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)
P.O. Box 1385, Heraklion, GR-71110, 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 protocol that uses context-independent page-remapping to reduce packet processing overheads, thread management overhead and synchronization in the common case, and reduce affinity issues in NUMA multicore CPUs. Our design result in the full 40 Gbits/s of available one-way Ethernet bandwidth and in about 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.

1 Introduction

Historically, high performance applications satisfy their communication needs through the use of specialized (and thus expensive) communication networks [4] that offer high-bandwidth, low-latency communication. Recent improvements in Ethernet technologies [1] however, promise link rates in the range of 40-100 Gbits/s, matching those of traditional high-performance interconnects. This paper explores the possibility of leveraging Ethernet as a cost-effective commodity solution for high-performance communication.

One of the challenges with using Ethernet as a high performance interconnect is the CPU and memory overhead typically associated with data transfer 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) [16]; protocol offloading [13]; and new network interface card (NIC) designs [22]. 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 in order to leverage increased network bandwidth while simultaneously freeing processor resources for application use.

In this paper we investigate efficient network data transfer over Ethernet networks in the 10-100 Gbits/s range assuming standard NIC programming interfaces and using multiple 10 GBit/s links to form a single logical connection. Our aim in expecting no special support from the network, either at the core (switches, routers, etc.) or its edge (NICs), is to take advantage of economies of scale in standard Ethernet NICs. 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. Also, we believe that future communication protocols will make use of spatial parallelism over multiple physical links to achieve high end-to-end throughput.

The system we present in this paper combines an appropriate networking API with RDMA semantics at kernel-level with an efficient implementation of a network communication protocol over Ethernet. The contributions of this paper are:

- An Ethernet transport protocol efficiently parallelized over multiple CPUs and cores on modern processors;
- A novel *context-independent* copy reduction technique based on VM page remapping that reduces packet processing cost and thread management overhead and synchronization; and
- An identification of challenges in parallel communication protocol design in NUMA multicore processors.

Both features require no special hardware support. This paper extends earlier work [14, 18] describing a non-parallel version of our Ethernet network transport protocol, named *MultiEdge*, which has the following characteristics:

- API support for RDMA and mixed, in-order and application-specific order message delivery.
- Lightweight flow-control optimized for inter-domain topologies.
- Transparent use of multiple physical links for data transmission.

Our results show that a base protocol that uses a more traditional page-remapping technique and is not parallel achieves a maximum one-way throughput of about 27.9 Gbits/s out of ideal 40 Gbits/s, whereas two-way throughput increases to about 37.3 Gbits/s out of ideal 80 Gbit/s. Parallelization of the transport protocol and using context-independent remapping improves one-way aggregate link 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 about 57.6 Gbits/s (74% of ideal) at 50% of total CPU utilization. Finally, our work is a first step towards mapping network protocols on emerging heterogeneous multicore CPUs and breaking the traditional bounding between CPU and NIC.

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

The high-performance communication protocol used in this paper [14] is geared towards high-speed Ethernet-based local-area networks. It 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 [23]. *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 [2]. *MultiEdge* is a fully end-to-end protocol that does not require any support from the network core.

In a departure from traditional socket (send/receive) based APIs over Ethernet, *MultiEdge* presents applications with a remote read/write memory API [16]. 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 *MultiEdge* API does not require any hardware support for RDMA and is implementable over standard Ethernet NIC hardware.

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 [23]. 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 [5, 8]. According to it, 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 [5]. 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 prohibit the application from accessing the 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 [4, 16] showed that enabling application preposting of receive buffers directly with the NIC offers such a mechanism. However this capability requires extensive NIC support and is thus expensive and outside the scope of our work. 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 Parallelization of Protocol Processing

Parallelization of the send path is typically straightforward, as multiple user contexts (threads or processes) that initiate communication operations can be mapped to different CPUs, cores, and NICs. We now focus on the parallelization of the receive path, which conceptually includes the following steps (Figure 1):

- Packets arrive in per-NIC (network) rings that are located in host memory. Each *Ethernet* ring may contain packets from different connections at the same time, as we would like to use all available physical throughput for a single connection.
- Packets are removed from the Ethernet rings and are placed to a per-connection (Rx) ring structure (possibly using the sequence number for indexing to avoid synchronization).
- Depending on message order semantics, in the common path, where there are no acknowledgements or retransmissions:
  - With *out-of-order* message delivery, packets are removed from the connection ring and processed. Processing involves remapping the buffers to the application address space for avoiding copying and doing all protocol book-keeping.
  - With *in-order* delivery message semantics, before packets are processed as above, the receiver needs to ensure that all packets of previous messages have been received.
- If the processed packet is the last segment of a message carrying a notification, send a notification to the application process.
- When necessary (uncommon path), send an acknowledgment based on certain conditions of packet ordering and system thresholds.
- Poll NICs for more work and start over.

