Computer Science Department
University of Crete

Analysis and Optimization of Overheads in Communication Protocols Over High Speed Ethernet-based Cluster Interconnects

Master's Thesis

Stavros Passas

March 2008

Heraklion, Greece

University of Crete

Computer Science Department

Analysis and Optimization of Overheads in Communication Protocols Over High Speed Ethernet-based Cluster Interconnects

Thesis submitted by

Stavros Passas

in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Stavros Passas

Committee approvals: _____

Angelos Bilas

Associate Professor, Thesis Supervisor

_____

Manolis Katevenis

Professor

_____

Evangelos Markatos

Professor

Departmental approval: _____

Panos Trahanias

Professor, Director of Graduate Studies

Heraklion, March 2008

Abstract

At the core of contemporary high performance computer systems is the communication infrastructure. For this reason, there has been a lot of work on providing low-latency, high-bandwidth communication subsystems for clusters. In this work, we introduce MultiEdge, a connection oriented communication system designed for high-speed commodity hardware.

We use MultiEdge to examine the behavior of edge-based protocols. We examine the implications of building a single logical link out of multiple physical links and we see how overheads and performance scale with the number and speed of links. Finally, we examine the costs associated with data copying at the 15-30 GBits/s range. We implement and show the effectiveness of zero-copy data transfers, analyzing the impact of additional costs associated with this approach.

We find that: (a) Our base protocol reaches almost 99.2% of the nominal aggregate throughput for up-to 2 psychical links of 1 GBit/s link rate. (b) When up-to 8 physical links are used, our protocol achieves up-to 65% of the nominal aggregate throughput. (c) The impacts of interrupts and data copies are significant, and when they are removed, protocol reaches 80% and 100% of the nominal throughput respectively. (d) With bi-directional link rates of 30 GBits/s, data copies limits the effective throughput to about 8.8 GBits/s on our systems. (e) The use of our zero-copy mechanism results in 80-90% improvement and reaches up-to 57% of the available bi-directional throughput. Finally, we believe that efficiently distributing protocol pro-

cessing over multiple host CPUs emerges as the main challenge in achieving higher transfer rates on modern architectures.

Supervisor professor: Angelos Bilas

Περίληψη

Στον πυρήνα των σύχρονων υπολογιστικών συστημάτων υψηλής επίδοσης είναι η υποδομή επικοινωνίας. Για αυτόν τον λόγο, έχουν εργαστεί πολλοί άνθρωποι για την δημιουργία υποσυστημάτων επικοινωνίας για συστοιχίες υπολογιστών, με χαρακτηριστικά την χαμηλή καθηστέρηση και την υψηλή ταχύτητα. Σε αυτήν την δουλειά, εισάγουμε το $MultiEdge$, που είναι ένα βασισμένο σε συνδέσεις, σύστημα επικοινωνίας και είναι ειδικά σχεδιασμένο για υψηλής ταχύτητας καθημερινό εξοπλισμό.

Χρησιμοποιούμε το $MultiEdge$ για να εξετάσουμε τη συμπεριφορά των πρωτοκόλλων που εφαρμόζονται μόνο στα άκρα του δικτύου. Εξετάζουμε τις συνέπειες της δημιουργίας μιας λογικής ζεύξης από πολλές φυσικές ζεύξεις και βλέπουμε πως η επίδοση και οι επιβαρύνσεις κλιμακώνονται με των αριθμό των ζεύξεων. Τέλος, εξετάζουμε τα κόστη που σχετίζονται με την αντιγραφή δεδομένων στο εύρος των 15-30 $GBits/s$. Υλοποιούμε και δείχνουμε την αποτελεσματικότητα τις μεταφοράς δεδομένων, χωρίς την ανάγκη αντιγραφής τους, αναλύοντας την επίπτωση στα επιπρόσθετα κόστη που σχετίζονται με αυτήν την προσέγγιση.

Βρήκαμε ότι: (α) Το πρωτόκολλό μας μπορεί να χρησιμοποιήσει περίπου το 99.2% της συνολικής ταχύτητας όταν χρησιμοποιούνται μέχρι 2 φυσικές ζεύξεις. (β) Όταν χρησιμοποιούνται μέχρι 8 φυσικές ζεύξεις, το πρωτόκολλο φτάνει μέχρι το 65% της συνολικής ταχύτητας. (γ) Η επίπτωση των διακοπών και των αντιγραφών δεδομένων είναι σημαντική, και όταν αυτά αφαιρεθούν, το πρωτόκολλο φτάνει το 80% και 100% της συνολικής ταχύτητας αντίστοιχα.

(δ) Όταν η αμφίδρομη ταχύτητα ζεύξης είναι κοντά στα 30 $GBits/s$, οι αντιγραφές δεδομένων περιορίζουν τη ταχύτητα του πρωτοκόλλου περίπου στην τάξη των 8.8 $GBits/s$ στα συστήματά μας. (ε) Η χρησιμοποίηση του μηχανισμού που δεν κάνει αντιγραφές δεδομένων, έχει σαν αποτέλεσμα την βελτίωση της απόδοσης του συστήματος κατα 80-90%, και φτάνει στο 57% της διαθέσιμης αμφίδρομης ταχύτητας. Τέλος, πιστεύουμε οτι η αποδοτική κατανομή του πρωτοκόλλου σε πολλούς επεξεργαστές φαίνεται σαν η κύρια πρόκληση στην επίτευξη υψηλότερων ταχυτήτων στην σύγχρονη αρχιτεκτονική υπολογιστών.

Επόπτης καθηγητής: Άγγελος Βίλας

Acknowledgments

I feel grateful to my supervisor, Angelos Bilas, for his valuable assistance and guideline in my academic steps in the field of Computer Science. The extensive discussions about my work and his wide knowledge have been of a great value for me.

A big thanks to Sven Karlsson for his support and help to a major part of my work. Moreover, I would like to thank my friend and colleague, Georgios Kotsis, for his help, especially in the beginning of my thesis.

My warmest appreciation to the following, past and current, members of the CARV Laboratory of the ICS/FORTH, whom I feel to be friends more than just colleagues: Mixalis Flouris, Stamatis Kavadias, Jesus Luna, Manolis Marazakis, Thanos Makatos, Giorgos Nikiforos, Giorgos Panagiotakis and Vassilis Papaefstathiou.

I would like also to thank my friends : Giannis Avgouleas, Sofia Doulgeraki, Giannis Georgalis, Fay Grivokostopoulou, Giorgos Iakovidis, Grigoris Kanakousakis, Maria Karagiorgaki, Socratis Kartakis, Lito Kriara, Vassilis Lekakis, Alexandra-Iuliana Magdalenoiu, Georgia Margariti, Zinovia Mitraka, Giannis Nikolopoulos, Panagiotis Palias and Eugenia Pantouvaki. Their encouragement and discussions have been a great value of me.

Last but not least, I would like to thank my family, my parents Giannis and Georgia and my brother Lefteris for their support and encouragement they provided me with.

Stavros Passas

Heraklion, February 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication infrastructure for scalable systems has recently gone through a wave of commoditization. Most scalable systems today, such as parallel systems for scientific and commercial applications rely on interconnects that plug in the I/O bus and are designed independently of the system motherboard and CPU [7, 21, 31]. This has had a significant effect on system cost-effectiveness and has allowed for extensive use of scalable systems in new application domains.

However, such communication subsystems require not only the use of specialized network interface cards (NICs) but switches as well. The result is that scalable systems need to employ multiple interconnects for different purposes. Typically, such systems are already interconnected with high-end Ethernet-based networks. In addition, they require one or two interconnects for different application domains, e.g. a system area network for compute-oriented applications and a storage area network for access to storage. This physical partitioning of systems based on their connectivity, results in excessive system costs and management complexity.

These existing, physically partitioned architectures are not able to satisfy requirements in new application domains. For instance, emerging networked storage systems require high communication throughput among stor-

age nodes as well as between storage and application nodes. Thus, although it is possible to use a domain-specific interconnect such as Fiber Channel or Infiniband among storage nodes, this may not be possible for application nodes.

A computer cluster is a group of coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks.

Clusters are usually deployed to improve performance and/or availability over that provided by a single computer, while typically being much more cost-effective than single computers of comparable speed or availability. 81.3% of the 500 most powerful computers are clusters, including the fourth and the fifth fastest systems in the world, as they appear in top supercomputer sites list at November 2007 [38].

Over the decade, many networks have been designed specifically for computer clusters. These networks, such as MyriNet, Infiniband or Quadrics, are custom made, which increases significantly their cost. Their main advantage is that their custom design usually increases network speed and reduces CPU overhead. However, Ethernet-based networks continue to improve in speed, This, combined with the low cost of Ethernet hardware, makes a large number of scalable systems use Ethernet. For instance, 66.5% of the cluster systems in the Top500 list use Gigabit Ethernet as their interconnect.

The main difference of the communication subsystems used traditionally in scalable systems has been the degree of support required from the network for the communication protocol. Based on this, we can divide interconnects in two categories: core-based and edge-based.

Most cluster interconnects today are core-based, e.g. Myrinet [7], Infiniband [21], and Quadrics [31]. These interconnects rely on the network core, i.e., the switches for providing FIFO ordering, flow-control, and reliable com-

munication. On the other hand, edge-based interconnects, such as Ethernet, incorporate all "intelligence" at the networked edge, i.e., NICs and hosts, and only simple forwarding functions are required from the network core. Besides this important difference, both core- and edge-based interconnects are already relying on the same type of physical links, i.e., 2.5-10 GBit/s serializers-deserializers.

An emerging aspect of high-end communication subsystems that may further blur differences is the use of spatial parallelism. Spatial parallelism is a new dimension in the design of high-end interconnects that uses multiple physical paths in a decoupled manner to carry the traffic of a single, end-to-end communication channel. Multiple links are already used today in high-end communication systems for byte-level parallelism: A single data unit sliced in bytes, is transmitted over multiple physical links that are tightly controlled by the sender and the receiver. However, as the number of links increases, it becomes difficult to control the links tightly and to achieve efficient byte-level parallelism.

Another approach to exploiting spatial parallelism is to transparently send full frames on top of separate links. We believe that for technology reasons similar to multi-core CPUs, the use of spatial parallelism in this decoupled manner will be a main factor in improving communication throughput. However, such systems may exhibit increased congestion, out-of-order delivery, and impose increased processing demands at the edge of the communication subsystem.

