HIGH Performance Computing using Portals on Intercept

Diploma Thesis

Gérard Basler[1]
March 2003

Winter Term 2002/03

Swiss Federal Institute of Technology Zürich, Switzerland
The University of New Mexico, Albuquerque USA

[1] gbasler@cs.unm.edu

Proposal

Implementing the Portals API for the Interceptor Network

Diploma Thesis Proposal
Gérard Basler

This work will be conducted in the Computer Science Department at the University of New Mexico under the supervision of Professor Arthur B. Maccabe starting in November of 2002.

Abstract

The goal of this project is to develop a high quality implementation of the Portals API for the Interceptor network. This implementation will be based on the Portals reference implementation and an existing Linux driver for Interceptor. Because the Portals API and the original Interceptor driver both support MPI, we will evaluate the success of this project by measuring performance on MPI applications. In addition to the standard bandwidth and latency tests, we will also consider the ability of the implementation to overlap computation with communication.

Background

We are primarily interested in providing support for "resource constrained" applications, applications that can be scaled to consume all of at least one of the resources provided by a computing system (e.g., memory, communication bandwidth, processor capacity, etc.). Traditionally, these applications were called "Grand Challenge" problems. We prefer the name "resource constrained" because it emphasizes the need to minimize overhead in the design and implementation of systems level software.

The Portals API was developed jointly between the University of New Mexico and Sandia National Laboratories. The API was designed to support high-performance communication in parallel, distributed memory systems consisting of tens to hundreds of thousands of nodes. This API provides the basis for variety of other services, including the system management software used on ASCI/Red and Cplant, the Lustre files system being developed by Peter Braam, and a high-performance MPI implementation.

The Interceptor network is a high-performance, scalable networking technology being developed at Super Computing Systems (SCS) in Zurich, Switzerland. This networking technology is aimed at the same environment that the Portals API is aimed at and, as such, developing an implementation of Portals for the Interceptor network will provide a good evaluation of the applicability of this technology for future systems.

Approach

To avoid duplicating unnecessary work, our approach is to blend an existing Linux driver for Interceptor from SCS with the prototype Portals implementation available from Sandia/UNM. The Linux driver will provide much of the basic functionality needed and will provide examples of sending and receiving messages on the Interceptor network.

The prototype Portals implementation was designed to support moving functionality among the application, the operating system, and specialized processors, e.g., processors on programmable Network Interface Cards (NIC). The Portals implementation achieves this ability to migrate functionality by using "call forwarding." In call forwarding, any internal library call can be replaced by an indirect call which builds a parameter block and issues the appropriate linkage (e.g., using an OS trap or by appending the parameter block to a buffer of NIC requests) to forward the call to the location where the needed functionality is best implemented.

In addition to call forwarding, the Portals prototype implementation uses a Network Abstraction Layer (NAL) to hide the details of the network activities involved in sending and receiving packets.

Porting the Portals prototype implementation involves implementing the NAL and deciding where different parts of the functionality needed for the interface should be implemented. The latter of these tends to take more time and has a greater impact on the quality of the resulting implementation.

Because the Interceptor NICs are not programmable, we can not consider implementing any of the Portals functionality on the NIC. However, the ability to move functionality between the application and the operating

system will be critical to providing a high quality implementation of Portals on Interceptor NICs. As such, the implementation will be based on one or more kernel threads in addition to the Portals library.

Because we are interested in providing support for resource constrained applications, we will not consider the possibility of multiple processes on a node. A single application may be multithreaded; however, issues related to multithreading are assumed to be handled at a higher level in the Portals library and will not be considered directly in this implementation. As such, the implementation can use the virtual NICs provided by the Interceptor NICs for implementing different parts of the Portals API.

Specific Tasks and Milestones

Task 1. **Identify and evaluate workstations to be used in this project.**

The goal is to determine the requirements for the workstations used in the experiments and to run baseline benchmarks evaluating the performance of the basic Interceptor drivers on these machines.

Milestone: running the ping-pong and bandwidth applications available from SCS.

Task 2. **Become familiar with SCS Interceptor Linux driver.**

Here, the goal is to understand the Interceptor NICs and the code used in the Linux driver to the extent that we can consider alternative implementations for the Portals port.

Activities: read Interceptor documentation, read driver code, and interact with the appropriate engineers at SCS.

Milestone: Introduce a kernel thread that responds, using interrupts, to a message in one of the ring buffers. The message in the ring buffer will describe a logical memory region in the local process (e.g., a 32-bit starting address and a length). In response, the kernel thread will reply with the portion of the application's page table (physical addresses) covered by the logical memory region.

Task 3. **Become familiar with the Portals API and the prototype Portals implementation.**

Activities: read Portals documents, papers related to Portals, and the code for the prototype implementation.

Milestone: a design document describing the approach to be used in implementing Portals using Interceptor NICs.

Task 4. **Port the prototype Portals implementation to the Interceptor network.**

Activities: based on the design developed in Task 3, identify at least two intermediate steps and measurements that can be used to demonstrate progress.

Milestone: ability to run several MPI programs that use the MPI over Portals libraries.

Task 5. **Measure the quality of the implementation.**

Activities: compile, run, and evaluate a suite of MPI benchmark programs. As a minimum, these programs will include a bandwidth test, a latency test, and a test that measures the effectiveness of computation and communication overlap.

Milestone: Analysis of results

Task 6. **Write-up and defend diploma thesis.**

Prof. Barney Maccabe
The University of New Mexico
Albuquerque, USA

Prof. Gerhard Tröster
Swiss Federal Institute of Technology
Zürich, Switzerland

Contents

1 Introduction **1**
- 1.1 Basics . . . 1
- 1.2 Message Passing . . . 1
- 1.3 Zero Copy, OS Bypass and Application Bypass . . . 1

2 The Intercept Network **3**
- 2.1 The Hardware . . . 3
  - 2.1.1 Network Properties . . . 3
  - 2.1.2 Network Packets . . . 3
  - 2.1.3 64 Bit PCI-X Network Interface Card . . . 4
  - 2.1.4 Switches . . . 6
- 2.2 Software . . . 6
  - 2.2.1 System Overview . . . 6
  - 2.2.2 VDMA handler . . . 7

3 Portals **12**
- 3.1 Introduction . . . 12
- 3.2 Portal Addressing . . . 12
- 3.3 Architecture . . . 13
  - 3.3.1 NAL . . . 13
  - 3.3.2 Myrinet Driver . . . 14
  - 3.3.3 Portal dataflow . . . 14

4 Design **18**
- 4.1 Case study . . . 18
  - 4.1.1 Pure software - VDMA handler's CB . . . 18
  - 4.1.2 Pure software - VDMA handler . . . 19
  - 4.1.3 Pure software - VDMA handler & CB protocol in kernelspace . . . 20

4.1.4 Pure software - VDMA handler & CB protocol in userspace . . . 20
4.1.5 Software & VHDL . . . 21
4.2 Conclusion . . . 22

5 Implementation **23**
5.1 Provisorly solution - VDMA handler CB . . . 23
5.2 Final solution . . . 23
5.2.1 Architecture . . . 24
5.2.2 VDMA . . . 25
5.2.3 CB protocol . . . 29
5.2.4 Portals P3mod . . . 31
5.2.5 Current implementation . . . 31
5.3 Code extracts . . . 32
5.3.1 API-side NAL . . . 32
5.3.2 LIB-side NAL . . . 36

6 Testing **57**
6.1 First Development System . . . 57
6.1.1 Loopback tests . . . 57
6.1.2 Ping-pong tests . . . 57
6.2 Second Development System . . . 60
6.3 MPI . . . 62

7 Conclusion **63**
7.1 Results . . . 63
7.2 Prospects . . . 63
7.3 Portals Feedback . . . 63
7.4 Commentary . . . 64
7.5 Thank You . . . 64

A Source Code **66**
A.1 Ping-pong . . . 66
A.2 Bandwith . . . 77

B Portals Installation **89**

List of Figures

2.1 Packets from different senders may be shuffled at the receiver 5
2.2 System overview 7
2.3 Processing of read requests / VDMA_GET 10
2.4 Processing of write requests / VDMA_PUT 11

3.1 Portals 3 NAL architecture 13
3.2 Portals Myrinet Driver architecture 15
3.3 Simplified example of a Portal *put* 16
3.4 Portals send message 17

5.1 Comparing the myrinet driver and the FCI driver 24
5.2 The Portals Intercept driver architecture 25
5.3 Unaligned blocks; sender and receiver have the same displacement 26
5.4 Unaligned blocks; sender and receiver have a different displacement 27
5.5 Dataflow for long messages over VDMA transfer 28
5.6 Dataflow for short messages over the CB protocol 32
5.7 Architecture short messages over the CB protocol 33

6.1 1*th* development system at the Scalable Systems Lab, UNM . 58
6.2 Running NICs in the development system 58
6.3 Development System network topology 59
6.4 2*nd* development system at the Scalable Systems Lab, UNM . 61
6.5 Running NICs in the development system 61

List of Tables

5.1 Packet types for a simplified RMPP 30

Chapter 1

Introduction

1.1 Basics

In the past there where two approaches to build supercomputers. The first one was to build a very powerful CPU and to connect a few of them together. These **VLIW** (very long instruction word) CPUs operated on vectors and could therefore perform the same operation on a large number of variables at the same time. These architectures were expensive and their size was limited because of the limited scalability of the shared memory architectures. The second approach, that became the common case when cheap bulk hardware was available, was to use standard computers and connect them to a cluster.

1.2 Message Passing

A widely used paradigm for process intercommunication between different machines is **message passing**. The Message Passing Interface **MPI** is the de facto standard for writing programs that run on parallel machines (`http://www.mpi-forum.org/`, a good introduction can be found in [10]).

1.3 Zero Copy, OS Bypass and Application Bypass

A good message passing implementation will avoid memory copies because network bandwith approaches memory bandwith on high performance clusters. **Zero-copy** NICs that directly access the memory space of the application have become the state of the art. Intelligent NICs that have their own processor are even capable to control the transfer of incoming messages without needing to interrupt the CPU. Because this strategy does not

need to involve the OS on every message transfer, it is frequently called **"OS Bypass."** Many protocols that support OS Bypass still require that the application actively participate in the protocol to ensure progress. This complicates the runtime environment, requiring a thread to process incoming requests, and significantly increases the latency required to initiate a long message protocol. The term **"Application Bypass"** refers to the technique where no intervention of the application or an application level thread is needed to ensure progress. (After [1, Portals Documentation]).

Chapter 2

The Intercept Network

Intercept is a complete network including hardware and software for high performance computing that has been developed by **Supercomputing Systems (SCS)**, Zürich, Switzerland.

The information presented here is a brief summary of several talks from Supercomputing Systems.

2.1 The Hardware

2.1.1 Network Properties

All packets in the network are protected with a *CRC-32 checksum*. The network uses a **link-to-link retransmission** to retransmit lost and corrupt packets. One acknowledge packet is sent back at every link for every correctly received data packet. A retransmission request packet is sent back at every link, if a corrupt packet is received. Up to seven packets can be outstanding on a link.

2.1.2 Network Packets

Network Layer

Currently the firmware supports multicast and destination based unicast packets. A unicast packet contains a 16 bit *Intercept unicast address* which gives a total of 65536 possible nodes. Source based routing functionality will be added in the future.

Transport Layer

Currently there are three different types of packets.

- **Virtual DMA packets** contain a 2 bit *channel number* and a 25 bit *channel virtual address*.
- **Short message packets** contain 0-63 qwords of data (0-504 bytes).
- **Short message command packets** are used to send flow control commands.

Moreover the NIC supports

- **two-sided** communication for **MPI 1.2** (sender and receiver take part of a communication)
- and **one-sided** communication for **MPI 2.0** (GET / PUT from remote process without interaction of the remote CPU).

2.1.3 64 Bit PCI-X Network Interface Card

The NIC acts as 16 independent **virtual NICs**. Each application process gets its own virtual NIC. This eliminates the need to obtain a lock to access the hardware. Therefore an application can immediately access the NIC without any lock-latency. 15 VNICs are provided to the user, one is reserved for the VDMA handler 2.2.2. The network is built out of **four independent networks with 2.5 Gbit/s** each. These networks are called **planes**. Together they achieve the full bandwith of **10 Gbit/s**. No physical connections exist among them. Each packet has a mask that specifies over which planes it may be sent. The firmware distributes the packets equally over all available planes. The motivation for this architecture was to *increase fault tolerance* and to *lower the hardware cost*. To meet the low latency and high bandwith requirements **two protocols** are used:

- A **low latency** protocol for **short messages** with very little overhead.
- And a **high bandwith** protocol for **long messages** with *zero copy* and *direct memory access* which adds overhead.

Each VNIC provides one **receive circular buffer (RX-CB)** (1-128 MB) and two **transmit circular buffer (TX-CB)** (64 KB). Both buffers reside in host memory. The NIC has one physical interrupt that is demultiplexed in software over the 16 VNICs. A virtual interrupt may be raised for every received packet or only for marked packets.

CB Protocol

The TX-CB contains the transfer commands and data as well. This allows pipelined setup of several transfers without waiting for the NIC to became ready to transfer the next packet.

- **Direct transfers** contain short messages; the data is directly copied into the TX-CB (but it may be sent using zero copy as well). On the receiving side the data always has to be copied.
- **Indirect transfers** consist of the DMA command only.

As a consequence of the four planes, incoming packets from one sender *may arrive in a different order at the receiver* (if the packets are allowed to be sent over more than one plane). Packets from different senders may be shuffled at the receiver as well (Fig. 2.1). The **defragmentation** has to be done in software.

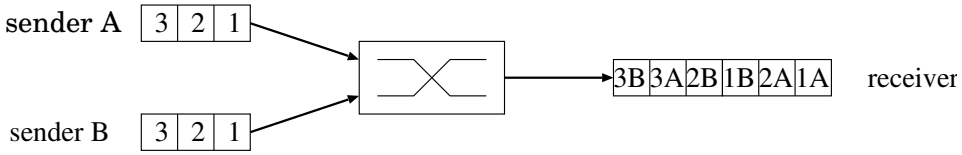

Figure 2.1: Packets from different senders may be shuffled at the receiver.

Virtual DMA Protocol

The virtual DMA protocol ressources consist of **four channels**, each channel has a **page table** with 256 entries, a **packet counter** and a **virtual interrupt**. With 1024 page table entries and 4 KB page size, 4 MB of memory can be mapped. Transfers must be segmented into 1 MB blocks by software (256 entries per pagetable x 4 KB page size = 1 MB). With this technique a VDMA transfer may be running and the software can set up the next three transfers in advance. Only one transfer is sent / received at a time in order to avoid network contention. Since the hardware counts the incoming packets, an interrupt is raised for every received 1 MB block. This reduces the number of interrupts to a very low amount (less than 1000 a second for full bandwith). The data packets arrive unordered but the software will not notice since the NIC notifies the software only when the full block is received.

Technical details

The 64 Bit PCI-X nic consists of the folliwing major parts:

- **CC**: The **communication controller** is a **XilinX XC2V3CCC-4** that provides 14336 slices with 2 flip-flop and 2 look-up tables each. The current firmware provides two-sided communication and 8 VNICs. With this configuration, FPGA is about 33% full.
- **SDRAM**: The **128 MByte SDRAM** will be used completely for *one-sided* communication (when the firmware is done). The nic may be upgraded to up to 512 MB of SDRAM.
- **SRAM**: A **4 MBit SRAM** chip is used for *end-to-end retransmission* (see 2.1.1). More than 50% are used. The nic can hold up to 16 MBit of SRAM.
- Four **optical modules**: Each **optical module** represents a **plane**. The user controlls the number of connected and available planes.

2.1.4 Switches

The Intercept network is build of 32 port switches. The contain a **full bandwith crossbar** and use **cut-through routing**. Since they switch each plane separately they can be configured as

- 8 x 4 planes switch module
- 16 x 2 planes switch module
- 32 x 1 planes switch module

2.2 Software

2.2.1 System Overview

The **MPI application** performs the actual work and communicates with the other instances on other nodes through the **MPI libarary**. The MPI library itself is based on the **FCI** (Fast Communication Interface) library that builds the heart of the communication interface. It supports **OS bypass** for short messages. Long messages are sent over the **Intercept Device Driver**. It contains a **VDMA handler** that coordinates all VDMA transfers of the application processes.

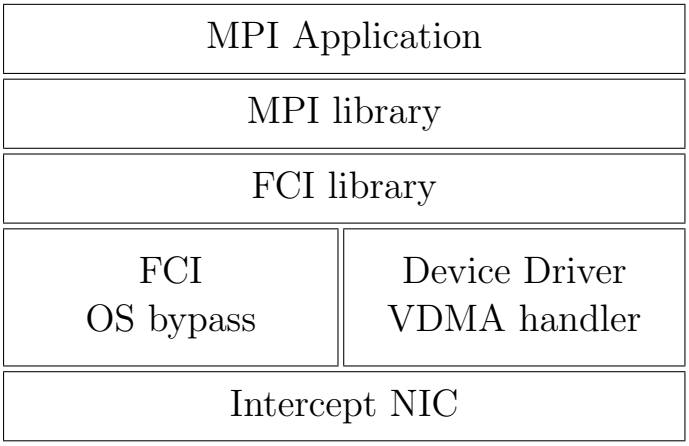

Figure 2.2: System overview.

