Chapter 46
Transparent Integration of a Low-Latency Linux Driver for Dolphin SCI and DX

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

Abstract High-speed interconnects like Dolphin's SCI and DX fulfil even high communication performance requirements. One of the prerequisites, though, is that the communication software must be either based on IP sockets or specifically adapted to the interconnect. Software written directly for Ethernet, arguably the most widespread interconnect today, cannot profit from this fast hardware. In this article, we present a Linux driver that fills this gap by allowing transparent usage of Dolphin hardware. ETHOM provides an Ethernet interface and makes use of the lowest message passing layer of Dolphin's driver stack in order to exchange Ethernet frames. It enhances the functionality of SCI and DX by offering an Ethernet and with that an IP interface.

Keywords Ethernet · SCI · Dolphin DX · Linux · TIPC

46.1 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 nodes – in these clusters were connected by Ethernet in one of its incarnations. Concerning the software, the predominant protocol used on top of Ethernet is the TCP/IP stack. With software running on

R. Finocchiaro (✉), L. Razik, S. Lankes, and T. Bemmerl
Chair for Operating Systems, RWTH Aachen University, Kopernikusstr. 16, 52056 Aachen, Germany
e-mail: finocchiaro@lfbs.rwth-aachen.de; razik@lfbs.rwth-aachen.de; lankes@lfbs.rwth-aachen.de; bemmerl@lfbs.rwth-aachen.de

the cluster that communicates intensively, the network more and more becomes the limiting factor of overall cluster performance.

So, there are two problems to cope with: (1) Networking hardware in the form of Gigabit Ethernet is too slow for several purposes; 10 Gigabit Ethernet is still in the beginnings and rather expensive. (2) Then, TCP/IP is a protocol suite designed for communication in wide area networks, offering elaborate mechanisms for routing, to deal with even extensive packet loss, etc. It is not so well suited for clusters.

To overcome these problems, there are mainly two approaches in order to allow faster communication (latency and bandwidth wise):

1. Usage of high-speed networks, each having their own low-level programming interface (API), most providing an implementation of the POSIX socket API, and some offering an IP interface. Examples of these networks include InfiniBand [10], Myrinet [15], QsNet [16], SCI [3], and Dolphin DX [4]. An IP interface for Dolphin DX has been presented in [12].
2. Replacing the software layer TCP, UDP – and sometimes IP as well – while keeping the Ethernet hardware. Examples of these replacement protocols include SCTP (Stream Control Transmission Protocol [7]), DCCP (Datagram Congestion Control Protocol [11]), UDP-Lite [13], AoE (ATA over Ethernet [9]), and TIPC (Transparent Interprocess Communication Protocol [14, 17]).

Being developed originally at Ericsson, the abovementioned TIPC has its origin 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 the prospect of less overhead for exchanging data within a cluster. TIPC is the transport layer of choice of the Kerrighed project [18], where it is used for kernel to kernel communication. Currently it cannot make use of high-speed networks like InfiniBand, SCI, or DX, as neither do they provide an Ethernet interface, nor does TIPC provide a specialised "bearer", which is the adaptation layer between TIPC and a native network interface.

A first approach to enable TIPC to make use of high-speed networks is described in [5], where we elaborate on ETHOS, an Ethernet driver built using Linux kernel-space UDP sockets to send and receive data. ETHOS therefore directly supports almost all high-speed interconnects. 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, *ETHOM (ETHernet Over Message-Queue driver)* provides an Ethernet interface for SCI and Dolphin DX hardware. Therefore, in addition to the TCP/UDP-Sockets already provided by the Dolphin 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.

In the next section, we give a short overview about the hardware that we enable to be used as Ethernet replacement. Section 46.3 elaborates on the design and the architecture of our driver and in Section 46.4, we provide some basic experimental results. We conclude with the current status in Section 46.5.