Given the lower cost and proliferation of edge-based networks, such as 1- and 10-Gigabit Ethernet, it becomes important to examine protocol layers that can support traditional end-to-end communication semantics and spatial parallelism for serving different application domains.

1.1 Thesis Contributions

In this thesis we present the design and implementation of an Edge-based communication subsystem, MultiEdge, which uses Ethernet and provides RDMA-type operations, FIFO ordering, reliable transmission. Then, we focus on data copies and show how we can avoid them, and the result of this change on system's performance.

Our contributions are:

1. We show how MultiEdge is able to support spatial parallelism. We also present a novel communication API that allows users to send data out-of-order in a single communication channel.

2. We design and implement zero-copy transfers, using commodity hardware, without hardware specific functions, and we present the associated challenges.

3. We use MultiEdge to examine in detail the impact of edge-based protocols on network traffic and system performance. We examine and understand the overheads in building high-throughput logical links by using a number of physical links and how overheads scale at the host-to-link interface. We investigate the impact of copying, interrupts, and packet scheduling over multiple links.

We find that our protocol can deliver about 99.2% and 66% of the nominal throughput when we use a single 1 and 10 Gbit/s physical link respectively. On the other hand, the use of multiple links is limited by interrupt and copy overheads. Replacing interrupts with polling results in similar maximum throughput (800 MBytes/s) in 8x1 and 1x10 configurations, limited by memory copies. Artificially removing protocol copies results in achieving 100% of nominal throughput with the one-way test in all link configurations.

Furthermore, when we use a 15 Gbit/s per direction network, copying limits maximum one-way throughput to about 7.7 GBits/s at 100% CPU utilization. Using our page remapping technique, we can achieve a maximum one way throughput of about 14.7 GBits/s out of ideal 15 GBits/s, whereas two way throughput increases to about 17 Gbits/s. However, overheads are more balanced and interrupt processing, remapping, packet processing, and NIC accesses all contribute significantly to CPU utilization. Finally, we find that the cost of page remapping mechanism itself is not very costly on today's CPUs, given that we already cross the kernel-user boundary once.

Overall, our work shows that edge-based protocols have the potential for significantly reducing the cost of scalable systems in the range of a few hundred nodes. We believe that appropriately distributing protocol processing on multiple cores of current future CPUs will result in end-to-end throughput in the range of 15-30 GBits/s reducing the performance gap between commodity and specialized interconnects.

Finally, the work in this thesis has appeared in [22, 29, 30].

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents the design and implementation of MultiEdge. We present and discuss our experimental platform and results in Chapter 3. Chapter 4 refers to related work. Finally, we summarize our work and draw conclusions in Chapter 5.

Chapter 2

Design

2.1 MultiEdge

MultiEdge is organized as three layers: the hardware drivers, the kernel level protocol layer, and a user-level library that interface applications to the kernel parts, see Figure 2.1. The Ethernet hardware drivers access the Ethernet hardware directly and provide a hardware independent interface to the protocol layer. The drivers perform hardware initialization, Ethernet frame reception and transmission, and low-level interrupt processing. The protocol layer is hardware independent, implements the programming API and adds higher level functionality such as end-to-end flow control, reliable data transfer, and high-level interrupt processing.

2.1.1 Communication primitives

MultiEdge provides a set of point-to-point, connection-oriented communication primitives; Before any communication can occur between two nodes, a connection has to be set up between the nodes. The programming API of MultiEdge has two primitives for this purpose: `connection_wait` and `request_connection`. The first one blocks until a connection is established while the second one initiates a connection to another node. For a connection to be established one node will have to execute `connection_wait` and

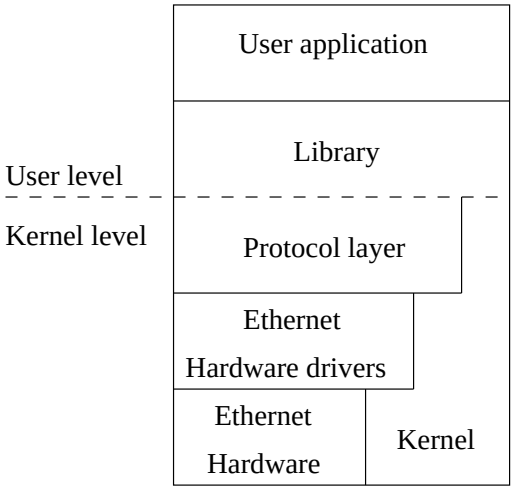


FIGURE 2.1: Overview of MultiEdge layers

the other will have to execute `request_connection`.

Once a connection is set up, communication is based on asynchronous remote memory operations. Currently, there are two remote memory operations: remote read and remote write. Each operation can access all the virtual address space of a process executing on a remote node. Both operations are fully asynchronous and are initiated by a single communication primitive that allows pointer arithmetic to remote addresses:

```
int RDMA_operation(connection,
                   remote_address,
                   local_address,
                   transfer_size,
                   operation,
                   flags);
```

`connection` refers to the connection on which the operation is initiated. `remote_address` and `local_address` indicate the virtual addresses on the remote and local nodes respectively. `transfer_size` specifies the size of

the data in bytes. `operation` specifies whether the operation is read or write. A remote memory write operation will cause a memory region of size `transfer_size` bytes starting at address `local_address` to be copied to the remote node at address `remote_address`. Similarly, a remote memory read operation will cause a memory region of size `transfer_size` bytes starting at address `remote_address` at the remote node to be copied to the local node at address `local_address`. `flags` is a bit-field of various options that modify the behavior of operations. Each operation can, when initiated, return a handle. The programmer can query the progress of each issued operation using the operation handle with the `query_event` primitive.

One important aspect of MultiEdge's API is that although the API includes primitives for registering memory regions, receive buffers need not be pre-registered. Data is instead copied directly into the virtual address space of the receiver.

Finally, the API provides a mechanism that delivers a notification to the remote node when selected remote memory write operations have finished. The programmer selects the remote memory write operations that will cause the delivery of notifications by setting a bit in the `flags` bit-field when invoking the `RDMA_operation` API call. Notifications are important to support asynchronous communication that is essential, e.g. for storage subsystems, and can be used to trigger handlers for incoming data.

2.1.2 Flow control

MultiEdge uses end-to-end flow control to ensure reliable communication. All operations and transfers are guaranteed to complete in the presence of dropped Ethernet frames due to transient problems, e.g. contention, bit errors, or transient link failures. We use a sliding window flow control algorithm with a fixed size window. The flow control algorithm operates on an Ethernet frame basis. The size of the window is set at compile time and the current default value is 512 frames.

The default behavior of MultiEdge is to deliver frames in-order. The receive and transmit paths each have a data buffer that is capable of holding as many Ethernet frames as there can be in the flow control window. These buffers are used for retransmissions and to reorder frames so that in-order delivery is achieved.

The receive path uses positive acknowledgments to notify the sender of received frames and negative acknowledgments to report back lost or damaged frames that need to be retransmitted. MultiEdge uses piggy-backing to reduce the number of explicit acknowledgments. All data frames carry positive acknowledgment information. Thus, when there is two way traffic in the system there is no need for explicit positive acknowledgment frames.

To further reduce the number of explicit acknowledgments, MultiEdge uses delayed acknowledgments: it will defer transmission of explicit positive or negative acknowledgments until after a number of frames have been received or dropped or a time-out occurs. We tune the related values experimentally to 48 frames for delaying positive acknowledgments, 256 frames from delaying negative acknowledgments, and a timeout period of 5ms.

Finally, to ensure data is delivered even in corner cases, such as link failures and lost acknowledgments, the sender will retransmit the last transmitted Ethernet frame if it has not received a positive acknowledgment for that frame within a coarse-grain timeout period.

2.2 Data Flow

For a more-in depth understanding of MultiEdge, we describe the transfer path for a remote memory write operation. We describe in detail both send- and receive-path packet processing.

In general, a single remote memory operation can generate several Ethernet frames if the operation data does not fit in a single Ethernet frame. Maximum Transfer Unit (MTU) is defined in network layer of MultiEdge,

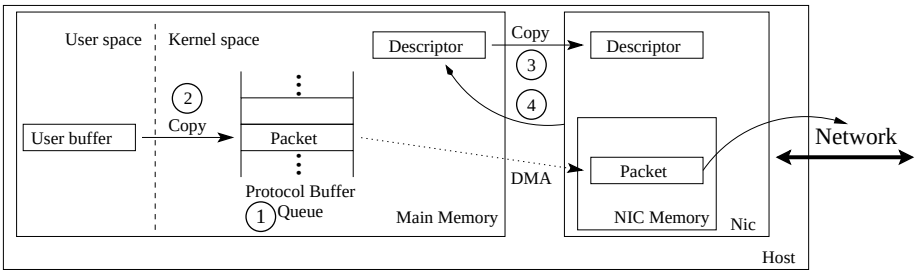


FIGURE 2.2: Send-path packet processing.

and network interfaces use it. Each Ethernet frame carries at the end a 32-bit cyclic redundancy check (CRC). The CRC covers all parts of the frame including the Ethernet header and is used at reception to verify that no bit errors have occurred in the transfer. The entire Ethernet frame is discarded if any bit errors are found.

We note that MultiEdge allocates all send- and receive-path buffers during initialization and re-uses buffers as soon as packets are delivered to the network (send path) or the application (receive path).