2.2.2 VDMA handler

The Linux device driver is the most relevant part in this system for the Intercept port of Portals. It manages the available nics and provides access to the ressources (e.g. *Virual NICs*) on them to other drivers and application processes. It loads the code for the *communication controller* and the *link controller* into the FPGAs on the nic and initializes it. The device driver also provides a **VDMA handler** that is used to bundle all VDMA transfers. Altough every userspaceprocess could start a VDMA transfer by allocating a VNIC from the device driver and writing a VDMA transfer into the TX-CB of the VNIC, there must be some kernelspace code that parses the page tables of the userspace process and sets the VDMA page table up. Since the VDMA engine is shared between all VNICs there must be a lock or a commont point where all the transfers are initiated. A separate driver has furthermore the advantage that VDMA transfers may be set up pipelined.

The following section gives a brief overview of the internal function of the VDMA handler as far as it's needed to understand the implementation of the Portals driver for Intercept. More information can be found in [8].

As soon as the Linux device driver is started up, it launches the **VDMA kernelthread** that handles all the VMDA transfer requests from the userspace clients or from a remote machine. *One VNIC is reserved for the VDMA handler* to exchange control and flow control messages between the VMDA handlers on different nodes.

All procedures that involve a communication with another VDMA han-

dler are done in the kernelthread. Therefore the VDMA handler internally uses a bunch of queues. Almost every *ioctl()* command adds a new request to a queue that is to be processed by the kernelthread.

The VMDA handler implements a **connection-based** communication protocol. Every userspace process that wants to communicate with a remote process has to announce at startup to which processes on which nodes it wants to talk.

The **flow control** is credit-based: Every nodes gets an initial credit to begin with. If the credits drop to zero it has to stop sending and must wait for incoming credits (the credit is updated by the receiver). The credit is big enough to hide the latency of the flow control mechanism in order to saturate the link.

There are two different VDMA commands: a **read** and a **write** command. This distinction is only made in software (it simulates the write command). The hardware knows of only one kind of VDMA transfer. If a process wants to read remote memory then the transfer is sliced into **1MB slices** (See 2.1.3). The handler sends a **VDMA_GET** message over the TX-CB of its own VNIC to the remote peer VDMA handler for every slice. To write to remote memory a **VDMA_PUT** command is used. The following is a short description of how these commands are implemented.

- **VDMA_GET**: The initiating VDMA handler sets up the page tables and sends a GET request to the remote VDMA handler for the next slice that sets up its page tables and starts transferring the data. When the packet counter on the receiver has reached zero, the NIC raises an interrupt and the VDMA handler starts the transfer of the next slice as described before. The packet counter on the sender will drop to zero first and the handler will release the VDMA channel. (Figure 2.3).

- **VDMA_PUT**: Since the page tables have to be set up first, it is not possible to start a VDMA transfer and hope the receiver will receive it. The receiver always has to set up page tables and to reserve a channel before it can actually receive data. Hence a real put operation is not possible. The initiator therefore simply sends a *VDMA_PUT* to the remote handler that cuts the transfer into slices and sends *VDMA_GET* messages back to get the message. After the whole message is transferred, the remote handler sends a *VDMA_DONE* control message back to the initiator that can notify the userspace process of the completion of the transfer. (The reason for this additional message is that the initator can know when a slice is sent but it cannot know when all the slices are sended.) (Figure 2.4).

The current hardware / firmware imposes the following **limitations to a VDMA transfer**:

- The *firmware* requests that every **buffer address** has to be aligned to one quadword (8 bytes). The *VDMA handler* accepts only transfers that are **aligned to four quadwords** (32 bytes). (I was told that the implementation was easier this way).
- The **length of a message** must be a **multiple of one quadword** (8 bytes) and must have a **minimal length** of **three quadwords** (24 bytes).

One VDMA handler kernelthread is started for every plugged network card.

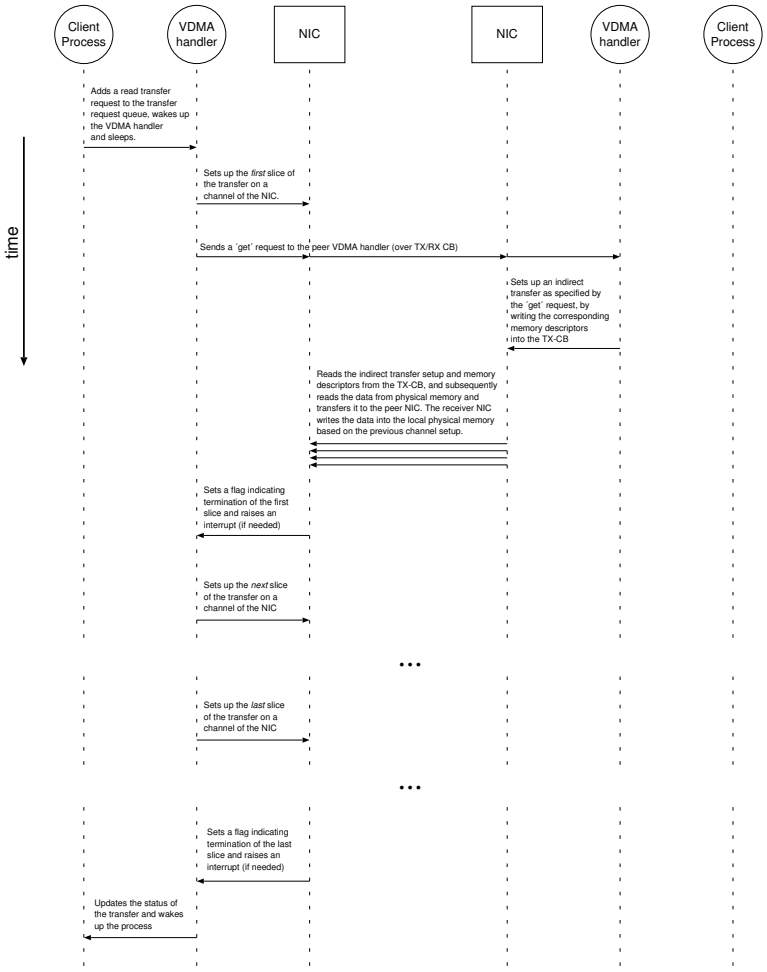

Figure 2.3: Processing of read requests / VDMA_GET. Figure taken from [8]

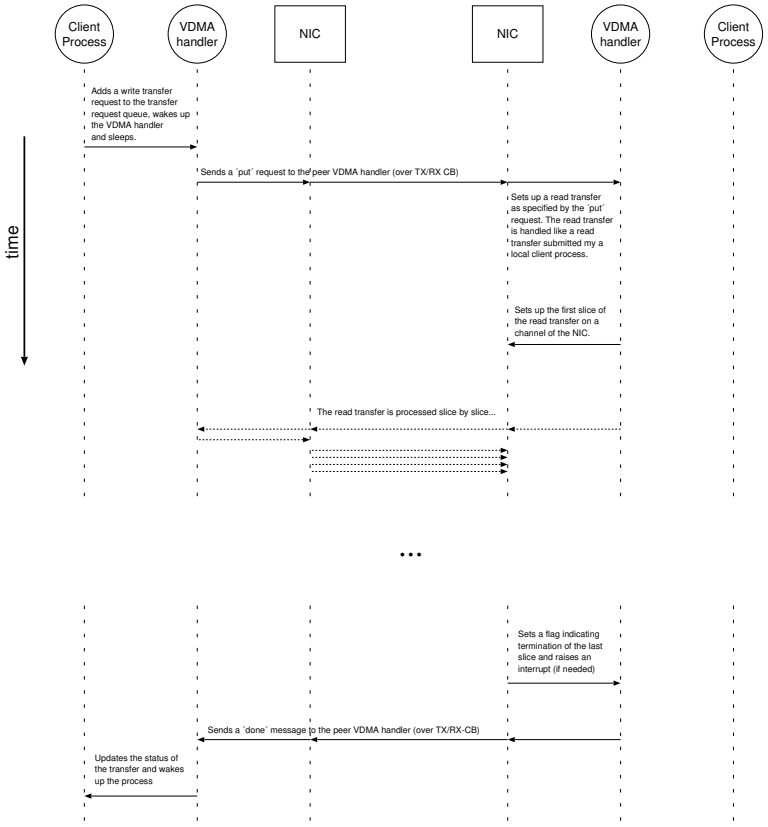

Figure 2.4: Processing of write requests / VDMA_PUT. Figure taken from [8]

Chapter 3

Portals

3.1 Introduction

Portals is the message passing technology that was developed for the Cplant [5] project at Sandia National Laboratories. It is designed to support the *scalability* requirements to construct a commodity cluster that can scale up to the order of ten thousand nodes. Portals is **connectionless**: a Process is not required to explicitly establish a point-to-point connection with another process to communicate. Portals is an **OS Bypass** and **Application Bypass** protocol, although the current reference implementation does not support them.

More information about Portals can be found in [1]. The source code is free available through Sourgeforge:
`http://sourceforge.net/projects/sandiaportals`.

3.2 Portal Addressing

This section will give a short introduction to the Portals 3 addressing mechanism. A portal represents an opening in the address space of a process. Portals combine the characteristics of both one-sided and two-sided communication. A Portal **matching get** operation reads data from another process, while a **matching put** performs a write operation.

Portals uses four components to address memory on a remote node: A **Process id**, a **memory buffer id / portal id**, an **offset within the memory descriptor** and a set of **match bits**. These match bits describe a specific pattern that the incoming data must match to use that match entry. Within each match entry is a list of memory descriptors that define

a region in memory as well as the behavior associated with that region, like how many and what kind of operations can be done using it. Although the match entry contains a list of memory descriptors, only the first one is considered when matching incoming data. Each memory descriptor can contain an **event queue** that is updated when an operation is performed on the region to let the application know what has happened.

3.3 Architecture

3.3.1 NAL

The implementation strategy of Portals 3 is to provide a highly platform and network independent API for message passing applications. The concept of a **network abstraction layer** (NAL) is used to make migration from one network to another easy. Portals 3 is divided into two parts: an application programming interface (API) and a library (LIB). Basically figure 3.1 shows the software hierarchy, highlighting the relevant sourcefiles for the NAL.

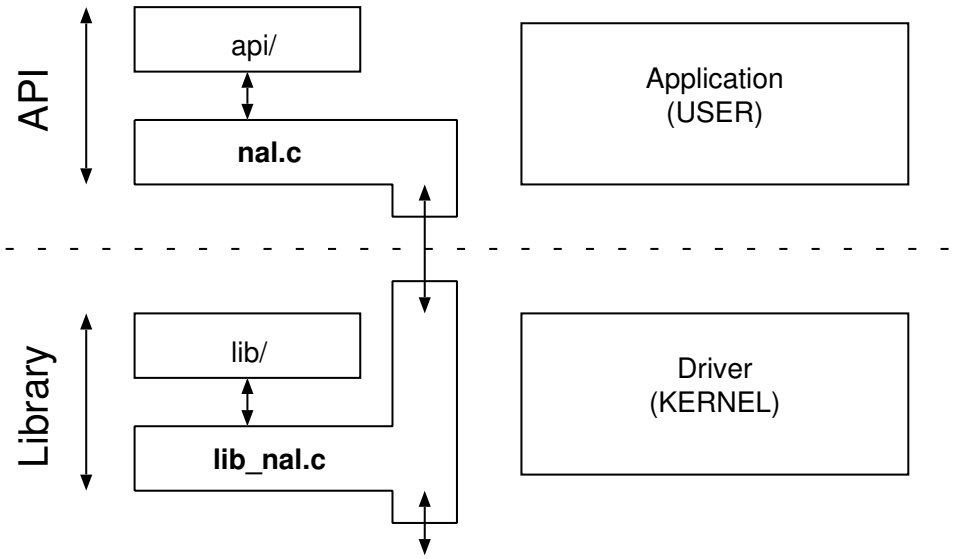

Figure 3.1: Portals 3 NAL architecture.

It's up to the user what kind of NAL they want:

- **"Kernel only NAL"**, both API-side and LIB-side live in kernel. Portals API calls can only be made from the kernel (qswnal, socknal, gm-

nal). Since these NALs are not accessable from userspace, only other kernel modules are able to use them. (E.g. [4, the lustre filesystem].)

- **"User only NAL"**, both API-side and LIB-side live in userspace. Portals API calls can only be made from user-level. (tcpnal, ipnal)
- **"Universal NAL"**, API-sides for both user-space and kernel-space, LIB-side lives in kernel. Portals API calls can be made from both user-level and kernel level. (no universal NALs currently exist)
- **"Split NAL"**, API-side in user space, LIB-side in kernel. Portals API calls can only be made from user-space. (rtscts NAL)
- **"NIC NAL"**, API-side in user space, LIB-side executes on the NIC. (no NIC NALs currently exist)
- **"NIC and Kernel NAL"**, API-side in user space, LIB-side executes partially on NIC and partially in Kernel. Portals API calls can only be made from user-level. (Mike Levenhagens Myrinet NAL)

3.3.2 Myrinet Driver

This section will give a brief overwiev of the Portals Myrinet Driver. It was used as a reference implementation to understand how a user accessible NAL works. Figure 3.2 illustrates the architecture.

The Portals api and the userspace side of the NAL are directly linked to the application. All calls into Portals are forwarded through a *forward* function in the userspace-side of the NAL. This function uses the *ioctl()* function in the module *P3mod*, that dispatches the calls to Portals. The whole Portals library resides completely in the kernel.

3.3.3 Portal dataflow

A Portal **put** will serve as an example look at the network abstraction layer and the dataflow in Portals 3. Figure 3.3 shows the simplified situation of a point-to-point put from the initiator to the target.

Figure 3.2 describes the simplified communication scheme that occurs during a call to *PtlPut*. After some initial processing by the API, control is passed down to the NAL. The NAL function *forward* calls *ioctl*, a Linux system call for communicating with a module. The library side of the NAL receives this call and immediately dispatches it to the Portals library, where all of the data necessary for communication with another node is assembled

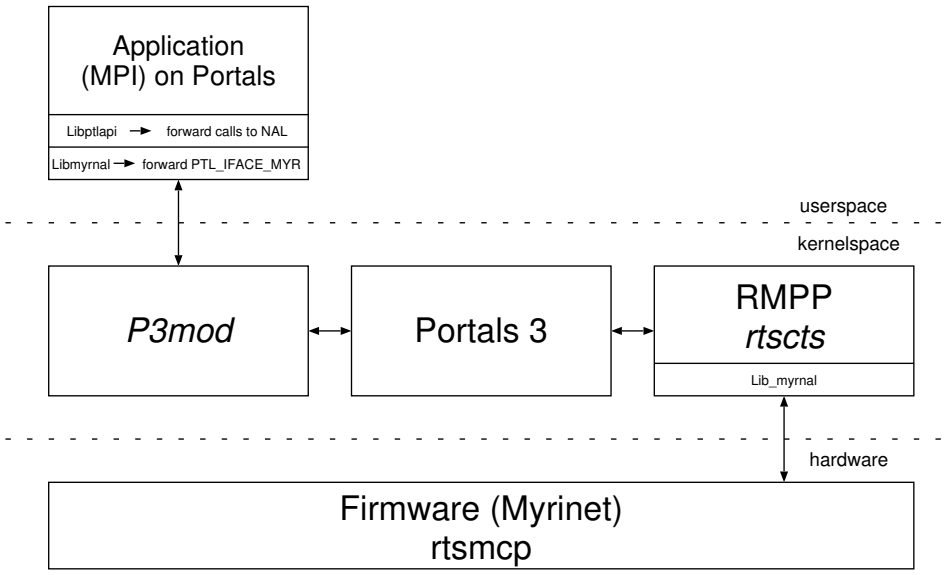

Figure 3.2: Portals Myrinet Driver architecture.

and handed back to the NAL again. The NAL then calls the routines that actually transmit the data over the NIC.

When the data arrives on the remote side, the incoming data handler passes the appropriate information to the Portals *lib_parse* routine and, consequently, to *parse_put*. This function finds the appropriate memory descriptor, if one has been setup, and calls the library side NAL to actually receive the data. The transaction concludes when *nal_recv* calls *lib_finalize* and the appropriate event queue is updated in user space with a *put event*.

Some high-performance network cards (e.g. Gigabit Ethernet, Myrinet) possess their own CPUs such that the underlying communication protocol can be splintered to the NIC. Portals goes one step beyond that making it possible for whole messages to be sent / received without interruption from the host CPU. On advanced nics the Portals matching procedure may be transfered to the NIC, in which case the application only receives *START and END events* when a process takes place or ended. Figure 3.4 visualizes this possibility. Portals supports another feature that some nics with on board memory offer: It reads only the Portals header from the wire; the nic is supposed to hold the transfer of the rest of the message back (it might temporarly store it in the on-board memory). After Portals has analyzed

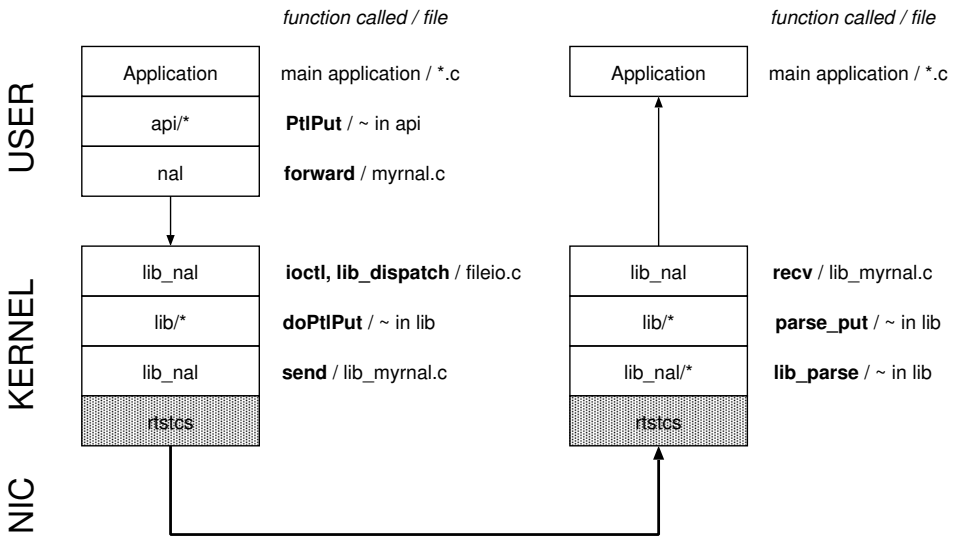

Figure 3.3: Simplified example of a Portal *put*.

the Portals header it uses a *callback* function in the NAL and that informs the nic where in memory it has to copy the message.

