Low-Latency Linux Drivers for Ethernet over High-Speed Networks

Rainer Finocchiaro, Lukas Razik, Stefan Lankes, Thomas Bemmerl *

Abstract—**Nowadays, high computing demands are often tackled by clusters of single computers, each of which is basically an assembly of a growing number of CPU cores and main memory, also called a node; these nodes are connected by some kind of communication network. With the growing speed and number of CPU cores, the network becomes a severe bottleneck limiting overall cluster performance. High-speed interconnects like InfiniBand, SCI, and Dolphin DX are good for alleviating this communication bottleneck, when the communication software is either based on IP or specifically adapted to the interconnect. Software written to communicate directly via Ethernet frames can not be used this way, though. In this article, we present two drivers for Linux that fill this gap. ETHOS is a very generic Ethernet over Sockets driver. With this driver it is possible to use any interconnect that offers a sockets interface as replacement for Ethernet. The second driver, ETHOM, sacrifices the compatibility with a wide range of interconnects in favour of higher performance on top of Dolphin's high-speed networks SCI and DX. It enhances their functionality by offering an Ethernet and with that an IP interface. Both drivers allow usage of layer-2 kernel functionality like interface bonding and bridging. By means of various measurements, we show that ETHOS and ETHOM with InfiniBand, SCI, or DX offer a two- to threefold increase in communication performance over Gigabit Ethernet.**

Keywords: Ethernet, InfiniBand, SCI, Dolphin DX, Linux, TIPC

1 Introduction

Computational power has always been a scarce resource and prognoses predict that this situation will not change any time soon. While computer performance increases, the demand for more computational power increases at least at the same pace.

Until very recently, CPUs as the main component of a computing system grew more powerful by raising the clock frequency. Today parallelism in the form of additional cores per die adds to the performance increase. From a hardware point of view, the next level of parallelism is the gathering of single computers to form a cluster. Traditionally, the single computers – called hosts or nodes – in these clusters were connected by Ethernet in one of its incarnations, as it is cheap, stable, and well supported. Concerning software, the predominant protocol used on top of Ethernet is the TCP/IP[1] stack. With software running on the cluster that communicates intensively, the network more and more becomes the real bottleneck that limits cluster performance.

So, there are two problems to cope with:

1. Ethernet networking hardware does not fit all purposes: In the form of Gigabit Ethernet it is too slow for several applications; 10 Gigabit Ethernet is still in the beginnings, not yet very wide-spread, and rather expensive.

2. There are protocols better suited for cluster computing than TCP/IP. This protocol suite was designed for communication in local to wide area networks, offering elaborate mechanisms for routing, to deal with even extensive packet loss, etc. With multiple checksums, a two-layer approach, and quite a lot of functionality that might not be needed in cluster computing scenarios, it causes some inefficiency.

To tackle these problems, there are mainly two approaches in order to allow faster[2] communication:

1. High-speed networks can be used to replace or complement Ethernet. Each of them has its own low-level programming interface (API), most provide an implementation of the POSIX socket API, and some offer an IP interface. Examples of these networks include InfiniBand [1], Myrinet [2], QsNet [3], SCI [4], and Dolphin DX [5]. An IP interface for Dolphin DX has been presented in [6].

2. Replace the software layer TCP and UDP[3] – and sometimes IP as well – with another protocol while keeping the Ethernet hardware. Examples of these replacement protocols include SCTP (Stream Control Transmission Protocol [7]), DCCP (Datagram Congestion Control Protocol [8]), UDP-Lite [9], AoE

*Chair for Operating Systems, RWTH Aachen University, Kopernikusstr. 16, 52056 Aachen, Germany,
E-mail: {finocchiaro,razik,lankes,bemmerl}@lfbs.rwth-aachen.de

[1] Transmission Control Protocol/Internet Protocol
[2] latency and bandwidth wise
[3] User Datagram Protocol

(ATA over Ethernet [10]), and TIPC (Transparent Interprocess Communication Protocol [11, 12, 13]).

Being developed originally at Ericsson, TIPC has its roots in the telecommunication sector, but provides some characteristics making it suitable for high performance computing (HPC) with clusters, such as an addressing scheme supporting failover mechanisms and less overhead for exchanging data within a cluster. TIPC is the transport layer of choice of the Kerrighed project [14], which is an enhancement of the Linux kernel with the objective to integrate all nodes of a cluster into a single unified view of operating system resources, a so called Single System Image (SSI). The TIPC protocol is used for kernel to kernel communication, but cannot currently make use of high-speed networks like InfiniBand or SCI, as they both do not provide an Ethernet interface, nor does TIPC provide a specialised "bearer", which is the adaptation layer between TIPC and a native network interface.

As a first approach for enabling TIPC to make use of high-speed networks, we developed *ETHOS (ETHernet Over Sockets driver)*, a driver offering an Ethernet interface supporting a wide variety of high-speed networks for communication. This driver is designed to use kernel-level UDP sockets to deliver data to peers; it enables any network interconnect providing kernel-space UDP sockets to be used as Ethernet replacement. Measurements with ETHOS on top of SCI and InfiniBand show significantly higher bandwidth and lower latency than Gigabit Ethernet.

In order to further reduce communication latency, we decided to sacrifice compatibility with other high-speed interconnects and use the next lower software layer available in the Dolphin Express stack, the Message Queue Interface. Using this interface for actually transferring data, *ETHOM (ETHernet Over Message Queue driver)* provides an Ethernet interface for SCI and Dolphin DX hardware. Therefore, in addition to the TCP and UDP Sockets already provided by the Dolphin Express software stack, ETHOM offers an Ethernet interface, enabling interface bonding, bridging and other layer 2 kernel features, as well as IP routing for the SCI and Dolphin DX interconnects. Furthermore, TIPC is enabled to make use of these two network technologies leveraging its Ethernet bearer, just like any other software that is stacked on top of an Ethernet interface.

The rest of this article is organised in the following way: In section 3, we shortly present the Linux network architecture, serving as background for understanding where the presented new Ethernet interfaces reside. After that, we present ETHOS and ETHOM in some detail, providing information about design decisions. Section 4 gives an overview of the performance of ETHOS and ETHOM when used with either TCP/IP or with TIPC, measured with popular microbenchmarks. Finally, in section 5, we conclude with the current status and plans for further improvements.

2 Technical Background

In this section, we will give a brief overview of the technologies involved in our development. We start with the high-speed networks, which are actively supported by our software, and continue with the TIPC protocol, which constitutes the main reason for developing these drivers.

2.1 High-Speed Networks

There are quite a few high-speed networks available today. In our description below, we concentrate on those directly available to us for developing the software and because of that actively supported by us. The above section "Introduction" names a few of the other available options.

2.1.1 Dolphin Express

Dolphin Interconnect Solutions designs interconnect chips and host adapters for PCI interfaces, and offers a complete software stack on top of this hardware for all major operating systems. This software stack allows to use the hardware efficiently from either user or kernel space. For reduced application complexity, several APIs are offered. The software stack consists of kernel drivers, libraries, tools and cluster management software, and it is published as open source under the (L)GPL.

Scalable Coherent Interface (SCI) The Scalable Coherent Interface [15, 16] is an established interconnect technology for transparent communication on the memory access level and/or the I/O read/write level. It maps (parts of) the physical address spaces of the connected nodes into one global address space, which allows to export and import memory and access it transparently via programmed input/output (PIO), or explicitly using direct memory access (DMA) transfers. Cache coherency between the nodes is supported by the standard, but not via I/O interfaces like PCI. The nodes are connected in multidimensional torus topologies without a central switch, as each host adapter also switches packets between its multiple links.

The current SCI hardware generation achieves remote store latencies starting at 220 ns and a maximum bandwidth of 334 MiB/s per channel.

Dolphin DX The Dolphin DX interconnect [6] is based on the protocols for the Advanced Switching Interface (ASI). As such, it also couples buses and memory regions of distributed machines, but is designed for PCI Express and not for coherent memory coupling. Also, it does

not use distributed switching like SCI; instead, all nodes connect to a central switch. Current switches offer 10 ports, and can be scaled flexibly.

Nevertheless, DX offers many of the same features as SCI from a programmers perspective, namely transparent PIO and DMA access to remote memory and remote interrupts. This makes it possible to integrate it into the existing software stack for SCI, offering the same APIs as for SCI.

The performance of DX has significantly improved compared to SCI for both, PIO and DMA transfers. The latency to store 4 bytes to remote memory is 40 ns, while the bandwidth reaches about 1.397 GiB/s already at 64 bytes transfer size.

2.1.2 InfiniBand

Usage of shared bus architectures for attaching I/O peripherals to the CPU/memory increasingly becomes a bottleneck. When launching the *InfiniBand Trade Association*[4] in 1999, its main aim was to define a high performance interconnect, which is called *InfiniBand* and breaks the bandwidth limitation of traditional shared bus architectures. In InfiniBand, computing nodes and I/O nodes are connected to a *switched fabric*. The fabric itself may consist of a single switch in the simplest case or a collection of interconnected switches and routers. Using such a switched fabric removes the bottleneck of shared bus architectures.