2.2.1 Transmit Path

Figure 2.2 shows the data path for each outgoing packet: (1) A number of new buffers are allocated in the kernel for an application send request (message). Each buffer is used to store one outgoing packet. Each buffer may consist of multiple pages, depending on the maximum allowable packet (frame) size. The header of each packet is filled in the corresponding buffer. (2) Then, the payload of each packet is copied from the user-space buffer. (3) A number of hardware descriptors is allocated, initialized, and copied to the network device memory. Each descriptor is used by the network interface to locate one packet buffer in the host memory and transfer it to the network interface (NIC) using DMA. (4) When the DMA from the host memory is complete, the NIC produces an interrupt to the host, to free

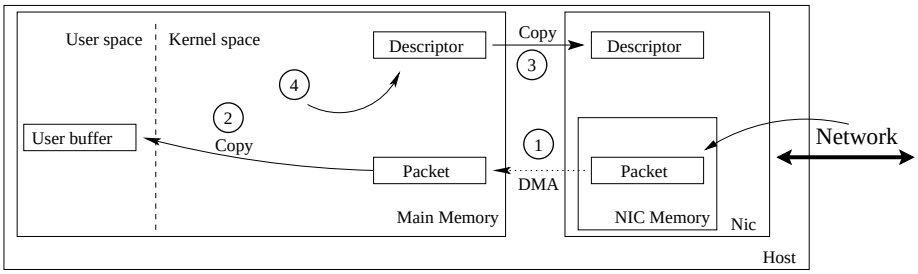


FIGURE 2.3: Receive-path packet processing.

the corresponding descriptors for future use. Send-path packet buffers are re-used only after the remote node acknowledges the corresponding packets.

2.2.2 Receive Path

Figure 2.3 shows the data transfer path for each incoming packet: (1) When an incoming packet is received, it is initially stored in the NIC memory. The next available descriptor on the NIC tells to the hardware where the packet should be stored in the host memory. The NIC uses DMA to transfer the packet (both header and payload) to this location. When the DMA is complete, if the NIC interrupts are enabled, the NIC interrupts the host to signal the new packet arrival. The communication layer interrupt handler is invoked by the host kernel to handle the new packet arrival. (2) Depending on ordering constraints, if the incoming packet can be processed, the protocol copies the payload from the kernel buffer to application buffer in user-space. (3) After the packet is delivered to the application, the kernel buffer is returned to the receive buffer pool and re-used for subsequent incoming packets. This requires re-initializing an existing, unused descriptor and writing the descriptor to the NIC. (4) When the protocol has no more work pending, it polls all devices for incoming packets. If there is no work to be done, the receive path enables NIC interrupts and goes to sleep.

2.3 Avoiding Copies

Except from the data transfer from host memory to the network interface, existing network protocols copy the payload data from the user-space buffers to the kernel-space buffers. Thus, for each data transfer over the network, data should cross over the memory bus at least three times on existing communication protocols. In a high-speed network, these memory transfers may become the bottleneck of system's performance.

To avoid memory copies, page remapping [8, 9, 11, 13] has been proposed earlier to eliminate the copy operation. Instead of copying the data between user and kernel buffers, the idea is to use virtual to physical address translation, page pinning, and page remapping to allow the application and the kernel to directly access the payload during packet sending or delivery.

However, the proposed mechanisms require hardware support to avoid copying the data which cannot be applied to a generic networking protocol. Moreover, some of these mechanisms can cause a copy-on-write as a penalty of removing the data copy overhead. Thus they don't avoid the copy overhead in the generic case, but it just postpone it when the first write occur.

We now examine how these techniques can be used to avoid memory copies when transmitting or receiving data (operation (2) in Figures 2.2 and 2.3). Each of the send and receive paths requires different techniques and thus, we discuss them separately.

2.3.1 Send path

The first challenge in the send path is accessing and transferring data directly from the user buffer. Typically, when a user performs a remote memory operation, data is copied from user to kernel space and into a pinned buffer that has been registered with the network interface. To eliminate this copy, we need to dynamically provide the NIC with the physical address of the

user buffer and to pin the buffer for the duration of the DMA. Furthermore, each buffer may consist of multiple virtual or physical pages and physical pages may not be contiguous in memory.

To pin a page into memory we walk the process' page table, locate the page structure, and increment the reference count of the page. The same process can be used to perform virtual to physical translation. Both of these operations can either be performed during buffer registration or during the communication operation itself. The first has lower overhead, especially if we assume that communication in programs exhibits locality and registration operations are less frequent than remote read/write operations. However, the latter approach does not require keeping large portions of memory pinned.

In this work we use the first approach and pin the pages once during registration. To avoid accessing the OS page tables at each read/write operation for obtaining the physical address of the buffer pages, we maintain a protocol page table that contains the virtual to physical translations of all registered (and pinned) buffer pages.

A second design decision is related to synchronous communication operations. Synchronous remote write operations return to the initiator as soon as the data have been read from the user buffer and the application can modify the buffer without affecting the previous write operation. Thus, synchronous operations can return either after the DMA on the send path is complete or before the DMA completes, if there is a mechanism to detect subsequent application accesses to the communication buffer. Copy-on-write has been used in the past to detect such accesses [13]. In our design we simply choose to return from the the initiating call after the DMA has completed. However, asynchronous write operations return as soon as the DMA has been posted, without waiting for the transfer to complete.

The second challenge in the send path is related to add the packet header

to the payload provided by the user application. Placing the buffer and the payload in a single buffer would require copying the user data. Instead we make use of the gather-DMA operation that is available on many NICs today. With gather-DMA, a single packet uses multiple descriptors, one for each buffer that contains part of the packet data.

2.3.2 Receive path

The protocol receive path is more involved. Data copying is required because the buffers used for delivering data from the network are typically allocated in the kernel, independent of the application buffers use for communication purposes.

Existing programmable network interfaces [7, 32] allow applications to directly post receive buffers from user level to the network interface. However, this ability requires extensive NIC support. It requires the ability to parse packet headers, detect and interpret protocol specific fields about memory locations where the payload should be delivered, and possibly perform virtual to physical address translation.

In our work we assume that the NIC does not have the ability to examine and interpret protocol specific headers and fields. Thus, when packets arrive at the NIC they are placed in buffers that have been already registered with the NIC by the protocol receive path. As it is not possible to predict or regulate the order of packet arrival, especially when multiple senders are involved, receive buffers where packets are delivered have no relationship to the corresponding application communication buffers, where packet data should be delivered.

To address this problem, we can use page remapping at the kernel level, and replace the user-level communication buffers with the kernel buffers where packet data has need delivered by the network interface (using DMA). This will result in delivering data to the application without copying them over the kernel-user boundary. However, page remapping for this purpose,

requires addressing four main problems:

- Implementing the page remapping itself.
- Replenishing receive buffers that have been handed over the application.
- Dealing with packet headers that proceed application data.
- Splitting large messages to multiple packets and inducing minimal data copying at the receive path.

Page remapping

Let's assume that both the kernel and user buffers are aligned to a page boundary and have the same (page) size. Mapping the physical page of the kernel buffer to the application buffer requires a walk through the receiving process' page table, identifying the entries that describe the application buffer, and replacing the physical page with the physical page of the kernel buffer. However, implementing this requires modifying the OS memory manager, which is an intrusive approach and imposes portability restrictions.

Instead, to achieve this effect without having to modify the operating system memory manager code we use the following technique. When a memory region is allocated by an application, the corresponding entries in the page tables are created. However, the physical pages for this entries are allocated when the virtual pages are actually accessed. For this reason the Linux OS defines a `no_page()` function pointer per mapped memory region and calls this function to allocate the empty page table entries. In our implementation we replace the generic `no_page()` function with a protocol-specific handler that instead of allocating a new page, it returns the physical page number we want to re-map the user buffer to. Then we invalidate the user virtual pages that correspond to the application buffer (by clearing the corresponding page table entry of the old page) and we generate a fake write

fault on the application page. This fault results in the memory manager invoking our `no_page()` handler, which in turn returns the kernel buffer page number that contains the packet data.

Then we flush the TLB entry for the user page. Finally, we examine if the user page was pinned into memory by the send path, in which case we also pin the new page and update our protocol page table used by the send path. Although the actual implementation of page remapping is somewhat more involved, we omit here some of the details. Later in this section we discuss buffer alignment issues.

Replenishing kernel buffers

After a kernel buffer has been remapped to the application, we need to replace it with a new buffer. This "replenishing" of kernel receive buffers can happen in two ways: (a) either by allocating a new kernel buffer or (b) by using the physical page that was freed from the user buffer. To avoid the cost of buffer allocation, we use (b) and we place the physical page of the user buffer as a receive buffer for a NIC descriptor.

The last issue with replenishing kernel buffers is packets that require multiple pages. For performance purposes the packet MTU in Ethernet-based networks can exceed a single page. For instance an MTU of 9000 bytes may require a receive buffer of size up to three pages for delivering a single packet. Thus, kernel buffers in the receive path need to be allocated in a manner that allows for individual, non-contiguous pages to be remapped and replenished. Thus, each buffer consists of a descriptor with pointers to multiple pages (depending on the MTU). This in turn requires the NIC to be able to handle descriptors with multiple pointers to physical pages. This feature however, is already available to many NICs as it is used also by certain TCP/IP implementations and offloading mechanisms.

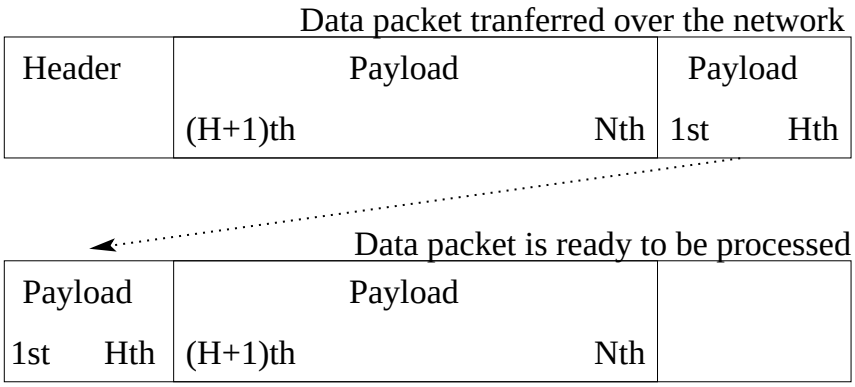


FIGURE 2.4: Header and data placement. H and N are the header and data bytes respectively.

Dealing with packet headers

Assuming that receive buffers are page aligned, we need the kernel receive buffer that will be remapped to also contain the packet data. However, incoming packets proceed data with the packet header. One approach to dealing with this is to deliver the packet header and the packet data in different. However, this requires performing variable size transfers from a single received packets to two kernel buffers, which is not possible with commodity network interfaces. Instead, commodity NICs, assume a pre-defined buffer size (e.g. page) and transfer the packet (both header and data) to as many such buffers are required by the packet size.