A very important characteristic of Portals is that it guarentees **in-order beginning** of transfers but not **in-order completion**. This is important when writing a driver.

(Parts of this chapter have been taken from [2]).

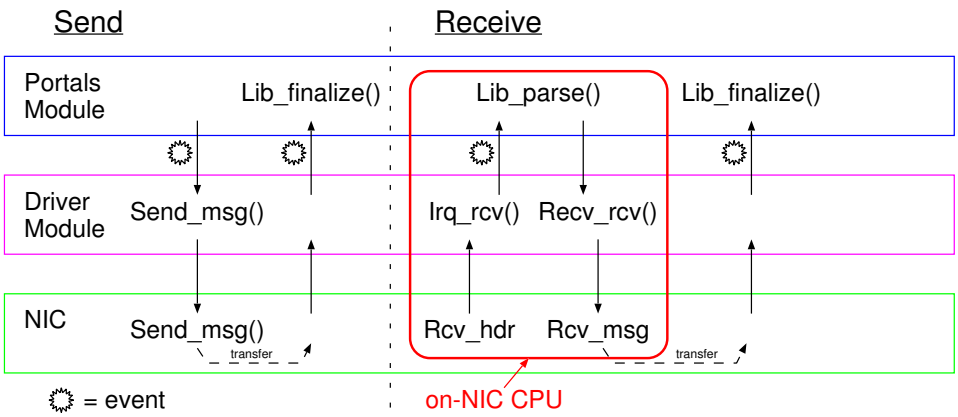

Figure 3.4: Portals send message.

Chapter 4

Design

This chapter contains the study on different implementation possibilities and the final design concept that has been chosen to build the Intercept driver for Portals 3.

4.1 Case study

Portals makes the assumption that not every packet that the nic receives has to be written into main memory. When a packet arrives it's first stored in on-nic memory. Afterwards the nic raises an interrupt and the host cpu decides how many bytes of the packet it wants and starts the DMA transfer from the nic memory to main memory. That methodology is not applicable with the current firmware of the Intercept nic. Every packet arrives in main memory. Therefore Portals has to be asked how many bytes to with memory address it wants, *before* the transfer is started.

4.1.1 Pure software - VDMA handler's CB

The simplest approach that someone could think of is to use the **circular buffer** protocol of the **VDMA handler** to send every message. Remember (2.2.2): Each VDMA handler allocates one VNIC for itself and uses its CB to transmit VDMA set up and flow-control messages. This approach is easy to implement but has several drawbacks:

1. **Only 504 bytes per message** can be transmitted through the CB send and receive function of the VDMA handler. In order to transmit bigger messages several major changes had to be made.

2. The VDMA handler makes usage of **extensive memory copies** to (un-)marshall messages.Maximal three copies are made on the sending side: A copy from the userspace buffer to a temporarly kernelspace buffer. Marshalling of the message into another buffer. And one copy more if the TX-CB is full and the message has to be copied into a temporarly queue. Two copies are needed on the receiver's side: One copy to unmarshall the message and one copy from kernelspace to userspace.

3. The VDMA handler is (unlike Portals) **connection based**. Every connection has to bestablished at program start in order to use it. Fundamental changes in the VDMA handler are needed to change that. Especially the **flow-control** is based on permanent connections.

4. This solution would completely miss out the **VDMA protocol** and therefore never achieve the best possible transfer rates because the memory copies would slow down the PCI-X bus.
   ⇒ The network bandwith is in the same region as the memory bandwith, hence each memory copy generates two additional memory accesses (read/write) - if even if we consider the fact that modern CPUs have large caches. Since the message has to be copied packetwise into the buffer, a copy is only about 500 bytes. The further copies are done very fast in the caches but in order to start the transfer of the message the driver uses a memory barrier causing the CPU to write all unwritten data back into main memory.
   Two additional memory copies would slow down the transfer rate to one fifth!

5. Since single packets may get shuffled at the receiver, the VDMA handler uses only **one plane** to keep the order of the packets. Therefore only **one quarter of the bandwith** will be available!

4.1.2 Pure software - VDMA handler

The next better approach is to use VDMA transfers for every message. Every message will be passed to the VDMA handler that sends it over the VDMA protocol. This requires some additional control messages (4.1). Also some restrictions concerning message size and alignement are present (2.2.2). A driver that uses this design will have to deal with the connection based nature of the VDMA handler as well.

4.1.3 Pure software - VDMA handler & CB protocol in kernelspace

This is a solution that combines the advantages of the short message and the long message protocol. If a message is big enough then it will be sent over the VDMA handler. The driver will allocats its own VNIC to send small messages with the shortest possible latency. The decision when to use which protocol will be made in the driver. In order to be able to use **all four planes** and to have a **flow-control**, a **new protocol** has be be implemented. The **TX/RX circular buffer** will be allocated in **kernelspace** in order to keep the whole driver in kernelspace as well. All commands will simply be forwarded from the userspace nal to the kernelspace nal. Furthermore a **locking mechanism** for the VNIC is still needed if more than one process of the application are allowed to use the nic.

4.1.4 Pure software - VDMA handler & CB protocol in userspace

The next better approach will move one of the two existing **TX circular buffer** into userspace. Each userspace-side of the nal and therefore each application would have its own virtual NIC. Short messages may be sent directly through the TX-CB without any intervention of the OS (**OS Bypass**). This saves one context switch from userspace to kernelspace but exposes the send buffer directly to the application. The application may currupt the send buffer and, because all control register can only me mapped into the addressspace as a whole block, it may also currupt the control register of other application that use a different VNIC. Longer messages that are to big for one packet but still to short to justify the delay of a VDMA transfer must be broken into several packets. This approach requires the same kind of **protocol** as the last one. The remaining packets may be sent over the second TX-CB of the same VNIC. That TX-CB may be in userspace or in kernelspace. A separate **thread** is needed to implement non-blocking calls. Special care has to be taken if more than one process are allowed to use the nic.

The receive buffer can be in userspace or kernelspace but since an interrupt will be raised (and all interrupts run in kernelspace) when a packet arrives, it's easier to keep that part of the driver in kernelspace as well.

That solution would be the most promising one. It's not feasible because of several reasons:

1. The current **Portals implementation doesn't support OS Bypass**. All function calls are forwarded into kernelspace because Portals

is a Linux module and modules live in kernelspace exclusively. To move Portals into userspace would be a project on it's own. Because more than one application may use Portals at the same time, something that coordinates all processes is still needed, therefore Portals must be split into a userspace library that is shared between all processes (and therefore requires context-switching).

2. Since some parts still need to be in kernelspace (interrupt handler), there will might be the need to share some structures between kernelspace and userspace. That's almost impossible and won't be a good concept because userspace code may compromise kernelcode.

4.1.5 Software & VHDL

A hardware solution of the Portals 3 library represents one of the final goals of the Portals implementation strategy.

The challenge on the software-side is that Portals doesn't offer any interface to the driver to put the match table to the nic.

The following changes in the firmware would furthermore require that the DMA limitations of the nic (addresses aligned to 32 bytes, length a multiple of 8 bytes) have to be removed first.

VHDL only

This solution would not justify the additional work needed to implement it for VDMA transfers because the page tables of the application space process has to be parsed to get the logical addresses for the VDMA transfer. The cpu load for this work is very low (1 interrupt for 1 MB of transferred data $\Rightarrow$ 800MB/s = 800 interrupts per second). On the other hand the code to setup and control the transfer is very complicated and therefore difficult to port on the nic.

A feasible project would be to port the **matching** of the memory descriptors and the **short message protocol** to the nic. The complexity of the firmware thus increases and requires more space on the FPGA. Since even dynamic memory structures are needed, memory is required and the memory that's already on the nic may not be big enough.

CPU

The easier hardware solution would be to bring a cpu on the nic. The FPGA is to 33% full, there might be enough space to load a cpu core into the FPGA.

Free CPU cores are available from `http://www.opencores.org/`. Nowadays there are even tools that can generate a customized cpu at the push of a button. The SRAM or the SDRAM might then be used as program and data memory. That solution offers faster development cycles and seems feasibler to me.

If someone is really interested in doing that then even a redesign of the nic might be considered. The FPGA could be replaced with one that has a cpu core integrated (e.g. XilinX virtex2pro with integrated PowerPC core).

4.2 Conclusion

An approach that changes anything in the firmware will go beyond the scope of a diploma thesis. Even if all the needed design tools were available, the success of the project may not be guaranteed because either the FPGA is too small to hold all the a cpu core or the RAM on the nic might be to small to hold the data structures.

Thus a pure software solution is the only reasonable approach.

The idea of the whole project is to approach the final goal step by step while learning the concepts of both systems. Therefore a basic version the driver will be written and after successfull tests, the driver will be upgraded until the final version is reached.

1. The first implementation will only use the CB buffer protocol of the VDMA handler to transfer messages. This step is very important in order to understand the concept of Portals and to learn how to write a nal for Portals.

2. The second version will use VDMA handler and send everything over VDMA.

3. Last but not least a VNIC will be reserved and all the short message traffic will go over the TX/RX-CB protocol.

Chapter 5

Implementation

5.1 Provisorly solution - VDMA handler CB

The VDMA handler provides two commands to send messages (< 504 bytes) over its circular buffer. Whenever Portals wants to send a message the message is attached to the `cb_send_request_queue` of the VDMA handler and the VDMA kernelthread is woken up. Afterwards the VDMA handler kernelthread starts processing its queues and sends the message. These functions are initially intended for debugging of the driver and will be removed in a later stage. Nevertheless I learned a lot about what it takes to write a driver for Portals. The most important lesson from this approach was, that the VDMA handler uses a *connection-based communication* that contrasts with the Portals philosophy. Every application that wants to use VDMA transfers has to send a **INIT_VDMA** command over the `ioctl` interface together with a structure that announces all the nodes and all the processes with which it wants to exchange messages.

5.2 Final solution

The final solution consists of the original Intercept Linux driver that was extended with additional functionality to support Portals. The driver is needed to load the firmware into the FPGAs on the nic and to initialize the nic. The `FCI` library (a library similar to Portals) just relies on the Linux driver and can therefore be omitted. Another important point was to leave as much code from the driver as it was in order to prevent new bugs and to merge updates easier from the original driver into the Portals driver at a later date. The hybrid module supports Portals calls and VDMA commands

at the same time.

5.2.1 Architecture

Portals & FCI architecture

Figure 5.1 compares the Myrinet driver against the Intercept FCI driver. All calls into Portals from the application are made into the userspace API side of Portals. Every function call is forwarded to the userspace side of the Myrinet NAL. The NAL uses the module **"p3mod"** to pass the barrier to kernelspace where the Portals module resides. The Portals module eventually jumps into the kernelspace side of the NAL to send and receive messages.

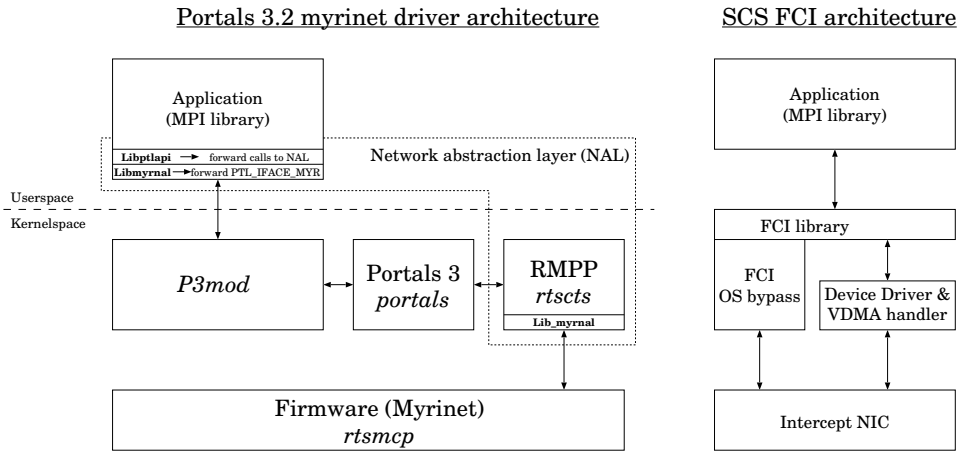

Figure 5.1: Comparing the myrinet driver and the FCI driver.

Portals Intercept driver architecture

The final architecture is shown in picture 5.2. The original driver part will be used to initialize the nic and to send *long messages* over the `VDMA handler`. It will be extended with functions to send *short messages* over another `VNIC`.

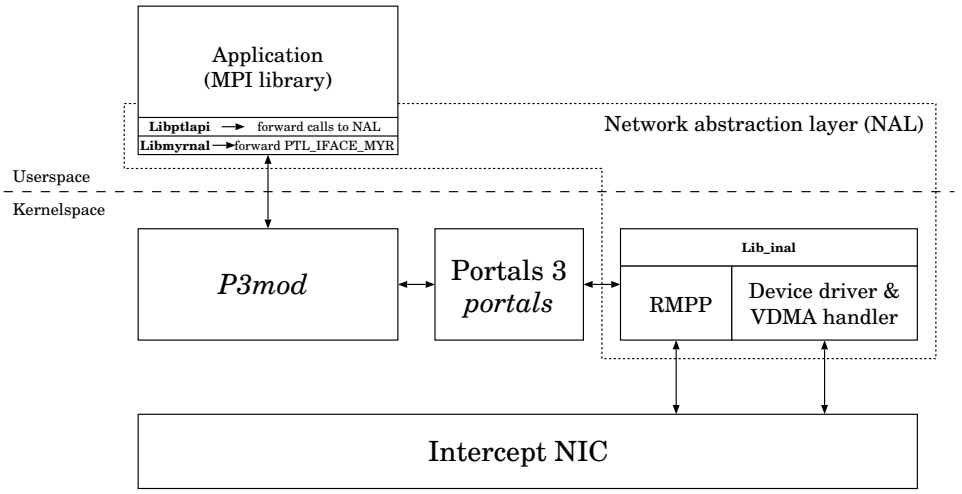

Figure 5.2: The Portals Intercept driver architecture.

5.2.2 VDMA

Removal of transfer limitations

The first change made to the VDMA handler was to introduce new control messages and to add intermediate buffer to get the ability to transfer a message of any size and any alignement (see 2.2.2). The *VDMA_GET* message was extended with the following fields:

```
typedef struct get_msg{
...
  Icpt_UInt64 key_remaining;
  Icpt_UInt64 first_block_length;
  Icpt_UInt64 first_block_addr;
  Icpt_UInt64 last_block_length;
  Icpt_UInt64 last_block_addr;
...
} get_msg_t;
```

Whenever a VDMA transfer request doesn't meet the hardware requirements the address and the length of the buffer are ajusted to meet them and a structure of the type `portals_wait_get_ext_reply_t` is allocated and attached to the queue `get_ext_reply_queue`. This struct holds informations that will be needed to store the remaining bytes and to identify the messages

that will contain them. The first/last_block fields contain the address and the number of bytes that are needed to be transferred. The GET message for the first slice will hold the additional information. The peer VDMA handler will analyze that message and send a `VDMA_PORTALS_GET_EXT_REPLY` message back, containing the remaining data.

Figure 5.3 illustrates how the transfer gets ajusted when the sender and the receiver have the same missalignement.

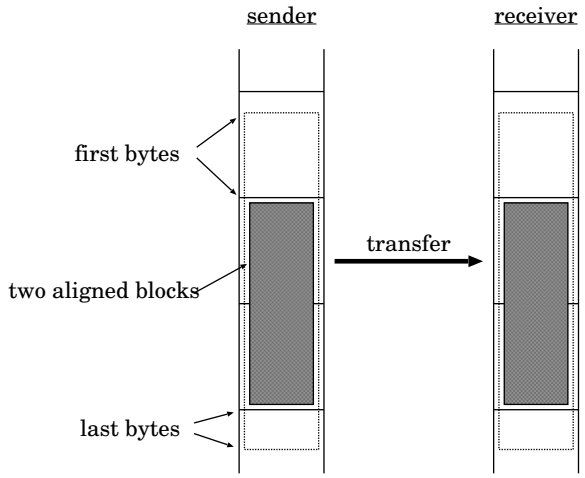

Figure 5.3: Unaligned blocks; sender and receiver have the same displacement.

If the sender and the receiver have different missalignements then a *memory copy at the sender's side* is needed. To accomplish this task some temporary buffers have been added to the VDMA handler that contain the temporary 1 MB blocks. This procedure is shown in figure 5.4.