These steps typically mandate specific mechanisms in the protocol design and implementation. First, removing packets from the Ethernet rings has to occur in per-NIC (network) threads that are woken up by NIC interrupts at packet arrival. Second, packet processing occurs in per-application threads that require access to application memory structures for page-remapping purposes. Third, for scaling protocol processing, the system needs to employ both multiple network as well as per-application protocol threads. Fourth, removing packets from the Ethernet rings and placing them in the connection rings, as well as removing them from the connection rings requires synchronization among concurrent contexts[1]. These implications result in (a) a higher number of threads than necessary, (b) more complex affinity characteristics between thread scheduling and protocol meta-data placement, and (c) synchronization in the common packet processing path.

In this paper we present a design that improves on these issues. A main challenge is to perform page-remapping directly by the network threads. This is challenging because these threads have no default knowledge of application VM structures and essentially need to take a new personality for each packet.

[1]This synchronization, depending on the protocol semantics and implementation can be optimized to use only atomic instructions as opposed to locks in the common path.

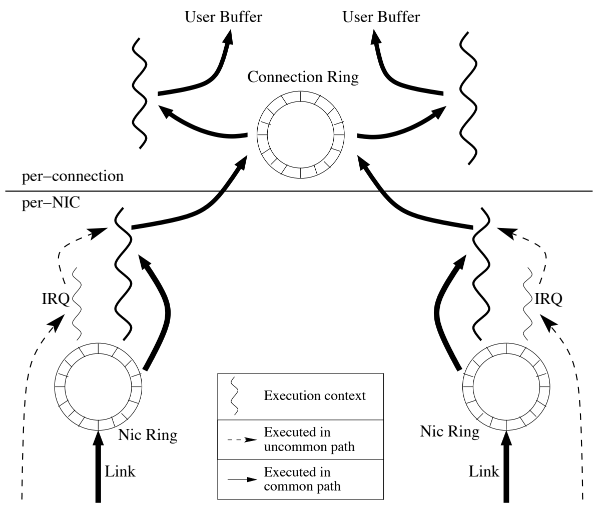


Figure 1. Flow diagram using per-connection threads.

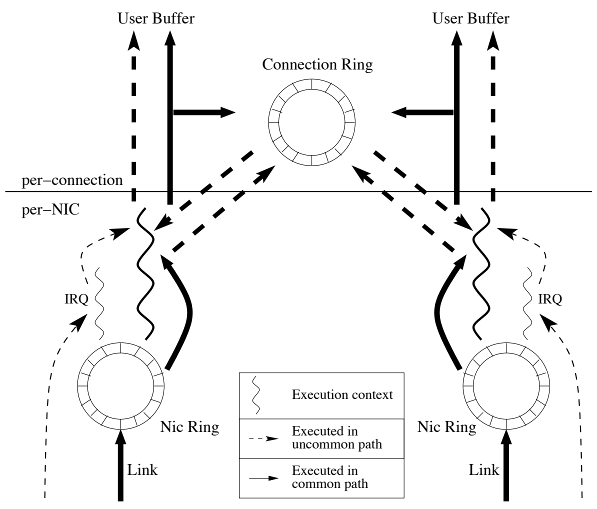


Figure 2. Flow diagram using per-NIC threads.

Once this is possible, then packet processing can happen at the network threads, eliminating the per-application process threads altogether. This, in turn, allows for removing all synchronization in the common, out-of-order delivery, case and simplifies to some extend affinity issues.