To address this issue we place message data in packets during packet sending as shown in Figure 2.4. First, we assume that the NIC uses buffers with size equal to the OS page size, e.g. 4 KBytes. This is what happens today in most NICs. If we assume a header of size H bytes and a payload of size N bytes, after we place the H header bytes in the packet, we skip the first H bytes of the payload and place in the packet the rest N-H bytes. Then, we place the H first bytes of the payload to the end of the packet.

When the packet is received, the first H bytes of the payload are copied from the end of the packet to the beginning of the receiver buffer overwriting the H header bytes and resulting in a page-aligned payload. Thus, the last H bytes of the packet that were placed on a separate buffer (as described next) are not needed any more and their buffer can be return to the buffer pool.

Splitting large messages to multiple packets

First, we assume that the source and destination buffers have the same alignment. Although this may not be necessary in APIs where the sender is aware of the destination virtual address, it is not an important restriction and it merely may increase the size of registered memory on the sender or the receiver.

Second, if the source address of the data (and thus the destination address as well) is not page aligned, then we create a first packet consisting of the data until the next page boundary. This packet will be delivered to the application using copying. This is required by the fact that the receiving NIC starts delivering packet at the beginning of the receive kernel buffer. When the source starting address is aligned to a page boundary or for subsequent packets of a large message we create MTU-size packets with the data to be sent as follows.

Large messages need to be broken down to multiple packets of size such that we don't exceed the MTU size and each packet contains data that will be delivered to a page boundary. Thus for an MTU size of 9000 bytes and a page size of 4 KBytes, each packet will consist of a header of H bytes and 8192 bytes of data (two pages). Each packet will be delivered in three pages due to the initial header. As described previously, the last H bytes of the packet that are located in the third page, will be copied over the first H bytes of the header in the first page, resulting in 8192 bytes of data in two pages that can be mapped directly to the receiver application. Finally, the last packet of a large message may be less than a page and thus will require

copying, as it is not possible use the re-shuffling technique described above.

This approach does not use the maximum MTU that may be available in the interconnect, however, given current trends in network overheads results in significant improvements in throughput as described next.

2.4 Interrupt handling

A major concern at high network speeds is how to reduce software overheads so that the data rates can be used efficiently. In particular, interrupts induce a significant overhead in modern operating systems. The design of MultiEdge tries to minimize interrupts both in the send as well as the receive path, as follows.

In commodity network interfaces, such as Ethernet, there are three event classes that may require interrupts: transmission of a frame for freeing the associated send buffer; reception of a frame for delivering to host and user memory; and, less frequently, management events, e.g. when a network cable has been removed or inserted. All events are commonly signaled using one single interrupt line.

MultiEdge handles interrupts in the following way: When an interrupt arrives, the interrupt handler disables subsequent interrupts and notifies the protocol layer. When the receive or send protocol path is invoked by an interrupt handler, it processes all pending events, e.g. send frame completions or newly received frames, by polling each network interface. The protocol layer enables interrupts when there are no more interrupt related events and no protocol kernel thread is active. The protocol can take advantage of multiple CPUs by using multiple contexts for processing send or receive path events. Receive path processing for a single user process can only happen in a single (protocol) kernel thread.

2.4.1 Polling

We design a version of MultiEdge that uses polling instead of interrupts to quantify the impact of interrupt handling. To implement the polling mechanism, when a device is registered with MultiEdge, we disable it's interrupts. As mentioned above, MultiEdge' receive path sleeps when there is no work to do. When interrupts have replaced by polling, receive path never sleeps. When it has no frames to process, it polls continuously the devices for pending and not-serviced work.

To check a device if it has pending work, we need a number of Programmed Input/Output (PIO) reads and writes. To avoid flooding the busses from these PIOs, we limit the time between two consecutive polls on the same device. In our current implementation we poll at 1ms intervals.

2.5 Multiple links

MultiEdge can make use of multiple network interfaces within a single communication channel. MultiEdge support up-to 64 Ethernet links, which is limited by the atomic bit operations we use. To able to take advantage of multiple links, we need to schedule efficiently the data frames to the available links. Moreover, the use of multiple paths to reach a destination, occur out-of order delivery of the transmitted frames, that the communication infrastructure should handle properly.

2.5.1 Out of Order frames

To better take advantage of multiple links, it is possible to relax operation ordering constraints. Remote memory operations that need not be ordered with respect to other operations, may be processed at the receive path as soon as they arrive without need for buffering. However, we believe that, to be meaningful, out-of-order delivery can not be done indiscriminately. To support both in-order and out-of-order delivery we extend the communication API to support the following operation flags:

- Backward fence: This remote memory operation will be performed on the destination node only after all previous operations issued by this source node to the same destination have been performed.

- Forward fence: Any subsequent operation issued by this source node to the same destination will be performed only after this operation has been performed.

These flags are orthogonal and so a single operation can specify both flags. The default behavior is to allow all operations to be re-ordered.To specify ordering constraints, the programmer can use the `flags` bit-field of the `RDMA_operation` API call. Individual frames originating from the same, or even different, operations can be re-ordered with respect to each other if only the semantics of the operation flags are upheld.

2.5.2 Packet Schedulers

Whenever a frame needs to be transmitted, MultiEdge uses one of the available network interfaces. For this interface selection we implement three different packet schedulers:

1. Static round robin (SRR): schedules one packet per link in a round-robin fashion, without any further knowledge.

2. Weighted static round robin (WSRR), where each link has a static weight (W) which indicates its peak, relative with the other links, speed. The WSRR schedules W frames per link before switching to the next link in a round-robin fashion.

3. Weighted dynamic (WD), where each link has a static weight (W), similar to WSRR. The WD scheduler dynamically estimates the number of bytes (B) already scheduled for transmission in each link and schedules the next packet over the link with the lowest B/W ratio.

2.6 Implementation

In the current implementation of MultiEdge, the protocol layer is implemented as a Linux kernel character device driver and requires no changes to the Linux kernel itself. The Ethernet hardware device drivers are managed by the protocol layer. Both the protocol layer and the Ethernet device drivers can be compiled as dynamically loaded kernel modules. Multiple Ethernet hardware device drivers can be simultaneously loaded and they can dynamically configured at run-time. This makes it possible to use multiple and different types of Ethernet controllers. The current version has support for Broadcom Tigon 3, Intel 1000 and Myricom 10 Gigabit Ethernet hardware.

2.6.1 Synchronization

MultiEdge uses multiple concurrent contexts for achieving high throughput. The send path runs in the context of the calling process, the interrupt context for detecting asynchronous events when not polling, and the receive path that executes in a private protocol thread. Synchronization among these contexts can be incur a significant overhead at high data rates. MultiEdge uses a combination of locks and atomic operations to minimize overheads, as follows:

- Operating system spin lock: In one occasion, when we post a send DMA descriptor.
- Atomic compare-and-swap: In four occasions, two when a send buffer is allocated and freed and two when sequence numbers are read and updated in the send and receive paths.
- Atomic increment: In one occasion, at frame reception, to record frame arrival for later processing.

- Atomic fetch-and-add: In two occasions, when we update variables but need to read their previous value: When reclaiming send DMA descriptors after they are used and when the round-robin load-balancing policy decides which link to use next.

Finally, we should note that further reducing synchronization requires either reducing concurrency in the protocol or additional network interface support, such as virtualizing the DMA post queue.

Chapter 3

Experimental Results

3.1 Methodology

We evaluate our system using three micro-benchmarks:

- one-way: One of the two nodes performs remote memory writes back to back without waiting for any response from the receiving node. The receiving node signals the end of the benchmark after it has received all the data. This benchmark exercises the send path at the sending and the receive path on the receiving node.

- two-way: Both nodes simultaneously perform back to back remote memory writes. The throughput in this case accounts for all data sent and received in each node, as both the send and receive paths are exercised at the same time.

- ping-pong: This is a request-reply benchmark using remote memory writes. Request and replies carry the same amount of data. Each node waits for receiving the full data before replying.

To understand MultiEdge' performance we use the following metrics:

- Throughput: the amount of useful payload data that has been delivered to the remote node.

- Latency: one way, end-to-end, application time for delivering a message.

- CPU utilization breakdown: the CPU time used for network processing. To represent the two CPUs available in our nodes, we use a maximum CPU utilization of 200%. Our CPU utilization results are approximate as we cannot account for the time between a NIC issues an interrupt, until the interrupt handler is executed on the host CPU.

For the CPU utilization breakdowns, we use the following labels:

- Poll/IRQ: Interrupt handling or polling for servicing the NICs.

- TxCopy/Translate: Overhead spent on preparing the payload in the send path. This component includes either the pinning and translation overheads or the data copy for the send path.

- RxCopy: The overhead of packet processing on the receive path, including copying, where appropriate. This component does not include the actual overhead for remapping, which is measured separately.

- Remapping: Overhead for page remapping on the receive path.

- Packet: Other protocol overhead for packet preparation processing. This includes header preparation and processing, ordering of packets, and flow control.

- Device: I/O overhead for communicating with the NIC both at the send and receive paths.

Our experimental setups are consisted of two systems connected with multiple network interfaces (NICs). Both nodes have two Opteron 244 CPUs running at 1.8 GHz and a Tyan S2892 motherboard. The operating system is the 64-bit version of Debian with Linux kernel version 2.6.18.8 compiled

with GCC version 4.1.2. The network interfaces on both nodes are connected pairwise with cross-cables without a switch.

In our results we conduct experiments with various MTU sizes. We believe that large MTU sizes are more representative of future trends and thus we present results for an MTU size of 9000 Bytes (Ethernet jumbo frames).

In the rest of this chapter, we present our experimental results. We present two evaluation setups that each of them focus on a different concept. Sections 3.2 presents our experimental platform and our results for which are focused on multiple links, without zero-copy transfers. Sections 3.3 present our experimental platform and our results for which present results using our protocol with zero-copy transfers.