Pseudo-connectionless VDMA transfers

As mentioned in 5.1 the VDMA handler uses a *connection-based protocol*. The proper solution to change this into a *connection-less protocol* would have been to reimplement this part wherever necessary. But this would have needed major changes that I didn't have the time for. Furthermore existing applications that are based on the old protocol would't run anymore. Therefore the old behavior was kept but a new control message **VDMA_PORTALS_UPD_CONN** has been added. If a process wants to send a message to a remote process that wasn't in mentioned when the

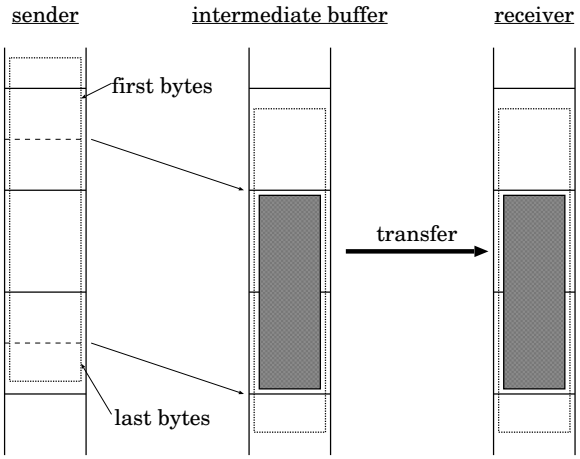

Figure 5.4: Unaligned blocks; sender and receiver have a different displacement.

INIT_VDMA was executed then it uses the new control message to have the remote VDMA handler updated its connection table and updates its own connection table.

A driver that'll be used for productive usage will therefore need some changes concerning this point.

No self connect

Every process participating in future VDMA transfers must be listened in the structure passed with the **INIT_VDMA** command. The VDMA handler expects that there's a switch beetween its nic and any other nic. It connects to *every* nic listed in that structure. Since the processes of the application that runs under that particular node are listened as well, it tries to connect to *itself* over the network.

If only two nics without any switch in between are present then the following happends at startup: The VDMA handler tries to connect to the first node in the list. Let's say that's itself. It sends the message intended for itself, but since there's *no software or hardware loopback*, the remote VDMA handler gets that message. It accepts the connection and sends an *initial credit update* back. Hereafter the first VDMA handler tries to connect to the second node in the list (see `vcomm_connect()`). Let's say that's the other nic. The other nic will refuse the connection because it thinks it's already

connected...

The VDMA handler code was changed to prevent that this scenario will happen.

Integration into Portals

Whenever a message has to be sent over the network, Portals passes a `ptl_hdr_t` structure to the driver and expects the driver to prepend that header to the message. The driver transferrs the Portals header in a new control message `VDMA_PORTALS_TRY_MATCH` over the CB protocol to the peer VDMA handler. The receiver passes that header to Portals and that lets the driver know how many bytes of the message it wants and where it should put the message (**matching receive**). Therefore only the receiver knows the details of the transfer. There are two kinds of VDMA transfers: *read and write*. This distinction is made in software, it simulates the write command, the hardware knows only one kind of VDMA transfers (see 2.2.2). The receiver causes the VDMA handler to read the message via **VDMA_GET** commands. After successfull completion of the transfer a `VDMA_PORTALS_MATCH_REPLY` message is sent back to the sender such that it can inform Portals that generates the appropriate `PTL_EVENT_..._END`. Figure 5.5 illustrates the dataflow for long messages.

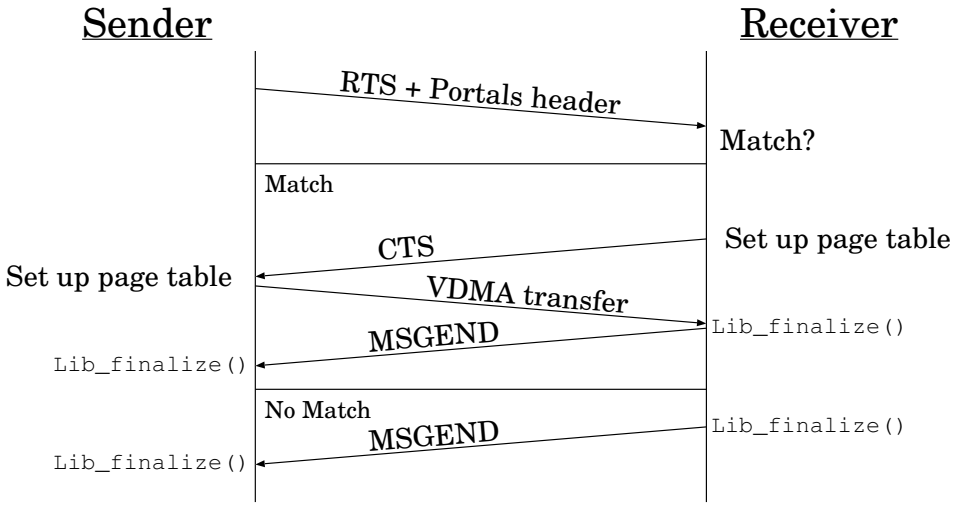

Figure 5.5: Dataflow for long messages over VDMA transfer.

The **Portals node id** is transposed 1:1 into the **Intercept Unicast**

Address (IUCA). The systemwide unique **physical rank** used by the VDMA handler is composed from the **Portals process id** and the **node id** number.

5.2.3 CB protocol

This section explains the implementation for the short message protocol.

VNIC

To be able to send messages over the CB protocol a **virtual NIC** is reserved upon startup of the driver. The receive buffer (RX-CB) and one send (TX-CB) buffer are assigned to the Portals Intercept driver kernelthread. Only the send function though which Portals jumps into the driver has access to the second send buffer.

Kernelthread

A kernelthread is launched for every application that uses the driver (more precisely: for every application process, if more than one processes are allowed to send messages). This kernelthread has the advantage that the processing of the arriving packets doesn't have to be done in interrupt. Even polling of the device is easily possible this way (see 6.1.2).

RMPP

Rolf Riesen [14] proposes a new protocol specially designed for high-performance computing. This protocol meets the demands of Portals and the Intercept nic:

- It's **connectionless**: the needed structures to support connections are allocated dynamically.
- It supports **ordered delivery** of messages: since a message is sent over all **four planes** the order of messages that the receiver sees may differ from the order in which they were sent. Even individual packets may get shuffled (see 3.3.3 and 2.1.3).
- A **flow control** to prevent flooding of the receiver is present as well.

The choosen implementation slightly differs from the original one that Rolf made for the Myrinet network. The protocol doesn't have to provide a

Type	Description
`RTS` <*data*>	Request to send a message
`CTS` *[n]*	Clear to send n data packets & hint how many bytes Portals wants from that message
`NEXT_CTS` *[n]*	Clear to send n data packets
`DATA` *[s]*	Data packet with sequence number s
`STOP_DATA`	Data packet, last of granted block
`LAST_DATA`	Data packet, last of message
`MSGEND`	Message successfully received, or no more data wanted, finish
`MSGERROR`	An error happend, abort transmission with error message

Table 5.1: Packet types for a simplified RMPP.

reliability layer because the hardware does all the necessary retransmission and therefore each sended packet will eventually arrive at its destination.

Table 5.1 gives an overview of the different packet types.

The message header consists of only the necessary fields:

```
typedef struct{
  pkthdr_type_t type;
  unsigned long long msgNum;
  size_t len;
} pkthdr_t;
```

The `msgNum` identifier is needed to guarantee the order of the messages. If a the message number is higher than expected, the whole message is copied to the queue `reordered_messages_queue` and `lib_parse()` (see 5.3.2 for explanation) will be called later, when the message before that has received.

A data packets contains only the number of the packet:

```
typedef struct{
  unsigned int number; /* Data packet [number]. (0-n) */
} pkthdr_data_t;
```

The receiver calculates the offset in the application via the packet number, therefore it doesn't matter in which order the data packets arrive.

The `CTS` message contains a clearance to send n packets and as a hint the total amount of bytes the memory descriptor can hold. It's the first CTS the receiver gets.

```
typedef struct{
```

```
  unsigned int n; /* Clear to send n data packets. */
  size_t mlen; /* Send requested length back, because this is more efficient.
} pkt_cts_t;
```

`NEXT_CTS` message is sent if the sender needs an other CTS. It consists of the number of granted packets only.

```
typedef struct{
  unsigned int n; /* Clear to send n data packets. */
} pkt_next_cts_t;
```

In order to send a message the sender transmitts a **ready to send (RTS)** together with the Portals header `ptl_hdr_t` and the first bytes of the message over the nic. This message is sent over the first TX-CB that is only used for that purpose (no lock latency). The receiver sends a **clear to send (CTS)** together with the number of packets its willing to accept back. This is done by the kernelthread and over the second TX-CB. In the current implementation this number is set to **16 packets**. In a further version this number should be proportional to the free space in the RX-CB and the number of nodes that want to sent messages to that node. Considering the fact that each packet holds up to ≈500 bytes of data, ≈8000 bytes may be sent with one CTS. The threshold when the driver switches from the short message protocol to the long message protocol is set to exactly that message length. The sender sents as many data packets as it was allowd to sent back. The last message is ot type `STOP_DATA`, if there are more packets (and it needs another CTS) or `LAST_DATA`, if the message was completely sent. When the message is completely arrived at the other side, a `MSGEND` message is sent back.

An overview of the whole architecture is given in figure 5.7, figure 5.6 illustrates the dataflow.

5.2.4 Portals P3mod

The module **P3mod** that provides user level access to kernel network abstraction layers (NAL) was *not SMP capable*. I modified the *cb_table*-functions and the register functions to use *spinlocks* and *rwlocks*. It should now be able to cope with several applications and several NALs at the same time.

5.2.5 Current implementation

The current driver version supports only one application.

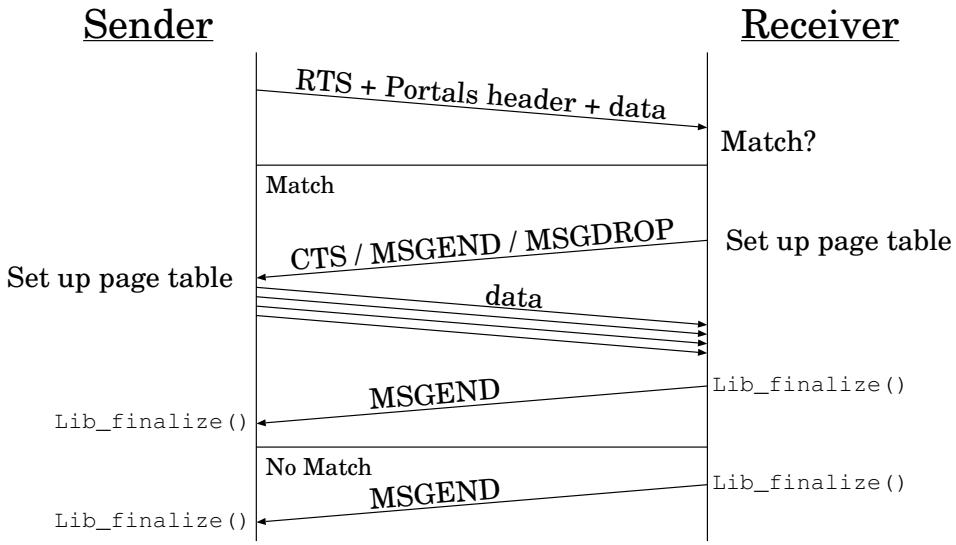

Figure 5.6: Dataflow for short messages over the CB protocol.

5.3 Code extracts

This section explains the interface function of the Portals Intercept driver. It may serve to understand the driver better and give an impression what's needed to write a NAL (there's no document of a newer data on that topic).

5.3.1 API-side NAL

The three main functionalities in *inal.c* are `open` & `close` and `forward`, which is used for all transactions out of the user mode.

First all the current and future pages are locked against swapping out. This is crucial for the correct operation of the driver. The `cb_write` function is supposed to copy data from userspace to kernelspace, however sometimes it's called in interrupt. If some pages are not valid it's not possible to page them in under a non-preemtible kernel (< 2.5), because page faulting is not allowed.

`P3DEV` addresses the **P3mod** module that forwards the calls from userspace to kernelspace. It's located under `/dev/portals3`. The `P3REGISTER` instruction is used to register the NAL with Portals.

Afterwards the initialization of the lib-side driver begins. The **IUCA** and the **allowed planes** are set through the *ioctl*-commands `SET_IUCA` and

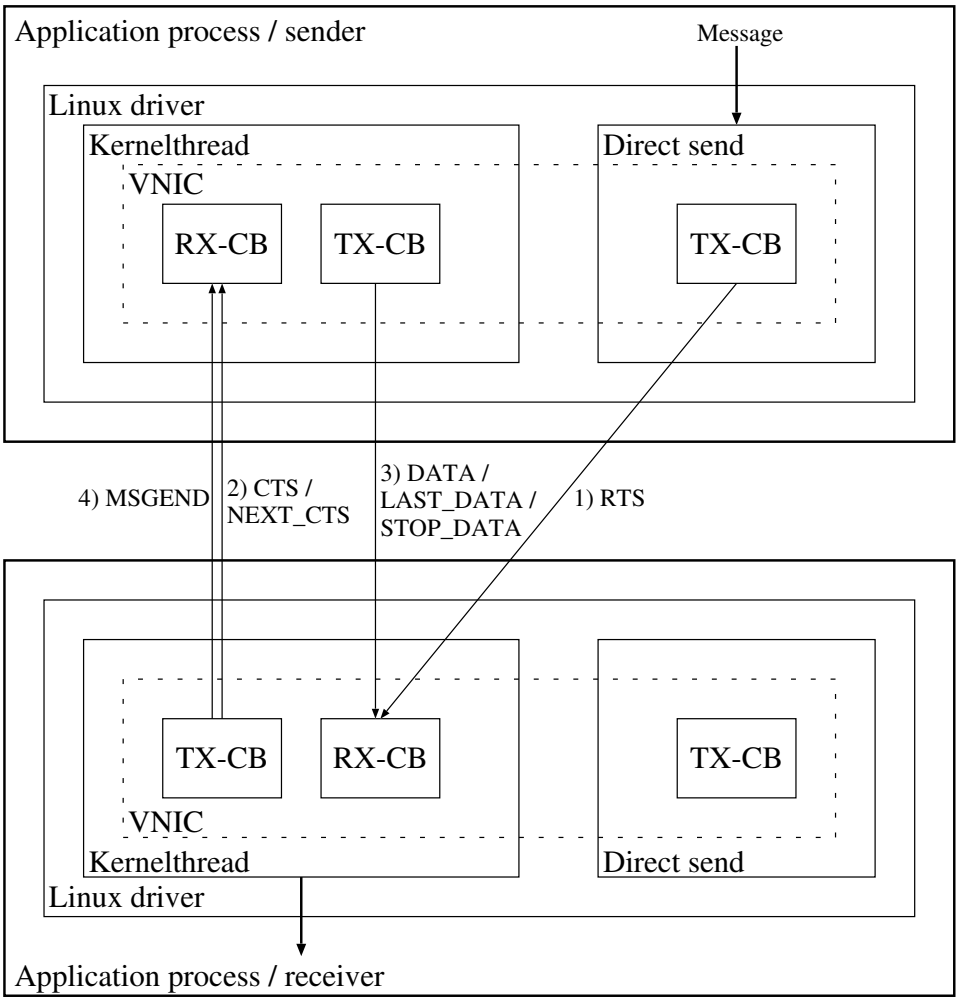

Figure 5.7: Architecture short messages over the CB protocol.

`SET_PLANE_MASK`.

Hence the process is registered with the VDMA handler by using the `INIT_VDMA_NOSELFCONNECT` *ioctl* command.

```
nal_t *PTL_IFACE_INAL(int interface_index,
            ptl_pid_t pid,
            ptl_ni_limits_t *desired,
            ptl_ni_limits_t *actual,
            int *rc)
{
  ...
  if( mlockall(MCL_CURRENT | MCL_FUTURE) != 0 )
  {
    ...
  }
  ...
  if ( (p3fd = open(P3DEV, O_RDWR)) < 0) {
    ...
  }
  ...
  rc_int = ioctl(p3fd, P3REGISTER, &mforward);
  ...
  inal_PtlGetId(__inal_ni_handle, &id);
  prank = PORTALS_ID_TO_ICPT_PRANK(id);
  ...
  if ((physfd = open(PATH "/" ICPT_FILENAME_EXT_PHYS_NIC, O_RDWR))<0){
    ...
  }

  /* Set IUCA for our NICS */
  command = SET_IUCA;
  iuca = id.nid;
  if(ioctl(physfd, command, &iuca)){
    ...
  }

  command = SET_PLANE_MASK;
  plane_mask = 1|2|8; /* plane 2(4) is fucked up! */
  if(ioctl(physfd, command, &plane_mask)){
    ...
```

```
  }

  if ((vdmafd = open(PATH "/" ICPT_FILENAME_EXT_VDMA, O_RDWR))<0){
    ...
  }

  /* Initialize the init_req structure */
  init_req.process_count = 1;
  init_req.processes    = malloc(init_req.process_count * sizeof(*init_req.proc
  init_req.processes[0]  = prank; /* only one process */
  init_req.nic_count    = 1;
  init_req.nics      = malloc(init_req.nic_count * sizeof(*init_req.nics));
  init_req.nics[0]      = 0;
  init_req.comm_table = malloc(init_req.nic_count * init_req.process_count *
                sizeof(*init_req.comm_table));
  init_req.comm_table[0]  = id.nid; /* IUCA = Portals nid */
  init_req.caller_v_rank  = 0; /* index in table above */

  if(ioctl(vdmafd, INIT_VDMA_NOSELFCONNECT, &init_req)){
    ...
  }
  ...
  return &inal;
}
```

Data is transfered across the protection domain by *forward*, which after packing all arguments in *mforward* calls *ioctl* of **P3mod** which handles the message to the Portals library.

```
static int forward(nal_t * nal,
           int id,
           void *args,
           size_t args_len,
           void *ret,
           size_t ret_len)
{
  ssize_t rc;
  int p3fd;
  inal_forward_t mforward;
  ...
  /* Pack arguments */
```

```
    mforward.args = args;
    mforward.args_len = args_len;
    mforward.ret = ret;
    mforward.ret_len = ret_len;
    mforward.p3cmd = id;
    rc = ioctl(p3fd, P3CMD, (unsigned long) (&mforward));
    ...
}
```

Before the device can be closed, the Portals library has to be closed first. Both instructions are called in the *shutdown* function.

```
static int shutdown(nal_t *nal, int interface)
{
    ...
    close(p3fd);
    ...
    close(vdmafd);
    ...
}
```

5.3.2 LIB-side NAL

This section explains briefly the lib side of the NAL. Only the direct interface to Portals is described, the actual send and receive functions that access the hardware directly won't be discussed here.

