes 1 through H are copied to the beginning of the message overwriting the header. The payload thus turns out to be page-aligned and therefore appropriate for VM page remapping.

Application buffer alignment and large-message handling. To efficiently handle the case where either the source or the destination (or both) buffers are not page-aligned, our protocol works as follows: The sender takes into account the alignment of the receiver buffer and splits the transfer such that the first packet consists of the data until the next page boundary on the receiver buffer. At the receive side this packet will be delivered to the application using copying. Subsequent packets are MTU-sized and consist of H bytes of protocol headers (48 bytes in our implementation, including the Ethernet header) followed by the payload (up to 8 KBytes in our implementation, which is twice the VM page size, when using 9000 byte Jumbo Ethernet frames). Our header handling ensures that the payload is suitably aligned in memory for VM page remapping. A small part of the payload (smaller than a page) at the beginning or at the end (or both) of the packet may require copying to the application buffer.

4. EXPERIMENTAL PLATFORM

Our experimental platform consists of two systems connected back-to-back with multiple NICs. Both nodes have two, quad core, Opteron 2354 CPUs running at 2.2 GHz and a Tyan S2915 motherboard. The operating system is the 64-bit version of Debian testing with Linux kernel version 2.6.18.8, compiled with GCC version 4.1.2. Each node is equipped with four Myricom 10G-PCIE-8A-C cards. Each card is capable of about 10 Gbits/s throughput in each direction for a full-duplex throughput of about 80 Gbits/s. Each Opteron 2354 CPU has a TLB size of 1024 entries and per core L1, L2, and shared L3 caches, with sizes of 4x32 KBytes, 4x512 KBytes and 1x2 MBytes respectively. Each processor is equipped with 4 DIMMs of 512 MByte DDR-667 for a total of 4 GBytes of main memory. Linux is configured with NUMA features enabled. Figure 5 shows a schematic of the internal data paths in each node from various memories to network links and the maximum throughput in each component of the path.

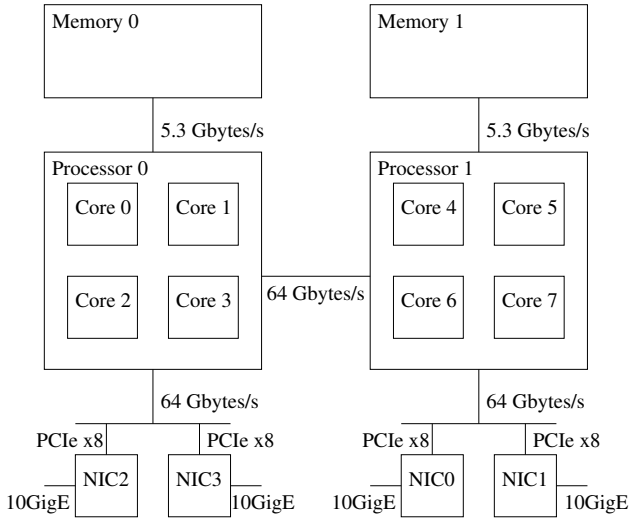


Figure 5: Internal data paths in each system node

We conduct experiments using MTU size of 9000 bytes (Ethernet Jumbo frames). This MTU size is widely used in high-performance systems and is in line with current technology trends.

4.1 Methodology

We evaluate the system using three micro-benchmarks: *one-way,* where one of the two nodes reliably sends messages back to back using remote writes without waiting for any response from the receiver. This benchmark exercises the send path at the sending and the receive path at the receiving node. *two-way,* where both nodes simultaneously transmit data back to back using remote writes. The throughput in this case reflects all traffic in the system, including both sent and received data. *ping-pong* is a request-reply benchmark using remote write. Both request and reply are of the same size.

To understand system behavior, we use the following metrics: (a) Throughput, which is calculated over the amount of application data that has been delivered to the remote node; (b) One-way, end-to-end latency; and (c) CPU utilization breakdowns. CPU utilization is approximate as we cannot account for the time between the moment a NIC issues an interrupt and until the interrupt handler executes on the host CPU and consists of the following components: *IRQ* is the cost for interrupt handling or polling; *TxCopy/Translate* is the overhead spent on preparing the payload in the send path. This component includes either the pinning and translation overheads or the data copy; *RxCopy/SetupRmap* is the overhead of packet processing in the receive path, including copying, where appropriate. This component does not include the actual overhead for remapping, which is measured separately; *Remapping* is the page remapping cost in the receive path; *Packet* is the packet processing overhead.

operation	time (μ s)
ioctl	0.28 - 0.35
alloc buffer	0.1 - 0.3
pin page	2.2 - 7.6
remap page	0.1 - 1.0

Table 2: Basic kernel costs.

