An advanced PCI–SCI bridge with VIA support

Mario Trams, Wolfgang Rehm, and Friedrich Seifert

{mtr,rehm,sfr}@informatik.tu-chemnitz.de

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

Abstract

Recent developments in networking technology and rise in cluster computing have driven many research studies in high performance communication architectures.
The so–called *Virtual Interface Architecture* (VIA) seeks to provide an operating system independent infrastructure for high–performance user–level networking in a generic environment. Therefore it defines mechanisms for low–latency, high–bandwidth message communication style.
Although low–latency is one of the major goals of VIA several research prototypes (software emulations) have shown that it couldn't be achieved satisfying until now.
The *Scalable Coherent Interface* (SCI) is a high–speed cluster interconnect that offers extreme low latency based on distributed shared memory (DSM) facilities. Although initially not intended for message–passing style of communication it's well–suited for this purpose.
Hence our idea is to combine the architectural principles of SCI and VIA. In this paper we describe some concepts of an advanced PCI–SCI bridge we're currently developing at the Chemnitz University of Technology.
First we show the system throughput advantages on using DMA–based communication beside DSM copy operations for message passing. In the case of Remote DMA transfer model we show that SCI and VIA can be efficently combined to form a new communication hardware architecture. In addition several problems with current PCI–SCI implementations are discussed.
The facts highlited demonstrate the complexity of the design space as well as the need of a prototypical implementation.

1 Introduction

Our primary focus is on message passing applications following the standardized *Message Passing Interface* (MPI [MPI98]). The reason for this focus is that we are part of a research poject at our university called *Numerical Simulation on Massive Parallel Computers* (SFB 393). Mathematicans and physicists involved in this project use MPI for their FEM *Finite Elements Method* simulation software, and so our job is to provide appropriate optimized compute servers beginning at the MPI level down to the low hardware layers.

In the first half of this paper we want to discuss some measurements on currently available PCI–SCI bridges and their consequences. In the second half we want to show some of our ideas for an advanced PCI–SCI architecture solving several problems with todays implementations and which is optimized for message passing communication.

2 DMA and Shared Memory Performance of PCI–SCI

Until now, we heard only about performance measurements of the Shared Memory of Dolphin's PCI–SCI bridges (for example [AmRo98]). But there were not many facts known about the DMA performance of these cards. Implementations for SCI's hardest competitor Myrinet [MYRI, BIP] have shown performances close to the theoretical PCI performance limit. The reason for this high bandwidth is the extensive use of a PCI Master (DMA Engine) producing long PCI burst cycles. These long bursts are the key to get high throughputs. When data is transfered using the SCI Shared Memory, then the transfer is performed by the CPU itself. Although this reduces the communication overhead to an absolute minimum (no descriptors etc.), a better system throughput could be achieved by using DMA as we'll show later.
To get some more facts on DMA performance of Dolphin's PCI–SCI bridge (D310 with LC–2) we made some appropriate measurements. In view

*The work presented in this paper is sponsored by the SMWK/SMWA Saxony ministries (AZ:7531.50-03-0380-98/6) of memory protection, the D310 supports conventional DMA only. Hence, every DMA initiation has to be checked and confirmed by the OS kernel under regular circumstances. However, to get an idea of how far the kernel call affects DMA behaviour we tested both methods: raw DMA controlled directly by user–level software and kernel DMA initiated through kernel calls. For reference, we also tested SCI Shared Memory performance.