Several debug macros are used in the driver, each has a debug level indicator as the first parameter, it's inserted or omitted in the module, depending on the actual debug level used to compile the code:

- The `ICPT_FCT_ENTRY` and `ICPT_FCT_EXIT` / `ICPT_FCT_RETURN` macros print a log output depending on the loglevel, whenever a the execution of a function begins or ends.
- `ICPT_LOG` prints a debug message, depending on the loglevel.
- `ICPT_ERROR` prints an error message.

After a Linux driver is loaded the function marked with `__init` is called. The function `inal_module_init()` is actually called from the `__init` function of the Intercept driver because everything is compiled into one module. The *init* function registers the Intercept NAL with Portals and makes it

available for used. When the Intercept module is started via the *insmod* command, an optional parameter `IcptDrv_ptls_nid` sets the node number of the current machine. After the userspace api has sent the `P3REGISTER` *ioctl* command (see 5.3.1) to *P3mod* the `inal_destroy_nal` callbacks is called to startup the kernelspace part of the NAL. When the p3 *filedescriptor* is released during shutdown of the userspace NAL (e.g. `close()`) the `inal_destroy_nal` callback is called, that unregisters the driver.

```
int __init inal_module_init(void)
{
  ICPT_FCT_ENTRY(LOG_MED, "");

  /* Register with the P3 module */
  ptls_devid = p3register_dev("Intercept␣Portals␣NAL",
                IcptDrv_ptls_nid,
                inal_create_nal,
                inal_destroy_nal);
                ...
  ICPT_FCT_RETURN(0, LOG_MED, "");
}

void __exit inal_module_exit(void)
{
  ICPT_FCT_ENTRY(LOG_MED, "");

  /* Tell the Portals 3.0 module, that we're out */
  p3unregister_dev(ptls_devid);
  ...
  ICPT_FCT_EXIT(LOG_MED, "");
}
```

The most important parts of the `inal_create_nal` subroutine is that an object of the type `nal_cb_t` is allocated that holds a *jump table* that is used by Portals to *communicate* with the driver.

```
nal_cb_t *inal_create_nal(ptl_ni_limits_t *desired,
              ptl_ni_limits_t *actual)
{
  int rc;
  nal_cb_t *nal;

  ICPT_FCT_ENTRY(LOG_MED, "");
```

```
  /* We have to allocate the nal structure dynamically, because this
  structure contains informations about the userspace process.
  Otherwise only one process at a time would be able to use this
  nal. */

  /* allocate memory for nal */
  nal = (nal_cb_t *) kmalloc(sizeof(nal_cb_t), GFP_KERNEL);
  ...

  /* allocate memory for nal private data */
  nal->nal_data = (void *) kmalloc(sizeof(inal_data_t), GFP_KERNEL);
  ...

  /* fill in nal function pointers */
  nal->cb_send      = inal_send;
  nal->cb_recv      = inal_recv;
  nal->cb_write     = inal_write;
  nal->cb_malloc    = inal_malloc;
  nal->cb_free      = inal_free;
  nal->cb_invalidate = inal_invalidate;
  nal->cb_validate    = inal_validate;
  nal->cb_printf    = inal_printf;
  nal->cb_cli    = inal_cli;
  nal->cb_sti    = inal_sti;
  nal->cb_dist      = NULL;
  ...

  /* initialize portals data structures */
  rc = lib_init(nal, IcptDrv_ptls_nid, 0, current->uid, actual);
  ...

  ICPT_FCT_RETURN(nal, LOG_MED, "");
}
```

The `inal_send` function checks if the userspace address is valid and calls the actual send function `p3_send`.

```
static int inal_send(nal_cb_t *nal,
           void *private,
           lib_msg_t *cookie,
```

```
            ptl_hdr_t *hdr,
            ptl_nid_t dnid,
            ptl_pid_t pid_in,
            unsigned int niov,
            lib_md_iov_t *iov,
            size_t len)
{
  user_ptr data;

  ICPT_FCT_ENTRY(LOG_MED, "");

  if (niov == 1) {
    if (len != iov[0].mdiov_len)
      ICPT_FCT_RETURN(PTL_INV_MD, LOG_MED, "");

    data = iov[0].mdiov_base;

    if (access_ok(VERIFY_READ, data, len) == 0) {
      ICPT_ERROR("buf %p, len %d no read access!\n",
              data, (int) len);
      ICPT_FCT_RETURN(PTL_INV_MD, LOG_MED, "");
    }
  }
  else if (niov == 0) {
    data = NULL;
    len = 0;
  }
  else {
    ICPT_FCT_RETURN(PTL_INV_MD, LOG_MED, "");
  }

  ICPT_FCT_RETURN(p3_send(nal, private, data, len, dnid, hdr, cookie), LOG_MED
}
```

This function decides wheter a message is sent over the *short message protocol* or the *long message protocol*.

```
static int p3_send(nal_cb_t *nal,
            void *private,
            void *buf,
            size_t len,
```

```
            int dst_nid,
            ptl_hdr_t *hdr,
            lib_msg_t *cookie)
{
  ICPT_FCT_ENTRY(LOG_MED, "");
  ...

  /* Threshold for VDMA one CTS and maximum data bytes. */
  if( len <= RTSCTS_DATA_PACKET_LEN + MAX_DATA_PKTS * RTSCTS_NEXT_DATA_
  {
    ICPT_FCT_RETURN(send_short_msg(nal, private, buf, len, dst_nid, hdr
            LOG_MED, "");
  }
  else
  {
    ICPT_FCT_RETURN(send_long_msg(nal, private, buf, len, dst_nid, hdr
            LOG_MED, "");
  }
}
```

The following function sends a message over the short message protocol. First an object is allocated and passed to the kernelthread afterwards such that it'll be able to recognize the incoming **CTS**. The sending of a **RTS** may be blocked temporarly because the driver is waiting for a **CTS** that confirms implicitly that the message number is resetted. Hence the **RTS** is sent or postponed if there's not enough space in the **TX-CB**. If anytime an error occurs then `lib_finalize()` is called to notify Portals upon unsuccessfull completion of the message transfer.

```
int send_short_msg(nal_cb_t *nal,
            void *private,
            void *buf,
            size_t len,
            int dst_nid,
            ptl_hdr_t *hdr,
            lib_msg_t *cookie)
{
  int retval;
  rts_sent_t *rts_sent;
  icpt_IUCA_t IUCA = (icpt_IUCA_t)dst_nid;
```

```
ICPT_FCT_ENTRY(LOG_MED, "");

if((rts_sent = (rts_sent_t*)
  kmalloc(sizeof(rts_sent_t), GFP_KERNEL))
   == NULL)
{
  ...
  lib_finalize(nal, private, cookie, -ENOMEM);
  ...
  ICPT_FCT_RETURN(-ENOMEM, LOG_MED, "");
}

...

/* Sending of RTS messages may temporarly be blocked because we are waiting
   the remote machine receives the BOOTMSG to initialize the message counter
if( !send_msg_allowed[IUCA] )
{
  rtscts_postponed_msg_t *postponed_msg;

  if((postponed_msg = (rtscts_postponed_msg_t*)
    kmalloc(sizeof(rtscts_postponed_msg_t), GFP_KERNEL))
     == NULL)
  {
    ICPT_ERROR("Could␣not␣allocate␣memory␣for␣postponing␣message␣request!\n"

    kfree(rts_sent);
    lib_finalize(nal, private, cookie, -ENOMEM);

    ICPT_FCT_RETURN(-ENOMEM, LOG_MED, "");
  }

  ICPT_LOG(LOG_HIGH, LOG_AREA_PORTALS,
        "Cannot␣send␣RTS␣because␣I'm␣still␣waiting␣for␣the␣corresponding␣CTS."
        "Message␣has␣been␣postponed.\n"
        );

  ...
  send_msgNum[IUCA]++;
```

```
  ICPT_FCT_RETURN(0, LOG_MED, "");
}

/* Send packet. */
if( (retval = rtscts_send_rts(&txcb_process, rts_sent, len, hdr))
  < 0 )
{
  if(retval != -ENOBUFS)
  {
    ICPT_ERROR("Could not send message to peer.\n"
           "(rtscts_send() returned %d).\n",
           retval);

    kfree(rts_sent);

    ICPT_FCT_RETURN(retval, LOG_MED, "");
  }
  else
  {
    /* If there is no space in the TX CB,
    the message will be sent later. */
    rtscts_postponed_msg_t *postponed_msg;

    ICPT_LOG(LOG_HIGH, LOG_AREA_PORTALS,
          "Cannot send RTS because I don't have enough free space\n"
          "in the TX circular buffer.\n"
          "Message has been postponed.\n"
          );

    if((postponed_msg = (rtscts_postponed_msg_t*)
      kmalloc(sizeof(rtscts_postponed_msg_t), GFP_KERNEL))
       == NULL)
    {
      ICPT_ERROR("Could not allocate memory for postponing message re

      kfree(rts_sent);
      lib_finalize(nal, private, cookie, -ENOMEM);

      ICPT_FCT_RETURN(-ENOMEM, LOG_MED, "");
    }
```

```
        ...
        send_msgNum[IUCA]++;

        ICPT_FCT_RETURN(0, LOG_MED, "");
      }
    }
    ICPT_FCT_RETURN(0, LOG_MED, "");
}
```

The `send_long_msg()` function passes a long message to the VDMA handler. It checks first if a connection to the remote node already exists and establishes one if that isn't the case. `vcomm_connect()` is used to connect to the remote node, afterwards a `send_portals_upd_conn()` sends a request to update the connection table of the remote process. When this is done `send_portals_try_match_msg()` transmits the Portals header to the remote driver that'll get the message with a *VDMA read* transfer.

```
int send_long_msg(nal_cb_t *nal,
            void *private,
            void *buf,
            size_t len,
            int dst_nid,
            ptl_hdr_t *hdr,
            lib_msg_t *cookie)
{
    ...
    ICPT_FCT_ENTRY(LOG_MED, "");
    ...

    local_p_rank = PORTALS_HEADER_SRC_TO_ICPT_PRANK(*hdr);
    remote_p_rank = PORTALS_HEADER_DEST_TO_ICPT_PRANK(*hdr);

    local_clt = get_clt_by_prank(local_p_rank);
    ...

    while(down_interruptible(&local_clt->connection_sem))
    {
        ...
    }
```

```
/* Check if we can find the remote prank in the prank table */
/* to see communication with the remote is set up */
for(i=0, bConnected=FALSE; i<local_clt->process_count; i++)
{
  if(local_clt->physical_ranks[i] == remote_p_rank)
  {
    bConnected = TRUE;
    break;
  }
}

if(bConnected)
{
  ...
}
else
{
  /* connect... */
  ...

  /* Now send the new information to the peer VDMA handler. */
  /* We have to send the IUCA of every single NIC. */
  /* Currently only one NIC is supported. */

  retval = send_portals_upd_conn(local_clt, remote_p_rank);
  if( retval == 0 )
  {
    ...
  }
  else
  {
    ICPT_ERROR("Could not send PORTALS UPDATE CONNECTION\n"
           "message to remote VDMA handler!\n");
    lib_finalize(nal, private, cookie, retval);

    ICPT_FCT_RETURN(retval, LOG_MED, "");
  }
}

if(local_clt->task != current)
```

```
  {
    ICPT_ERROR("Current process (PID: %d) has task structure at %p, while"
            " the process registered as VDMA client on this file"
            " handle has task structure at %p\n",
            current->pid, current, local_clt ? local_clt->task : NULL);
    ICPT_FCT_RETURN(-EACCES, LOG_MED, "");
  }

  /* Connection to peer VDMA handler established */
  /* and comm_table etc contains remote processes */

  /* Send a Portals Try Match message */
  /* after the VDMA transfer is processed or */
  /* if the peer VMDA handler sends a non-match */
  /* message back, we use to queue to call lib_finalize */
  /* to notify Portals upon completion */
  retval = send_portals_try_match_msg(hdr, nal, private,
                    cookie, local_clt, remote_p_rank, buf);
  if( retval != 0 )
  {
    ICPT_ERROR("Could not send Portals Try Match message!\n"
            " - vcomm_send() returned %d\n", retval);

    lib_finalize(nal, private, cookie, retval);

    ICPT_FCT_RETURN(retval, LOG_MED, "");
  }

  /* the vdma handler will free the request descriptor, once it has */
  /* handled the request */

  ICPT_FCT_RETURN(INAL_OK, LOG_MED, "");
}
```

The following lines give a short overview of the kernelthread implementation. First it looks in the RX-CB if any new messages have arrived. If a **RTS** arrives then the `msgNum` field is checked to see if a message with this number is expected. If the number is too high then the RTS is copied into a reorder queue. If it's exactly the expected number, then `lib_parse()` is called that sends the `PTL_EVENT_..._START` event to the application. It

furthermore analyzes the passed `ptl_hdr_t` object and looks for a matching memory descriptor. Regardless if it finds one or not it'll jump back into the driver through the function `recv_short_msg()`.

If a data packet arrived then it's copied into the application. If it's the last packet then `lib_finalize()` is called, that generates the `PTL_EVENT_..._END` event.

It's worth mentioning that the common `copy_to/from_user()` functions, that copy memory between kernelspace and userspace, can't be used at this place. These functions are indended for *ioctl* calls, when the task context is still the userspace process. Since we have our own thread context (kernelthread) an other approach must be gone. The function `memcpytouser()` takes as an additional argument pointer `struct task_struct* task` that specifies the application process. That function is the reason, why all pages have to be locked before any transfer can happen (see 5.3.1).

If the `RTSCTS_POLL` variable is defined then the kernelthread starts polling the nic. That might result in shorter respond time (see 6.1.2).

```
for(;;)
{
#ifndef RTSCTS_POLL
  set_current_state(TASK_INTERRUPTIBLE);
#endif
  /*##############################*/
  /* Perform the Portals work */
  /*##############################*/

  /*--------------------------------------------------------------*/
  /* Process the next packet from the RX CB (if available)    */
  /*--------------------------------------------------------------*/
  while(rtscts_read_next(recv_msg_buffer, &recv_IUCA) >= 0)
  {
    pkthdr_t *hdr = (pkthdr_t*)recv_msg_buffer;
    ...

    switch(hdr->type)
    {
    case RTS: {
      ptl_hdr_t *phdr = (ptl_hdr_t *)(recv_msg_buffer + sizeof(pkthdr_t
      nal_cb_t *nal;
      struct task_struct *task;
```

```
inal_private_t private;
struct list_head *tmp;
reordered_rts_t *reordered_rts;

/* Only if we received a RTS check the message number. */
if( hdr->msgNum == BOOTMSGNUM )
{
  ...
  lastMsgNum[recv_IUCA] = hdr->msgNum;
}
else
{
  /* Did we expect this message number? */

  if( hdr->msgNum != lastMsgNum[recv_IUCA] + 1)
  {
    reordered_rts_t *reordered_rts;

    /* No, don't call lib_parse(), since we have to guarantee
       in-order-beginning. */
    ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
          "Received message number: %Lu\n"
          "Expected message number: %Lu\n"
          "(Message will be processed later.)\n",
          hdr->msgNum,
          lastMsgNum[recv_IUCA]
          );

    if( hdr->msgNum < lastMsgNum[recv_IUCA] )
    {
      /* Message number to low (expecting
         ascending numbers!) */
         ...
    }
    else
    {
      /* Add RTS to the reorder queue. */
      ...
    }
    goto reordered_msg;
```

```
      }
      else
      {
        lastMsgNum[recv_IUCA] = hdr->msgNum;
      }
    }
    nal = get_cb(cb_tbl, phdr->dest_pid, &task);
    if (nal == NULL)
    {
      ICPT_ERROR("nid %Lx pid %x, has no NAL CB\n",
              phdr->dest_nid,
              phdr->dest_pid);
              ...
    }
    else
    {
      ...
      lib_parse(nal, phdr, &private);
    }
    /* Process reordered messages... */
    ...

  }
  reordered_msg:
    break;
    ...
    break;

  case CTS:
  ...
    break;

  case NEXT_CTS:
    break;

  case DATA:
  case STOP_DATA:
  case LAST_DATA:
    if(!list_empty(&(this_portals_handler->cts_sent_messages_queue)))
    {
```

```
...
    memcpytouser(cts_sent->task,
            dst,
            recv_msg_buffer +
            sizeof(pkthdr_t) + sizeof(pkthdr_data_t),
            len);
            ...

    /* Got full data packet? (LAST_DATA)
       -> send MSGEND back. */
    if( cts_sent->recvlen >= cts_sent->mlen )
    {
      ...

      /* Use the same msg number that we received!
         (Remote thread identifies msg by this) */
      if( (retval = rtscts_send_msgend(&txcb_kthread,
                        recv_IUCA,
                        hdr->msgNum))
        < 0)
      {
        if(retval != -ENOBUFS)
        {
          ICPT_ERROR("Could not send message to peer.\n"
                "(rtscts_send() returned %d).\n",
                retval);

          lib_finalize(cts_sent->nal,
                cts_sent->private,
                cts_sent->cookie,
                retval);
          list_del(&cts_sent->list_link);
          kfree(cts_sent);
        }
        else
        {
          /* If there is no space in the TX CB,
             the message will be sent later. */
             ...
        }
```

```
    }
    else
    {
      lib_finalize(cts_sent->nal,
              cts_sent->private,
              cts_sent->cookie,
              0);
      list_del(&cts_sent->list_link);
      kfree(cts_sent);
    }
  }
  else if( hdr->type == STOP_DATA )
  {
    /* We need to send the next CTS. */
    unsigned int retval;

    ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
          "Sending NEXT_CTS ...\n");

    /* Use the same msg number that we received!
       (Remote thread identifies msg by this) */
    if( (retval = rtscts_send_next_cts(&txcb_kthread, cts_ser
      < 0 )
    {
      if(retval != -ENOBUFS) call
      {
        ICPT_ERROR("Could not send message to peer.\n"
              "(rtscts_send() returned %d).\n",
              retval);

        lib_finalize(cts_sent->nal,
              cts_sent->private,
              cts_sent->cookie,
              retval);
        list_del(&cts_sent->list_link);
        kfree(cts_sent);
      }
      else
      {
        /* If there is no space in the TX CB,
```

```
                        the message will be sent later. */
                        ...
                    }
                }
                else
                {
                    ...
                }
            }
            goto data_found;
        }
      }
    }
    ICPT_ERROR("Received unexpected DATA message!\n"
            "Sender IUCA: %d\n"
            "msgNum: %Lu\n",
            recv_IUCA,
            hdr->msgNum
            );
  data_found:
    break;

  case MSGEND:
    break;

  default:
    ICPT_ERROR("Bullshit! Received unidentifiable message type!\n");
    break;
  }
#ifndef RTSCTS_POLL
  /* Notice that we did some work... */
  set_current_state(TASK_RUNNING);
#endif
 }

 /* Send packets if we have any to send. */

 /* Send postponed packets first. */
 {
  ...
```

```
  }

  /* Send data packets if we have any to send. */
  {
    ...
  }

  ICPT_LOG(LOG_LOW, LOG_AREA_PORTALS,
        "Kernel thread %s - one iteration done.\n",
        current->comm);

#ifdef RTSCTS_POLL
  /* Try to not monopolize the PCI-X bus. */
  udelay(1);
#else
  ...
  /* Although we're busy, it isn't nice to monopolize the CPU by
     taking advantage of being a kernel thread */
  if(current->need_resched)
  {
    /* if we yield the CPU, we must make sure we'll be re-scheduled
       later on.
    */
    ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
          "Kernel thread %s - rescheduling.\n",
          current->comm);

    set_current_state(TASK_RUNNING);

    schedule();
  }
#endif
  /* Memory barrier to see the correct flag value */
  mb();
  ...
}
```

