Memory Management in a combined VIA/SCI Hardware

Mario Trams, Wolfgang Rehm, Daniel Balkanski and Stanislav Simeonov *
{mtr,rehm}@informatik.tu-chemnitz.de
DaniBalkanski@yahoo.com, stan@bfu.bg

Technische Universität Chemnitz
Fakultät für Informatik**
Straße der Nationen 62, 09111 Chemnitz, Germany

Abstract In this document we make a brief review of memory management and DMA considerations in case of common SCI hardware and the Virtual Interface Architecture. On this basis we expose our ideas for an improved memory management of a hardware combining the positive characteristics of both basic technologies in order to get one completely new design rather than simply adding one to the other. The described memory management concept provides the opportunity of a real zero–copy transfer for Send–Receive operations by keeping full flexibility and efficiency of a nodes' local memory management system. From the resulting hardware we expect a very good system throughput for message passing applications even if they are using a wide range of message sizes.

1 Motivation and Introduction

PCI–SCI bridges (Scalable Coherent Interface [12]) become a more and more preferable technological choice in the growing market of Cluster Computing based on non–proprietary hardware. Although absolute performance characteristics of this communication hardware increases more and more, it still has some disadvantages. Dolphin Interconnect Solutions AS (Norway) is the leading manufacturer of commercial SCI link chips as well as the only manufacturer of commercially available PCI–SCI bridges. These bridges offer very low latencies in range of some microseconds for their distributed shared memory and reach also relatively high bandwidths (more than 80MBytes/s). In our clusters we use Dolphins PCI–SCI bridges in junction with standard PC components [11]. MPI applications that we are running on our cluster can get a great acceleration from low latencies of the underlying SCI shared memory if it is used as communication medium for transferring messages. MPI implementations e.g. such as [7] show a

* Daniel Balkanski and Stanislav Simeonov are from the Burgas Free University, Bulgaria.

** The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6). It is also carried out in strong interaction with the project GRANT SFB393/B6 of the DFG (German National Science Foundation).

bandwidth of about 35MByte/s for a message size of 1kByte which is quite a lot (refer also to figure 1 later).

The major problem of MPI implementations over shared memory is big CPU utilization on long message sizes due to copy operations. So the just referred good MPI performance [7] is more an academic peak performance which is achieved with more or less total CPU consumption. A standard solution for this problem is to use a block–moving DMA engine for data transfers in background. Dolphins PCI–SCI bridges implement such a DMA engine. Unfortunately, this one can't be controlled directly from a user process without violating general protection issues. Therefore kernel calls are required here which in end effect increase the minimum achievable latency and require a lot of additional CPU cycles.

The Virtual Interface Architecture (VIA) Specification [16] defines mechanisms for moving the communication hardware closer to the application by migrating protection mechanisms into the hardware. In fact, VIA specifies nothing completely new since it can be seen as an evolution of U–Net [15]. But it is a first try to define a common industry–standard of a principle communication architecture for message passing — from hardware to software layers. Due to its DMA transfers and its reduced latency because of user–level hardware access, a VIA system will increase the general system throughput of a cluster computer compared to a cluster equipped with a conventional communication system with similar raw performance characteristics. But for very short transmission sizes a programmed IO over global distributed shared memory won't be reached by far in terms of latency and bandwidth. This is a natural fact because we can't compare a simple memory reference with DMA descriptor preparation and execution.