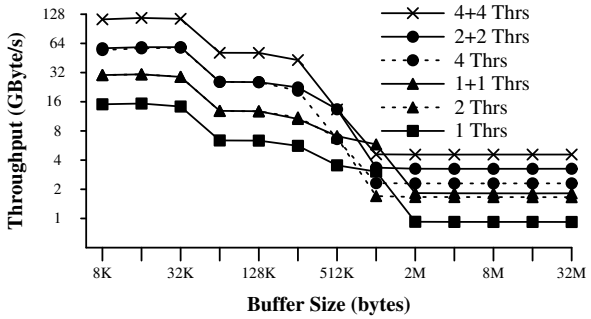


Figure 6: Memory copy throughput.

This includes header preparation, ordering of packets, and flow control; *Device* is the cost for communicating with the NIC both at the send and receive paths.

In our experiments we use the following protocol configurations: *CP:* This is our base version, where the protocol uses one copy in the send and one copy in the receive path. *Map:* This is the protocol version with copies eliminated in the send and receive path using address translation and page remapping. *NoCP:* This is an "ideal" version with both copies artificially removed, showing the maximum achievable performance without copy, remapping, or translation overheads. In all experiments, we report the average of five measurements for each data point. Finally, in the parallel protocol we indicate the number of send, receive threads with *nRT:* and *nST:* respectively.

4.2 System Costs and Memory Subsystem

Table 2 shows the overhead of certain basic kernel operations we use in our design. An empty `ioctl` costs about 0.3 μs. Allocating a kernel buffer (MTU size) costs about 0.2 μs. This cost could be higher if we had memory fragmentation. In addition to both pinning and remapping costs increase almost linearly with the number of pages. Pinning is fairly expensive as it requires locating the corresponding virtual memory area, walking the page table to locate the requested physical pages, and finally increasing their reference count. Pinning the first page is more expensive than the rest, because consecutive virtual pages are placed in consecutive locations in the page table. For page remapping the average overhead is about 0.5 μs. The overhead is lower than pinning because the virtual memory area for each page is stored in a protocol cache during receive buffer registration. This allows us to walk directly the page table, find the table entry, and update it.

Figure 6 shows memory copy throughput in each node. The knees at 32 KBytes, 256 KBytes, and 1 MByte correspond to the L1, L2, and L3 cache sizes. In all these runs each core accesses only memory attached to its CPU. Sustained memory copy throughput for a single core is about 920 MBytes/s, while for all 8 cores it is about 4.56 GBytes/s. With 2 and 4 cores, throughput is significantly higher when cores are split between the two CPUs (memories) rather than placed in a single CPU (memory).

	local memory			remote memory		
cores	1+1	2+2	4+4	1+1	2+2	4+4
read	4008	7760	14344	3724	6680	9536
write	5200	7480	8336	4000	5240	5280

Table 3: Memory throughput(Mbytes/s) when cores access data located on local or remote memory.

Table 3 shows the aggregate throughput when we use one, two, or four cores from each processor for memory read and write. For one core accesses to local memory have a sustained rate of about 1.95 and 2.5 GBytes/s for reads and writes respectively, but it drops significantly when accessing remote memory.

Finally, we see that the maximum memory throughput with local memory accesses for concurrent reads and writes for all the cores is around 10 Gbytes/s, while for remote memory accesses throughput is 33% lower. This indicates that NUMA placement should be taken into consideration when interpreting our results.

4.3 NUMA Affinity Issues

The NUMA architecture of recent multicore processors, such as the ones we use (Figure 5), results in possible variations in memory throughput depending on the relative placement (*affinity*) between host and NIC buffers as well as between application and protocol threads. The main types of affinity are between:

- Application threads and cores;
- Interrupt handlers and cores;
- Protocol receive threads and cores;
- Memories and NICs;
- DMA transfer direction and memory modules;

We obtain all our results with application threads running on separate CPUs, interrupt handlers running on any core of the CPU where the NIC is attached, and protocol receive threads running on different cores of the CPU where the NIC is attached. Memory-NIC and DMA direction-memory affinities are more involved. Memory-NIC affinity means that each NIC performs DMAs only to the memory on the same CPU where the NIC is attached. DMA direction affinity means that the protocol operates in a manner such that it performs only read (write) DMAs from (to) a specific memory module. Our protocol configuration so far does not try to exploit affinities to a significant degree, as such tuning cannot be expected from typical applications.