In contrast to SCI and DX, InfiniBand does not support transparent communication on the memory access level. The other features like DMA access to remote memory and remote interrupts are also supported by InfiniBand.

The current InfiniBand hardware generation (x4 QDR) achieves remote store latencies starting at 1500 ns and a maximum bandwidth of 2.8 GiB/s per channel.

2.1.3 Driver Stacks

Figure 1 shows the current network driver architecture. There are several paths of execution that could forward a message from an application at the top of the illustration to a network interface controller (NIC) at the bottom.

The most popular and established interface for using a network is the Berkeley sockets interface, which is part of the `glibc`. The address family switch decides the further path through the kernel. By using the socket family `AF_INET` (in the middle of the illustration), the message has to pass through the TCP/IP or UDP/IP stack. These stacks are not optimal for reaching the maximum bandwidth and the lowest possible latency. Therefore, high-speed networks provide mechanisms to bypass these stacks, thereby offering excellent performance, while still maintaining compatibility to the established Berkeley socket interface. If the highest possible performance is essential for the applications, high-speed networks also provide interfaces, which disclose all available features to the applications. Usage of these interfaces, implies a redesign of the applications for each network.

The SISCI API [17] is the most efficient possibility to use SCI or DX as high-speed interconnect. SISCI is a shared-memory programming interface that makes the features of the SCI interconnect accessible from user space. It consists of a user-space shared library (`libsisci`, shown on the left in the illustration) which communicates with the SISCI kernel driver via `ioctl()` operations to create and export shared memory segments, map remote memory segments to the address space of the calling process, send and wait for remote interrupts, and perform DMA transfers from and to remote memory segments. Next to this, functions for error checking and information querying, plus other miscellaneous operations are included.

These means allow processes running on different machines to create common, globally distributed shared memory regions and read and write data from and to there either via PIO or DMA operations. Synchronization can be performed via shared memory or via remote interrupts.

To obtain optimal communication performance, data transfers need to be aligned to suitable SCI packet and buffer sizes (16, 64 and 128 bytes), and remote read operations should be avoided except for very small data sizes. Error status checks should be performed as rarely as possible, as they involve a costly read access via the PCI bus.

SISCI does not provide means to pass messages between processes except for writing to some shared memory location and synchronizing via either shared memory or remote interrupts. While it is not very complicated to create a simple message queue on top of this, Dolphin supplies a thin software layer for communication via message queues (`MBox/Msq`). It allows to establish unidirectional communication channels between machines which can be operated via simple `send()` and `recv()` operations, either using PIO or DMA. This software layer takes care of alignment, data gathering, error checking and so forth, and offers different optimized protocols for small, medium, and large data sizes.

It is also the basis for Dolphin's SuperSockets (`SSocks` in Figure 1), which in user space offer a Berkeley API compliant sockets interface via `libksupersockets`. By using this interface, applications are able to bypass the TCP/IP software stack of the kernel, thereby reducing latency. Support of this established and widely used in-

[4]The home page of the InfiniBand Trade Association can be found at `http://www.infinibandta.org`

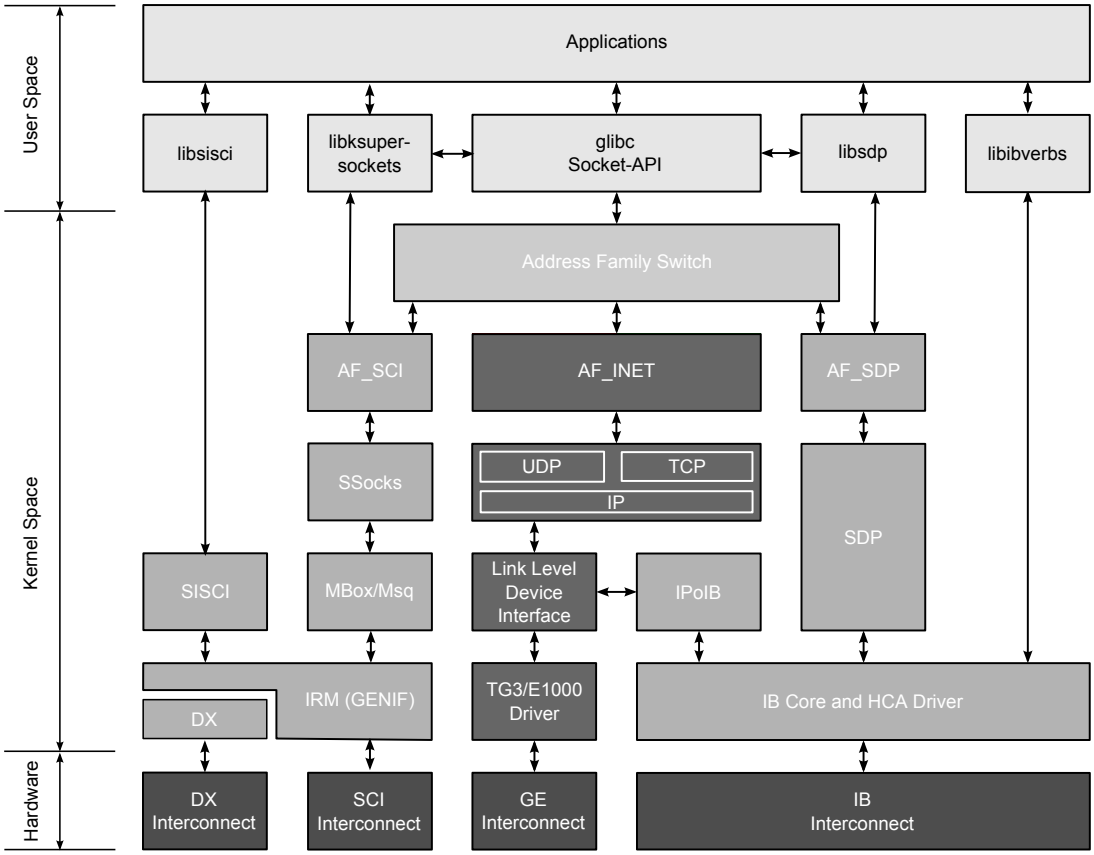

Figure 1: Network Driver Architecture of Dolphin DX, SCI, Ethernet and InfiniBand

terface increases the usability of the SCI and DX network.

InfiniBand provides a similar socket interface via the *Sockets Direct Protocol* (SDP) supplied by the `SDP` kernel module in kernel space and `libsdp` in user space. By using remote direct memory access (RDMA) with a zero-copy data transfer mechanism, SDP provides a low latency and a high bandwidth. However, this interface deals only with stream sockets. All other socket types are supported by the Linux IP stack and operate over the *IP over InfiniBand* (IPoIB) link to the driver (compare Figure 1).

The counterpart of the SISCI interface for the InfiniBand interconnect is *IB Verbs*. The IB Verbs interface – provided by the IB Core module and the Host Channel Adapter (HCA) driver in kernel space and by `libibverbs` in user space – is a programming interface that makes it possible to use RDMA transfers directly in user space. It provides the lowest overhead, lowest latency, and the maximum bandwidth, however, requires a network-dependent rework of existing applications.

2.2 The TIPC Protocol

Our main aim was to supply Kerrighed with support for communication over high-speed networks. Kerrighed uses TIPC instead of IP as communication protocol for all its communication needs, so we turned our focus to enabling TIPC to use high-speed networks. As TIPC is not so well known, but offers a lot of interesting features for the realm of cluster computing, it will be presented in this section.

Since it introduces new terminology, we start with describing the network topology as it is seen by TIPC.

2.2.1 Network Topology

A TIPC network consists of individual computers, called *nodes*, that are grouped in a hierarchical manner as depicted in Figure 2:

Nodes Nodes represent the lowest level of a TIPC network. They are individual computers, that communicate with peers over direct links.

Cluster A group of nodes forms a *cluster*, if every one of these nodes can reach any other node directly (*one hop*). Directly refers to the logical link; i. e. nodes connected via a switch – as is standard with Ethernet – are considered directly connected. Two nodes that are connected via a third one, which is equipped with two network interface cards (NICs), would not be considered

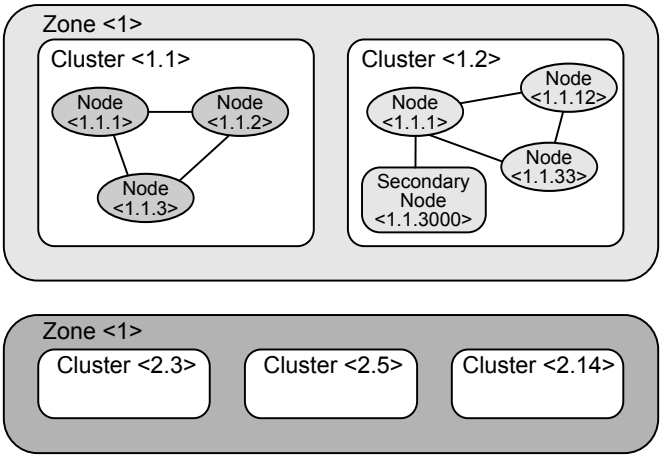

Figure 2: TIPC Network Topology

directly connected. In standard configuration, a cluster can comprise up to 4095 nodes.