3.2 Base Results

3.2.1 Base Experimental Platform

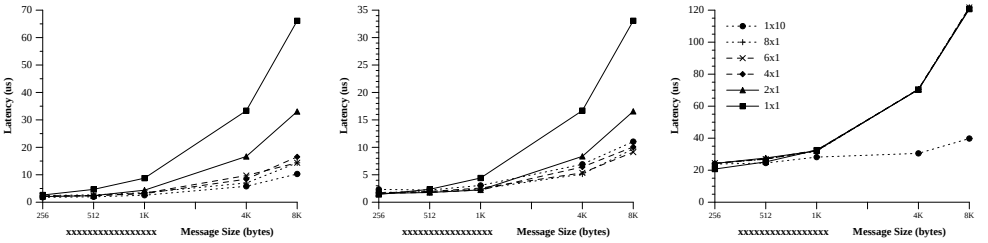
On the Base experimental platform both nodes have 2 GBytes of main memory. Each node is equipped with the following network interfaces:

- One Myricom 10G-PCIE-8A-C card;
- One Intel PRO/1000 PT Quad Port PCI-E card;
- One Intel PRO/1000 GT Quad Port PCI-X card.

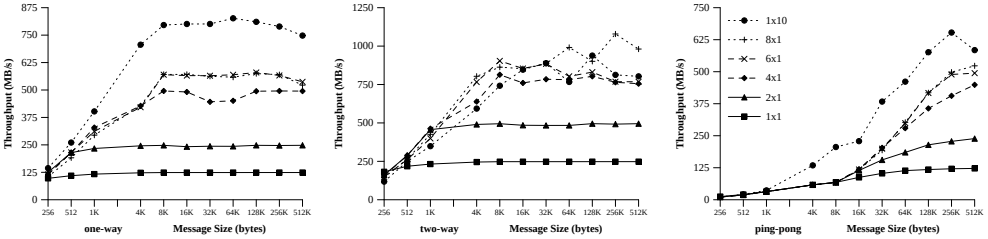
In our experiments we show the behavior of our protocol using 1 Gbit/s and 10 Gbit/s network interfaces. Moreover, we vary the number of 1 GBit/s links from one to eight and we contrast our multiple-link results with the single Myrinet 10 GBit/s link. Finally, we show the behavior of name using different packet schedulers when we use heterogeneous network interfaces.

We organize our experiments around the following system configurations:

- *Base:* Our base protocol.



(a) Latency



(b) Throughput

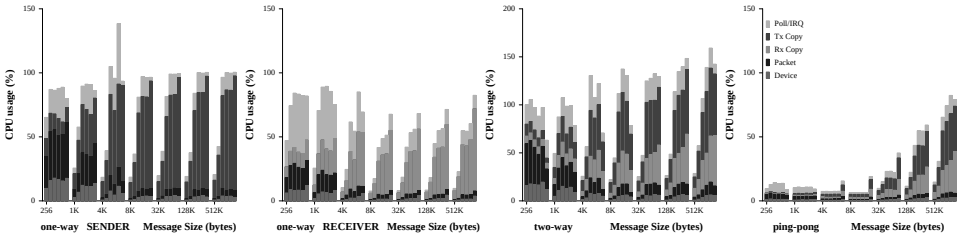
FIGURE 3.1: Latency and Throughput in the base system.

- *Polling:* Our base protocol with interrupts disabled and replaced by a polling mechanism.
- *noCopy:* Our base protocol with copies to and from user-space artificially disabled. In this configuration the exact amount of data is transferred over the network.

On throughput and latency figures we present one curve for each link configuration and we use nxk to denote n links of k Gbits/s each. On CPU utilization figures we show one bar per configuration, with the leftmost bar referring to 1x1 setup and the two rightmost bars to 8x1 and 1x10 respectively.

3.2.2 Base protocol results

Figures 3.1(a,b) show latency and throughput for the base configuration. Latency in ping-pong reflects one-way memory to memory time for each operation. We see that minimum latency in all configurations is about 23



(a) CPU Utilization

FIGURE 3.2: CPU utilization in the base system. We show one bar per configuration, with the leftmost bar referring to 1x1 and the two rightmost bars to 8x1 and 1x10 respectively.

μs. For ping-pong, we see that latency does not depend on the number of links, as long as the MTU is 9 KBytes and physically only one link is used every time. Latency in one-way and two-way reflect the host overhead to initiate an operation and does not represent the end to end latency. In both cases we see that minimum overhead is about 2 μs and is not affected by bi-directional traffic.

In terms of throughput, MultiEdge can fully utilize link throughput in the 1x1 and 2x1 configurations delivering a maximum throughput of about 99.2% of the maximum nominal throughput. In 1x10, the maximum throughput is about 825 MBytes/s of the peak 1250 MBytes/s, or about 66%.

We see that in one-way and ping-pong benchmarks, data throughput is approximately proportional to the number of links. The throughput continues to increase for up-to six links but remains almost at the same level (about 550 MBytes/s) for eight links. In the two-way benchmark, we see that throughput scales well up-to four links. With higher number of links, throughput fluctuates around 800-900 MBytes/s and exhibits a large variance.

Second, throughput is limited be the available CPU resources. Figure 3.2

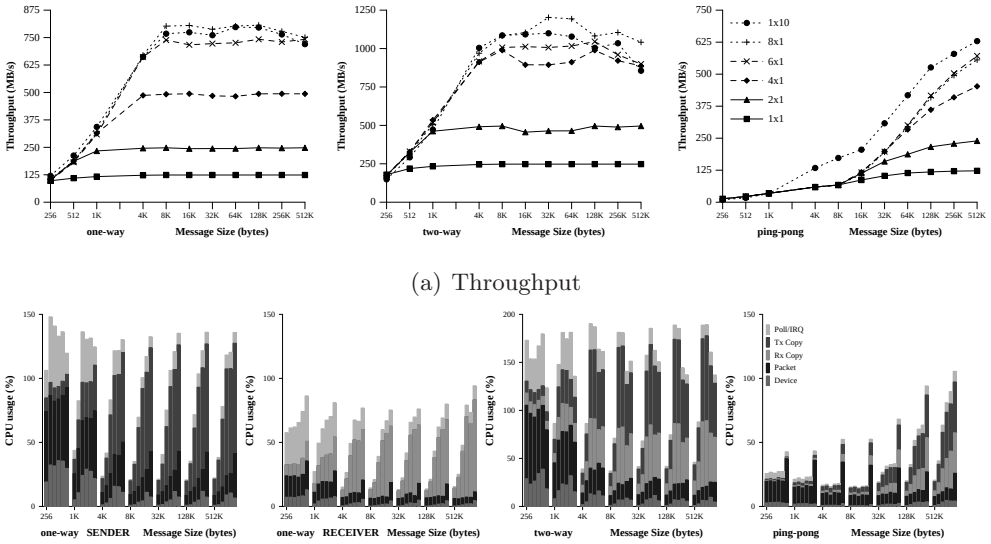
shows our CPU utilization breakdowns. We present one CPU utilization breakdown bar per link configuration for each message size. The leftmost bar refers to the 1x1 configuration and the two rightmost to the 8x1 and 1x10 configurations respectively. In one-way we show breakdowns for both the sender and receiver. In two-way and ping-pong we only present breakdowns for one of the two communicating nodes; in these benchmarks both nodes behave in a similar manner because the send and receive paths of the protocol are exercised.

One-way throughput is limited by the send path overhead. Our protocol is able to use a single CPU for send path execution, only partially offloading transfer completions to the second CPU. Two-way is limited by load imbalances in the protocol, which is not able to fully utilize both CPUs.

For all benchmarks we see that data copies are the dominant portion of the CPU utilization. Moreover, copy overhead increases proportional with message size. For example, with 128-Kbyte messages and 8x1 links, copy overhead accounts for 75% and 65% on the send and receive paths respectively in one-way, and about 70% on both send and receive paths in two-way and ping-pong.

Third, interrupt cost increases with the number of links but not the speed of each link. Note that for 4-KByte messages and 1x8 links, the interrupt overhead is higher than in other configurations as the hardware and protocol interrupt coalescing mechanisms are not effective, resulting in a large number of interrupts being delivered. Overall, the cost for interrupts, packet processing, and device management decrease when we increase the message size and remain at the same level for messages larger than 8-KBytes.

Finally, comparing 1x10 and 8x1 links, we see that 1x10 exhibits higher throughput in cases where the CPU utilization is lower, namely one-way and ping-pong. However, in two-way different message sizes result in different behavior.



(a) Throughput

(b) CPU Utilization

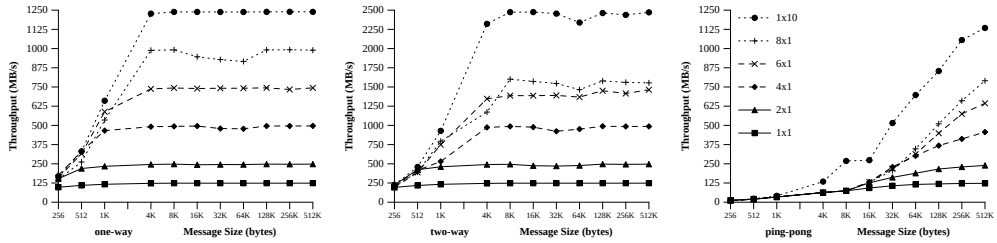
FIGURE 3.3: Throughput and CPU utilization when interrupts are replaced by polling.

3.2.3 Impact of interrupts

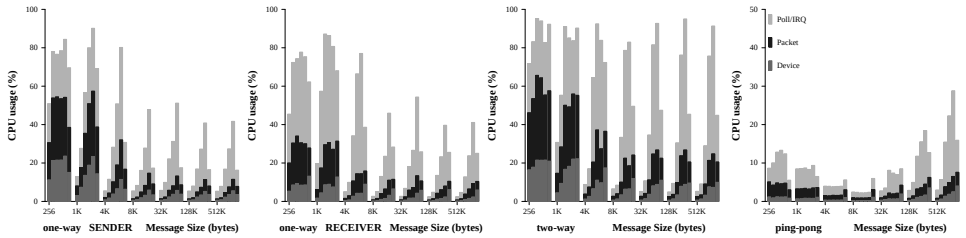
Figure 3.3 shows the data throughput and CPU utilization when interrupts are replaced by a polling mechanism.

Compared to Base, we notice that throughput increases up-to 38% and 25% in one-way and two-way respectively, but remains at the same level in ping-pong. This is due to two reasons: a) we avoid high interrupt pressure when there is large numbers (hundreds) of outstanding data transfers and b) the receive path is able to run on a different CPU from the send path. Between 1x10 and 8x1 links, we see that they have almost the same performance in one-way and two-way benchmarks. We believe that this is due to memory throughput limitations.