We statically map the running threads as follows: The *interrupt handlers* and *protocol receive threads* run on the CPU to which the device they handle is attached. We don't specify statically where *application threads* run, but we use fewer application threads than the system cores and each thread runs on a separate core avoiding contention between application threads.

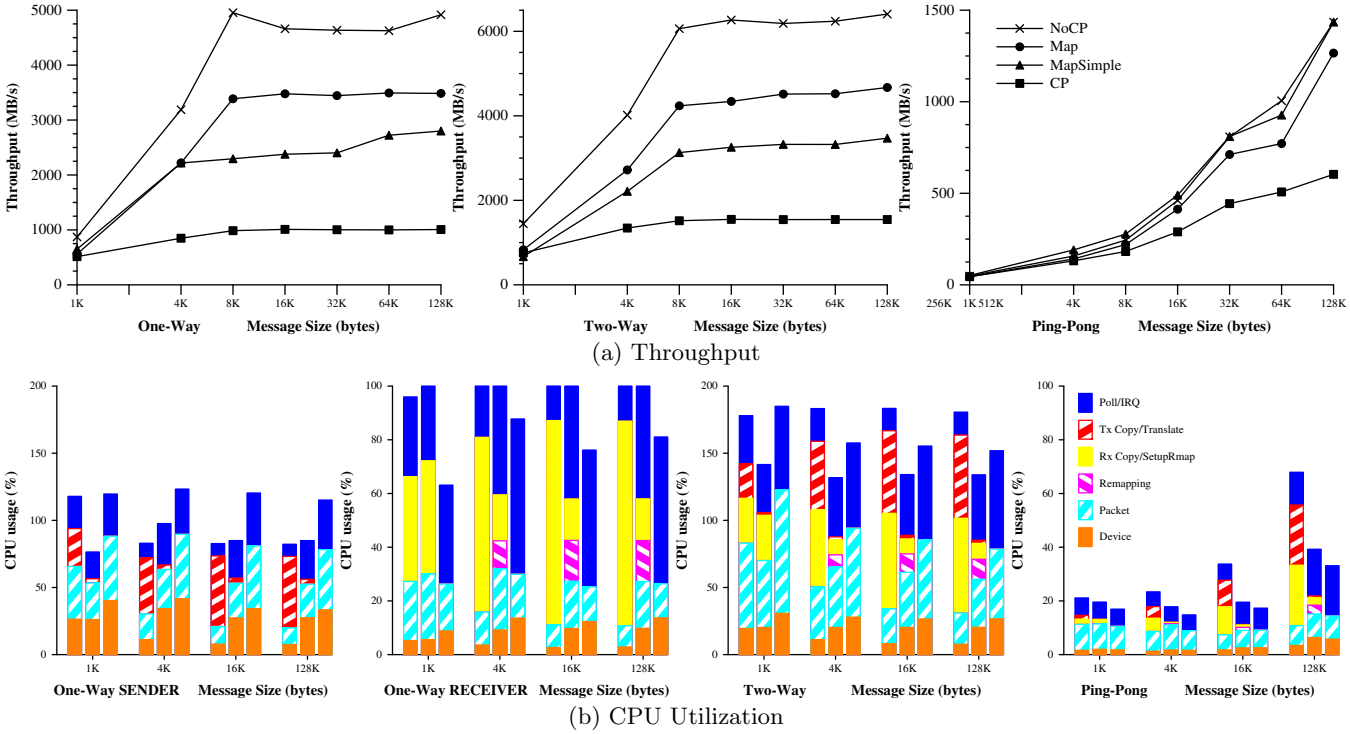


Figure 7: Impact of copy avoidance on throughput and CPU utilization. For CPU utilization (b) we show one bar per protocol configuration (left to right): CP, Map, NoCP.

5. RESULTS

We structure results around the following questions: (a) The benefits from page remapping in the range of 40-80 Gbits/s; (b) Protocol scaling with multiple CPUs and protocol threads; (c) The impact of different TLB invalidation schemes and buffer alignment; and (d) Affinity issues.