Zone The next bigger organisational unit is called a *zone*. A zone consists of interconnected clusters. In standard configuration, a zone can include up to 4095 clusters.

Network A group of interconnected zones constitutes a *network*, if all of these zones are directly linked to each other. Up to 255 zones are supported inside of one network.

Typically, geographical position of nodes leads to a grouping in clusters, zones, and networks. In that sense, commonly, nodes put together inside a cabinet or a room form a cluster; clusters inside the same building or site usually form a zone. One of the assumptions that TIPC makes is that the far biggest part of communication takes place between nodes inside a cluster, while communication between clusters is rare, even more so between zones.

Each TIPC network is attributed a *network identifier* (netID), so that different logical networks can utilise the same physical media (e. g. Ethernet cards and cables) without interfering with each other.

A node as part of a TIPC network communicates with its peers using one or more network interfaces, each of which must be connected to a physical medium supported by TIPC. If configured correctly, TIPC automatically detects peers of a node and sets up logical links to each of them.

2.2.2 Addressing

While TIPC supports *physical* addressing, which is quite similar to the IP addressing scheme[5], its real strength lies in *functional* addressing, which allows for location-agnostic services, supporting e. g. redundancy, if a service is provided by more than one node.

For physical addressing, each node (not interface!) is attributed a network address consisting of the three parts zone identifier (`Z`), cluster identifier (`C`), and node identifier (`N`); i. e. `<Z.C.N>`. A particular connection endpoint is described with a *port identifier* as `<Z.C.N:ref>`, i. e. a network address plus a unique reference number. Apart from nodes, clusters can be addressed (`<Z.C.0>`), as well as zones (`<Z.0.0>`) – the address `<0.0.0>` has a special meaning and is operation specific.

Functional addressing is accomplished by means of *port names*, denoted as `{type, instance}`. Often, `type` indicates a class of service provided by a port and `instance` a sub-class. Port names do not have to be unique inside a TIPC network. Multiple port names can be assigned to one port or vice versa. When binding a port name to a port, an application can restrict the visibility or *scope* of the name to e. g. node or cluster scope.

Using functional addressing, a client directly gains two advantages: First, his message is sent to the nearest destination delivering the specific service as quickly as possible; second, load balancing is achieved as TIPC automatically selects one of the service providers at the nearest distance in a round-robin manner.

2.2.3 Further Functionality

In addition to load balancing by using multiple service providers, described above, TIPC automatically shares network load across the available physical links, if they are set to the same *priority*. This way, the aggregated bandwidth is used. Furthermore, TIPC supports a mechanism for *link changeover*, which cares for the correct order of messages. This mechanism comes into action in case another link with the same priority comes up and even if one of the available links goes down or fails. TIPC *actively monitors* the state of peer nodes, allowing it to directly recognise failing nodes, nodes taken off the net, or nodes added to the net. In high-load situations, TIPC provides *flow control* effectively slowing down a fast sender, if the receiver is not able to cope with the amount of incoming data.

[5]The major difference is that TIPC attributes one address per node, while IP attributes one address per interface.

2.2.4 Interfaces to Upper and Lower Layers

TIPC provides a software layer between applications, interacting with TIPC from above, and packet transport services below. Although the protocol specification [13] does not make any assumptions on the API used for interacting with TIPC from above, the available implementations provide a standard socket API and a so called *native interface*. As transport service providers, the specification mentions various interfaces like Ethernet, protocols like DCCP and SCTP, and hardware mechanisms like mirrored memory. The combination of each of these transport services with a corresponding small software adapter are abstractly called *bearers*.

The only available bearer today in an existing TIPC implementation is the *Ethernet bearer*, although an IP bearer is in the works. For that reason, only Ethernet is currently supported as hardware transport.

3 Architecture of the Drivers

In order to describe the architecture of our Ethernet replacement drivers, an overview of the different hardware and software layers involved is given in section 3.1. Building on that, we describe our first development ETHOS, which uses sockets as "communication medium", in section 3.2. With this background, we go into some detail about the implementation and design decisions of our second approach ETHOM in section 3.3. Where appropriate, we cross reference to the ETHOS explanations.

3.1 Linux Ethernet Network Architecture

Seen from a rather high level of abstraction, the Linux Ethernet network architecture consists of three layers that are used by an application to communicate with a counterpart on another node (see Figure 3). The first layer consists of a programming interface like for example the well-known and wide-spread *sockets interface*. This layer is available for user-space application processes as well as for the kernel space. It expects user data in any form from the application and builds Ethernet frames which are passed to the next lower layer. The layer below is represented by the *Ethernet interface layer*. This layer takes Ethernet frames from above and passes them on to the hardware. It is usually implemented in the device driver for a specific Ethernet network interface card (NIC). The third and lowest is the *hardware layer*, as represented by the Ethernet NIC itself, which is responsible for the physical transmission of data from a sender to its peer.

3.2 Architecture of ETHOS

ETHOS is an Ethernet driver built to use the Linux kernel sockets API instead of real hardware to send and receive data. Looking again at Figure 3, you could think of the ETH interface layer being implemented by ETHOS

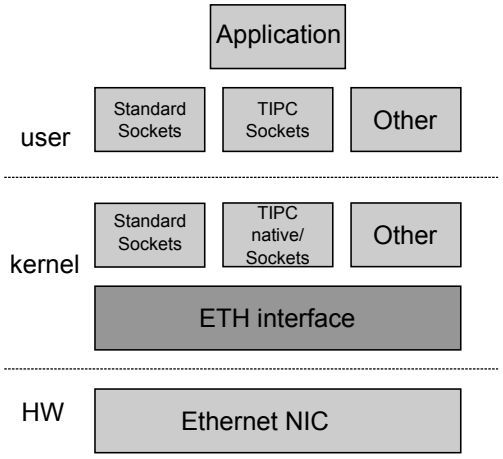

Figure 3: Linux Network Architecture

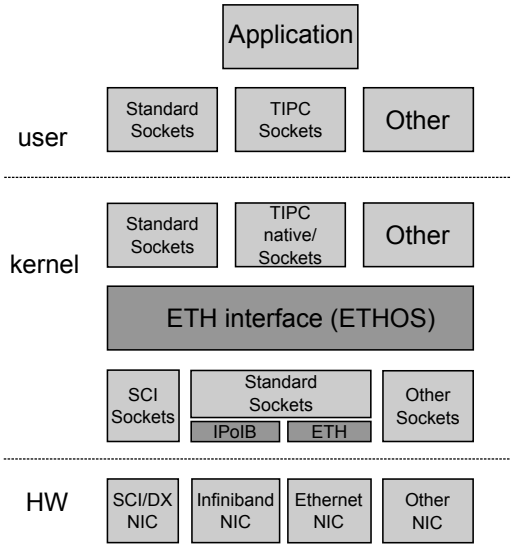

Figure 4: Network Architecture with ETHOS

instead of the NIC driver, and the hardware (HW) layer being replaced by another layer of sockets. These sockets must certainly have some hardware somewhere below in order to physically transfer data from one host to another. This hardware layer, however, can be provided by any network offering kernel-space UDP sockets, not only by Ethernet. If a network device does not offer native UDP kernel sockets but an IP driver, as depicted in Figure 4, standard UDP kernel sockets on top of this IP driver are deployed.[6]

In the following sections, we will go into some level of detail for three important aspects of ETHOS, the configuration, connection establishment, and the communication phase.

[6]Note that the Ethernet NIC shown in Figure 4 is used for testing and demonstration purposes only. It is in general not reasonable to use it for applications, as performance should by principle always lag behind the native interface.

3.2.1 Configuration

Simplified, configuration of ETHOS can be subdivided into three phases according to the question: "At what time are the parameters needed?" The categories would then be: at compile time of the driver, at load time of the driver, or at run time of the driver.

An incomplete list of necessary parameters contains the following: the socket family of the UDP socket to be used for low-level transmission, a unique host id and Ethernet "hardware address"[7], a transmit timeout value, the number of hosts in the network and their physical IP addresses, the number of ETHOS interfaces, and the IP addresses and Maximum Transmission Units (MTUs) of these ETHOS interfaces.

For simplicity of the implementation of this prototype, many parameters have to be specified at compile time. Exceptions are host id and transmit timeout, that have to be specified at load time, and the IPs and MTUs of the ETHOS interfaces, which can be specified at run time.

Summarising, before compiling the driver, a table has to be configured for every desired ETHOS interface, holding information about the socket family to be used for low-level communication and the IP addresses that all nodes in the given subnet can be reached by in the order of their `host_id`s. Again, these IP addresses are for the low-level interfaces, i. e. the addresses of the IP over InfiniBand (IPoIB) interfaces for communication over InfiniBand, the addresses that Dolphin's SuperSockets are configured with for SCI and DX, and the IP addresses assigned to the physical Ethernet devices for "Ethernet-over-Ethernet" communication (compare Figure 4 and the lower *Receive Socket* addresses in Figure 5). In order to be able to use the same binary on every node in the network, the `host_id` is specified as a module parameter; with this `host_id`, a unique Ethernet hardware address is generated. After loading the driver, the provided ETHOS interface behaves just like a normal Ethernet interface; IP addresses etc. can be configured with `ifconfig`.