Moreover, similar to Base, one-way throughput is limited by send path processing. In two-way, 8x1 uses up both CPUs, whereas 1x10 is limited by



(a) Throughput



(b) CPU Utilization

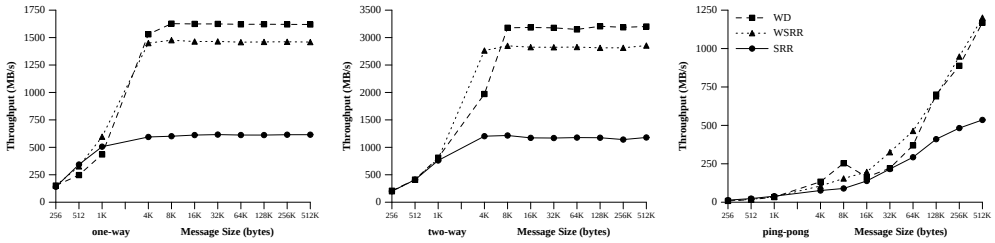
FIGURE 3.4: Throughput and CPU utilization when copies are artificially disabled.

send path processing, which is not offloaded to the free cycles of the second CPU.

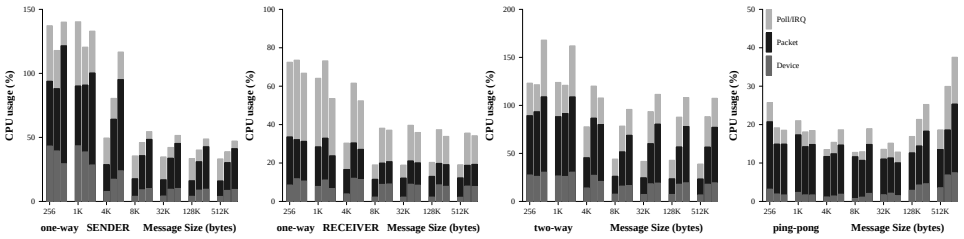
3.2.4 Impact of copies

Figure 3.4 shows the data throughput and CPU utilization when copies are artificially disabled. We see that the one-way and ping-pong benchmarks exhibit throughput approximately proportional to the number of links, reaching the maximum possible throughput for each configuration. In comparison to Base with 1x8 links, throughput increases by 66,5% and 87,5% for one-way and two-way benchmarks respectively.

In all cases, a large portion of CPU time is spent servicing interrupts, and it increases proportionally with the number of links. This is also the reason that 1x8 configuration doesn't scale in two-way test.



(a) Throughput



(b) CPU Utilization

FIGURE 3.5: Throughput and CPU utilization when using different schedulers. In CPU utilization we show one bar for each scheduler: SRR (left), WSRR (middle), WD (right).

3.2.5 Impact of scheduling

To examine the behavior of different schedulers, we present results for one configuration with five links, 1x10 and 4x1. Moreover, to avoid the limitations discussed above we disable both copies and interrupts. Figure 3.5 shows the data throughput and CPU utilization for the send path using different link schedulers.

We note that SRR treats all links in a similar manner and does not take advantage of the full link throughput available. WSRR and WD exhibit almost the same behavior in ping-pong, as the synchronous nature of the benchmark allows for only a small number of outstanding packets.

In one-way and two-way benchmarks, we see that for large message sizes, WD outperforms WSRR by 11% and 12.5% respectively. Finally,

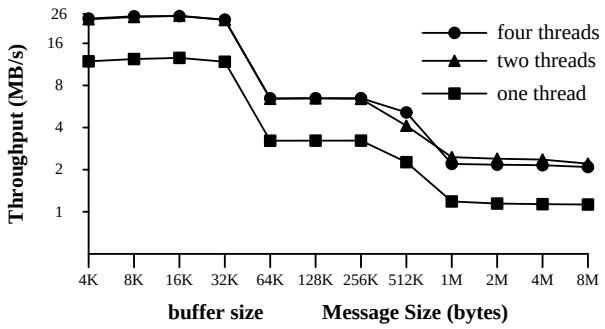


FIGURE 3.6: Memory to memory copy throughput for a single node.

Figure 3.5(b) shows that for large messages WD spends more CPU time for packet management, which is affected by the time to prepare and schedule each data frame. The difference is approximately 10% and 20% in one-way and two-way respectively, similar to the throughput benefit of WD over WSRR.

3.3 Zero-Copy Results

3.3.1 Zero-Copy Experimental Platform

For this experimental platform, both nodes are equipped with two Myricom 10G-PCIE-8A-C card. Each card is capable of about 10 GBits/s throughput in each direction for a full-duplex throughput of about 40 GBits/s. Due to a PCI-Express limitations in our motherboards, one of the two cards runs at half speed. Thus, we are limited to a maximum throughput of about 15 GBits/s per direction, for a full-duplex maximum throughput of about 30 GBits/s.

3.3.2 Basic operation overheads

The Opteron 244 CPUs in our system have a TLB size of 1024 entries and L1 and L2 cache sizes of 128 KBytes and 1 MByte respectively. Each processor is equipped with 2 DIMMs of 1 GByte DDR-400 for a total of 4 GBytes

	local data (MB/s)	remote data (MB/s)	
	1 CPU	1 CPU	2 CPUs
read	2500	1845	3086
write	1875	1461	2546

TABLE 3.1: Memory throughput for reads/writes when threads access data located on local and remote (to their CPU) memory modules.

main memory. Also, Linux is configured with NUMA (Not-Uniform Memory Access) features enabled.

Figure 3.6 shows the memory throughput in the nodes we use. For data copies up to 64 and 512 KBytes, copy throughput is higher due to L1 and L2 hits. Sustained memory copy throughput for a single CPU is about 1 GByte/s. and for two CPUs is about 2 GBytes/s which is the maximum memory throughput.

Table 3.1 shows throughput for memory reads and writes separately. We see that for one CPU, accesses to memory attached to the CPU have a sustained rate of about 2.5 and 1.9 GBytes/s for reads and writes respectively, but it drops significantly when accessing remote memory modules.

The two NICs over PCI-Express are capable of full-duplex throughput of about 3.75 GBytes/s (30 Gbit/s). NICs are able to deliver data to host memory with this rate, as long as using all memory modules, we achieve 5 GBytes/s and 3.75 GBytes/s throughput for reads and writes respectively. Copying data on transmit and receive paths, would increase by 2 the number of accesses on the memory, which require more memory throughput than the available on our systems.

Table 3.2 shows the overhead of certain basic operations we use in our design. An empty `ioctl` costs about 0.2 μs. Allocating a kernel buffer (MTU size) costs about 1 μs. This high cost is due to the kernel memory allocator

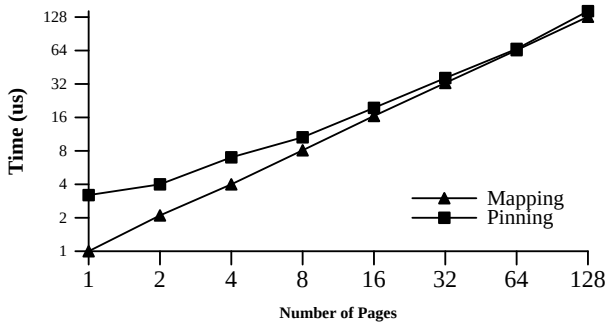
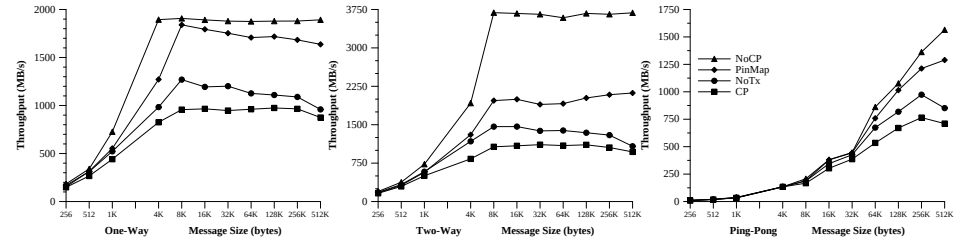


FIGURE 3.7: Pinning and Mapping overheads for a number of pages

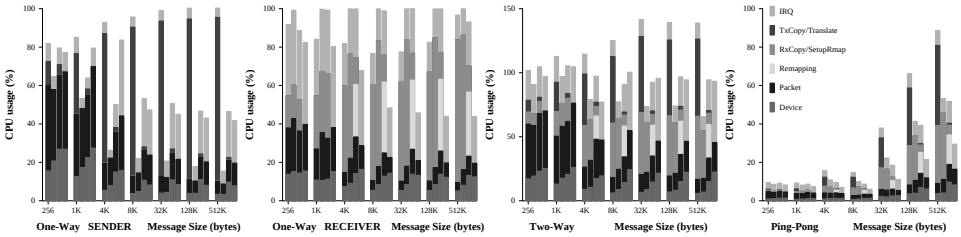
operation	time (μ s)
ioctl	1.15 - 0.25
alloc buffer	0.85 - 1.05
pin page	3.2 - 3.8
remap page	0.9 - 1.1

TABLE 3.2: Base cost for page remapping, pinning, and buffer allocation in the kernel.

that tries to minimize memory fragmentation. Pinning a single page costs about 3.6 μs and remapping a page costs about 1 μs. Pinning a page 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. In Figure 3.7 we see that both pinning and remapping costs increase almost linearly with the number of pages. Pinning the first page is more expensive that the rest, because the common case is that consecutive virtual pages are places in are also consecutive in the page table. For page remapping the average overhead is lower than pinning, because virtual memory area is stored when receive buffer is registered. Thus, we walk directly the page table to find the page table entry and



(a) Throughput



(b) CPU Utilization

FIGURE 3.8: Throughput and CPU utilization for different protocol configurations. For CPU utilization we show one bar per configuration: CP, NoTx, PinMap, NoCP (left to right).

update it.