The callback function `recv_short_msg()` is a called by Portals, after the driver calls `lib_parse()`. The argument `rlen` specifies the length of the incoming message. `mlen` is the length of the memory descriptor and

indicates how many bytes the driver should write into the buffer pointed by `data`.

```
int recv_short_msg(nal_cb_t *nal,
           void *private,
           void *data,
           size_t mlen,
           size_t rlen,
           lib_msg_t *cookie)
{
  int retval = 0;
  inal_short_msg_private_t *priv =
       &((inal_private_t*)private)->private.short_msg;
  icpt_IUCA_t IUCA = priv->IUCA;
  __u8 *msg_buffer = priv->msg_buffer;
  pkthdr_t *hdr = (pkthdr_t*)msg_buffer;
  /* Length of data in the first packet. */
  size_t datalen = hdr->len - (sizeof(pkthdr_t) + sizeof(ptl_hdr_t));

  ICPT_FCT_ENTRY(LOG_MED, "");

  /* Copy data from the first packet. */
    if( memcpytouser(priv->task,
             data,
             msg_buffer + sizeof(pkthdr_t) + sizeof(ptl_hdr_t),
             min(mlen, datalen)) != 0 )
    {
      ICPT_ERROR("Could not copy incoming message\n"
                 "to userspaceprocess!\n");

      if( (retval =
           rtscts_send_msgend(&txcb_kthread, IUCA, hdr->msgNum))
        < 0)
      {
        if(retval != -ENOBUFS)
        {
          ICPT_ERROR("Could not send message to peer.\n"
                 "(rtscts_send() returned %d).\n",
                 retval);
```

```
          lib_finalize(nal, private, cookie, retval);

          ICPT_FCT_RETURN(retval, LOG_MED, "");
        }
        else
        {
          /* If there is no space in the TX CB,
          the message will be sent later. */
          ...
        }
      }
    }

  ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
       "Copied %u bytes to userspace.\n",
       min(mlen, datalen)
       );

  if( mlen <= datalen )
  {
    /* Send MSGEND back. */
    ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
         "Got last message... sending MSGEND back.\n");

    if( (retval =
         rtscts_send_msgend(&txcb_kthread, IUCA, hdr->msgNum))
      < 0)
    {
      if(retval != -ENOBUFS)
      {
        ICPT_ERROR("Could not send message to peer.\n"
                "(rtscts_send() returned %d).\n",
                retval);

        lib_finalize(nal, private, cookie, retval);

        ICPT_FCT_RETURN(retval, LOG_MED, "");
      }
      else
      {
```

```
        /* If there is no space in the TX CB,
        the message will be sent later. */
        ...
      }
    }
    else
    {
      /* No errors. */
      lib_finalize(nal, private, cookie, 0);
    }
  }
  else
  {
    /* Send CTS back. */
    cts_sent_t *cts_sent;

    ICPT_LOG(LOG_MED, LOG_AREA_PORTALS,
          "Sending CTS back...\n");

    if((cts_sent = (cts_sent_t*)
      kmalloc(sizeof(cts_sent_t), GFP_KERNEL))
       == NULL)
    {
      ICPT_ERROR("Could not allocate memory\n"
                 "for CTS sent message request!\n");

      lib_finalize(nal, private, cookie, -ENOMEM);

      ICPT_FCT_RETURN(-ENOMEM, LOG_LOW, "");
    }
    ...

    /* Send packet. */
    if( (retval = rtscts_send_cts(&txcb_kthread, cts_sent))
      < 0)
    {
      if(retval != -ENOBUFS)
      {
        ICPT_ERROR("Could not send message to peer.\n"
                "(rtscts_send() returned %d).\n",
```

```
                retval);

        lib_finalize(nal, private, cookie, retval);

        ICPT_FCT_RETURN(retval, LOG_MED, "");
      }
      else
      {
        /* If there is no space in the TX CB,
        the message will be sent later. */
      }
    }
    else
    {
      ...
    }
  }
  ICPT_FCT_RETURN(retval, LOG_MED, "");
}
```

Altogether the Portals driver consists more than 5000 lines of code and is almost twice as big as the original Intercept driver.

Chapter 6

Testing

6.1 First Development System

To compile and test the driver an environment for testing was set up. It consists of two rack mounted nodes, each with the following configuration:

- Motherboard: **Supermicro P4DPR**, **Intel E7500** chipset.
- **Dual Xeon P4 2.2 Ghz** CPUs with hyperthreading enabled.
- **512 MB DDR-266 Ram**.
- **One Intercept NIC**, in PCI-X 66 Mhz mode.
- **Redhat 8.0**, Redhat Kernel **2.4.18-14** with SMP enabled, gcc 3.2.

6.1.1 Loopback tests

The characteristic that was tested first is the **bandwith**. The program `VDMATransfer` of Martin Maletinsky (SCS) was run with an unmodified version of the Intercept driver. The transfer of a message of 100 MB size, revealed a bandwith of 47.2 MB/s. A similar program under Portals reached 49.6 MB/s. This showed that the Portals implementation doesn't have to be slower, even if there's a Portals driver between the Intercept driver and the application.

6.1.2 Ping-pong tests

The ping-pong test consists of two programs: a client program that sends a message to the server program and measures the time and a server program

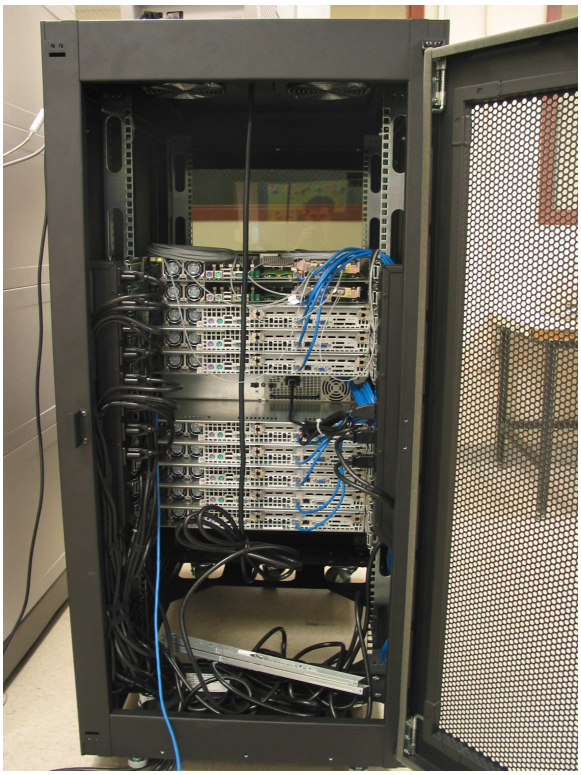

Figure 6.1: 1^{th} development system at the Scalable Systems Lab, UNM.

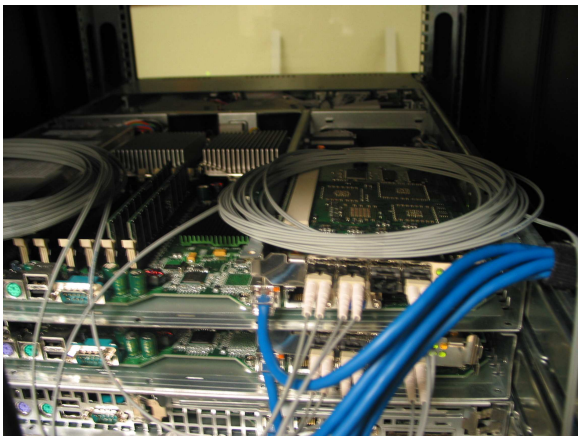

Figure 6.2: Running NICs in the development system.

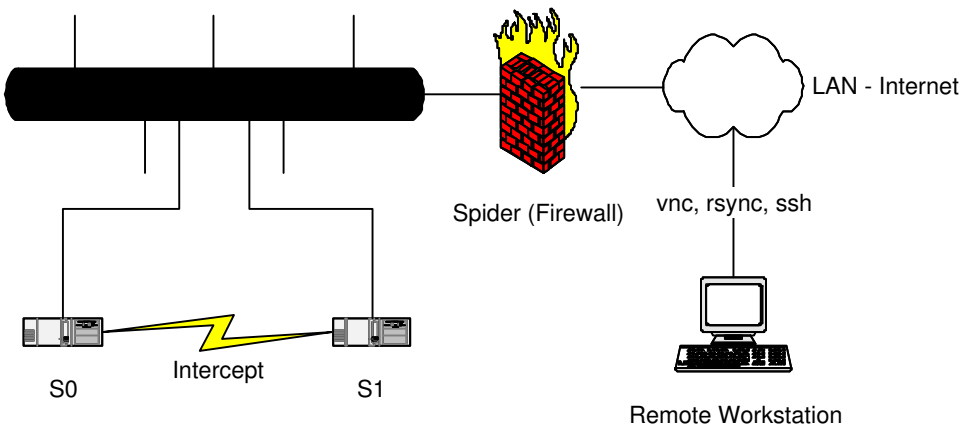

Figure 6.3: Development System network topology.

that returns the message. *30* ping message were send back and forth in one test. The following 5 series show that the average **latency** is $\approx 45\mu s$.

min $[\mu s]$	max $[\mu s]$	average $[\mu s]$
40	91	48
39	93	45
33	102	40
40	101	48
31	107	42

An interesting approach is to poll the nic instead of using interrupts. At first sight this might be a waste to spend a whole CPU with polling a device. However latest CPUs like the Intel Pentium 4 offer *two instruction paths* instead of only one. From outside it looks like two independent CPUs but from inside they share the *same datapath*. This feature called "hyperthreading" may increase the throughput of a single CPU significantly when two processes run that don't occupy the same ressources on the CPU. For instance a program that renders an image and one that compiles source code might run in parallel. Therefore it doesn't make sense to run one computational process twice, because the two processes would compete for the calculation units on the CPU. But it makes sense to use one of the "virtual" CPUs for computation and one for communication exculsively.

The next five series show that polling reduces significantly the latency to $\approx 30\mu s$.:

min [μs]	max [μs]	average [μs]
23	145	32
22	90	30
23	87	29
22	91	27
23	91	27

Finally a message of 100 MB size was transferred from one node to another to measure the bandwith. It's $\approx$ 56.8 MB/s with VDMA transfers. If the message is not perfectly aligned then the bandwith drops to 52.4 MB/s. If the driver is force to transfer everything over the CB protocol then the bandwith is even lower: 47.8 MB/s.

The motherboards refused constantly to operate faster than 66 Mhz. The fact that even the testprogram `VDMATransfer` for the unmodified driver from SCS was slow reached the conclusion the there must be a hardware problem and that the Portals part is not the bottleneck.

These numbers are so ricidulous that I asked the people from SCS to run `VDMATransfer` on their systems. They got 357 MB/s over PCI-X 133 MHz on their Supermicro P4DL6 boards (textbf(ServerWorks GC-LE chipset).

Neither a `BIOS` upgrade nor a completely disabled logging of the driver nor a firmware upgrade of the NICs cured them.

6.2 Second Development System

Since the first system didn't produce the expected results, I looked for another system. The problem was to find another PCI-X system to reach the maximum performance. I eventually could organize two Itanium machines (HP rx2600) that were configured as follows:

- **HP zx-1** chipset
- **Dual Itanium-2 900 Mhz**, 1.5 MB L3 cache.
- **8 GB Ram**
- **One Intercept NIC**, in 133 Mhz mode, plugged in the slow 0.5 GB/s slot because it did not fit in the fast 1 GB/s slot.
- **Redhat Advanced Server 2.1**, Redhat Kernel 2.4.18-e.12 with original configuration from SCS, gcc 2.96

The bandwith results with the original SCS drivers so far are:

Figure 6.4: 2^{nd} development system at the Scalable Systems Lab, UNM.

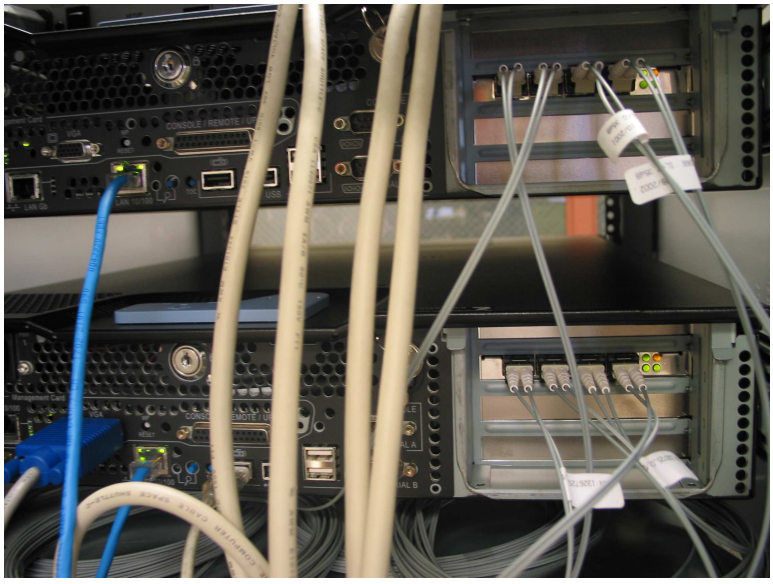

Figure 6.5: Running NICs in the development system.

210 MB/s in loopback mode, with regardless of what planes are activated.

240 MB/s two NICs and only one plane.

400 MB/s two nics with two or more planes.

6.3 MPI

The final goal of the project was to run some MPI programs over Portals. Further research discovered that there's no MPI library for Sourceforge Portals 3.2. The only MPI library available is for Cplant Portals 3.0. Since a compilation of the MPI library under the new Portals was not possible, migrating the driver to Cplant Portals was considered. Examination of the Portals 3.0 code revealed that major changes in the driver itself would have been needed to get it run under the older version. In the end it was just a matter of time, if this seamed feasible or not. Because a port of the MPI library to the newer Portals version is under way, that part of the project was dropped.

Chapter 7

Conclusion

7.1 Results

The presented driver shows one possibility among several how a Portals driver may look like. A comparison between a program that accesses the NIC on a very low level (`VDMATransfer`) and a Portals application pointed out that the additional overhead imposed by Portals is neglectable.

7.2 Prospects

The most important drawback of the current implementation is that only one userspace process is currently supported. If, at a later date, the NIC will be redesigned then it is very likely that FPGAs with integrated CPU cored will be used. This will open the field for a bunch of new possibilities for a future implementation.