3.2.2 Connection Establishment

After loading and configuring the local ETHOS interface, data can be sent to remote hosts. Looking at Figure 5, this could be an application on node 1 willing to send data to an application on node 2. The application on node 1 addresses its data to 192.168.0.5.

At this time, however, the kernel on node 1 does not yet know which node in the network provides an interface with IP address 192.168.0.5; it still has to find out, which Ethernet address to put as destination address into the

[7]With ETHOS, this is the 6-byte-address assigned to an ETHOS interface; i. e. not a real hardware address.

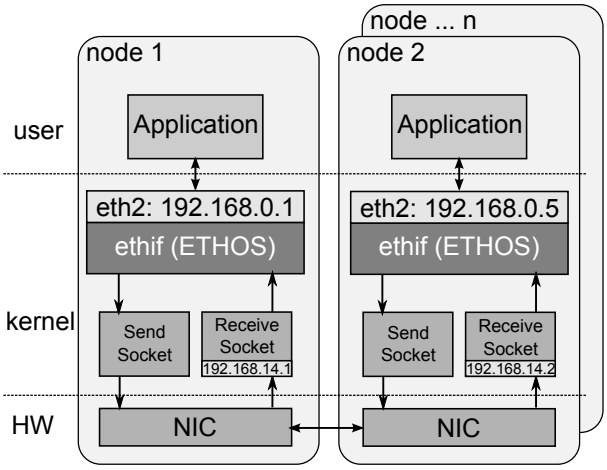

Figure 5: Implementation of ETHOS

Ethernet frame that it passes down to ETHOS. Therefore, prior to sending IP packets with data to the destination, the Address Resolution Protocol (ARP) is used to determine the Ethernet hardware address of the remote interface providing the requested IP. For this purpose, the kernel sends so called ARP requests, Ethernet frames with the hardware address ff:ff:ff:ff:ff:ff, that ETHOS forwards to all hosts in the network. The interface providing the IP address encapsulated in the request answers with its hardware address and after that the correct mapping between destination's IP address and Ethernet hardware address is known at the sending kernel.

3.2.3 Communication Phase

We will now have a closer look at what happens when a node transfers data to another node in the cluster. Figure 6 gives a quite detailed overview of the internal architecture of ETHOS. This illustration will serve as a basis for the following explanation.

Sending Figure 6 shows two nodes, one with ETHOS host ID 1 on the left – the sending node – and one with ETHOS host ID 4 on the right side – the receiver. Inside of each host, various elements involved in the send and receive process are depicted, roughly subdivided into the user-space elements at the top, kernel-space elements in the middle, and the hardware at the bottom. The transfer starts at the left node's top – a user-space application has opened a TCP socket and starts writing data to it – and it ends on the right node's top, where a receiving user-space application reads the incoming data again from a TCP socket. ETHOS host 1 provides two Ethernet interfaces (eth2 and eth3), which are each configured with an IP address, a corresponding network, and a hardware address. The same applies to ETHOS host 4 with Ethernet interfaces eth3 and eth5.

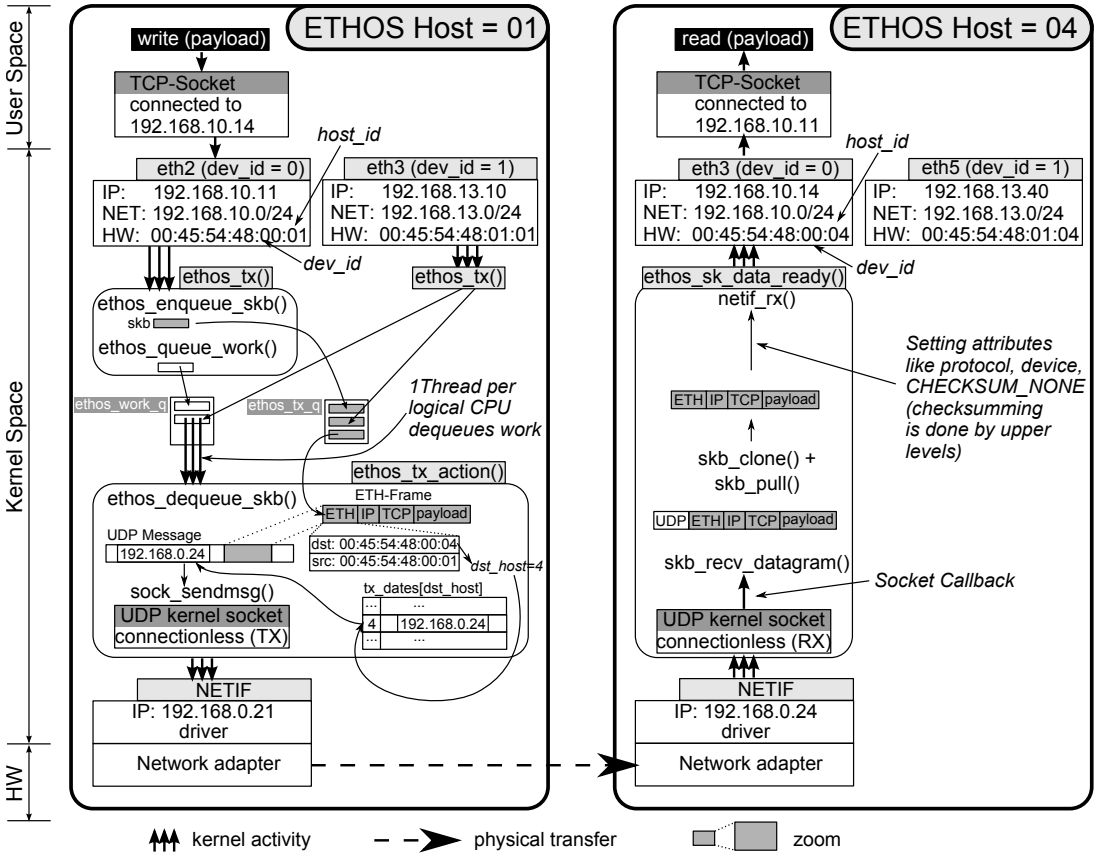

Figure 6: Data Transfer through ETHOS from Sender to Receiver

In the example provided, user data shall be sent over the TCP socket connected to 192.168.10.14 on ETHOS host 4; therefore Ethernet interface eth2 is chosen. The user data is split by the kernel into packets (Ethernet frames) the size of which must not exceed the specified maximum transmission unit (MTU) of interface eth2. These packets are then handed over to the driver of this Ethernet interface, i. e. ETHOS.

Every single Ethernet frame is encapsulated in a socket buffer (skb), which is passed to ETHOS by the kernel calling the `ethos_tx()` function. This way, ETHOS receives an Ethernet frame, that can be transferred via a UDP kernel socket. As `ethos_tx()` is executed in the so called *interrupt context* (see [18] and [19] for details), the Ethernet frame cannot be sent by `sock_sendmsg()`, because this function might *sleep*, which is not allowed in interrupt context. To circumvent this problem, the skb is enqueued in an `ethos_tx_queue` with `ethos_enqueue_skb()`. This `ethos_tx_queue` is a single queue per driver and therefore accepts skbs from other ETHOS interfaces as well. In order to initiate further processing of the enqueued data, a `work_struct` is enqueued in the `ethos_work_q` at the end of `ethos_tx()`.

The `ethos_work_q` is a *Linux Kernel Work Queue* [18] and is used in order to postpone the execution of tasks that must not be executed from within `ethos_tx()` until a later time. It is important to mention that the `ethos_work_q` is processed by the Linux kernel very soon after `ethos_tx()` returns; i. e. this does not cause too much added latency.

After termination of `ethos_tx()` and possibly other interrupt handlers, a kernel thread dequeues the work struct from the `ethos_work_q` and calls the embedded handler, in this case `ethos_tx_action()`. The handler function is executed, and it starts dequeuing and processing skbs until `ethos_tx_q` is empty. If the host possesses more than one execution unit (CPU, core, or logical CPU) and there are more than one work structs in the `ethos_work_q`, each of these execution units could possibly run one kernel thread dequeuing work simultaneously. For every skb that `ethos_tx_action()` pulls from the queue, the encapsulated Ethernet frame is inspected. The Ethernet frame consists of the Ethernet header and Ethernet data, that in turn in our example in Figure 6 consists of an IP header, followed by a TCP header and the user data that our application wants to transfer. In order to determine the IP address of the receiving host, the destination hardware address (dst) as part of the Ethernet header is examined; more specifically the `host_id` part (*dst_host*) residing in the last byte of this address (see Figure 6, inside `ethos_tx_action()` of

the left node).

Looking at the Ethernet frame (ETH-Frame) in the picture, one could ask why not simply take the IP address (192.168.10.14) available in the IP header as distinguishing factor to determine the destination node's IP address. The reason is simple: As we want to support other protocols on top of Ethernet, we can not count on an IP header inside the Ethernet frame. With TIPC for example, the addressing is different and there is no IP header, that we could use. The Ethernet hardware address on the other hand is always available.