3.3.3 Benefits of page remapping

Figure 3.8(a-b) shows data throughput and CPU utilization breakdown respectively for the different setups we use. The base protocol, 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. Moreover, copy overhead is a dominant cost in all cases, except for the smaller message sizes, reaching up to 10% of CPU utilization. Two-way utilization for base setup is close to 100% for message sizes up-to 4 KBytes and increases to 130-140% for larger.

Next, we see that artificially removing the data copy in transmit path (NoTx bar in Figure 3.8) increases throughput in all benchmarks. Especially for one-way and ping-pong the increase is approximately 27%. In one-way,

while the transmit path is not a bottleneck (unlike the base setup), the receive path saturates one CPU.

Replacing copies with our remapping mechanism results in a large performance improvement, compared to the base (copy-based) configuration: Throughput increases by 87%, 98%, and 70% for one-way, two-way, and ping-pong respectively. In one-way we see that receiver path utilization is almost 100% and throughput reaches up to 14.7 GBits/s , that is the 98% of the maximum throughput when copies are artificially removed (NoCP). Thus, any further improvement in throughput can mainly come from better distributing receive path protocol processing to multiple cores in future CPUs. In two-way the throughput reaches up to 57% of maximum throughput of NoCP.

Overall, throughput is limited by three reasons: (a) the maximum memory throughput in our systems, (b) the high CPU requirements of the receive path that saturates a single CPU, and (c) the inability of the receive path to effectively utilize the second CPU. In our results, the main bottleneck for PinMap is not (a), as memory throughput is higher than the achievable maximum throughput.

For (b) we see that (e.g. in one-way, receiver node) interrupt handling and page remapping are the two main costs reaching up to 74.1% of CPU utilization in the receiver of one-way. Also, NIC access and packet processing are up to 36.3% of CPU utilization. Given that our protocol processing has been optimized and that the current bottleneck is receive path CPU utilization, we believe that any further improvements will come from improving (c), more efficient distribution of protocol processing on multiple cores. However, this is not straight-forward due to the fact that costs related to interrupt handling, NIC access, and packet distribution are not trivial to balance at fine-grain without NIC support or extensive synchronization.

Figure 3.9 shows the system's latency for one-way and ping-pong. Ping-

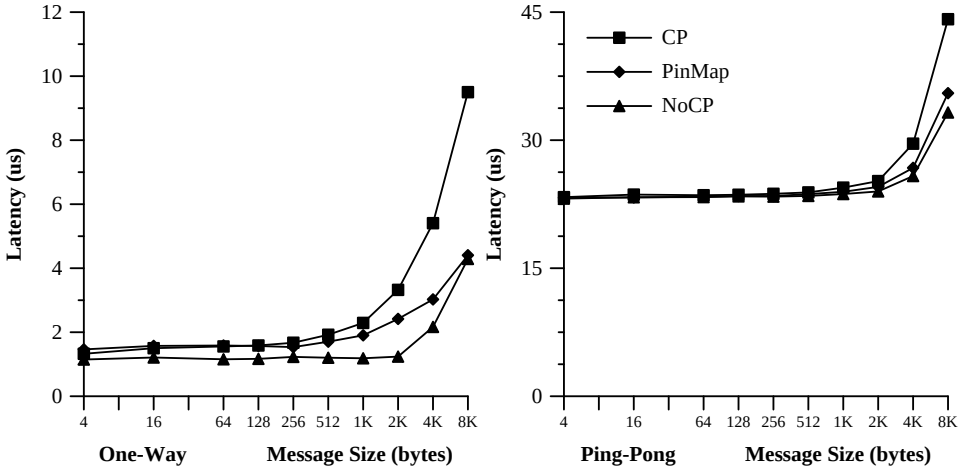


FIGURE 3.9: Message latency for different protocol configurations.

Pong exhibits a latency of 23 μs for 4-Byte messages and reaches 24-26 μs for 2-KByte messages. One-way shows the overhead of posting a write request. We see that for messages up-to 2 KBytes PinMap performs better due to the lack of copy on the send path. Smallest packet size in Gigabit-Ethernet is 60 Bytes. Packets with this size, are limited from the hardware to fit into a single hardware descriptor. Our header size is 48 bytes, thus if payload size is less or equal with 12, we copy it. We see that PinMap performance is always equal or better from CP, thus it isn't needed to set a larger threshold between copying and translating. This is because our pages are already pinned and we spent time only to take the pages from protocol's page table.

3.3.4 Impact of TLB flushing mechanism

After remapping the receive buffers, we need to invalidate existing stale TLB entries in the CPU. Another option would be to directly update the TLB entries with the new mapping, however, this is not possible on many modern CPUs. TLB entries may be invalidated either selectively or by flushing the full TLB. Figure 3.10 shows these two cases for one-way. We also include a curve where we artificially do not flush any TLB entries, to show the best

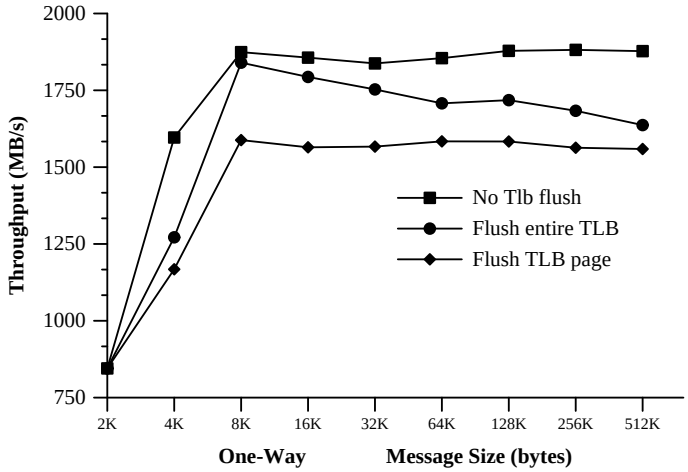


FIGURE 3.10: Throughput for different TLB invalidation mechanisms.

possible case.

We see that the overhead when flushing the entire TLB depends on the message size. Full and selective TLB invalidations are 1-13% and 17% worse than the ideal throughput with no invalidations, respectively. Moreover, as expected the throughput when the TLB is not flushed is close to the NoCP throughput of Figure 3.8. 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.

3.3.5 Impact of data alignment

Until now we have presented results using appropriate data alignment on the send and receive buffers, such that page remapping is possible on the receive path for messages equal or larger than 4 KBytes. Also, message size is a power of two resulting in full page remappings for large messages. In all cases, data placement in the actual packets requires shuffling the first part of the packet, as explained in Section 2.3 (Figure 2.4). Also, we assume in all cases that the send and receive buffers are aligned at page boundaries.

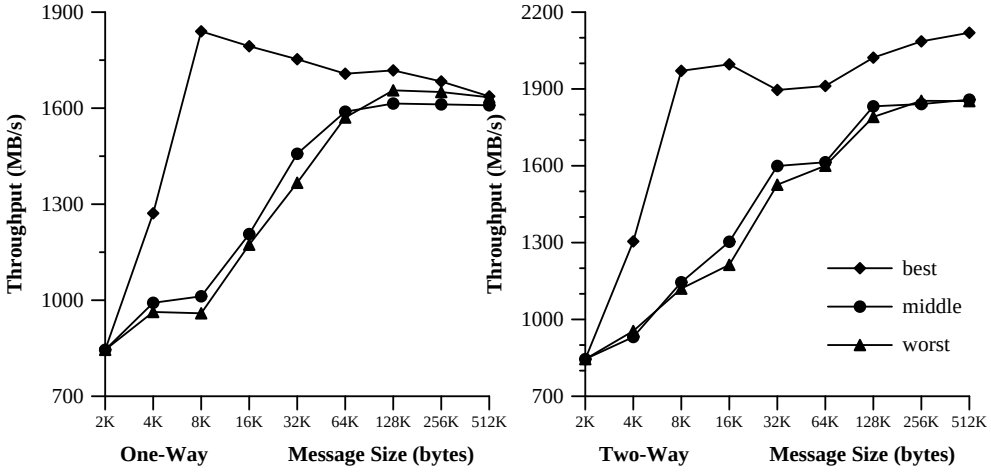


FIGURE 3.11: Throughput with varying data alignment.

In case the send and receive buffers are not page-aligned, then there is a need to copy part of the data and to use a larger number of packets, as discussed in Section 2.3. Figure 3.11 presents throughput for one-way and two-way when different buffer alignment is used. In these experiments we keep use a fixed address for the source buffer and change the alignment of the destination buffer. The worst case occurs when the buffer starts from the second byte of the page and the best when the buffer is page aligned, with all other cases varying in between. The curve labeled "middle" refers to the case where the destination buffer starts at the middle of a page.

For messages larger than a page we see that the middle and worst cases exhibit similar behavior. In both cases 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 their size is a multiple of 4 KBytes. Thus, in both cases the same number of packets and bytes need to be transmitted and copied. Finally, when buffers are not properly aligned, throughput increases with messages size as an increasing number of packets uses remapping on the

receive path, asymptotically reaching the maximum throughput of the fully aligned case.

Chapter 4

Related Work

With the advent of clusters and their extensive use as a computational platform there has been a lot of research on scalable communication subsystems in clusters. There has been extensive previous work in improving base communication performance by enabling user-level communication, eliminating copies of data, and reducing host overheads and context switches [3, 14, 17, 28, 33, 37]. Also, there has been work on network interface architectures and support for high-performance cluster communication [5, 6, 7, 19, 20, 21, 32]. Finally, previous work has evaluated low-latency, high-speed interconnects in various contexts [1, 23]. Our work differs from these efforts and builds on previous work in two important ways: (a) We advocate kernel-level, edge-based communication subsystems that provide high level semantics and transparency, important for commercial applications and (b) we introduce spatial parallelism and examine the impact on edge-based protocols.

Previous efforts that are related to our work in terms of the underlying platform. The authors in [39] 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 di-

rectly 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 [35] present a user-level, zero-copy protocol design and implementation on top of 1 GBit/s Ethernet. In our work we examine 1- and 10-GBits/s links, multiple links, and we present detailed network statistics on the impact of edge-based protocols on network traffic.