46.2 Dolphin's High-Speed Interconnects

46.2.1 Scalable Coherent Interface (SCI)

The Scalable Coherent Interface [1,8] 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 (D352) achieves remote store latencies starting at 220 ns and a maximum bandwidth of 334 MiB/s per channel.

46.2.2 Dolphin DX

The Dolphin DX interconnect [4] 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.

46.2.3 Dolphin Software Stack

The SISCI API [2] is the basic and 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 and DX interconnects accessible from user space. It consists of a user-space shared library (`libsisci`), 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.

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. Synchronisation 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.

SISCI does not provide means to pass messages between processes except for writing to some shared memory location and synchronising via either shared memory or remote interrupts. Therefore, based on this shared memory interface, Dolphin supplies a thin software layer for communication via message queues (`MBox/Msq`). It allows to establish uni-directional communication channels between machines which can be operated via simple `send()` and `recv()` operations. This software layer takes care of alignment, data gathering, error checking and so forth, and offers different optimised protocols for small, medium, and large data sizes.

It is also the basis for Dolphin's SuperSockets, which in user space offer a Berkeley API compliant sockets interface via `libksupersockets`.

46.3 Architecture of ETHOM

In order to bring together the two worlds of Ethernet-based software and Dolphin's high-speed networks, we inserted a thin layer of indirection below the Ethernet interface (see Fig. 46.1). 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 46.2). Compared with ETHOS, we sacrifice 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.

46.3.1 Configuration

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`

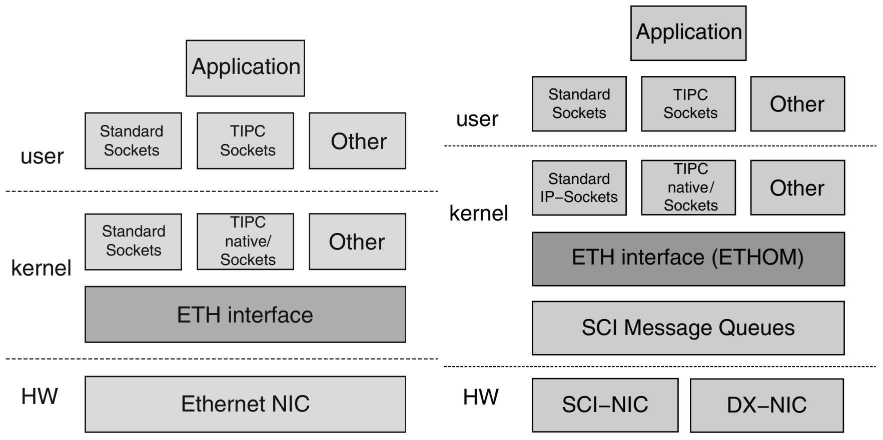

Fig. 46.1 Network architecture of ETHOM (*right*) in Comparison to Standard Architecture (*left*)

has to be passed as a parameter allowing to 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.

46.3.2 Connection Establishment

As shown in Fig. 46.2, 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 queues 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,

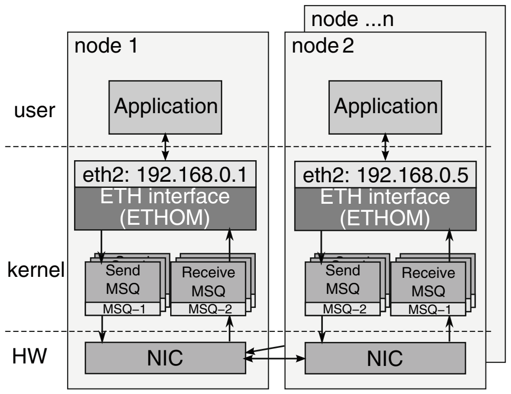

Fig. 46.2 Implementation of ETHOM

effectively reducing the number of remaining threads to one. On the occasion that a peer node does not connect directly, a new connection attempt is made periodically.