The destination hardware address in our example is the hardware address of the receiver's NIC (eth3), i. e. 00:45:54:48:00:04. The last byte of this address indicates the ETHOS destination host (`dst_host=04`). The last byte of the source hardware address 00:45:54:48:00:01 represents the ETHOS source host (`src_host=01`). These hardware addresses are assigned during initialisation of the driver; every ETHOS interface (e. g. eth2, eth3 on host 1 in Figure 6) in the whole network possesses a unique address. After determining the destination host (`dst_host=4`), the IP address of the specific destination NIC can be read from a lookup table; this table is specified during the configuration phase (see section 3.2.1). The IP address is written to the header of a UDP message, which is finally sent through a UDP kernel socket with `sock_sendmsg()`.

If the software stack of the NIC provides kernel-space UDP sockets, as does the Dolphin stack for SCI and DX, this UDP kernel socket can be used to transfer the Ethernet frames directly via the high-speed interconnect.

Receiving Our message has now left host 1 and is physically transferred to host 4 using whatever kind of network offered the UDP kernel socket interface. When the UDP message arrives on host 4, ETHOS could use the function `sock_recvmsg()`, which blocks until a message has been received. However, as a long time (typically about 1 ms) may pass between message reception and the return of `sock_recvmsg()`, a faster asynchronous mechanism is used for fetching the message, which is received by the UDP kernel socket.

On the receiving side, ETHOS uses a *call-back mechanism* in order to get directly informed of arriving messages. To achieve that, a *call-back function* is registered with the UDP kernel socket. For this purpose, the socket provides the function pointer `sk_data_ready()`, which is called by the kernel as soon as a message is available. The call-back function is registered at loading time of the ETHOS driver, where our function `ethos_sk_data_ready()` replaces the standard `sk_data_ready()` function.

When a message arrives at the receiving UDP kernel socket, the first thing that `ethos_sk_data_ready()` does is to fetch the message with `skb_recv_datagram()`. The UDP message or *UDP datagram*, which is encapsulated in an skb consists of a UDP header and UDP data, where the UDP data is nothing else but the Ethernet frame that has been sent through the network from the sending host.

The UDP header provides a checksum that could be used to proof the integrity of the UDP message. However, the encapsulated user data – which is still prepended by the Ethernet, IP, and TCP header – is secured by a second checksum on the Ethernet frame level[8]. The IP header inside the Ethernet frame is again protected by its own checksum, and the whole TCP packet is protected by another checksum residing inside the TCP header. Again, we can not count on TCP/IP being used on top of our Ethernet driver, but we certainly transfer Ethernet frames. For that reason, the additional checksumming on the lower UDP level is not necessary, and we can strip off the UDP header without verifying the checksum.

The received skb is cloned with `skb_clone()` [19], i. e. the administrative data is copied so that it can easily be adapted, while the Ethernet frame is referenced by pointers. After that, the UDP header is stripped off by calling `skb_pull()`, effectively moving the start pointer from the beginning of the UDP header to the beginning of the Ethernet header. As shown in Figure 6, our message now consists of the Ethernet, IP, and TCP headers followed by the original user data.

At the end of the function `sk_data_ready()`, the attributes of the skb clone are set; `dev`, `protocol`, `ip_summed` are adapted, so that the kernel level above ETHOS accepts the skb. A very important part of that is to set `ip_summed` to `CHECKSUM_NONE` in order to inform the upper level that checksumming was not performed by ETHOS and therefore has to be done by the upper level. At last, the skb clone is passed upwards by the function `netif_rx()` and the original skb is freed. Freeing the clone and its data (the Ethernet frame) is done by the the upper kernel levels.

3.3 Architecture of ETHOM

As our second approach, a thinner layer of indirection is inserted below the Ethernet interface (see Figure 7). This layer passes the Ethernet frames to the *SCI Message Queues*, which represent the lowest message passing layer of the Dolphin software stack (compare section 2.1.3 and Figure 1), sacrificing compatibility with other high-speed interconnects for better performance at the additional cost of higher system load. At the lowest level, SCI or DX cards physically deliver the data to the peer nodes.

As depicted in Figure 8, on each node of the cluster there are two message queues for every peer. Each message

[8] To be precise, this checksum is located at the end of the Ethernet frame. For simplicity reasons it is not shown in our Figure.

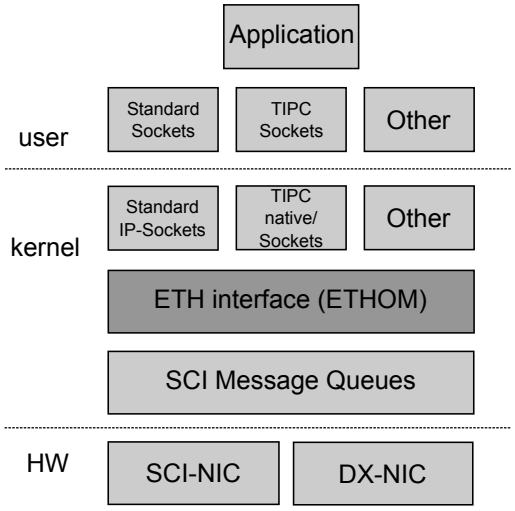

Figure 7: Network Architecture with ETHOM

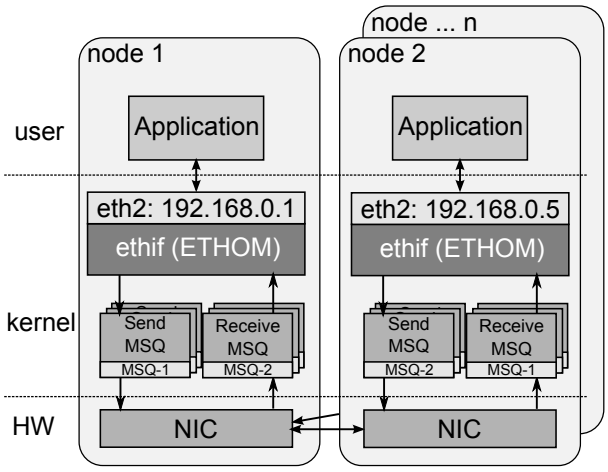

Figure 8: Implementation of ETHOM

queue is identified by a cluster-wide unique ID.

Memory requirements for each message queue are dominated by the buffer, which is currently hard-coded to 13 KB. As the SCI network allows a maximum of 512 nodes when setup as a 3D torus, total memory requirement per node would amount to: number of peers × 2 queues × size of queue = 511 × 2 × 13 KB = 13 MB. So, concerning memory consumption, our driver should be quite scalable.

As it is very improbable to have more than one card of either SCI or Dolphin DX inside of one node, currently only one Ethernet interface is supported by ETHOM, although most of the code is already prepared for using multiple.

3.3.1 Configuration

Just like ETHOS, ETHOM is configured in three phases: at compile time, at loading, and at run time of the driver. For simplicity reasons, basic configuration is rather static; number of peers in the network and their *ETHOM host_id* to *SCI node IDs* mapping have to be specified at compile time. At load time, most importantly the ETHOM `host_id` has to be passed as a parameter enabling the use one binary for all hosts in the network. Optionally, *direct flushing* after each call to `send_msg()` can be enabled for the sender side, *dynamic polling* for the receive thread. A transmit timeout can be specified that tells the kernel after which period of time to drop packets. With the above mentioned parameters, the Ethernet interface is set up and ready to go. The IP address, MTU etc. can be assigned at run time with `ifconfig`. All module parameters specified at load time can be changed at run time.

3.3.2 Connection Establishment

As shown in Figure 8, after loading the driver, on each node two unidirectional message queues are created for every peer node in the network (e. g. 14 message queues on each node in case of 7 peer nodes). Message queue IDs are calculated from the local and the peer node number as

$$\begin{aligned} \mathrm{ID}_{\mathrm{ReceiveQueue}} &= \#\mathrm{hosts} \times \mathrm{peer} + \mathrm{local} \\ \mathrm{ID}_{\mathrm{SendQueue}} &= \#\mathrm{hosts} \times \mathrm{local} + \mathrm{peer} \end{aligned}$$

This way they are guaranteed to be unique throughout the cluster.

For each peer node, two threads are started (e. g. 14 threads on each node in case of 7 peers), one trying to connect the local send to the distant receive queue and one waiting for a connection on the local receive queue. As soon as the first of the threads waiting on the local receive queue has accomplished its connection, this thread becomes the *master thread* that polls on all connected receive queues. All the other send and receive threads terminate as soon as their connection is established, effectively reducing the number of remaining threads to one. In case the peer node does not connect directly, a new connection attempt is made periodically.