Figure 2 shows the main structures and common-path operations in our protocol. Solid lines correspond to out-of-order processing mode, which is the common path in the protocol, while dashed lines correspond to in-order processing, which is the uncommon path. In the common case, network threads remove packets from ethernet rings independently, directly remap them to user buffers, and mark the packet as delivered in the connection ring. The uncommon path requires first placing packets in the connection ring, and then processing packets of single messages in parallel by scanning the ring using a block atomic increment operation (atomic increment by N).

Next, we discuss how our design achieves efficient network packet handling relying on (and benefiting from) use of context-independent VM remapping, how it handles synchronization issues arising from network threads to shared protocol states, and how it limits resource affinity challenges in NUMA architectures.

3.1 Context-independent VM remapping

Network threads have their own contexts, which differ from application contexts shared by per-process threads. Performing VM remap operations on application memory from system-level threads is a key challenge in our system.

A receive thread must be able to manipulate the virtual address space of the application where the packet is delivered. To achieve this, when a process initializes the device we store a pointer to the kernel's memory manager structure. During data packet processing, when the need arises to access a process's user address space we appropriately restore the corresponding memory manager. For x86/x86_64 architectures it suffices to replace the `cr3` register to point to the page table of the process we need to access and change certain system variables to leave the system in a consistent state.

As a further optimization to this mechanism, each time the application registers explicitly a buffer we store the kernel's page table stucture (usually the different levels of the page table entries, if they exist). During the VM remapping, we use the cached page table structures to locate the page table entry that points to the old page manually and replace it with the new page. This prevents invalidation of the proper page entry and causing a software page-fault to put the new page into the page table. On both places we explicitly flush the TLB to prevent reading stale data by the application. Finally, we update our cached page tables to keep our data consistent with the kernel's page tables.

3.2 Thread Synchronization

In our parallel design we ensure that threads synchronize over access to the connection ring to avoid con-

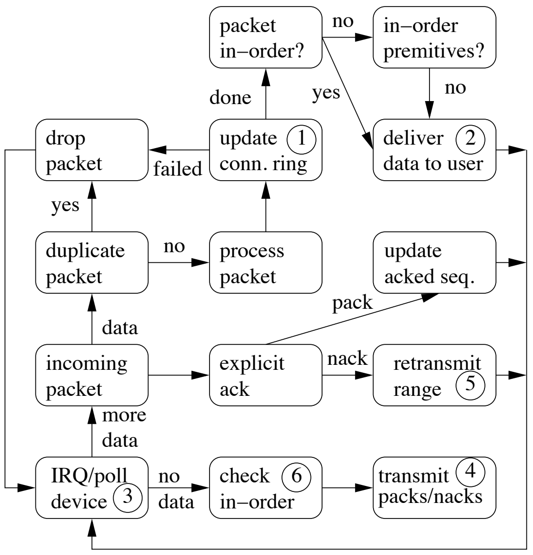


Figure 3. ***MultiEdge*** **protocol flow diagram. Numbers correspond to synchronization points in Figure 4.**

Type	Description
trylock (1)	Protect accesses per buffer entry
trylock (2)	Protect accesses per memory page
trylock (3)	Prevent concurrent tx. completions
trylock (4)	Prevent concurrent tx. of explicit acks
trylock (5)	Prevent concurrent retransmissions due to nacks
trylock (6)	Prevent concurrent scanning of the ring in-order
atomic	all shared flow control sequence numbers

Figure 4. Summary of synchronization points.

flicts in the following cases (Figures 3 and 4):

1. When a data packet is assigned to a protocol thread, that thread must determine whether the incoming packet is a duplicate. To serialize on accesses to the connection receive ring, threads use atomic instructions to update shared variables such as the maximum sequence number and individual entries on the receive ring.

2. In the (uncommon) case that overlapping messages are processed concurrently, a per-buffer lock is required to avoid concurrent remapping of a single buffer.

3. When multiple threads concurently poll the devices for transmission completion events, ensure a single thread is processing all such events.

4. In the case of a positive or (uncommon) negative acknowledgment, the protocol uses a per-connection try-lock to ensure that only a single thread transmits the ack.

5. In the (uncommon) case of retransmitted packets, the same packet may end up being processed by multiple threads. This case requires a try-lock per entry on the connection receive ring to ensure that only one of the duplicates is processed.