5.1 Benefits of Page Remapping

Figure 7(a-b) shows throughput and CPU utilization when using copy (CP) vs. page remapping (Map) and contrasts them with the ideal version that artificially avoids both (NoCP). We see that CP, which uses one copy on the send and one copy on the receive path, is limited by CPU in both one-way and two-way, reaching a maximum bi-directional throughput of about 1.5 GBytes/s over all NICs. Moreover, copy overhead dominates in all cases, except for the smaller message sizes. CPU utilization for CP in two-way reaches up to 180% for larger messages, saturating both the send and receive path CPUs.

Replacing copies with remapping in Map results in a large performance improvement: Throughput increases by almost a factor of three in one-way and two-way and by a factor of two in ping-pong. In one-way we see that receiver path utilization is almost 100% and throughput reaches up to 3.5 GBytes/s. Results are similar in two-way where bi-directional throughput is about 4.5 GBytes/s, however, CPU utilization is about 150%, reflecting the saturation of the receive path. Any further improvement in throughput can mainly come from better distributing receive path protocol processing to multiple cores in future CPUs.

Figure 7(a) shows the performance improvement when using context-independent remapping (Map) vs. remapping that uses per-connection threads (MapSimple). We see that especially in one-way and two-way, throughput improves by up to 34%. In the rest of our evaluation, we use only the context-independent remapping technique.

Overall, delivering end-to-end wire throughput can be limited by two factors: (a) maximum memory throughput in our systems or (b) high CPU requirements for protocol processing and especially the receive path. If we artificially remove data copies and remapping (NoCP) throughput increases in all benchmarks to saturate either available link (one-way) or (single) memory bandwidth (two-way).

Figure 9(a) shows system latency for one-way and ping-pong. In one-way, the overhead of posting a write request is about 2 μs for small messages, increasing slightly with message size, for all configurations. In ping-pong, latency is 11.7(NoCP)-13.4(CP) μs for 4 Bytes messages, reaching 14.2(NoCP)-17.5(CP) μs for 1 KByte messages. Our results show that Map outperforms CP in all cases. We thus reserve the use of copying only in cases where our network hardware forces us to fit the Ethernet packet within a single hardware descriptor (for messages less or equal to 12 Bytes).

5.2 Protocol Scaling

Now we examine the scaling of the network protocol with increasing number of processing threads. Figure 8 shows throughput and CPU utilization for different protocol configurations. First, we consider the case of a single send thread with an increasing number of receive threads. In this case, one-way throughput scales from 3.5 GBytes/s to