In case of IP communication on top of ETHOM, IP addresses can be specified arbitrarily, they do not have to correspond to node numbers. Just like with hardware Ethernet devices and as described in the corresponding ETHOS section 3.2.2, the ARP protocol is used at first contact to find the node that provides the sought-after IP address. The ARP broadcast request is sent to each connected node sequentially, as Dolphin's message queue

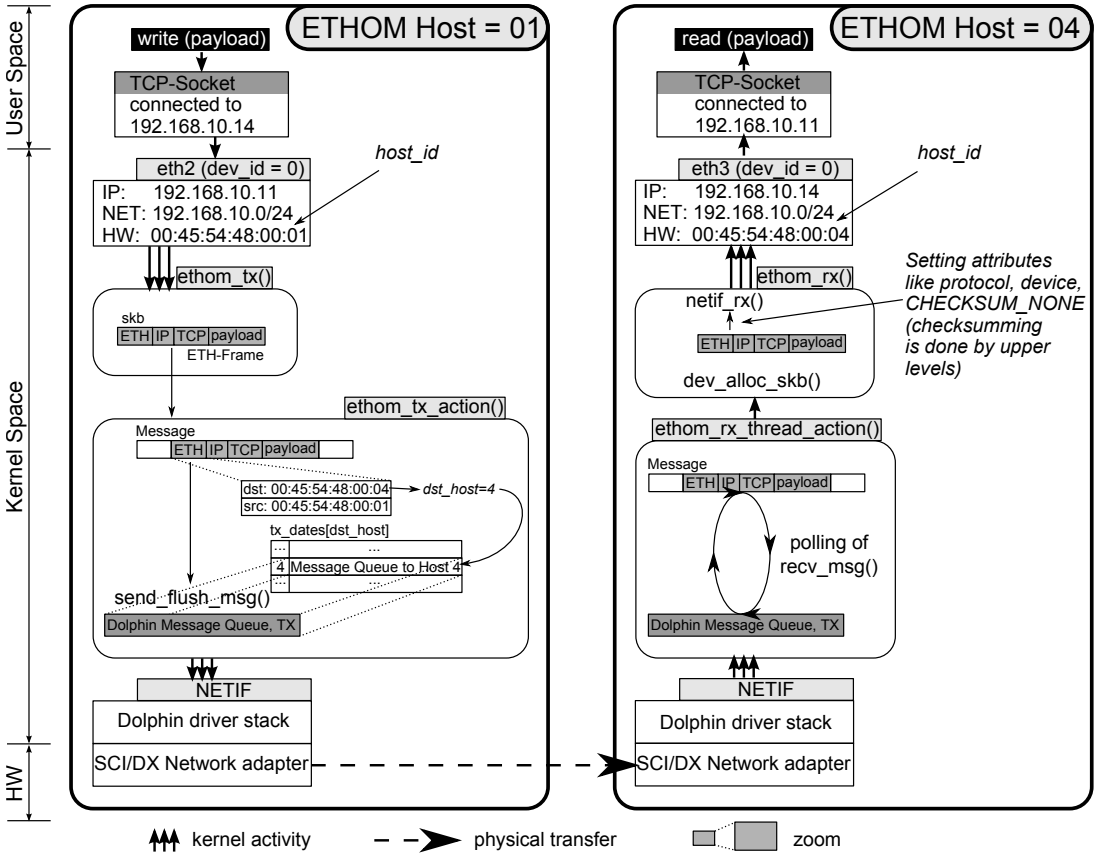

Figure 9: Data Transfer through ETHOM from Sender to Receiver

API does not provide a broadcast function, and answered by the node in question. From that point on, communication over IP is possible.

3.3.3 Communication Phase

Exchanging data between two nodes in a network is described on the basis of Figure 9. Exactly as in Figure 6, an application on ETHOM host 1 on the left sends data through a TCP socket to an application on ETHOM host 4 on the right.

Sending When an application on host 1 writes data to a TCP socket connected to a receiver on host 4, this data is passed to the kernel networking stack. The kernel then splits it into packets fitting into the previously specified MTU (Fragmentation) – if necessary – and equips each packet with an Ethernet header. This newly constructed *Ethernet frame* is passed to ETHOM by calling its `ethom_tx()` function. There, the minimum length of the packet is checked and if needed padding bytes are added, before the Ethernet frame is given to `ethom_tx_action()`. In `ethom_tx_action()`, the last byte of the destination hardware address (indicating `dest_host`, here "04") encapsulated in the Ethernet header is used to find the send (TX) message queue which is connected to the receive (RX) message queue on the destination host. Depending on the `flush` parameter either `send_msg()` or `send_flush_msg()` is called to forward the message to the Dolphin driver stack and finally the hardware. `send_msg()` should be beneficial for data throughput, while `send_flush_msg()` – which we chose for our measurement and general operation – should reduce latency.

Comparing this send process to the ETHOS send process, no asynchronous forwarding of the message between `ethom_tx()` and `ethom_tx_action()` is needed, as the low-level functions used to finally send the message (either `send_msg()` or `send_flush_msg()`) are completely non-blocking. Hence, we do not have to leave interrupt context.

Receiving On the receiving side on host 4 on the right hand side in Figure 9, the data is directly written to the message queue's data space in main memory by the Dolphin hardware; no interrupt is called to signal the arrival of data. As described in section 3.3.2, a thread is started executing the function `ethom_rx_thread_action()` that either dynamically or not polls on the receive message queue. This thread, repeatedly calling Dolphin's

`recv_msg()` function fetches the data shortly after arrival and passes it upwards to `ethom_rx()`. In `ethom_rx()`, an skb structure is allocated with `dev_alloc_skb()`, the same attributes are set as in the ETHOS case (compare section 3.2.3) and the Ethernet frame is passed upwards with `netif_rx()`. Here, the IP packets are reassembled from several Ethernet frames (if they were fragmented before), IP and TCP headers are stripped off again and the user data reaches its final destination, the application on host 4.

By handling Ethernet frames from above layers, ETHOM – unlike IP interfaces like IPoIB – directly supports all protocols that rely on Ethernet (like e. g. TIPC) in addition to IP.

As the Dolphin message queue API does not support broadcasting of data to all connected peers, broadcast is implemented in ETHOM. Currently, data is sent to each peer sequentially in a simple Round-Robin fashion.

In case of a node failure or shutdown, all other nodes continue working as before. Reconnection of message queues as soon as a node comes up again is not yet implemented, though.

4 Experimental Results

Measurements were performed on two clusters as SCI and DX cards are built into separate clusters:

1. The first cluster, called Xeon throughout this paper, consists of two nodes equipped with two Intel Xeon X5355 four-core CPUs running at 2.66 GHz. The mainboard is an Intel S5000PSL with two on-board Gigabit Ethernet controllers (Intel 82563EB). Each node is equipped with a DX adapter from Dolphin (DX510H, 16 Gb/s) in a PCIe x8 slot, and an InfiniBand adapter from Mellanox (MHGS18-XTC DDR, 20 Gb/s), which is plugged into a PCIe x8 slot, too. The DX cards are connected directly without an intermediate switch, while Gigabit Ethernet and InfiniBand use a switch; Cisco Catalyst 2960G-24TC-L (Ethernet) and Mellanox MTS-2400-DDR (InfiniBand).

2. The second cluster, called PD, consists of 16 nodes, two of which were used for measurements. Each node features a Pentium D 820 dual-core CPU running at 2.8 GHz. The mainboard is from ASUSTek (P5MT-M). It is equipped with two on-board Gigabit Ethernet controllers (Broadcom BCM5721). In addition to that, each node is equipped with an SCI card from Dolphin (D352, 10 Gb/s), which resides in a PCIe x4 slot, and with an InfiniBand adapter from Mellanox (MHGS18-XTC DDR, 20 Gb/s), which is plugged into a PCIe x8 slot. The SCI cards are connected in a 4x4 torus topology.

All nodes run an unpatched kernel 2.6.22, 64-bit on Xeon and 32-bit on PD. Kernel preemption was enabled. For InfiniBand, we used the drivers included in kernel 2.6.22, for SCI and DX, version 3.3.1d of Dolphin's software stack.

In order to measure the performance of our driver in comparison with hardware drivers for Ethernet and InfiniBand, we measured TCP socket performance, concentrating on latency (see section 4.1.1) and bandwidth (see section 4.1.2) for various message sizes. In addition to that, we performed two measurements with TIPC replacing TCP/IP (see section 4.2) in order to see what performance to expect from our approach to enable TIPC over high-speed interconnects.

For the TCP experiments, we used NPtcp from the NetPIPE[9] suite in version 3.7.1, which is described in [20] and Dolphin's sockperf[10] in version 3.3.1d as it records information about interrupt and CPU usage. TIPC performance was measured with tipcbench[11], which is part of the "TIPC demo v1.15 package".

After testing with several different MTU settings, we chose to use the biggest possible MTU for ETHOS and ETHOM, as it does not have a negative effect on latency but proved positive for bandwidth. Apart from that, we did not touch the default settings for the other parameters of our Ethernet, SCI, DX, and InfiniBand NICs (like interrupt coalescing, message coalescing, and any other).

Currently, we use a protocol of the message queues which is restricted to 8 KB, therefore the largest MTU for ETHOM is 8 KB minus header length at the moment. We expect greatly improved bandwidth when usage of a protocol supporting larger messages is implemented.

4.1 TCP/IP Benchmarks

As TCP/IP is far more wide spread than TIPC, our first curiosity concerned basic benchmark performance using this protocol suite. We focus on latency and bandwidth and then give some information on system resource utilisation.