6. In the (uncommon) case that in-order processing is dictated by message semantics, a try-lock is required to ensure that packets are processed in-order. This effectively allows a single thread to do the in-order processing, while the remaining threads are free to process other pending work.

3.3 NUMA Affinity Issues

The NUMA architecture of recent multicore processors such as the one used in this study (Figure 5) results in possible variations in memory throughput depending on the relative placement (or *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 direction and memory modules;

We obtained results corresponding to 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

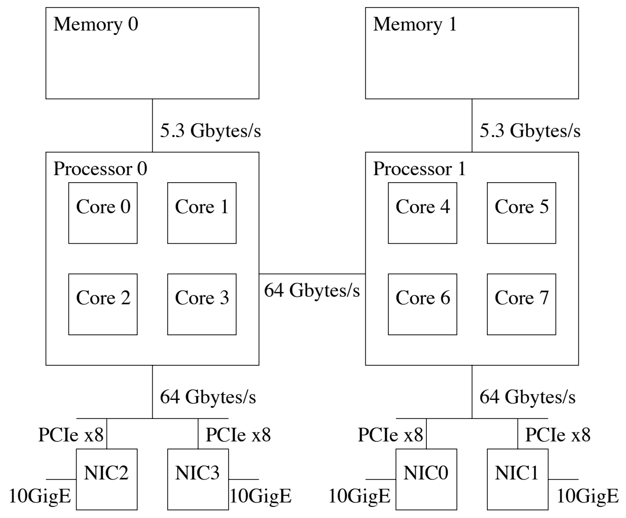


Figure 5. Internal data paths in each system node

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) each memory module. Our protocol configuration so far does not try to exploit affinities to a significant degree, as such tuning would not be expected of typical applications.

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 cache 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 (Not-Uniform Memory Access) 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.

We conduct experiments using MTU size of 9000

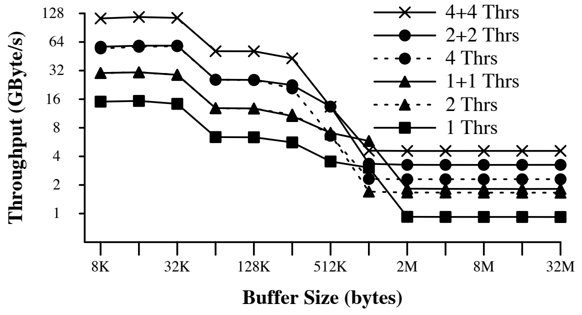


Figure 6. Memory copy throughput.

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

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 the send and receive paths that are exercised simultaneously. *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 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. 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.

	local memory (MBytes/s)				remote memory (MBytes/s)			
	1 core	1+1	2+2	4+4	1 core	1+1	2+2	4+4
read	2004	4008	7760	14344	1862	3724	6680	9536
write	2600	5200	7480	8336	2000	4000	5240	5280

Table 1. Memory throughput when cores access data located on local or remote memory.

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.

Finally, 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, and translation overheads. In all experiments, we report the average of five measurements for each data point.

Figure 6(b) 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). Table 1 shows throughput for memory read and write separately. For one core accesses to local memory have a sustained rate of about 2 and 2.6 GBytes/s for reads and writes respectively, but it drops significantly when accessing remote memory.

Table 2 shows the overhead of certain basic 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.

5 Results

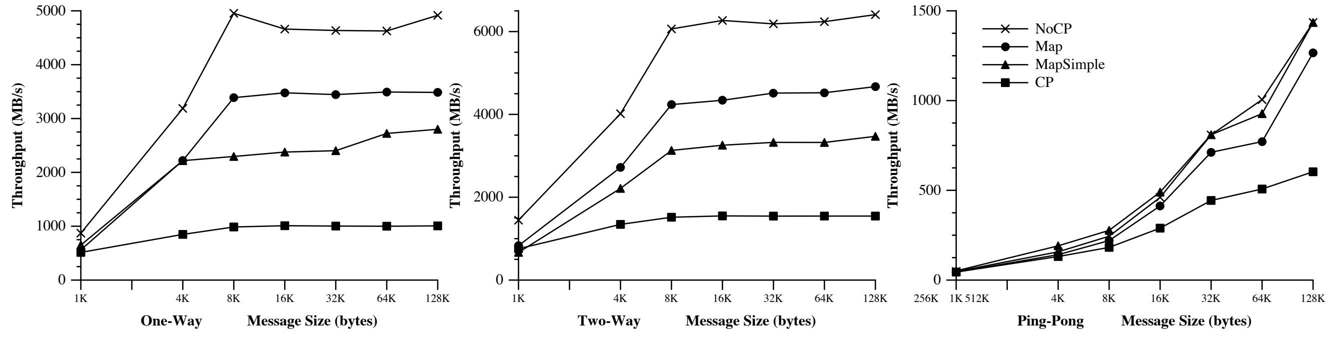
We structure results around the following questions: (a) The benefits from page remapping in the range of 40-80 Gbits/s; (b) Protocol scaling with multiple CPUs and protocol threads; (c) Affinity issues; and (d) The impact of different TLB invalidation schemes and buffer alignment.