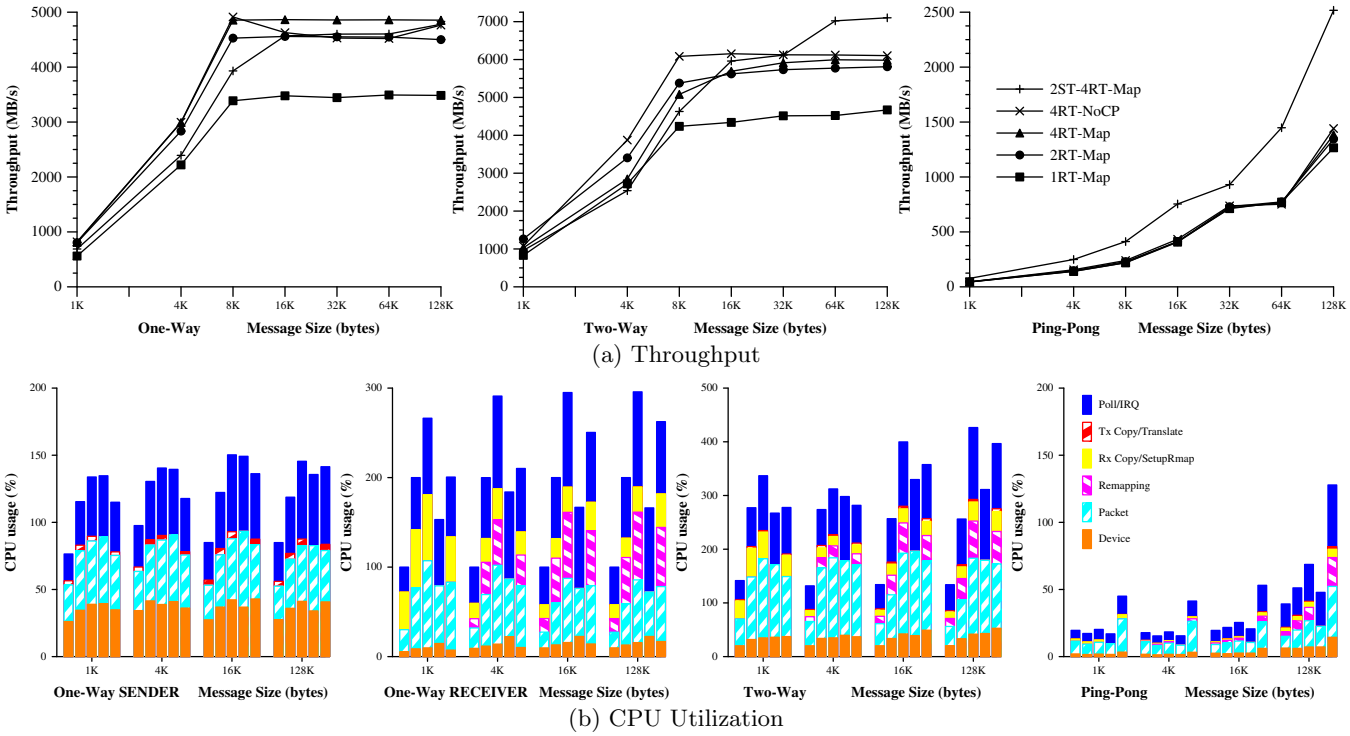


Figure 8: Protocol scaling with the number of protocol threads. For CPU utilization we show one bar per configuration (left to right): 1RT, 2RT, 4RT, 4RT-NoCP, 2ST-4RT.

about 4.9 GBytes/s (Figure 8(a)) reaching the maximum one-way throughput achievable on four NICs. We see that two receive threads are almost adequate for achieving this maximum throughput. Figure 8(b) shows that CPU utilization on the receive path for 1RT, 2RT, and 4RT is about 100%, 200%, and 300%, respectively. Thus, 2RT saturates two cores while managing to service all four NICs at link speed. In two-way, throughput scales from 4.5 GBytes/s to about 6.0 GBytes/s, with 2RT almost reaching maximum throughput. Similarly to one-way, in 2RT CPU utilization is about 280%, indicating that the two receive threads saturate two cores with the rest of the CPU utilization attributed to the sending thread. In 4RT, although there are spare CPU cycles available, throughput does not increase significantly beyond 6.0 GBytes/s since the bottleneck is the single memory throughput when using a single send thread.

Increasing the send threads to two for two-way (2ST-4RT-Map) results in a maximum throughput of about 7.2 GBytes/s at similar CPU utilization levels as in 4RT-Map. The increased throughput is a result of using both memories the system as opposed to mostly one memory when using a single send application thread. We discuss this issue further in the following subsection.

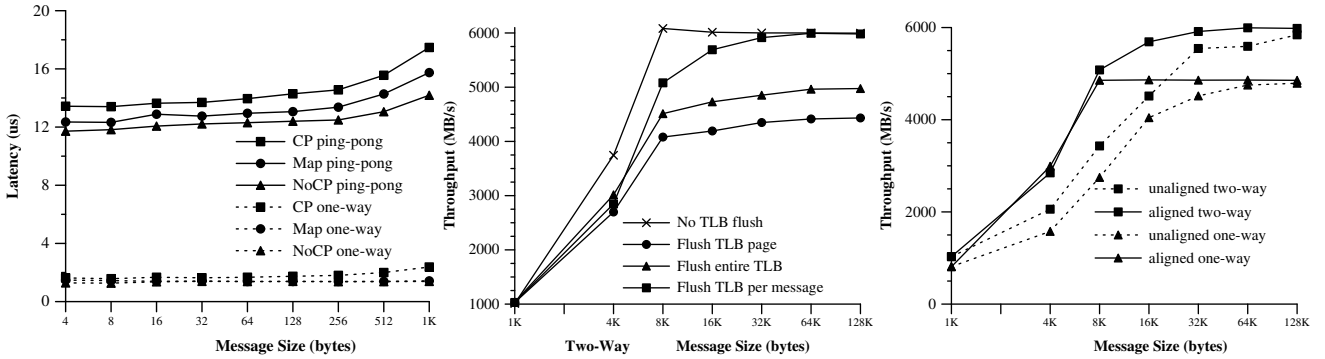
5.3 Affinity Issues

Our protocol configuration used so far does not try to exploit affinities to a significant degree, as such tuning cannot be expected from typical applications. Our highest achieved throughput of 7.2 GBytes/s in configuration 2ST-4RT-Map uses essentially send memory-NIC affinity but no receive memory-NIC affinity.