7.3 Portals Feedback

Portals offers a very coarse-grained interface to its drivers. If the NIC has its own CPU that does the main part of the work, then that is sufficient enough. However in recent and future NICs that CPU becomes the bottleneck (according to the SCS guys) because it only can process the traffic in a *serial* manner. FPGA based NICs like Intercept have the ability to process packets in *parallel* because the number of available state machines is limited only by the number of offered flip-flops of the FPGA.

Especially the Intercept NIC supports broadcast packets and one-sided / two-sided communication in hardware. The current Portals interface does

not support those sophisticated methods, everything is pushed up to the Portals library level or even to the MPI library level. A fine-grained interface might allow the driver to forward those methods to the hardware. If the driver does not offer its own methods then Portals could "fall back" and emulate those function calls via the more simplified ones supported by the hardware (e.g. there is only a send functino in the NAL, no distinction is made between a *get* and a *put*). Portals uses its own protocol on top of the driver. Depending on the driver's implementation some packet types (e.g. acknowledge packets) might be sent twice, once by the driver and once by Portals. An interface that offers the possibility to attach Portals messages to the lower level ones that are caused by the same event might reduce the amount of sent packets.

One might compare this architecture to driver models for graphic cards (like OpenGL, DirectX and so on).

7.4 Commentary

It's always difficult to read other people's code ... especially if the code is 15000 lines long as the VDMA handler is! Nevertheless it's been an challenging and very interesting work. Very frustrating were my experiences with software and hardware that didn't meet the actual specifications. For example I wasted one week trying to get Portals run and my machines simply kept crashing all the time. I eventually figured out that Portals is SMP safe but the rtscts driver I used wasn't and therefore continously crashed on my SMP machines. A had a similar experience with the Intercept network card: My first test programs that used VDMA got stock and nothing ever happend. I even sent my code back to the SCS team and they couldn't tell me why it crashed. I was finally told that it'd be better to align every buffer to 4096 bytes. I did so and magic happend! Well I figured out that my firmware was too old and that the new update I got met the specifications ... At the very ending of my driver writing when I started to test my implementation I couldn't get the expected bandwith figures because of some unexplainable hardware incompatibilities. I sent my programs back to SCS and it run five times faster on their motherboards ...

7.5 Thank You!

At this place I would like to thank the following people. Without them this project would never have been realized:

- Prof. Barney Maccabe for hiring another useless Swiss guy ;-) and his support.
- Prof. Tony Gunzinger and Adrian Riedo who maked my exchange possible.
- Ben Andrews for proofreading my report and being a good friend. The other guys from the "Obscure Acronym Lab" (OAL) for the moral support. Special thanks to Darko Stefanovic to lend me two Itanium machines.
- Wenbin Zhu for all the help with Portals.
- Josh Karlin, Edgar Leon and Carl Sylvia for helping me out with answering questions and help for organizing and installing equipment.
- Kevin Pedretti of Sanida National Laboratories for the jump start with Portals and for answering my never ending questions.
- Rolf Riesen of Sanida National Laboratories for helping me finding a protocol for me driver.
- Martin Maletinsky, Martin Uehli and Vincenzo di Pompeo of Supercomputing Systems Zürich

Appendix A

Source Code

This section contains test programs to examine the Portals driver. Every program run successfully with the written implementation both using VDMA transfer and CB transfers.

A.1 Ping-pong

Listing A.1: ping-pong.h

```
//#define DEBUG

#define HW_BUF_LEN   (LOOPS*1024)
#define HW_SRVR_PID      (5)
#define HW_SRVR_NID      (0)
#define SRV_PORTAL       (2)
#define CLI_PORTAL       (4)

#define MAX_MES 10
#define MAX_MDS 10
#define MAX_EQS 10
#define MAX_ACI  0
#define MAX_PTI 10

#define LOOPS 30

#ifdef NAL_INTERCEPT
#define NAL_INTERFACE PTL_IFACE_INAL
```

```
#endif

#ifdef NAL_MYRINET
#define NAL_INTERFACE PTL_IFACE_MYR
#endif

#ifdef DEBUG
#define PDEBUG(fmt, args...) printf(fmt, ##args)
#else
#define PDEBUG(fmt, args...)
#endif
```

Listing A.2: ping-pong_cli.c

```
/* ping-pong */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <sys/time.h>
#include <portals/p30.h>
#include "ping-pong.h"

#define min(a,b)        (((a)<(b))?(a):(b))
#define max(a,b)        (((a)>(b))?(a):(b))

char *
event_to_str(int event)
{
  if (event == PTL_EVENT_GET_START)    return "PTL_EVENT_GET_START";
  if (event == PTL_EVENT_GET_END)      return "PTL_EVENT_GET_END";
  if (event == PTL_EVENT_GET_FAIL)     return "PTL_EVENT_GET_FAIL";

  if (event == PTL_EVENT_PUT_START)    return "PTL_EVENT_PUT_START";
  if (event == PTL_EVENT_PUT_END)      return "PTL_EVENT_PUT_END";
  if (event == PTL_EVENT_PUT_FAIL)     return "PTL_EVENT_PUT_FAIL";

  if (event == PTL_EVENT_REPLY_START) return "PTL_EVENT_REPLY_START";
  if (event == PTL_EVENT_REPLY_END)    return "PTL_EVENT_REPLY_END";
```

```
  if (event == PTL_EVENT_REPLY_FAIL)  return "PTL_EVENT_REPLY_FAIL";

  if (event == PTL_EVENT_ACK)         return "PTL_EVENT_ACK";

  if (event == PTL_EVENT_SEND_START)  return "PTL_EVENT_SEND_START";
  if (event == PTL_EVENT_SEND_END)    return "PTL_EVENT_SEND_END";
  if (event == PTL_EVENT_SEND_FAIL)   return "PTL_EVENT_SEND_FAIL";

  if (event == PTL_EVENT_UNLINK)      return "PTL_EVENT_UNLINK";

  return NULL;
}

int main(int argc, char **argv)
{
  int rc;
  int i;
  ptl_ni_limits_t desired, actual;
  char inbuf[HW_BUF_LEN];
  char outbuf[HW_BUF_LEN];
  ptl_handle_ni_t          ni;
  ptl_handle_me_t          me;
  ptl_md_t                 md_incoming, md_outgoing;
  ptl_handle_md_t          md, md2;
  ptl_handle_eq_t          eq;
  ptl_event_t              ev;
  ptl_process_id_t         id_local, id_remote;
  struct timeval tv1, tv2, tv3, tv4;
  unsigned long long delay, start, stop;
  unsigned long long dmin, dmax, sum;
  const int num_eq_slots = 10;

  /* PtlInit() must be called first thing */
  rc = PtlInit(NULL);
  if (rc) {
    printf("PtlInit() failed. rc = %d\n", rc);
    abort();
  }

  desired.max_match_entries    = MAX_MES;
```

```
desired.max_mem_descriptors = MAX_MDS;
desired.max_event_queues    = MAX_EQS;
desired.max_atable_index    = MAX_ACI;
desired.max_ptable_index    = MAX_PTI;

/* initialize the intercept portals interface */
rc = PtlNIInit(NAL_INTERFACE, PTL_PID_ANY, &desired, &actual, &ni);
if (rc) {
  printf("PtlNIInit() failed. rc = %d\n", rc);
  abort();
}

printf("ping_client started...\n");

strcpy(outbuf, "hello from ping-ping client.");

id_local.nid = PTL_NID_ANY;
id_local.pid = PTL_PID_ANY;

if ((rc = PtlMEAttach(ni, CLL_PORTAL, id_local, 0, ~0, PTL_RETAIN,
            PTL_INS_AFTER, &me))) {
  printf("PtlMEAttach failed with %d\n", rc);
  abort();
}

/* create an event queue */
if( (rc = PtlEQAlloc( ni, num_eq_slots, NULL, &eq )) ) {
  printf("PtlEQAlloc failed with %d\n", rc);
  abort();
}

/*
 * Fill in the MD and attach it
 */
md_incoming.start                = inbuf;
md_incoming.length               = sizeof(inbuf);
md_incoming.threshold = PTL_MD_THRESH_INF;
// md_incoming.max_offset = IOSIZE;
md_incoming.options              = PTL_MD_OP_PUT;
md_incoming.user_ptr             = NULL;
```

```
md_incoming.eventq                  = eq;
memset(inbuf, 0, sizeof(inbuf));
if( (rc = PtlMDAttach( me, md_incoming, PTL_RETAIN, PTL_RETAIN, &md
  printf("PtlMDAttach failed with %d\n", rc);
  abort();
}

/* Setup the outgoing buffer */

md_outgoing.start                   = outbuf;
md_outgoing.length                  = strlen(outbuf) + 1;
md_outgoing.threshold               = PTL_MD_THRESH_INF;
// md_outgoing.max_offset             = args->size+128;
md_outgoing.options                 = PTL_MD_OP_PUT;
md_outgoing.user_ptr                = NULL;
md_outgoing.eventq                  = eq;
/*   md_outgoing.eventq.handle_idx    = -1; */

if ((rc = PtlMDBind(ni, md_outgoing, &md2))) {
  printf("PtlMDBind failed with %d\n", rc);
  abort();
}

id_remote.nid = HW_SRVR_NID;
id_remote.pid = HW_SRVR_PID;

dmin = __LONG_LONG_MAX__;
dmax = 0;
sum = 0;

sleep(1);

printf("Sending %d ping(s) to NID %Lu\n", LOOPS,
     (unsigned long long)id_remote.nid);

gettimeofday(&tv1, NULL);
if( (rc = PtlPut( md2, PTL_NOACK_REQ, id_remote, SRV_PORTAL,
          0, 0, 0, 0 )))
{
  printf("PtlPut failed with %d\n", rc);
```

```
    abort();
  }

  while (1)
  {
    if ( PtlEQWait( eq, &ev ) != PTL_OK )
    {
      fprintf(stderr,"PtlEQWait() failed\n");
      exit(1);
    }

    PDEBUG("got event: %s\n", event_to_str(ev.type));

    switch(ev.type)
    {
    case PTL_EVENT_SEND_END:
      gettimeofday(&tv2, NULL);

      start = tv1.tv_sec*1000000 + tv1.tv_usec;
      stop = tv2.tv_sec*1000000 + tv2.tv_usec;

      delay = stop - start;
      sum += delay;
      dmin = min(dmin, delay);
      dmax = max(dmax, delay);

      PDEBUG("Sent %Lu bytes in %Lu us (round trip delay).\n",
            md_outgoing.length, delay);
      PDEBUG("Latency %Lu us.\n", delay/2);

      usleep(1000000/5);

      if( ++i >= LOOPS )
      {
        printf("Summary:\n");
        printf("Minumum latency: %Lu\n", dmin);
        printf("Maximum latency: %Lu\n", dmax);
        printf("Average: %Lu (%u tries)\n", sum / LOOPS, LOOPS);

        PtlMDUnlink( md );
```

```
            PtlMDUnlink( md2 );

            PtlMEUnlink( me );

            /* graceful cleanup */

            PtlEQFree( eq );

            PtlNIFini(ni);
            PtlFini();

            return 0;
          }
          /* send the md to the hello world server */
          gettimeofday(&tv1, NULL);
          if( (rc = PtlPut( md2, PTL_NOACK_REQ, id_remote, SRV_PORTAL,
                    0, 0, 0, 0 )))
          {
            printf("PtlPut failed with %d\n", rc);
            abort();
          }
          break;

        default:
          break;
        }
      }
}
```

Listing A.3: ping-pong_srvr.h

```
/* hello world server */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <portals/p30.h>
#include "ping-pong.h"

//typedef void(*sighandler)(int) steht schon in <asm/signal.h>
```

```
__sighandler_t old;
ptl_handle_ni_t ni_h;

void catchabort(int nr)
{
    printf("abort.\n");

  /* graceful cleanup */
  PtlNIFini(ni_h);
  PtlFini();

    exit(0);
}

void catchignore(int nr)
{
    printf("caught signal no: %d\n", nr);
    signal(nr, catchignore);
}

char *
event_to_str(int event)
{
  if (event == PTL_EVENT_GET_START)    return "PTL_EVENT_GET_START";
  if (event == PTL_EVENT_GET_END)      return "PTL_EVENT_GET_END";
  if (event == PTL_EVENT_GET_FAIL)     return "PTL_EVENT_GET_FAIL";

  if (event == PTL_EVENT_PUT_START)    return "PTL_EVENT_PUT_START";
  if (event == PTL_EVENT_PUT_END)      return "PTL_EVENT_PUT_END";
  if (event == PTL_EVENT_PUT_FAIL)     return "PTL_EVENT_PUT_FAIL";

  if (event == PTL_EVENT_REPLY_START) return "PTL_EVENT_REPLY_START";
  if (event == PTL_EVENT_REPLY_END)    return "PTL_EVENT_REPLY_END";
  if (event == PTL_EVENT_REPLY_FAIL)  return "PTL_EVENT_REPLY_FAIL";

  if (event == PTL_EVENT_ACK)          return "PTL_EVENT_ACK";

  if (event == PTL_EVENT_SEND_START)  return "PTL_EVENT_SEND_START";
  if (event == PTL_EVENT_SEND_END)     return "PTL_EVENT_SEND_END";
```

```
  if (event == PTL_EVENT_SEND_FAIL)    return "PTL_EVENT_SEND_FAIL";

  if (event == PTL_EVENT_UNLINK)       return "PTL_EVENT_UNLINK";

  return NULL;
}

int
main(int argc, char **argv)
{
  int rc;
  int i;
  ptl_ni_limits_t desired, actual;
  char inbuf[HW_BUF_LEN];
  char outbuf[HW_BUF_LEN];
  ptl_handle_ni_t          ni;
  ptl_handle_me_t          me;
  ptl_md_t                 md_incoming,md_outgoing;
  ptl_handle_md_t          md_in_handle, md_out_handle;
  ptl_handle_eq_t          eq;
  ptl_event_t              ev;
  ptl_process_id_t         id_local;
  const int num_eq_slots = 10;

  /* PtlInit() must be called first thing */
  rc = PtlInit(NULL);
  if (rc) {
    printf("PtlInit() failed. rc = %d\n", rc);
    abort();
  }

  desired.max_match_entries    = MAX_MES;
    desired.max_mem_descriptors = MAX_MDS;
  desired.max_event_queues     = MAX_EQS;
  desired.max_atable_index     = MAX_ACI;
  desired.max_ptable_index     = MAX_PTI;

  /* initialize the intercept portals interface */
  rc = PtlNIInit(NAL_INTERFACE, HW_SRVR_PID, &desired, &actual, &ni);
  if (rc) {
```

```
    printf("PtlNIInit() failed. rc = %d\n", rc);
    abort();
  }

  printf("ping_server started...\n");

  strcpy(outbuf, "hello from ping-ping server.");

  id_local.nid = PTL_NID_ANY;
  id_local.pid = PTL_PID_ANY;

  if ((rc = PtlMEAttach(ni, SRV_PORTAL, id_local, 0, ~0,
              PTL_UNLINK, PTL_INS_AFTER, &me)) ) {
    printf("PtlInit() failed. rc = %d\n", rc);
    abort();
  }

  /* create an event queue */
  if( (rc = PtlEQAlloc( ni, num_eq_slots, NULL, &eq )) ) {
    printf("PtlEQAlloc failed with %d\n", rc);
    abort();
  }

  /* Setup the outgoing buffer */
  md_outgoing.start               = outbuf;
  md_outgoing.length              = strlen(outbuf) + 1;
  md_outgoing.threshold      = PTL_MD_THRESH_INF;
  // server->md_outgoing.max_offset= TXIOSIZE;
  md_outgoing.options             = PTL_MD_OP_PUT;
  md_outgoing.user_ptr    = NULL;
/*   md_outgoing.eventq.handle_idx                = -1; */
  md_outgoing.eventq              = eq;
  if ((rc = PtlMDBind(ni, md_outgoing,
            &md_out_handle))) {
    printf("PtlMDBind() failed. rc = %d\n", rc);
    abort();
  }

  /*
   * Fill in the MD and attach it
```

```
 */
md_incoming.start                    = inbuf;
md_incoming.length                   = sizeof(inbuf);
md_incoming.threshold      = PTL_MD_THRESH_INF;
// server->md_incoming.max_offset= (RXIOSIZE);
md_incoming.options                  = PTL_MD_OP_PUT;
md_incoming.user_ptr       = NULL;
md_incoming.eventq                   = eq;

if ((rc = PtlMDAttach(me, md_incoming,
            PTL_UNLINK, PTL_UNLINK, &md_in_handle)) ) {
    printf("PtlMDAttach failed with %d\n", rc);
  abort();
}

i = 0;

while (1)
{
  if ( PtlEQWait( eq, &ev ) != PTL_OK )
  {
    fprintf(stderr,"PtlEQWait() failed\n");
    exit(1);
  }

  PDEBUG("got event: %s\n", event_to_str(ev.type));

  if (ev.type == PTL_EVENT_PUT_END)
  {

    PDEBUG("got a message!!!\n");

    PDEBUG("received message from node %d\n"
          "request length: %lu, manipulated length: %lu\n"
          "message: %s\n",
          (int) ev.initiator.nid,
          (unsigned long)ev.rlength,
          (unsigned long)ev.mlength,
          inbuf);
```

```
        if( ++i >= LOOPS)
        {
          PtlMDUnlink( md_in_handle );
          PtlMDUnlink( md_out_handle );

          PtlMEUnlink( me );

          /* graceful cleanup */

          PtlEQFree( eq );

          PtlNIFini(ni);
          PtlFini();

          return 0;
        }

        if ((rc = PtlPut(md_out_handle, PTL_NOACK_REQ,
                ev.initiator, CLL_PORTAL, 0, 0, 0, 0))) {
          printf("PtlPut failed with %d\n", rc);
          abort();
        }
      }
    }
}
```