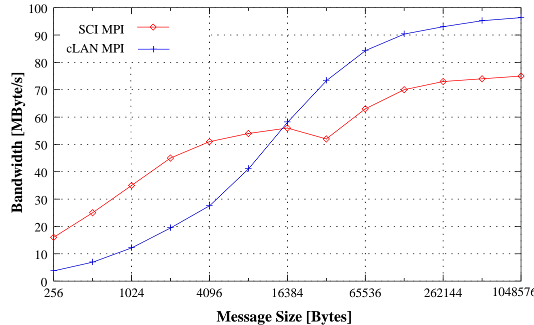

Figure1. Comparison of MPI Implementations for Dolphins PCI–SCI Bridges and GigaNets cLAN VIA Hardware

Figure 1 shows bandwidth curves of MPI implementations for both an SCI and a native VIA implementation (GigaNet cLAN). The hardware is in both cases based on the PCI bus and the machines where the measurements were taken are comparable. The concrete values are based on ping–pong measurements and where taken from [7] in case of SCI, and from [10] (Linux case) for the cLAN hardware.

As expected, the bandwidth in case of SCI is looking better in the range of smaller message sizes. For larger message sizes the cLAN implementation demonstrates higher bandwidth because of its advanced DMA engine. But not less important is the fact that a DMA engine gives the CPU more time for computations. Details of such CPU utilization considerations are outside the scope of this paper and are already discussed in [14] and [8].

As summarization of these motivating facts we can state that besides a powerful DMA engine controllable from user–level a distributed shared memory for programmed IO is an important feature which shouldn't be missed in a communication system.

2 What are the Memory Management Considerations?

First of all we want to make a short definition what belongs to memory management regarding this document.
This can be stated by the following aspects expressed in the form of questions:

1. How a process' memory area is made available to the Network Interface Controller (NIC) and in what way main memory is protected against wrong accesses?
2. At which point in the system a DMA engine is working and how are the transactions of this DMA engine validated?
3. In which way memory of a process on a remote node is made accessible for a local process?

Based on these questions we can classify the different communication system architectures in terms of advantages/disadvantages of their memory management. In the analysis that is presented in the following sections we'll reveal these advantages and disadvantages arisen from common PCI–SCI architecture and the VI Architecture.

3 PCI–SCI vs. VIA discussion and comparison

3.1 Question 1: How a process' memory area is made available to the NIC and in what way main memory is protected against wrong accesses?

Common PCI–SCI case: Current PCI–SCI bridges developed by Dolphin realize a quiet static memory management [4] to get access to main memory or rather PCI address space. To avoid unwanted accesses to sensitive locations, the PCI–SCI bridge is set up to allow accesses only to a dedicated memory window. Memory access requests caused by remote machines are only allowed if they fall within the specified window. This causes two big disadvantages:

– Continuous exported regions must also be continuous available inside the physical address space. Additionally, these regions must be aligned to the minimum exportable block size which is typically quite large (512kB for Dolphin's bridges).

– Exported Memory must reside within this window.

To handle these problems it is required to reserve main memory only for SCI purposes. This, in practice, 'wastes' a part of memory if it is not really exported later.

In consequence these disadvantages of common PCI–SCI bridge architecture make their use with MPI applications very difficult. Especially in view of zero–copy transfer operations. Because data transfers can be processed using the reserved memory region only, it would require that MPI applications use special `malloc()` functions for allocating data structures used for send/receive purposes later. But this violates a major goal of the MPI standard: Architecture Independence.

VIA case: The VI Architecture specifies a much better view the NIC has on main memory. Instead of a flat one–to–one representation of the physical memory space it implements a more flexible lookup–table address translation. Comparing this mechanism with the PCI–SCI pendant the following advantages become visible.

- Continuous regions seen by the VIA hardware are not required to be also continuous inside the host physical address space.
- Accesses to sensitive address ranges are prevented by just not including them into the translation table.
- The NIC can get access to **every** physical memory page, even if this may not be possible for all physical pages at once (when the translation table has less entries than the number of physical pages).

The translation table is not only for address translation purposes, but also for protection of memory. To achieve this a so–called *Protection Tag* is included for each translation and protection table entry. This tag is checked prior to each access to main memory to qualify the access. For more information about this see later in section 3.2.

Conclusions regarding question 1: It is clear, that the VIA approach offers much more flexibility. Using this local memory access strategy in a PCI–SCI bridge design will eliminate all of the problems seen in current designs.
Of course, the drawback is the more complicated hardware and the additional cycles to translate the address.