M-VIA is an implementation of the Virtual Interface Architecture (VIA) [16] over Gigabit Ethernet networks. It only supports single network interfaces. Moreover, previous work with M-VIA has only examined performance issues on 1 GBit/s Ethernet networks. The authors in [2] examine the base send/receive performance of VIA on native and Ethernet implementations. They find that 1 GBit/s Ethernet implementations of VIA have the potential of delivering higher throughput than TCP/IP-based protocols. However, native VIA implementations provide about 30-60% better throughput. The authors in [27] compare various MPI implementations in a cluster interconnected with Gigabit Ethernet. The MPI implementations rely either on TCP/IP or a VIA-type substrate for basic communication capabilities. They find that using TCP/IP imposes significant overheads and that VIA-type base communication on top of Gigabit Ethernet has significant potential for improving MPI performance.

Our multi-link approach bears similarity with inverse multiplexing [15, 18]. Inverse multiplexing is a technique to implement a single high-capacity logical channel by using several lower-capacity channels, where the distribution and aggregation of the traffic flow to and from the individual channels is transparent to the higher layers. Although the concept is similar, the tradeoffs and required mechanisms in our setup, i.e., scalable systems, are very different from previous applications of inverse multiplexing [10] that target mostly communication over wide area networks.

The addition of inverse multiplexing to the current network protocols

seems promising to increase networks performance. The implementation of inverse multiplexing in software should distribute individual packets among multiple multiple psychical channels.

Although, existing network protocols usually require a single route for the same flow, that limits the efficiency of inverse multiplexing. Moreover, inverse multiplexing will introduce out-of-order delivery even for packets of the same flow which which degrade the performance of the existing protocols. We aim to design a protocol that is designed to support inverse multiplexing and maximize the performance without any hardware support.

Previous work has examined issues in building multi-rail network configurations. The authors in [12] use simulation to examine rail allocation methods in multi-stage, cluster-based networks. They find that certain allocation methods can result in significant improvements in latency and bandwidth. In contrast, we aim to examine in a real system the overheads and benefits of using multiple rails on edge-based communication subsystems. The authors in [24] examine how multiple rails can be used in Infiniband interconnects under MPI. Our work on one hand uses Ethernet as the interconnect and on the other hand is more transparent in that all higher system layers are able to take advantage of multiple rails.

There have been efforts to improve routing and link utilization aspects of Ethernet. The authors in [34] show how the spanning tree architecture of Ethernet can be improved for Metropolitan Area and Cluster networks. This work is orthogonal to our effort. Currently, most scalable systems use topologies that are structured and well controlled. In fact, a large number of systems tend to be built with a small number of high-radix switches. This, combined with dense SMP nodes leads to systems that can support hundreds of processors with a small number of switches. The authors in [36] design and build a multi-dimensional hyper crossbar network using multiple Gigabit Ethernet interfaces. They find that their system delivers more than

90% of the peak throughput for different micro-benchmarks. To the best of our knowledge, the system they designed supports spatial parallelism but, and in contrast to our work, does not support remote memory operations.

The issue of load balancing protocol processing on multiple CPUs has been examined in [40]. The authors show the hardware extensions and synchronization required for distributed Ethernet processing on multiple CPUs in a high-speed network interface. In our work, we examine the limitations of using multiple host CPUs. We believe that our approach is in-line with current technology trends of using multi-core CPUs as host processors.

The authors in [25, 26] have examined techniques for reducing interrupt cost and its impact on communication throughput. Thus, we believe that the main challenge for future communication protocols is packet processing; Techniques for distributing and balancing protocol processing on multiple cores without increasing synchronization overheads dramatically on the critical path or requiring extensive architectural support are not well understood today. Thus, we believe that protocol processing over multiple, general-purpose cores emerges as a main problem for commodity, high-speed cluster interconnects.

Other research [22] has used MultiEdge to examine the performance and scalability of software shared memory system (GeNIMA [4]). The authors ported GeNIMA to MultiEdge used real applications derived from the SPLASH-2 benchmark suite [41] on a 32-nodes cluster, interconnected with dual 1-GBit/s links, with promising results.

Chapter 5

Conclusions

In this work, we examine the viability of edge-based communication protocols for building scalable servers. We examine both the level of performance they may offer, how they may take advantage of spatial parallelism in the interconnect, and the the impact they may have on network traffic and behavior. Our intention is to explore the extreme configuration point of cluster interconnects where all protocol processing is placed at the edge of the network and does not require any support from the network core.

We design and implement MultiEdge, a communication subsystem that support remote read and write memory operations over ordinary 1- and 10-Gigabit Ethernet interfaces, using raw Ethernet frames. Although our approach is currently intrusive and requires low-level, hardware driver modifications, we believe that future network to host interfaces can provide the necessary support to allow for higher portability.

We also examine the implications of host-level copies for high-speed communication protocols over Ethernet-based interconnects. We first analyze the impact on throughput and CPU utilization on the send and receive paths. We then examine how copies can be eliminated using page remapping. Finally, we explore the dependency of page remapping on communication buffer alignment.

Our micro-benchmark results show that MultiEdge is able to deliver about 66% of the nominal link throughput with 10-GBit/s links and about 99.2% with 1-GBit/s links. The minimum latency is about 23μs.

Moreover, we find that multiple link configurations are limited by a larger number of interrupts and poor load balancing of the send and receive paths over multiple CPUs. The impact of memory copies is significant, and removing them results in up-to 66% improvement in maximum throughput. Furthermore, removing interrupts results in up-to 38% improvement in maximum throughput.

We find that eliminating copies with address translation and page remapping results in 80-90% improvement and allows reaching a maximum of 98% and 53% of available throughput in one-way and two-way respectively. After copies are eliminated, the bottleneck is mainly receive path processing. Moreover, interrupt processing, remapping, packet processing, and NIC accesses contribute similarly to CPU utilization.

Overall, we believe that using page remapping for dealing with copies is an effective technique for communication protocols at high-speeds. We believe that the main challenge for future communication protocols is packet processing; Techniques for distributing and balancing protocol processing on multiple cores without increasing synchronization overheads dramatically on the critical path or requiring extensive architectural support are not well understood today. Thus, we believe that protocol processing over multiple, general-purpose cores emerges as a main problem for commodity, high-speed cluster interconnects.

Bibliography

[1] 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.

[2] M. Baker, P. A. Farrell, H. Ong, and S. L. Scott. Via communication performance on a gigabit ethernet cluster. In *Euro-Par 2001, published as Lecture Notes in Computer Science (LNCS)*, volume 2150, pages 132–142, Jan. 2001.

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

[4] A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA26)*, May 1999.

[5] M. Blumrich, C. Dubnick, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *Proc. of The 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA2)*, Feb. 1996.

[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proc. of the 21st International Symposium on Computer Architecture (ISCA21)*, pages 142–153, Apr. 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[8] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *IEEE Infocom 1997*, Apr. 1997.

[9] J. Chase, A. Gallatin, and K. Yocum. End-system optimizations for high-speed tcp. *IEEE Communications*, 39(4):68–74, 2001. Special issue on TCP Performance in Future Networking Environments.

[10] F. M. Chiussi, D. A. Khotimsky, and S. Krishnan. Generalized inverse multiplexing of switched atm connections. In *Global Telecommunications Conference 1998 (GLOBECOM'98)*, volume 5, pages 3134–3140, Nov. 1998.

[11] H. K. J. Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[12] 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.

[13] 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, December 1993.

[14] 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.

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

[16] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V). Stanford, CA, USA.*, Aug. 1997.

[17] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture (ISCA19)*, pages 256–266, May 1992.

[18] P. Fredette. The past, present, and future of inverse multiplexing. In *IEEE Communications Magazine*, volume 32, pages 42–46, Apr. 1994.

[19] Giganet. Giganet cLAN family of products. http://www.emulex.com/products.html, 2001.

[20] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proc. of the IEEE Spring COMPCON '96*, Feb. 1996.

[21] An infiniband technology overview. Infiniband Trade Association, http://www.infinibandta.org/ibta.

[22] S. Karlsson, S. Passas, G. Kotsis, and A. Bilas. MultiEdge: An edge-based communication subsystem for scalable commodity servers. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007.

[23] 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.

[24] J. Liu, A. Vishnu, and D. K. Panda. Building multirail infiniband clusters: Mpi-level design and performance evaluation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 33, Washington, DC, USA, 2004. IEEE Computer Society.

[25] M. Marazakis, V. Papaefstathiou, and A. Bilas. Optimization and Bottleneck Analysis of Network Block I/O in Commodity Storage Systems. In *Proc. of the 21st ACM International Conference on Supercomputing (ICS07)*, June 2007.

[26] M. Marazakis, K. Xinidis, V. Papaefstathiou, and A. Bilas. Efficient Remote Block-level I/O over an RDMA-capable NIC. In *Proc. of the 20th ACM International Conference on Supercomputing (ICS06)*, June 2006.

[27] H. Ong and P. A. Farrell. Performance comparison of lam/mpi, mpich, and mvich on a linux cluster connected by a gigabit ethernet network. In *4th Annual Linux Showcase and Conference*, Oct. 2000.

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

[29] S. Passas and A. Bilas. Reducing host-level copy overhead at the 15-30 gbits/s range over commodity ethernet-based cluster interconnects. In *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS 2008). [Under Submission]*, June 2008.

[30] S. Passas, G. Kotsis, S. Kalrsson, and A. Bilas. Spatial parallelism in gigabit ethernet cluster communication subsystems. In *Proceedings*

of the 2008 Workshop on Communication Architectures for Clusters (CAC2008). Held in Conjuction with IPDPS2008, Apr. 2008.

[31] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network (qsnet): High-performance clustering technology. In *Proc. of The 2001 IEEE Symposium on High Performance Interconnects (HOT Interconnects 9). Stanford, CA, USA.*, Aug. 2001.

[32] F. Petrini, W. Feng, A. Hoisie, S. Coll, and F. E. The quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.

[33] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In *PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA*, March 30 – April 3 1998.

[34] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: a multi-spanning-tree ethernet architecture for metropolitan area and cluster networks. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2283–2294, Mar. 2004.

[35] 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.

[36] 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.

[37] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.

[38] TOP500.Org. Top500 supercomputing sites, Nov. 2007. http://www.top500.org.

[39] 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)*, February 1997.

[40] P. Willmann, H. Kim, S. Rixner, and V. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *International Symposium on High Performance Computer Architecture (HPCA), San Francisco, CA*, Feb. 2005.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA22)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.