5.1 Benefits of page remapping

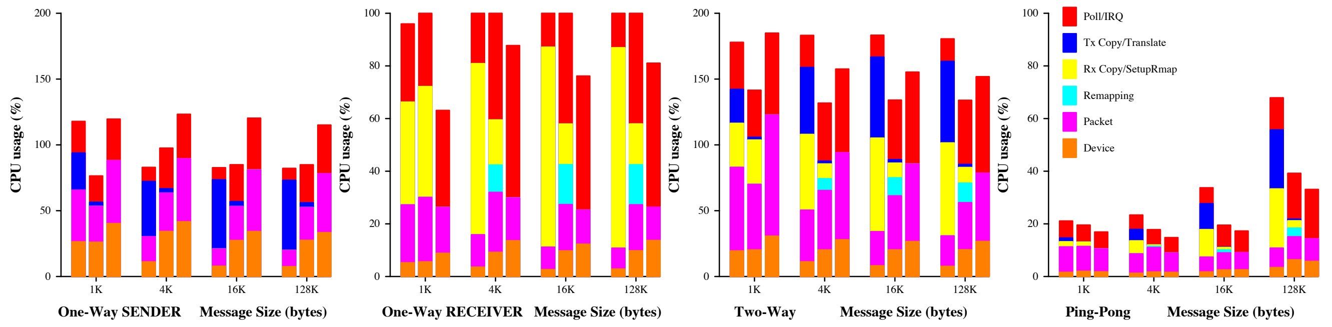
Figure 7(a-b) shows throughput and CPU utilization when using copy (CP) vs. page remapping (Map) and contrasts them with the ideal version that artificially avoids both (NoCP). We see that CP, which uses one copy on the send and one copy on the receive path, is limited by CPU in both one-way and two-way, 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. Thus, 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.



(a) Throughput



(b) CPU Utilization

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.

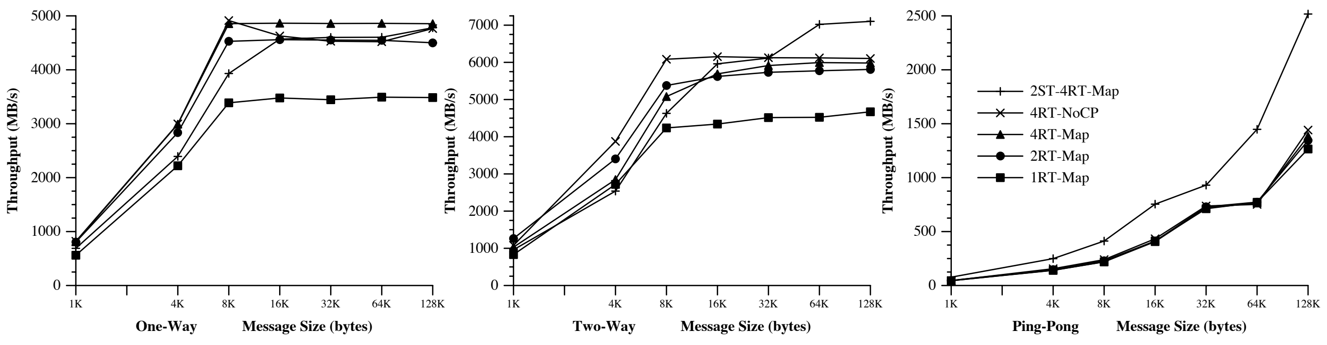
remapping that uses per-connection threads (MapSimple). We see that especially in one-way and two-way, throughput improves by up to 34%. In the rest of our evaluation, we use only the context-independent remapping technique.