3.2 Question 2: At which point in the system a DMA engine is working and how are the transactions of this DMA engine validated?

Common PCI–SCI case: The DMA engine accesses local memory in the same way as already discussed in section 3.1. Therefore it inherits also all disadvantages when dealing with physical addresses on the PCI–SCI bridge.

For accesses to global SCI memory a more flexible translation table is used. This *Downstream Translation Table* realizes a virtual view onto global SCI memory — similar as the view of a VIA NIC onto local memory. Every page of the virtual SCI memory can be mapped to a page of the global SCI memory.

Regarding validation, the DMA engine can't distinguish between regions owned by different processes (neither local nor remote). Therefore the hardware can't make a check of access rights on–the–flow. Rather it is required that the DMA descriptor containing the information about the block to copy is assured to be right. In other words the operating system kernel has to prepare or at least to check any DMA descriptor to be posted to the NIC. This requires OS calls that we want to remove at all cost.

VIA case: A VIA NIC implements mechanisms to execute a DMA descriptor from user–level while assuring protection among multiple processes using the same VIA hardware. An user process can own one or more interfaces of the VIA hardware (so–called *Virtual Interfaces*). In other words, a virtual interface is a virtual representation of a virtual unique communication hardware. The connection between the virtual interfaces and the VIA hardware is made by *Doorbells* that represent a virtual interface with its specific control registers. An user–level process can insert a new DMA descriptor into a job queue of the VIA hardware by writing an appropriate value into a doorbell assigned to this process. The size of a doorbell is equal to the page size of the host computer and so the handling which process may access which doorbell (or virtual interface) can be simply realized by the hosts' virtual memory management system. Protection during DMA transfers is achieved by usage of *Protection Tags*. These tags are used by the DMA engine to check if the access of the current processed virtual interface to a memory page is right. The protection tag of the accessed memory page is compared with the protection tag assigned to the virtual interface of the process that provided this DMA descriptor. Only if both tags are equal, the access is legal and can be performed. A more detailed description of this mechanism is outside the scope of this document (refer to [13] and [16]).

Conclusions regarding question 2: The location of the DMA engine is in both cases principally the same. The difference is that in case of VIA a real lookup–table based address translation is performed between the DMA engine and PCI memory. That is, the VIA DMA operates on a virtual local address space, while the PCI–SCI DMA operates directly with local physical addresses. The answer for the access protection is simple: The common PCI–SCI DMA engine supports no protection in hardware and must trust on right DMA descriptors. The VIA hardware supports full protection in hardware where the DMA engine is only one part of the whole protection mechanism.

3.3 Question 3: In which way memory of a process on a remote node is made accessible for a local process?

Common PCI–SCI case: Making remote memory accessible is a key function in a SCI system, of course. Each PCI–SCI bridge offers a special PCI memory window which is practically the virtual SCI memory seen by the card. So the same SCI memory the DMA engine may access can be also accessed via memory references (also called programmed IO here). The procedure of making globally available SCI memory accessible for the local host is also referred as *importing global memory into local address space.*

On the other side, every PCI–SCI bridge can open a window to local address space and make it accessible for remote SCI nodes. The mechanism of this window is already described in section 3.1 regarding question 1. The procedure of making local memory globally accessible is also called *exporting local memory into global SCI space.*

Protection is totally guaranteed when dealing with imported and exported memory in point of view of memory references. Only if a process has got a valid mapping of a remote process' memory page it is able to access this memory.

VIA case: The VI Architecture offers principally no mechanism to access remote memory as it is realized in a distributed shared memory communication system such as SCI. But there is an indirect way by using a so–called Remote DMA (or RDMA) mechanism. This method is very similar to DMA transfers as they are used in common PCI–SCI bridges. A process that wants to transfer data between its local memory and memory of a remote process specifies a RDMA descriptor. This contains an address for the local VIA virtual address space and an address for the remote nodes' local VIA virtual address space.