To explore further the impact of affinity we consider two additional configurations: one featuring DMA-direction and memory affinity and another featuring (both send and receive) memory and NIC affinity. First, we note that these types of affinity are mutually exclusive if one desires the simultaneous use of all available NICs and memories. For instance, DMA-direction and memory affinity, where the protocol performs only read or write DMAs to each of the two memories in the system, requires that send and receive buffers of all NICs be located in separate memory modules, breaking memory and NIC affinity. DMA direction-memory affinity results in maximum bidirectional throughput of about 6 GBytes/s, similar to single-memory performance, while memory-NIC affinity results in maximum bidirectional throughput of about 7 GBytes/s showing a measurable improvement to overall performance. A more detailed evaluation of different affinity types and configurations as well as dynamic protocol adaptation is beyond the scope of this work and we leave it for future work.

5.4 Impact of TLB Invalidation Mechanism

After remapping a receive buffer, TLB entries may be invalidated either selectively or by flushing the full TLB. In addition, TLB entries may be invalidated eagerly as soon as a page is remapped, or lazily, only after all pages related to a single packet or message are remapped. Our previous results use lazy-full TLB invalidations. Figure 9(b) shows two additional cases, eager-full and eager-selective TLB invalidations for *two-way*. Lazy-selective invalidations are not interesting as eager-selective would always result in less or equal overheads. We also include a curve where we artifi-



(a) latency (b) TLB (c) alignment

Figure 9: Message latency (a), impact of TLB invalidation mechanisms (b), and impact of data alignment (c).

cially do not flush any TLB entries, to illustrate the best possible case.

We see that the overhead when flushing the entire TLB depends on message size. Eager-full and eager-selective TLB invalidations are 16% and 25% worse than the ideal throughput with no invalidations. Lazy-full TLB invalidations scale better as message size increases, and for messages larger than 32 KBytes reach the same throughput as the ideal case. Also, it is important to note that when flushing the full TLB, although it appears to incur a lower CPU overhead, it may have an impact on overall application performance as the TLB may need to be refilled with flushed entries, especially for compute intensive applications.

5.5 Impact of Buffer Alignment

Finally, until now we have presented results using appropriate data alignment for send and receive buffers, such that page remapping is possible on the receive path for messages equal to or larger than 4 KBytes. Also, message size is a power of two, resulting in full page remappings for large messages. When send and receive buffers are not page-aligned, there is a need to copy part of the data and to use a larger number of packets. To fix alignment the first packet is used to align data appropriately and the last one to transmit the remaining, non-aligned data. These two packets have a total payload of 4 KBytes for messages larger than 4 KBytes, since the transfer size is a multiple of 4 KBytes. Figure 9(c) shows throughput for *one-way* and *two-way* when source and destination buffers are not aligned. For unaligned addresses, throughput increases with messages size, as more packets use remapping on the receive path, asymptotically reaching the maximum throughput of aligned buffers.

6. RELATED WORK

The last two decades there has been extensive research on communication subsystems for building cost-effective, high-performance clusters. To a large extent this research has focused on examining issues in the host-NIC interface, such as eliminating data copies, system call overhead in the communication path, and context switches [7, 16, 23]. Through this work, NIC architectures have evolved dramatically to low-latency, high-throughput designs that are decoupled from the processor-memory architecture [2]. Similarly there has been extensive work in evaluating various aspects of cluster interconnects and in different contexts [1, 13]. Our work in this paper differs from these efforts in that (a) we assume no protocol-specific support from the network interface, (b) we target 100 Gbits/s Ethernet-based networks, and (c) we take advantage of multicore CPUs in protocol design.