A.2 Bandwith

Listing A.4: bandwith.h

```
//#define DEBUG

#define BUFFER_ALIGNEMENT 4096 //32

#define BUFFER_SIZE (100*1024*1024UL)

#define HW_SRVR_PID      (5)
#define HW_SRVR_RCV_PTL (5)

#define MAX_MES 10
```

```
#define MAX_MDS 10
#define MAX_EQS 10
#define MAX_ACI  0
#define MAX_PTI 10

#define min(a,b)        (((a)<(b))?(a):(b))
#define max(a,b)        (((a)>(b))?(a):(b))

#ifdef NAL_INTERCEPT
#define NAL_INTERFACE PTL_IFACE_INAL
#endif

#ifdef NAL_MYRINET
#define NAL_INTERFACE PTL_IFACE_MYR
#endif

#ifdef DEBUG
#define PDEBUG(fmt, args...) printf(fmt, ##args)
#else
#define PDEBUG(fmt, args...)
#endif
```

Listing A.5: bandwith_cli.c

```
/* simple bandwith measurement tool
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <sys/time.h>
#include <unistd.h>
#include <portals/p30.h>
#include "bandwith.h"

unsigned char buf[BUFFER_SIZE] __attribute__ ((aligned(BUFFER_ALIGNEMEN

char *
event_to_str(int event)
{
```

```
  if (event == PTL_EVENT_GET_START)    return "PTL_EVENT_GET_START";
  if (event == PTL_EVENT_GET_END)      return "PTL_EVENT_GET_END";
  if (event == PTL_EVENT_GET_FAIL)     return "PTL_EVENT_GET_FAIL";

  if (event == PTL_EVENT_PUT_START)    return "PTL_EVENT_PUT_START";
  if (event == PTL_EVENT_PUT_END)      return "PTL_EVENT_PUT_END";
  if (event == PTL_EVENT_PUT_FAIL)     return "PTL_EVENT_PUT_FAIL";

  if (event == PTL_EVENT_REPLY_START) return "PTL_EVENT_REPLY_START";
  if (event == PTL_EVENT_REPLY_END)    return "PTL_EVENT_REPLY_END";
  if (event == PTL_EVENT_REPLY_FAIL)  return "PTL_EVENT_REPLY_FAIL";

  if (event == PTL_EVENT_ACK)          return "PTL_EVENT_ACK";

  if (event == PTL_EVENT_SEND_START)  return "PTL_EVENT_SEND_START";
  if (event == PTL_EVENT_SEND_END)     return "PTL_EVENT_SEND_END";
  if (event == PTL_EVENT_SEND_FAIL)    return "PTL_EVENT_SEND_FAIL";

  if (event == PTL_EVENT_UNLINK)       return "PTL_EVENT_UNLINK";

  return NULL;
}

const char *get_argument(const char **argv,
             const char *keystr,
             const char *deflt)
{
  int i;
  for (i=1; argv[i]; i++)
    if (strstr(argv[i],keystr))
      return argv[i+1];
  return deflt;
}

int
main(int argc, const char **argv)
{
  int rc;

  int num_eq_slots = 10;
```

```
ptl_process_id_t snd_target;
ptl_match_bits_t snd_match_bits  = 0;
ptl_md_t         snd_md;
ptl_event_t      event;

ptl_handle_ni_t ni_h;
ptl_handle_eq_t eq_h;
ptl_handle_md_t md_h;

ptl_ni_limits_t desired, actual;
struct timeval tvstart, tvstop;

/* Offset in send buffer, to introduce unaligned transfers. */
const unsigned long missalign = atol(get_argument(argv, "-m", "0"));

/* Length of the whole message (just for limitation, includes the
   filename string) */
const unsigned long length = atol(get_argument(argv, "-l", "0"));

unsigned long msglen;
unsigned char *missalignedbuf;

double b, ddelay, dmsglen, dstart, dstop;

missalignedbuf = buf + missalign;
msglen = (length != 0) ? min(1024*1024*length, sizeof(buf) - missalig
  : (sizeof(buf) - missalign);

printf("Bufferaddress: %p\n"
     "Bufferaddress + missalignement: %p\n"
     "Messagelength: %lu\n",
     buf,
     missalignedbuf,
     msglen);

/* PtlInit() must be called first thing */
rc = PtlInit(NULL);
if (rc) {
  printf("PtlInit() failed. rc = %d\n", rc);
```

```
    abort();
  }

  desired.max_match_entries    = MAX_MES;
  desired.max_mem_descriptors = MAX_MDS;
  desired.max_event_queues     = MAX_EQS;
  desired.max_atable_index     = MAX_ACI;
  desired.max_ptable_index     = MAX_PTI;

  /* initialize the myrinet portals interface */
  rc = PtlNIInit(NAL_INTERFACE, PTL_PID_ANY, &desired, &actual, &ni_h);
  if (rc) {
    printf("PtlNIInit() failed. rc = %d\n", rc);
    abort();
  }

  /* create an event queue */
  rc = PtlEQAlloc(ni_h, num_eq_slots, NULL, &eq_h);
  if (rc) {
    printf("PtlEQAlloc() failed. rc = %d\n", rc);
    abort();
  }

  /* setup a md */
  snd_md.start      = missalignedbuf;
  snd_md.length     = msglen;
  snd_md.threshold  = 1;
  // snd_md.max_offset = ?;
  snd_md.options    = 0;
  snd_md.eventq     = eq_h;

  /* bind the md */
  snd_target.nid = 0;
  snd_target.pid = HW_SRVR_PID;

  printf("sending to nid=%d, pid=%d\n",
       (int) snd_target.nid, (int) snd_target.pid);

  rc = PtlMDBind(ni_h, snd_md, &md_h);
  if (rc) {
```

```
    printf("PtlMDBind() failed. rc = %d\n", rc);
    abort();
  }

  sleep(1);

  /* send the md to the hello world server */
  gettimeofday(&tvstart, NULL);
  rc = PtlPut(md_h, PTL_NOACK_REQ, snd_target, HW_SRVR_RCV_PTL,
        0, snd_match_bits, 0, 0);
  if (rc) {
    printf("PtlPut() failed. rc = %d\n", rc);
    abort();
  }
  /*  gettimeofday(&tvstop, NULL); */

  while (1) {
    if ( PtlEQWait( eq_h, &event ) != PTL_OK ) {
      fprintf(stderr,"PtlEQGet() failed\n");
      exit(1);
    }
    PDEBUG("got event: %s\n", event_to_str(event.type));
    switch(event.type)
    {
    case PTL_EVENT_SEND_START:
      /*            gettimeofday(&tvstart, NULL); */
      break;

    case PTL_EVENT_SEND_END:
      gettimeofday(&tvstop, NULL);
      break;

    default:
      break;
    }
    if( event.type == PTL_EVENT_UNLINK) break;
  }

  PtlMDUnlink( md_h );
```

```
  PtlEQFree( eq_h );

  /* graceful cleanup */
  PtlNIFini(ni_h);
  PtlFini();

  dstart = (double)tvstart.tv_sec * 1000.0 + (double)tvstart.tv_usec / 1000.0;
  dstop = (double)tvstop.tv_sec * 1000.0 + (double)tvstop.tv_usec / 1000.0;

  ddelay = dstop - dstart;

  printf("Sent %lu bytes in %f ms.\n", msglen, ddelay);

  dmsglen = msglen/1024/1024;
  b = dmsglen/ddelay*1000;

  printf("Bandwith: %f MB/s\n", b);

  return 0;
}
```

Listing A.6: bandwith_srvr.h

```
/* filetrans server */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <getopt.h>
#include <portals/p30.h>
#include "bandwith.h"

//typedef void(*sighandler)(int) steht schon in <asm/signal.h>

__sighandler_t old;
ptl_handle_ni_t ni_h;

unsigned char buf[BUFFER_SIZE] __attribute__ ((aligned(BUFFER_ALIGNEMENT)));

char *
```

```
event_to_str(int event)
{
  if (event == PTL_EVENT_GET_START)    return "PTL_EVENT_GET_START";
  if (event == PTL_EVENT_GET_END)      return "PTL_EVENT_GET_END";
  if (event == PTL_EVENT_GET_FAIL)     return "PTL_EVENT_GET_FAIL";

  if (event == PTL_EVENT_PUT_START)    return "PTL_EVENT_PUT_START";
  if (event == PTL_EVENT_PUT_END)      return "PTL_EVENT_PUT_END";
  if (event == PTL_EVENT_PUT_FAIL)     return "PTL_EVENT_PUT_FAIL";

  if (event == PTL_EVENT_REPLY_START) return "PTL_EVENT_REPLY_START";
  if (event == PTL_EVENT_REPLY_END)    return "PTL_EVENT_REPLY_END";
  if (event == PTL_EVENT_REPLY_FAIL)  return "PTL_EVENT_REPLY_FAIL";

  if (event == PTL_EVENT_ACK)          return "PTL_EVENT_ACK";

  if (event == PTL_EVENT_SEND_START)  return "PTL_EVENT_SEND_START";
  if (event == PTL_EVENT_SEND_END)     return "PTL_EVENT_SEND_END";
  if (event == PTL_EVENT_SEND_FAIL)    return "PTL_EVENT_SEND_FAIL";

  if (event == PTL_EVENT_UNLINK)       return "PTL_EVENT_UNLINK";

  return NULL;
}

void catchabort(int nr)
{
  printf("abort.\n");

  /* graceful cleanup */
  PtlNIFini(ni_h);
  PtlFini();

  exit(0);
}

void catchignore(int nr)
{
  printf("caught␣signal␣no:␣%d\n", nr);
  signal(nr, catchignore);
```

```
}

const char *get_argument(const char **argv,
             const char *keystr,
             const char *deflt)
{
  int i;
  for (i=1; argv[i]; i++)
    if (strstr(argv[i],keystr))
      return argv[i+1];
  return deflt;
}

int
main(int argc, const char **argv)
{
  int rc;

  int               num_eq_slots     = 10;
  int               rcv_ptl_indx     = HW_SRVR_RCV_PTL;
  ptl_match_bits_t  rcv_match_bits   = 0;
  ptl_match_bits_t  rcv_ignore_bits  = 0xFFFFFFFFFFFFFFFF;
  ptl_process_id_t  rcv_match_id;
  ptl_md_t          rcv_md;
  ptl_event_t       event, put_end_event;

  ptl_handle_eq_t eq_h;
  ptl_handle_me_t me_h;
  ptl_handle_md_t md_h;

  ptl_ni_limits_t actual, desired;

  int got_unlink_event;
  int got_put_end_event;
  int got_put_start_event;

  const unsigned long missalign = atol(get_argument(argv, "-m", "0"));

  unsigned char *missalignedbuf;
```

```
missalignedbuf = buf + missalign;

/* PtlInit() must be called first thing */
rc = PtlInit(NULL);
if (rc) {
  printf("PtlInit() failed. rc = %d\n", rc);
  abort();
}

desired.max_match_entries    = MAX_MES;
desired.max_mem_descriptors = MAX_MDS;
desired.max_event_queues     = MAX_EQS;
desired.max_atable_index     = MAX_ACI;
desired.max_ptable_index     = MAX_PTI;

/* initialize the myrinet portals interface */
rc = PtlNIInit(NAL_INTERFACE, HW_SRVR_PID, &desired, &actual, &ni_h)
if (rc) {
  printf("PtlNIInit() failed. rc = %d\n", rc);
  abort();
}

/* create an event queue */
rc = PtlEQAlloc(ni_h, num_eq_slots, NULL, &eq_h);
if (rc) {
  printf("PtlEQAlloc() failed. rc = %d\n", rc);
  abort();
}

/* create and attach a match entry to a portal */
rcv_match_id.nid = PTL_NID_ANY;
rcv_match_id.pid = PTL_PID_ANY;

rc = PtlMEAttach(ni_h, rcv_ptl_indx, rcv_match_id,
        rcv_match_bits, rcv_ignore_bits,
        0, PTL_INS_AFTER, &me_h);
if (rc) {
  printf("PtlMEAttach() failed. rc = %d\n", rc);
  abort();
}
```

```
/* create and attach a md to the me */
rcv_md.start      = missalignedbuf;
rcv_md.length     = sizeof(buf) - missalign;
rcv_md.threshold  = 1;
/* rcv_md.max_offset = HW_BUF_LEN; */
rcv_md.options    = PTL_MD_OP_PUT | PTL_MD_TRUNCATE;
rcv_md.eventq     = eq_h;

printf("buf: %p\n"
     "buf + missalign: %p\n"
     "bufferlength: %lu\n",
     buf, missalignedbuf, (unsigned long)rcv_md.length);

rc = PtlMDAttach(me_h, rcv_md, PTL_UNLINK, PTL_UNLINK, &md_h);
if (rc) {
  printf("PtlMDAttach() failed. rc = %d\n", rc);
  abort();
}

/* install ctrl-c signal handler for proper shutdown */
signal(SIGINT, catchabort);

while (1) {
  printf("waiting for a message...\n");
  got_unlink_event = 0;
  got_put_end_event = 0;
  got_put_start_event = 0;

  while (1) {
    if ( PtlEQWait( eq_h, &event ) != PTL_OK ) {
      fprintf(stderr,"PtlEQGet() failed\n");
      exit(1);
    }
    fprintf(stderr,"got event: %s\n", event_to_str(event.type));
    if (event.type == PTL_EVENT_PUT_END) {
      put_end_event = event;
    }
    if( event.type == PTL_EVENT_UNLINK) break;
  }
```

```
    printf("got a message!!!\n");

    printf("received message from node %d\n"
           "request length: %lu, manipulated length: %lu\n",
           (int) put_end_event.initiator.nid,
           (unsigned long)put_end_event.rlength,
           (unsigned long)put_end_event.mlength);

    /* reattach the md to the me */
    rc = PtlMDAttach(me_h, rcv_md, PTL_UNLINK, PTL_UNLINK, &md_h);
    if (rc) {
      printf("PtlMDAttach() failed. rc = %d\n", rc);
      abort();
    }
  }

  /* graceful cleanup */
  PtlNIFini(ni_h);
  PtlFini();

  return 0;
}
```

Appendix B

Portals Installation

This section contains a quick guide how to install Portals. This is an extract from an email correspondance with Kevin Pedretti.

1. Patch a kernel with the Luster 2.4.18 patch
2. Configure Portals:
   `./configure --with-linux=path_to_patched_kernel_src`
3. Edit `portals/api/api-init.c` to turn off the default extremely verbose debugging. There should be a line like:
   `unsigned int portals_debug = ~0;`
   change this to:
   `unsigned int portals_debug = 0;`
4. Do a make in `portals/user/ptrxtx`
5. Do a make in `portals/user/hello_world`
6. Edit the `mac-map` file in `/user/etc`. The `README` in the same directory describes this file.
7. Startup Portals:
   cd to `portals/user/etc` `./portals.eth start node_id eternet_device_name`
   e.g.,
   `./portals.eth start 0 eth0`
   This script should load the modules `portals.o`, `p3mod.o`, `rtscts.o`, and `kptrxtx` into the kernel with no unresolved symbols.
8. See if hello world works in `portals/user/hello_world`.
   Start the `hello_world_srvr` on node 0. Then run `hello_world_cli` in another window/on another node. The output of `hello_world_srvr` should look something like...

```
waiting for a message...
got a message!!!
received msg from node 0: this is a test of the emergency broadcasting
system

waiting for a message...
```

Bibliography

[1] Ron Brightwell, Arthur B. Maccabe, Rolf Riesen :
The Portals 3.2 Message Passing Interface Revision 1.0
`http://www.sandiaportals.org`.
Sandia National Laboratories & The University of New Mexico, 11/2002. 2, 12

[2] Adrian Riedo :
The Portals over TNet website
`http://hpc.fribyte.ch`. NOTE: Similar project with the TNet network card from Supercomputing Systems AG. 16

[3] Alessandro Rubini & Jonathan Corbet :
Linux Device Drivers, 2nd ed.
O'Reilly, ISBN 0-596-00008-1

[4] *The Lustre cluster file system*
`http://www.lustre.org/`. 14

[5] *The Computational Plant project*
`http://www.cs.sandia.gov/cplant`. 12

[6] Hansueli Wyss, Marc Oberholzer, Patrick Müller, Martin Heimlicher :
Project Intercept - NIC HW Programmer's Manual
Supercomputing Systems AG, 10/2002

[7] Martin Maletinsky :
Project Intercept - Linux Device Driver Design Description
Supercomputing Systems AG, 6/2002

[8] Martin Maletinsky :
Project Intercept - Virtual DMA Software Design Description
Supercomputing Systems AG, 6/2002 7, 10, 11

[9] Roland Paul :
Intercept Packet Definition
Supercomputing Systems AG, 6/2002

[10] Peter S. Pacheco :
Parallel Programming with MPI
Morgan Kaufmann Publishers, ISBN 1-55860-339-5, 1997 1

[11] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schröter, D. Verworner :
Linux Kernelprogrammierung
Addison-Wesley, München, 6. Auflage, ISBN 3-8273-1659-6, 2001

[12] Maurice J. Bach :
The Design of the Unix Operating System
Prentice-Hall, London, ISBN 0-13-201799-7, 1990

[13] *Cross-Referencing Linux*
`http://lxr.linux.no/`
NOTE: Indispensable kernel cross reference. Very recommendable.

[14] Rolf Riesen :
Message-Based, Error-Correcting Protocols for Scalable High-Performance Networks
PhD thesis, University of New Mexico, 7/2002 29