Conclusions regarding question 3: While a PCI–SCI architecture allows processes to really share their memory globally across a system, this is not possible with a VIA hardware. Of course, VIA was never designed for realizing distributed shared memory.

4 A new PCI–SCI Architecture with VIA Approaches

In our design we want to combine the advances of an ultra–low latency SCI Shared Memory with a VIA–like advanced memory management and protected user–level DMA. This combination will make our SCI hardware more suitable for our message passing oriented parallel applications requiring short as well as long transmission sizes.

4.1 Advanced Memory Management

In order to eliminate the discussed above restrictions with continuous and aligned exported memory regions that must reside in a special window, our PCI–SCI

architecture will implement two address translation tables — for both local and remote memory accesses. In contrast, common PCI–SCI bridges use only one translation table for accesses to remote memory. This new and more flexible memory management combined with reduced minimal page size of distributed shared memory leads to a much better usage of the main memory of the host system.

In fact, our targeted amount of imported SCI memory is 1GB with a page granularity of 16kB. With a larger downstream address translation table this page size may be reduced further to match exactly the page size used in the host systems (such as 4kB for x86 CPUs).

In case of the granularity of memory to be exported in SCI terminology or to be made available for VIA operations there's no question: It must be equal to the host system page size. In other words, 4kB since the primary target system is a x86 one. 128MB is the planned maximum window size here.

4.2 Operation of Distributed Shared Memory from a memory–related point of view

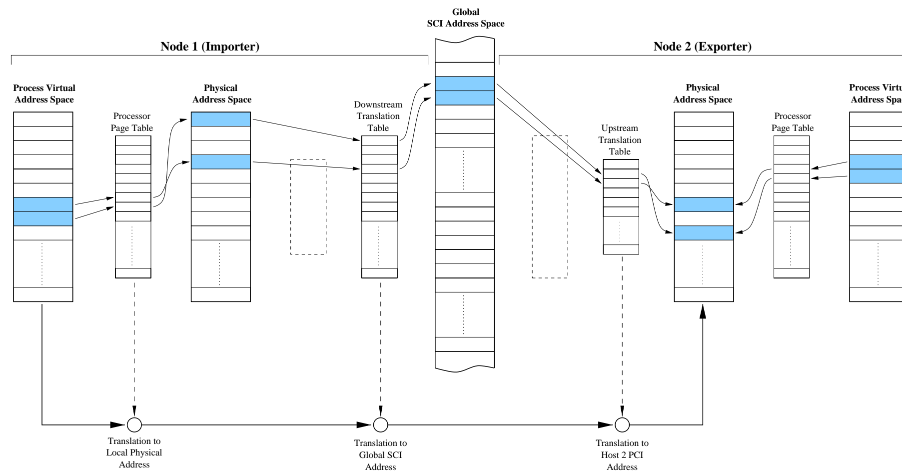

Figure2. Address Translations between exporting and importing Processes for programmed IO

Figure 2 gives an overall example of exporting/importing memory regions. The example illustrates the address translations performed when the importing process accesses memory exported by a process on the remote node.

The exporting process exports some of its previously allocated memory by registering it within its local PCI–SCI hardware. Registering memory is done on a by–page basis. Remember that in case of a common PCI–SCI system it would be required that this exported memory is physically located inside this special memory area reserved for SCI purposes. But here we can take the advantage of the virtual view onto local memory similar to this in VI Architecture.

Once the upstream address translation table entries are adjusted, the exported memory can be accessed from remote machines since it became part of the global SCI memory. To access this memory, the remote machine must import it first. The major step to do here is to set up entries inside its downstream address translation table so that they point to the region inside the global SCI memory that belongs to the exporter. From now, the only remaining task is to map the physical PCI pages that correspond to the prepared downstream translation entries into the virtual address space of the importing process.