In [12], MultiEdge has been evaluated on a cluster of 32 nodes, using multiple 1-Gbit/s and single 10-Gbits/s Ethernet links, running an optimized software shared memory protocol and real applications. The emphasis there is on examining the impact of the lack of protocol support from the network switches on out-of-order delivery and packet loss. In [17] we examine the scalability of MultiEdge up to eight 1-Gbit/s links and present a detailed evaluation on the impact of different protocol costs on CPU utilization. In this work we design a protocol that avoids copying overheads without breaking existing APIs and scales on multiple cores to achieve a maximum bidirectional end-to-end transfer rate of more than 7 GBytes/s out of a maximum bandwidth of about 10 GBytes/s.

Address translation and page remapping have been proposed previously for eliminating the cost of crossing the user-kernel boundary in various contexts [3, 6]. The authors in [6] present a mechanism for transferring data over this boundary and deals with alignment issues and concurrent accesses. We use a similar technique in our receive path design. In addition we deal with alignment issues that are induced by Ethernet and the lack of protocol support at the NIC. Copy offloading is also achieved using hardware support in some processors [9]. While this approach results in delivering wire-speed for 10 Gbits/s link rates, CPU utilization remains high.

The packet re-shuffling technique we use is similar to header patching proposed in [3]. The main difference is that a scatter-list mechanism can be used on the receive buffers used by the hardware and each segment of this list can be transfered to a user buffer. However, this doesn't work in our case because we can only avoid copies using page-sized scatter list buffers. Moreover, we evaluate the effectiveness of this technique at much higher network speeds.

Distributing packet processing over multiple cores has been examined in [25]. The authors present the design of a network interface that uses multiple CPUs for 10 Gbits/s Eth-

ernet processing. However, they focus on NIC design rather than the host CPU communication stack. In contrast, our work do not rely on NIC support and we examine communication rates up to 100 Gbits/s. We believe that our approach is inline with current technology trends of using multicore CPUs as host processors.

Previous efforts that are related to our work in terms of the underlying platform include [20, 24]. The authors in [24] provide a communication protocol, UNet, on top of Fast Ethernet and ATM interconnects. Their goal is to provide high-bandwidth, low-latency communication on top of commodity interconnects. They focus on data transfers and describe how they can be performed directly from user space when the NIC provides a programmable CPU and what support is required at the kernel-level for less aggressive NICs. The authors in [20] present a user-level, zero-copy protocol design and implementation on top of 1 Gbit/s Ethernet, using a programmable Ethernet NIC. They achieve a minimum latency of 23 μs and a maximum bandwidth of 880 Mbits/s, close to our kernel-level protocol over a single 1 Gbit/s link. In our work, our goal is not to bypass the kernel. Instead, we are interested in eliminating the copy overheads while crossing the user-to-kernel boundary for transparency purposes.

The concept of end-to-end multi-link communication channels is similar to *inverse multiplexing* [8]. Inverse multiplexing has previously been applied to wide area network communication [4]. Moreover, this concept has been explored in the context of cluster interconnects: Multi-rail communication tries to take advantage of spatial parallelism and has been examined by previous work. The authors in [5] examine rail allocation methods for multi-stage cluster interconnects.

Finally, there are recent efforts to build multi-stage interconnects out of Gigabit Ethernet switches and NICs. The authors in [22] build a multi-dimensional hyper crossbar network using multiple Gigabit Ethernet interfaces in each node. They find that for a set of micro-benchmarks the system delivers more than 90% of the peak throughput. This work is orthogonal to our work in this paper as it focuses on the impact of the multi-stage interconnect rather than the degree of spatial parallelism.

7. CONCLUSIONS

In this work we design a parallel protocol for high-speed communication protocols over Ethernet-based interconnects. We examine how copies can be eliminated using page remapping and how protocol processing on the receive path can scale over multiple cores, taking advantage of current technology trends without at the same time imposing restrictions on existing APIs and buffer management semantics. We use a page-remapping technique that is context-independent to reduce the number of protocol threads and related overheads. We also discuss and optimize synchronization issues for parallelized protocol data structures.

We find that our page remapping results in 2-3x improvement and allows reaching a maximum of about 70% of available throughput in one-way and two-way respectively. After copy avoidance, the bottleneck is mainly receive path processing. Interrupt processing, page remapping, packet processing, and NIC accesses are all important to the extent that they are essential processing steps in the receive path and cannot be eliminated. Distributing receive protocol processing over multiple cores and optimizing the synchronization points, allows the protocol to scale to a maximum end-to-end throughput of 7.2 GBytes/s (57.6 Gbits/s or 72% of maximum bidirectional bandwidth of 80 Gbits/s). To our knowledge this is the highest throughput achieved with commodity systems and transparent, kernel-level communication protocols.