4.1.1 Latency

In Figure 10, the round-trip latency (RTT/2) for messages of varying sizes measured with NPtcp is shown.

The green curve represents Gigabit Ethernet, the reference that ETHOS and ETHOM compete with. The lowest latencies are delivered by ETHOM on SCI, followed

[9] The NetPIPE benchmark suite is available for download at `http://www.scl.ameslab.gov/netpipe/`

[10] The sockperf benchmark is part of the Dolphin Driver Package available from `http://www.dolphinics.com`

[11] The tipcbench benchmark as part of the TIPC demo package is available at `http://tipc.sourceforge.net/tipc_linux.html`

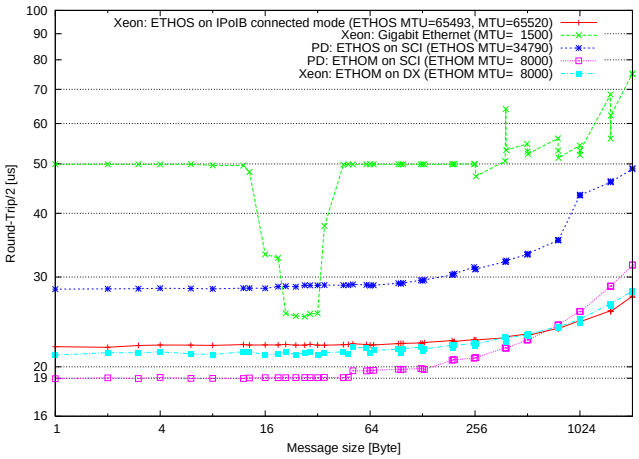

Figure 10: Latency measured with NPtcp

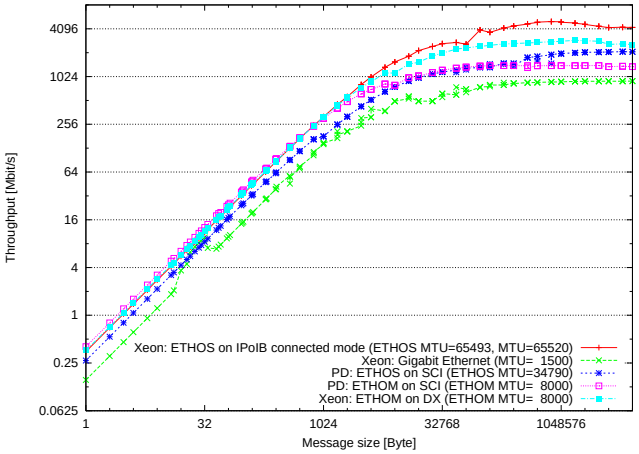

Figure 11: Throughput measured with NPtcp

by ETHOM on DX; the highest times are the Ethernet times. A dramatic decrease in latency can be seen for Ethernet with message sizes between 16 and 48 B, which indicates polling for new messages on the receiving side. For larger messages, the high raw bandwidths of InfiniBand and DX lead to lower latency as for SCI. Comparing ETHOM on SCI with ETHOS on SCI, an improvement in latency of around 10 μs for small messages and around 15 μs for larger ones can be observed.

Summarising, ETHOM on SCI provides an improvement in latency by a factor of two and above on our measurement platform over Gigabit Ethernet and about a 30% improvement over its companion ETHOS.

4.1.2 Bandwidth

Figure 11 shows the bandwidth for varying message sizes measured with NPtcp.

Gigabit Ethernet delivers for all message sizes the lowest bandwidth (excluding the aforementioned interval between 16 and 48 B). For small messages, ETHOM on SCI performs best with ETHOM on DX and ETHOS on IPoIB close by. At about 1 KB, the three curves split again each gradually approaching its maximum, which is at 1.5 Gb/s for SCI and 3 Gb/s for DX (with their current limitation to an MTU of 8 KB) and about 5 Gb/s for ETHOS on InfiniBand. Comparing ETHOM with ETHOS on SCI, it can be noticed that for small messages (until 8 KB) ETHOM provides a 50% increase in bandwidth. For large messages (256 KB and above) ETHOS benefits from the support for larger low-level packets and maybe additional buffering in the sockets layer.

To sum up, ETHOM on SCI exhibits a twofold increase in bandwidth for messages up to 1 KB over Gigabit Ethernet and about a 50% increase over ETHOS.

4.1.3 Interrupts and CPU Utilisation

In this section, we show the drawbacks of ETHOM, which uses polling and therefore a higher system load to achieve its high performance. Figure 12 and Figure 13 have to be examined together in order to get some meaningful statement. In Figure 12 the number of interrupts triggered by each device are recorded over the message size.

Two points are immediately eye-catching:

1. ETHOM on SCI and DX does not trigger any interrupts, which is obvious considering that ETHOM uses polling mechanisms instead of relying on an interrupt.

2. With 45000 IRQs/s, ETHOS on InfiniBand puts by far the highest load onto the IRQ-processing routines, sharply decreasing with messages bigger than 512 Byte. At a second glance, it can be seen that our Ethernet adapter is limited to 20000 IRQs/s, which is a clear indication of coalescing intermediate interrupts.

As mentioned before, we have to keep the interrupts in mind when discussing the system load. Figure 13 shows only the system time and not the time needed for IRQ processing. Several aspects are worth mentioning here: First of all, ETHOM on SCI has a very high system load, as one thread is constantly polling for new messages, effectively occupying one of the 2 cores available on each node of PD. The curve for ETHOM on DX, which is measured on the 8-core Xeon system, indicates that a multi-core platform is a much better basis for our polling approach and alleviates the high load. The system load amounts to 12.5%, which again reflects one core fully occupied with polling, and there is very low IRQ load to be expected as no interrupt is triggered by the DX adapter.

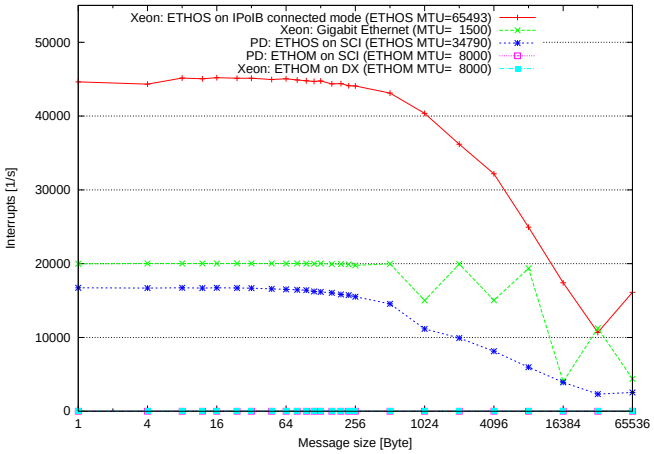

Figure 12: Interrupts measured with sockperf

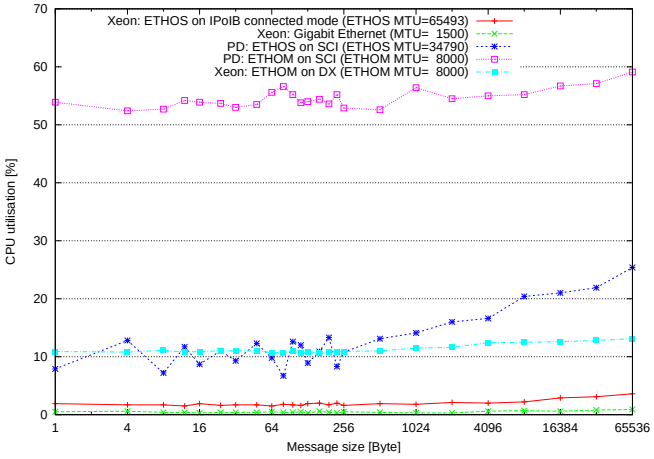

Figure 13: Systime measured with sockperf

The next point worth mentioning is the very low system time for Gigabit Ethernet and ETHOS on InfiniBand. On InfiniBand, due to the high interrupt rate, non-negligible time will be spend in the interrupt routines, but on Ethernet, the overall system load in general is very low.

The unsteadiness of the ETHOS on SCI curve seems to stem from changing between polling and interrupt mode in Dolphin's SCI sockets, which ETHOS uses.

System utilisation can be summarised with very low load for Gigabit Ethernet, moderate load for ETHOS on SCI and ETHOS on InfiniBand. At the other end is ETHOM, which causes a very high load. ETHOM will clearly benefit very much from upcoming *many-cores*[12].

4.2 TIPC Benchmarks

As one of the main aims in our motivation was to speed up communication over TIPC, we finally measured performance of Ethernet, SCI, DX, and InfiniBand with the TIPC protocol replacing TCP/IP. The only "semi-official" benchmark, that was publicly available at the time of our measurements, is *tipcbench*.

[12] *Many-core* or sometimes *massively multi-core* is the term used for a very high number of cores per CPU die.