When the importing process accesses the imported area, the transaction is forwarded through the PCI–SCI system and addresses are translated three times. At first the host MMU translates the address from the process' virtual address space into physical address space (or rather PCI space). Then the PCI–SCI bridge takes up the transaction and translates the address into the global SCI address space by usage of the downstream translation table. The downstream address translation includes generation of the remote node id and address offset inside the remote nodes' virtual local PCI address space. When the remote node receives the transaction, it translates the address to the correct local physical (or rather PCI) address by using the upstream address translation table.

4.3 Operation of Protected User–Level Remote DMA from a memory–related point of view

Figure 3 shows the principle work of the DMA engine of our PCI–SCI bridge design. This figure shows principally the same address spaces and translation tables as shown by figure 2. Only the process' virtual address spaces and the corresponding translation into physical address spaces are skipped to not overload the figure.

The DMA engine inside the bridge is surrounded by two address translation tables, or more correct said by two address **translation and protection** tables. On the active node (that is, where the DMA engine is executing DMA descriptors — node 1 here) both translation tables are involved. However, on the remote node there has practically nothing changed compared to the programmed IO case. Hence the remote node doesn't make any difference between transactions whether they were generated by the DMA engine or not.

Both translation tables of one PCI–SCI bridge incorporate protection tags as described in section 3.2. But while this is used in VIA for accesses to local memory, here it is also used for accesses to remote SCI memory. Together with VIA mechanisms for descriptor notification and execution the DMA engine is unable to access wrong memory pages — whether local (exported) nor remote (imported) ones. Note that a check for right protection tags is really made only for the DMA engine and only on the active node (node 1 in figure 3). In all other cases the same translation and protection tables are used, but the protection tags inside are ignored.

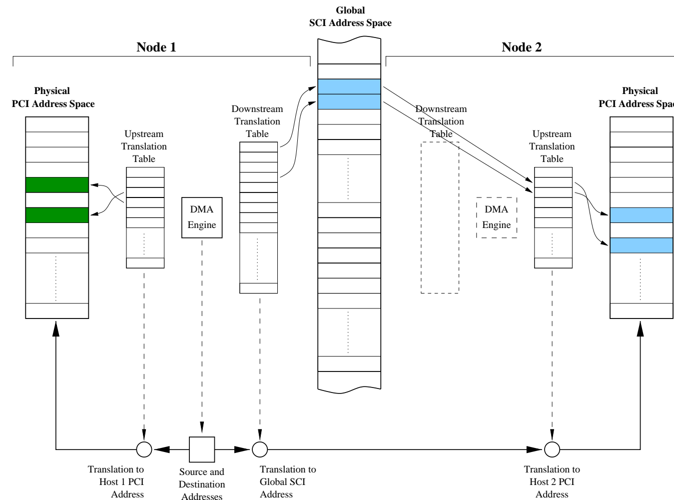

Figure3. Address Translations performed during RDMA Transfers

4.4 A free choice of using either Programmed I/O or User–Level Remote DMA

This kind of a global memory management allows applications or more exactly communication libraries to decide on–the–fly depending on data size in which way it should be transferred. In case of a short message a PIO transfer may be used, and in case of a longer message a RDMA transfer may be suitable. The corresponding remote node is not concerned in this decision since it doesn't see any differences. This keeps the protocol overhead very low.

And finally we want to remember the VIA case. Although we already have the opportunity of a relatively low–latency protected user–level remote DMA mechanism without the memory handling problems as in case of common PCI–SCI, there's nothing like a PIO mechanism for realizing a distributed shared memory. Hence the advantages of an ultra–low latency PIO transfer are not available here.

5 Influence on MPI Libraries

To show the advantages of the presented advanced memory management we want to take a look at the so–called *Rendezvous Protocol* that is commonly used for Send–Receive operations.