Overall, we believe that our approach of using multiple NICs (and network links) for increasing end-to-end throughput matches well the current technology trends in building multicore CPUs and that our protocol design is effective for delivering high throughput through standard and well-defined kernel-level APIs. We believe that the main issue remaining for future work is a more detailed examination of buffer and thread affinity issues and techniques for dynamically adapting protocol behavior to deal with application locality issues over heterogeneous multicore CPUs.

8. ACKNOWLEDGMENTS

We would like to thank Sven Karlsson for his contribution on earlier versions of this work. We thankfully acknowledge the support of the European FP6-IST program through the UNIsIX project and the HiPEAC Network of Excellence.

9. REFERENCES

[1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *IEEE/ACM Conference on Supercomputing*, Orlando, Florida, Nov. 1998.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet: A Gigabit-Per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] J. Brustoloni. Interoperation of Copy Avoidance in Network and File I/O. In *18th Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM)*, New York, Mar. 1999.

[4] F. Chiussi, D. Khotimsky, and S. Krishnan. Generalized Inverse Multiplexing of Switched ATM Connections. In *Global Telecommunications Conference (IEEE GLOBECOM)*, Sydney, Australia, Nov. 1998.

[5] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits. Using Multirail Networks in High-Performance Clusters. In *IEEE Cluster Computing*, Newport Beach, California, Oct. 2001.

[6] P. Druschel and L. Peterson. Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *14th ACM symposium on Operating systems principles (SOSP)*, Asheville, North Carolina, Dec. 1993.

[7] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects V*, Stanford, California, Aug. 1997.

[8] J. Duncanson. Inverse multiplexing. *IEEE Communications Magazine*, 32(4):34–41, Apr. 1994.

[9] B. Goglin. Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In *IEEE Cluster Computing*, Tsukuba, Japan, Oct. 2008.

[10] IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/.

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

[12] S. Karlsson, S. Passas, G. Kotsis, and A. Bilas. MultiEdge: An Edge-based Communication Subsystem for Scalable Commodity Servers. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, California, Mar. 2007.

[13] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. Panda, and P. Wyckoff. Microbenchmark Performance Comparison of High-Speed Cluster Interconnects. *IEEE Micro*, 24(1):42–51, Jan.-Feb. 2004.

[14] K. Magoutis, S. Addetia, A. Fedorova, and M. Seltzer. Making the Most out of Direct-Access Network-Attached Storage. In *2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, California, Mar. 2003.

[15] J. Mogul. TCP Offload is a Dumb Idea Whose Time has Come. In *9th Conference on Hot Topics in Operating Systems (HOTOS)*, Lihue, Hawaii, May 2003.

[16] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 5(2):60–72, Apr.-Jun. 1997.

[17] S. Passas, G. Kotsis, S. Karlsson, and A. Bilas. Expoiting Spatial Parallelism in Ethernet-based Cluster Interconnects. In *Workshop on Communication Architectures for Clusters (CAC). Held in Conjuction with IPDPS*, Miami, Florida, Apr. 2008.

[18] J. Pinkerton. The Case for RDMA, 2002. RDMA Consortium, http://www.rdmaconsortium.org/home/The_Case_for_RDMA02053.pdf.

[19] Stream Control Transmission Protocol (SCTP). http://www.ietf.org/rfc/rfc2960.txt.

[20] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing. In *ACM/IEEE Supercomputing*, Denver, Colorado, Nov. 2001.

[21] W. R. Stevens and G. R. Wright. *TCP/IP Illustrated (vol. 2): The Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] S. Sumimoto, K. Ooe, K. Kumon, T. Boku, M. Sato, and A. Ukawa. A scalable communication layer for multi-dimensional hyper crossbar network using multiple gigabit ethernet. In *20th Annual International Conference on Supercomputing (ICS)*, Cairns, Australia, June 2006.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *15th ACM symposium on Operating systems principles (SOSP)*, Cooper Mountain Resort, Colorado, Dec. 1995.

[24] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *3rd International Symposium on High-Performance Computer Architecture (HPCA)*, San Antonio, Texas, Feb. 1997.

[25] P. Willmann, H. Kim, S. Rixner, and V. S. Pai. An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. In *11th International Symposium on High-Performance Computer Architecture (HPCA)*, Palace Hotel, San Francisco, Feb. 2005.