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).

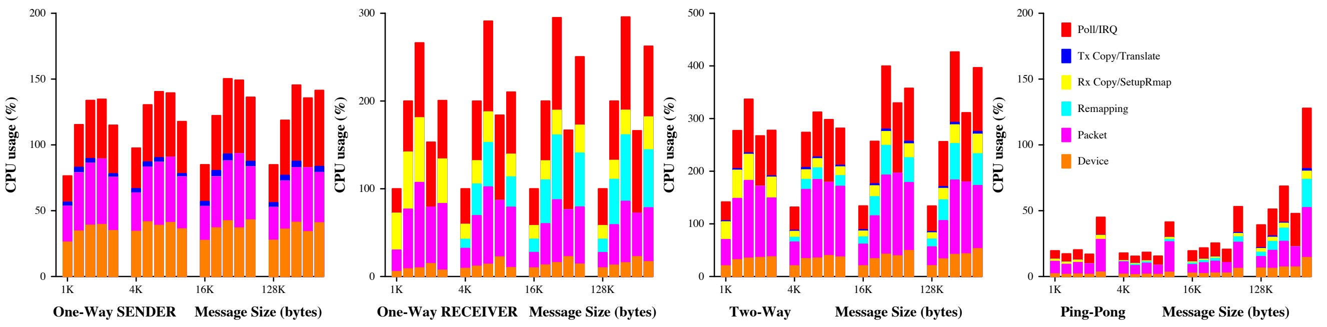
Figure 9(a) shows the system's latency for one-way and ping-pong. Ping-Pong exhibits a latency of 11.7-13.4 μs for 4 Byte messages and reaches 16.5-21.2 μs for 2 KByte messages. One-way shows the overhead of posting a write request to be about 2μs for 4 Byte messages for all configurations and increase slightly with message size. We see that for messages up to 2 KBytes Map performs better due to the lack of copy on the send path. The smallest packet size in Ethernet is 60 bytes. Packets of this size are required from our hardware to fit in a single hardware descriptor. Our header size is 48 bytes, thus, if the payload size is less or equal to 12 bytes, we need to perform a copy when creating each packet. We see that Map performs always equal or better compared to CP, thus, there is no need to set a larger threshold to regulate between copying and creating the gather list on transmit side.

5.2 Protocol scaling

Next 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 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



(a) Throughput



(b) CPU Utilization

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.

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 more in the next subsection.

5.3 Affinity issues

Our protocol configuration used so far does not try to exploit affinities to a significant degree, as such tuning would not be expected of 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-memory affinity and another featuring (both send and receive) memory-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-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-NIC affinity. A test with DMA direction-memory affinity results in maximum bidirectional throughput of about 6 GBytes/s, similar to single-memory performance, while a test with memory-NIC affinity results in maximum bidirectional throughput of about 7 GBytes/s underlining a measurable impact 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.