Figure 4 illustrates the principle of the Rendezvous protocol used in common MPI implementations [7] based on Dolphins PCI–SCI bridges. One big problem in this model is the copy operation that takes place on the receivers' side to take data out of the SCI buffer. Although the principally increasing latency can be hidden due to the overlapping mechanism a lot of CPU cycles are burned there.

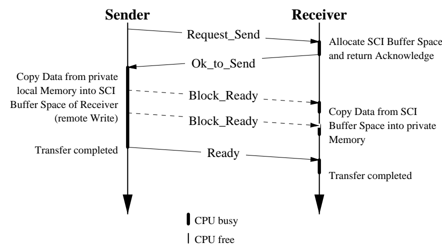

Figure4. Typical Rendezvous–Protocol in common PCI–SCI Implementations

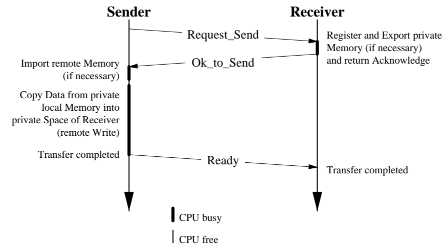

Figure5. Improved Rendezvous–Protocol based on advanced PCI–SCI Memory Management

With our proposed memory management there's a chance to remove this copy operation on the receivers' side. The basic operation of the Rendezvous protocol can be implemented as described in figure 5. Here the sender informs the receiver as usual. Before the receiver sends back an acknowledge it checks if the data structure the data is to be written to is already exported to the sender. If not, the memory region that includes the data structure is registered within the receivers' PCI–SCI bridge and exported to the sender. The sender itself must also import this memory region if this was not already done before. After this the sender copies data from private memory of the sending process directly into private memory of the receiving process. As further optimization the sender may decide to use the DMA engine to copy data without further CPU intervention. This decision will be typically based on the message size.

6 State of the project (November 1999)

We developed our own FPGA–based PCI–SCI card and have prototypes of this card already running. At the moment they only offer a so–called *Manual Packet Mode* for now that is intended for sideband communication besides the regular programmed IO and DMA transfers.

The card itself is a 64Bit/33MHz PCI Rev.2.1 one [8]. As SCI link controller we are using Dolphins LC–2 for now, and we are looking to migrate to the LC–3 as soon as it is available. The reprogrammable FPGA design leads to a flexible reconfigurable hardware and offers also the opportunity for experiments.

Linux low–level drivers for Alpha and x86 platforms and several configuration/test programs were developed. In addition our research group is working on an appropriate higher–level Linux driver for our card [5, 6]. This offers a software–interface (advanced Virtual Interface Provider Library) that combines SCI and VIA features such as importing/exporting memory regions, VI connection management etc. Also it emulates parts of the hardware so that it is possible to run other software on top of it although the real hardware is not available. As an example, a parallelized MPI–version of the popular raytracer *POVRAY* is already running over this emulation. This program uses an MPI–2 library for

our combined SCI/VIA hardware. This library is also under development at our department [3].

For more details and latest news refer to our project homepage at
`http://www.tu-chemnitz.de/~mtr/VIA_SCI/`

7 Other Works on SCI and VIA

Dolphin already presented some performance measurements in [1] for their VIA implementation which is a emulation over SCI shared memory. Although the presented VIA performance is looking very good, it's achieved by the cost of too big CPU utilization again.

The number of vendors of native VIA hardware is growing more and more. One of these companies is GigaNet [17] where performance values are already available. GigaNet gives on their web pages latencies of $8\mu s$ for short transmission sizes. Dolphin gives a latency for PIO operations (remote memory access) of $2.3\mu s$. This demonstrates the relatively big performance advantage a distributed shared memory offers here.

University of California, Berkeley [2] and the Berkeley Lab [9] are doing more open research also in direction of improving the VIA specification. The work at the University of California, Berkeley is concentrated more on VIA hardware implementations based on Myrinet. In contrast, the work at the Berkeley Lab is targeted mainly to software development for Linux.