In case that IP communication is performed 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, the *Address Resolution Protocol* (ARP) is used at first contact to find the node that provides the sought-after IP address. For this purpose, the kernel sends so called *ARP requests*, Ethernet frames with the hardware address `ff:ff:ff:ff:ff:ff`, that ETHOM forwards to all hosts in the network. The interface providing the missing IP address, which is 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.

46.3.3 Communication Phase

Exchanging data between two nodes in a network is described on the basis of Fig. 46.3: 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

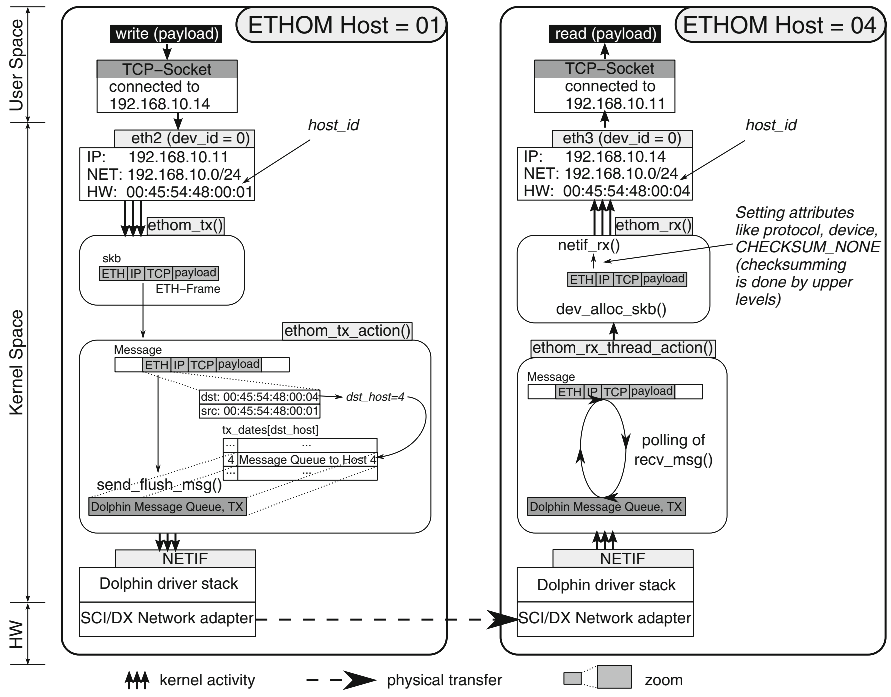

Fig. 46.3 Data transfer through ETHOM from sender to receiver

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.

Receiving. Arriving on host 4 (compare Fig. 46.3), 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 mentioned before, 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()`, attributes like `dev`, `protocol`, `ip_summed` are set, so that the kernel level above ETHOM accepts the skb, 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.

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.

46.4 Performance Evaluation

In this section, we briefly present basic experimental results. For a more detailed analysis, please refer to [6].

The measurements were performed on two different clusters: (1) *PD* consisting of nodes equipped with Pentium D 820 processors from Intel, on-board Gigabit Ethernet Controllers (BCM5721), a D352 SCI card from Dolphin, and an MHGS-18 DDR InfiniBand adapter from Mellanox (20 Gb/s). (2) *Xeon* consisting of nodes equipped with Xeon 5355 processors from Intel, on-board Gigabit Ethernet Controllers (Intel 82563EB), a DX510H adapter from Dolphin, and the same InfiniBand adapter as PD.

We chose NPtcp from the widely used NetPIPE benchmark suite in version 3.7.1 in order to generate easily comparable and reproducible low-level latency and bandwidth data.

46.4.0.1 Latency

Figure 46.4 shows the round-trip latency (RTT/2) for messages of varying sizes on the vertical axis and the message size on the horizontal axis.

The upper curve starting at 50 μs represents Gigabit Ethernet, the reference that ETHOS and ETHOM compete with. The lowest latencies are delivered by ETHOM on SCI, followed 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.

All in all, 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.