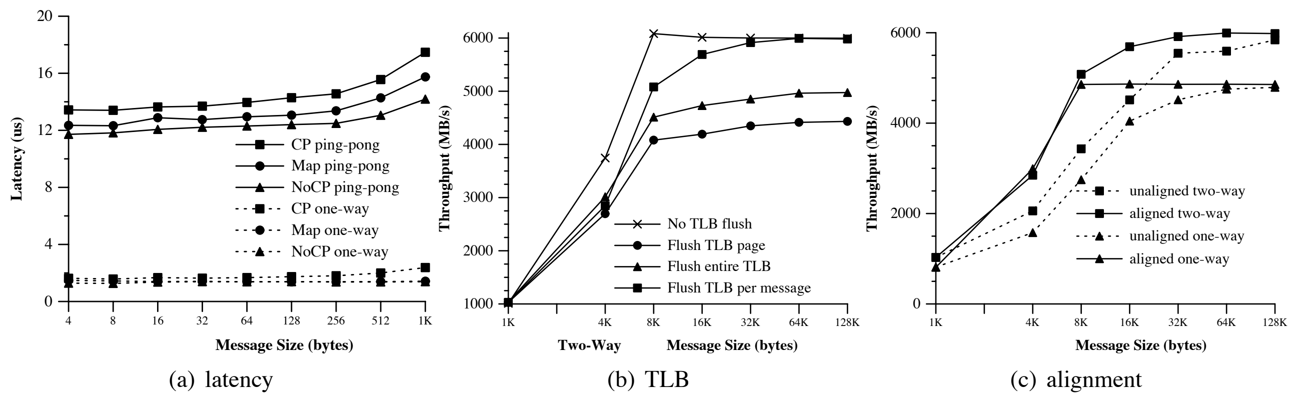


Figure 9. Message latency (a), impact of TLB invalidation mechanisms (b), and impact of data alignment (c).

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 artificially 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 [9, 15, 20]. Through this work, NIC architectures have evolved dramatically to low-latency, high-throughput designs that are decoupled from the processor-memory architecture [4]. Similarly there has been extensive work in evaluating various aspects of cluster interconnects and in different contexts [3, 12]. 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 40-80 Gbits/s Ethernet-based networks, and (c) we take advantage of multicore CPUs in protocol design.

In [14], *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 [18] 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 [5, 8]. The authors in [8] 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 [11]. 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 [5]. 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 is able to be transfered to a user buffer. However this cannot work on our operating system, we can only avoid copies using page size 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 [22]. The authors present the design of a network interface that uses multiple CPUs for 10 Gbits/s Ethernet processing. However, they focus on NIC design rather than the host CPU communication stack. In contrast in our work, we do not rely on NIC support and we examine communication rates up to 80 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 [17, 21]. The authors in [21] 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 [17] 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* [10]. Inverse multiplexing has previously been applied to wide area network communication [6]. 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 [7] 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 [19] 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 examine the implications of host-level copies 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 find that eliminating copies with address translation and 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 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 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 very 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 placement and techniques for dynamically adapting protocol behavior to application requirements (with respect to 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.

References

[1] IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force.

[2] Stream Control Transmission Protocol (SCTP).

[3] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *SC98: High Performance Networking and Computing*, Nov. 1998.

[4] 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.

[5] J. Brustoloni. Interoperation of copy avoidance in network and file I/O. *The 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99).*, 1999.

[6] F. M. Chiussi, D. A. Khotimsky, and S. Krishnan. Generalized inverse multiplexing of switched atm connections. In *Global Telecommunications Conference 1998 (GLOBECOM'98)*, Nov. 1998.

[7] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits. Using Multirail Networks in High-Performance Clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):625–651, 2003.

[8] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the Fourteenth Symposium on Operating Systems Principles (SOSP14)*, pages 189–202, Dec. 1993.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. Vmmc-2: Efficient support for reliable, connection-oriented communication. In *Hot Interconnects V*, 1997.

[10] J. Duncanson. Inverse multiplexing. In *IEEE Communic. Magazine*, pages 34–41, Apr. 1994.

[11] B. Goglin. Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In *Proceedings of the IEEE International Conference on Cluster Computing*, Sept. 2008.

[12] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. P. Kini, D. K. Panda, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, 2004.

[13] J. Mogul. Tcp offload is a dumb idea whose time has come. In *Proc. of the 9th conference on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[14] Reference omitted due to blind review process.

[15] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): efficient, portable communication for workstatin clusters and massively-parallel processors. *IEEE Concurrency*, 1997.

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

[17] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Proc. of the ACM/IEEE Conference on Supercomputing*, Nov. 2001.

[18] Reference omitted due to blind review process.

[19] 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 *Proc. of the 20th ACM International Conference on Supercomputing (ICS06)*, pages 107–115, June 2006.

[20] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP15)*, pages 40–53, Dec. 1995.

[21] M. Welsh, A. Basu, and T. von Eicken. Atm and fast ethernet network interfaces for user-level communication. *Proc. of The 3nd IEEE Symposium on High-Performance Computer Architecture (HPCA3)*, 1997.

[22] P. Willmann, H. Kim, S. Rixner, and V. Pai. An Efficient Programmable 10 Gbit Ethernet Network Interface Card. In *Intern. Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[23] G. Wright and R. Stevens. *TCP/IP Illustrated Vol. 2: The Implementation*. Addison-Wesley, 1996.