8 Conclusions and Outlook

The combined PCI–SCI/VIA system is not just a simple result of adding two different things. Rather it is a real integration of both in one design. More concrete it is an integration of concepts defined by the VIA specification into a common PCI–SCI architecture since major PCI–SCI characteristics are kept. The result is a hardware design with completely new qualitative characteristics. It combines the most powerful features of SCI and VIA in order to get highly efficient messaging mechanisms and high throughput over a broad range of message lengths.

The advantage that MPI libraries can take from a more flexible memory management was illustrated for the case of a Rendezvous Send–Receive for MPI. The final proof in practice is still pending due to lack of a hardware with all implemented features.

References

1. Torsten Amundsen and John Robinson: *High–performance cluster–computing with Dolphin's CluStar PCI adapter card*. In: Proceedings of SCI Europe '98, Pages 149–152, Bordeaux, 1998

2. Philip Buonadonna, Andrew Geweke: *An Implementation and Analysis of the Virtual Interface Architecture.* University of California at Berkeley, Dept.of Computer Science, Berkeley, 1998. www.cs.berkeley.edu/~philipb/via/
3. *A new MPI–2–Standard MPI Implementation with support for the VIA.* www.tu-chemnitz.de/informatik/RA/projects/chempi-html/
4. Dolphin Interconnect Solutions AS: *PCI–SCI Bridge Spec. Rev. 4.01.* 1997.
5. Friedrich Seifert: *Design and Implementation of System Software for Transparent Mode Communication over SCI.*, Student Work, Dept. of Computer Science, University of Technology Chemnitz, 1999. See also: www.tu-chemnitz.de/~sfri/publications.html
6. Friedrich Seifert: *Development of System Software to integrate the Virtual Interface Architecture (VIA) into Linux Operating System Kernel for optimized Message Passing.* Diploma Thesis, TU–Chemnitz, Sept. 1999. See also: www.tu-chemnitz.de/informatik/RA/themes/works.html
7. Joachim Worringen and Thomas Bemmerl: *MPICH for SCI–connected Clusters.* In: Proceedings of SCI–Europe'99, Toulouse, Sept. 1999, Pages 3–11. See also: wwwbode.in.tum.de/events/sci-europe99/
8. Mario Trams and Wolfgang Rehm: *A new generic and reconfigurable PCI–SCI bridge.* In: Proceedings of SCI–Europe'99, Toulouse, Sept. 1999, Pages 113–120. See also: wwwbode.in.tum.de/events/sci-europe99/
9. *M–VIA: A High Performance Modular VIA for Linux.* Project Homepage: http://www.nersc.gov/research/FTG/via/
10. MPI Software Technology, Inc. *Performance of MPI/Pro for cLAN on Linux and Windows.* www.mpi-softtech.com/performance/perf-win-lin.html
11. *The Open Scalable Cluster ARchitecture (OSCAR) Project.* TU Chemnitz. www.tu-chemnitz.de/informatik/RA/projects/oscar_html/
12. *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE Std. 1596-1992. SCI Homepage: www.SCIzzL.com
13. Mario Trams: *Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface.* Diploma Thesis, TU-Chemnitz, 1998. See also: www.tu-chemnitz.de/informatik/RA/themes/works.html
14. Mario Trams, Wolfgang Rehm, and Friedrich Seifert: *An advanced PCI–SCI bridge with VIA support.* In: Proceedings of 2nd Cluster–Computing Workshop, Karlsruhe, 1999, Pages 35–44. See also: www.tu-chemnitz.de/informatik/RA/CC99/
15. The U-Net Project: *A User–Level Network Interface Architecture.* www2.cs.cornell.edu/U-Net
16. Intel, Compaq and Microsoft. *Virtual Interface Architecture Specification V1.0.*, VIA Homepage: www.viarch.org
17. GigaNet Homepage: www.giganet.com