46.4.0.2 Bandwidth

Figure 46.5 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

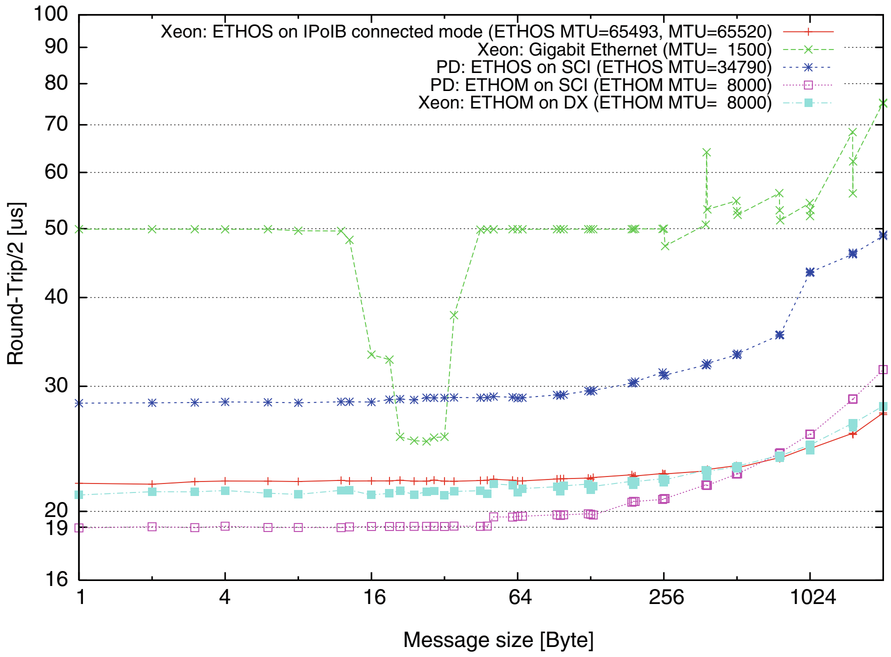

Fig. 46.4 Latency measured with NPtcp

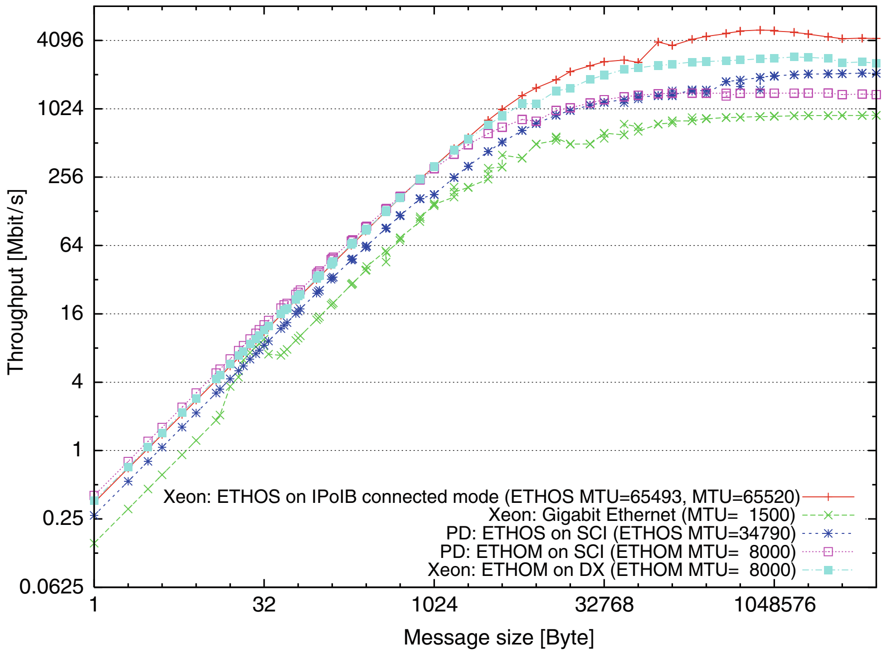

Fig. 46.5 Throughput measured with NPtcp

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 (up to 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.