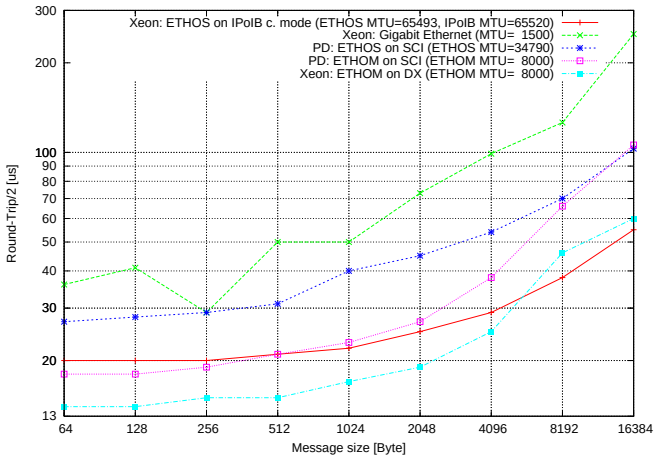

Figure 14: Latency measured with tipcbench

4.2.1 Latency

In Figure 14, the latency for varying message sizes measured with tipcbench is depicted.

With the TIPC protocol, ETHOM on DX reaches the lowest latencies we ever measured, amounting to 14 μs. ETHOM on SCI lies at 19 μs for small messages. Gigabit Ethernet has the highest latencies for all message sizes. ETHOS on InfiniBand has a latency of 20 μs for small messages, starting from 8 KB it has the lowest latency of the measured interconnects.

Comparing ETHOM and ETHOS on SCI, a decrease in latency of between 10 and 20 μs can be observed for messages smaller than 8 KB. For 8 KB and above the bigger packet size supported by ETHOS leads to a latency which is on par with ETHOM.

4.2.2 Bandwidth

Figure 15 shows the bandwidth for varying message sizes measured with tipcbench. Several facts are interesting about this measurement:

1. Gigabit Ethernet provides the best throughput for small messages, has a significant decrease at a message size of 4 KB and reaches a maximum of clearly below 500 Mb/s. It is not clear to us why Ethernet performs so well for small messages in this test, but we suspect that this is influenced by polling and very efficient buffering. The fact that the maximum throughput is rather low indicates that the strength

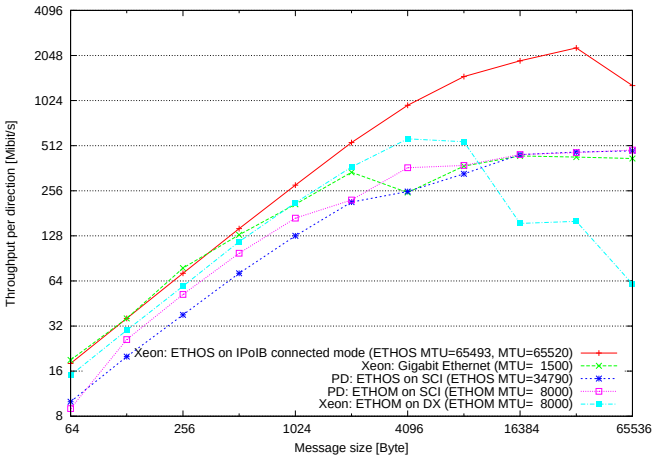

Figure 15: Bandwidth measured with tipcbench

of the TIPC protocol lies in low latency rather than high bandwidth.

2. The severe decrease in bandwidth of ETHOM on DX for messages of 8 KB and above is eye-catching. Part of the problem should be related to the limitation of 8 KB for packets that can in our implementation currently be sent via message queues. Another part seems to be the way that ETHOM deals with send failures. As this problem does not appear with NPtcp, we suspect another part of the problem in the way TIPC uses the ETH interface or in the way tipcbench works. We are still investigating this issue and hope for improvement when we get rid of the "8 KB limit".

Comparing ETHOM and ETHOS on SCI, we see an improvement in bandwidth for practically all message sizes. An increase for message sizes above 8 KB is expected after modification of ETHOM.

Summarising the TIPC benchmark results, we can recapitulate that ETHOM on SCI and DX shows a very low latency. Compared with Ethernet, it provides a decrease by the factor 2 to 3 for small messages and up to 4 at 4 KB. Concerning bandwidth, Ethernet performs surprisingly well for small messages. Tipcbench and the effects with ETHOM on DX have to be studied more thoroughly.

5 Conclusions and Outlook

The tests performed within the scope of this article show that ETHOS and ETHOM – making use of a high-speed interconnect like InfiniBand, SCI, or Dolphin DX – provide solutions that offer better performance than Gigabit Ethernet, latency wise and bandwidth wise. Regarding the completely different price range of Gigabit Ethernet and these high-speed interconnects, this comparison is only reasonable, when low-latency (and maybe high-bandwidth) Ethernet interfaces are required, that can not be provided by Gigabit Ethernet.

Comparing the results of ETHOS and ETHOM, we observe a 30%-70% improvement in bandwidth for small to medium-sized messages and about a 30% decrease in latency, when SCI is used.

The advent of *many cores* should have a twofold positive effect on ETHOM:

1. The network should become an even more constraining bottleneck for communicating applications, as the connection is shared by a larger number of cores; so better communication performance is highly appreciated.
2. Having a smaller ratio between the one core sacrificed for communication and the number of cores still available for computation reduces the relative communication overhead.

With ETHOM, we present a driver for Linux, that makes effective use of the lowest message passing layer of the Dolphin software stack. Processing Ethernet frames from the layer above, it enables a potentially wide range of software to make use of Dolphin's high-speed networks. It has a low overhead and is small with about 1000 lines of code. ETHOS on the other hand offers good performance for a broad range of high-speed interconnects.

Currently, ETHOM fulfils our main aim to enable TIPC – and any other software communicating via Ethernet frames – to use SCI and DX. Besides ETHOS, it provides the only Ethernet interface for SCI and DX; as a side effect, support for IP-routing is now offered using the standard kernel IP stack on top of ETHOM.

On the other hand, porting software to the native interfaces of high-speed interconnects almost always provides better performance and efficiency at runtime – obviously at the cost of porting effort. As usual, it remains to the user to balance the pros and cons.

Having succeeded to let TIPC run on top of InfiniBand, SCI, and DX, our next goal is to sacrifice compatibility to Ethernet (as the upper interface layer) and design a native TIPC bearer for SCI and DX. This way, we hope to further improve performance. Apart from that, the effect of the performance improvement onto applications and higher-level Single System Image (SSI) functionality will be studied.

References

[1] InfiniBand Trade Association, "Infiniband Architecture Overview." `http://www.infinibandta.org/events/past/it_roadshow/overview.pdf`, 2002.

[2] Myricom Inc., "Myrinet 2000 Product List." http://www.myri.com/myrinet/ product_list.html, 2008.

[3] Quadrics Ltd., "Quadrics QsNetII." http://www.quadrics.com, 2003.

[4] Dolphin Interconnect Solutions, "The Dolphin SCI Interconnect." http://www.dolphinics.com, 1996.

[5] Dolphin Interconnect Solutions, "The Dolphin DX Interconnect." http://www.dolphinics.com/products/ pent-dxseries-dxh510.html, 2007.

[6] V. Krishnan, "Towards an Integrated IO and Clustering Solution for PCI Express," in *Proc. IEEE International Conference on Cluster Computing (CLUSTER'07)*, (Austin, Texas), Sept. 2007.

[7] S. Fu and M. Atiquzzaman, "SCTP: state of the art in research, products, and technical challenges," in *Computer Communications, 2003. CCW 2003. Proceedings. 2003 IEEE 18th Annual Workshop on*, pp. 85–91, 2003.

[8] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)." http://ietfreport.isoc.org/rfc/PDF/rfc4340.pdf, 2006.

[9] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)." http://ietfreport.isoc.org/rfc/PDF/rfc3828.pdf, 2004.

[10] S. Hopkins and B. Coile, "AoE (ATA over Ethernet)." http://www.coraid.com/site/co-pdfs/AoEr10.pdf, 2006.

[11] J. Maloy, "TIPC: Providing Communication for Linux Clusters," in *Proceedings of the Ottawa Linux Symposium*, pp. 347–356, 2004. http://www.linuxsymposium.org/proceedings/ LinuxSymposium2004_V2.pdf.

[12] A. Stephens, J. Maloy, and E. Horvath, "TIPC Programmer's Guide." http://tipc.sourceforge.net/doc/ tipc_1.7_prog_guide.pdf, 2008.

[13] J. Maloy and A. Stephens, "TIPC Specification." http://tipc.sourceforge.net/doc/ draft-spec-tipc-02.html, 2006.

[14] The Kerrighed Team, "Kerrighed: a Single System Image operating system for clusters." http://www.kerrighed.org, 2008.

[15] H. Hellwagner and A. Reinefeld, eds., *SCI: Architecture and Software for High Peformance Compute Clusters*, vol. 1734 of *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1999.

[16] IEEE, *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI)*, 1992.

[17] Dolphin Interconnect Solutions, *SISCI Interface Specification 2.1.1*, May 1999.

[18] R. Love, *Linux Kernel Development (2nd Edition)*. Novell Press, 2 ed., 2005.

[19] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly Media, Inc., Dec. 2005.

[20] Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performace evaluator," in *In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, 1996. http://www.scl.ameslab.gov/netpipe/paper/ full.html.