46.5 Conclusions

The performance evaluation presented in Section 46.4 and more detailed in [6] shows that ETHOM – making use of a high-speed interconnect like either SCI or Dolphin DX – is a solution that offers better performance than Gigabit Ethernet, latency wise and bandwidth wise. Regarding the 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, which cannot be provided by Gigabit Ethernet.

Comparing the results with ETHOS [5], which implemented an Ethernet interface using kernel-level UDP sockets as its lower interface, 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: (1) The network should become an even bigger bottleneck for communicating applications, as the connection is shared by a bigger 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.

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 side, 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.

References

1. ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI): IEEE (2007)
2. SISCI Interface Specification 2.1.1 (1999). Dolphin Interconnect Solutions.
3. Dolphin Interconnect Solutions: The Dolphin SCI Interconnect (1996) http://www.dolphinics.com
4. Dolphin Interconnect Solutions The Dolphin DX Interconnect (2007). http://www.dolphinics.com/products/pent-dxseries-dxh510.html
5. Finocchiaro, R., Razik, L., Lankes, S., Bemmerl, T.: ETHOS, a generic Ethernet over Sockets Driver for Linux. In: Proceedings of the 20th International Conference on Parallel and Distributed Computing and Systems (PDCS) (2008)
6. Finocchiaro, R., Razik, L., Lankes, S., Bemmerl, T.: ETHOM, an Ethernet over SCI and DX Driver for Linux. In: Proceedings of 2009 International Conference of Parallel and Distributed Computing (ICPDC 2009), London, UK (2009)
7. Fu, Shaojian and Atiquzzaman, M.: SCTP: state of the art in research, products, and technical challenges. In: Proceedings of the IEEE 18th Annual Workshop on Computer Communications, CCW 2003, pp. 85–91 (2003)
8. Hellwagner, H., Reinefeld, A. (eds.): SCI: Architecture and Software for High Peformance Compute Clusters, Lecture Notes in Computer Science, vol. 1734. Springer-Verlag, Berlin, Germany (1999)
9. Hopkins, S., Coile, B.: AoE (ATA over Ethernet) (2006) http://www.coraid.com/site/co-pdfs/AoEr10.pdf
10. InfiniBand Trade Association: Infiniband Architecture Overview (2002). http://www.infinibandta.org/events/past/it_roadshow/overview.pdf
11. Kohler, E., Handley, M., Floyd, S.: Datagram Congestion Control Protocol (DCCP) (2006) http://ietfreport.isoc.org/rfc/PDF/rfc4340.pdf
12. Krishnan, V.: Towards an Integrated IO and Clustering Solution for PCI Express. In: Proceedings of IEEE International Conference on Cluster Computing (CLUSTER'07), Austin, TX (2007)
13. Larzon, L.-A., Degermark, M., Pink, S., Jonsson, L.-E., Fairhurst, G.: The Lightweight User Datagram Protocol (UDP-Lite) (2004). http://ietfreport.isoc.org/rfc/PDF/rfc3828.pdf.
14. Maloy, Jon (2004). TIPC: Providing Communication for Linux Clusters. In Proceedings of the Ottawa Linux Symposium, pages 347–356. http://www.linuxsymposium.org/proceedings/LinuxSymposium2004_V2.pdf
15. Myricom Inc.: Myrinet 2000 Product List (2008). http://www.myri.com/myrinet/product_list.html
16. Quadrics Ltd.: Quadrics QsNetII (2003). http://www.quadrics.com
17. Stephens, A., Maloy, J., Horvath, E.: TIPC Programmer's Guide (2008). http://tipc.sourceforge.net/doc/tipc_1.7_prog_guide.pdf
18. The Kerrighed Team Kerrighed: a Single System Image operating system for clusters (2008). http://www.kerrighed.org