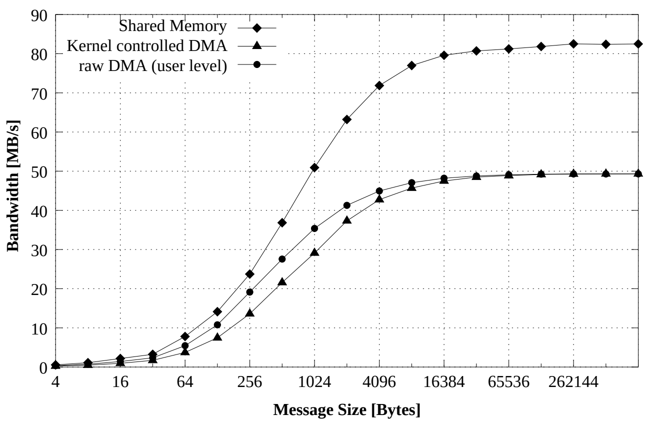
2.1 Test conditions

We tested two 440BX boards each equipped with 2 350MHz PII, coupled via Dolphins latest D310 PCI–SCI bridges, and running Linux 2.0.36. All of the eight available write stream buffers of the bridge [Dolph1] were used to get a maximum degree of parallelism during SCI transfers. For the Shared Memory routine a fast copy–routine with Intel's MMX instructions (64Bit moves) was used. The processors write–combining feature was not activated.
The measurements are based on a simple Ping–Pong test including a syncronization handshake. That is, after the message is sent to the receiver a "Ok" is sent. If the receiver detects the Ok, it sends the same message back followed by an Ok again. The sender measures the time from the beginning of send operation to the receive of the Ok. The round–trip time we got is divided by two to get the time for the message transmission. To avoid DMA status polling during DMA transfers, the *Chained DMA Mode* of the D310 was used. That is, the DMA descriptor queue consists of two descriptors: one for the data body and one for the handshake (Ok). So the receiver polls only in cached memory to check whether the data block has fully arrived or not.
All measurements taken are based on zero–copy. Hence, data is transfered only between memory regions accesible directly by the D310. For Shared Memory that's true for the destination and in case of DMA for both source and destination. In practice, there are some problems with the D310 to bring zero–copy up to the application level. These problems are described later in this paper (section 3.4). However, in this paper we want to discuss an advanced PCI–SCI bridge and we present measurements from Dolphin's hardware to show what can be achieved by raw hardware.
Figure 1 shows the results.

The measured values for the Shared Memory transmission confirm with measurements demonstrated in [AmRo98]. The advance of raw DMA over kernel controlled DMA is significant up to message sizes of 8–16kB. But general DMA performance, however, is looking much worse than expected. Neither does DMA performance exceed Shared Memory performance, nor Shared Memory performance is reached for large message sizes. This is especially sad, since real implementations on the relatively "slow" 1.280GBit/s Myrinet achieved higher rates than the current 4GBit/s LC–2 SCI implementation [BIP, MYRI, KuSt98].
The reason for the low DMA performance can only be found inside the DMA engine. There are only blocks of 64Bytes transfered over the PCI Bus in one burst. This seems to be the most affecting fact, especially when the PCI–SCI bridge reads data out of local memory. As described in [PCI98], reading from main memory as a PCI Master costs somewhat more than writing (2 cycles overhead on write, 12 cycles overhead on read).

Latency (0Byte Message):

Shared Mem 4μs	raw DMA 7μs	Kernel DMA 11.5μs



Note: To keep the graphs clear only message sizes with powers of 2 are shown. Otherwise the graphs are looking a bit jaggy due to the behaviour of the write stream buffers.

Figure 1: DMA and Shared Memory Performance

2.2 Potential for DMA Improvement

As previously mentioned, DMA performance depends heavily on PCI burst length. In general, the number of clock cycles required for a DMA transfer can be estimated by following formula (for 32Bit

PCI bus):

$$C = O + o \cdot \Big(1 + (N \text{ DIV } L)\Big) + \frac{N}{4}$$

where

C	—	number of total cycles
N	—	number of bytes to transfer
L	—	used PCI burst length
O	—	DMA startup overhead
o	—	overhead per PCI transfer

The bandwidth B can be calculated by

$$B = \frac{N \cdot F}{C}$$

where F is the frequency of the PCI bus (in our case 33MHz).
The DMA startup overhead O includes stuff like invoking DMA descriptor execution and descriptor fetching by the bridge out of main memory. Note that here's also the syncronization overhead included (handshake descriptor processing). The overhead o per PCI transfer is caused inside the DMA engine for several handling stuff per packet and during the PCI transfer (PCI address phase and waiting until the hostbridge is ready). In case of Dolphins D310 PCI–SCI bridge [Dolph1] only bursts up to 64 bytes are used ($L = 64$).

It's interesting to know concrete values of O and o. These values can be estimated by doing some graphical tests to approximate the real raw DMA graph. We did it and the best approximation is with $O = 250$cycles and $o = 25$cycles. Although this approximation is a bit inaccurate for transfers less than 128Bytes (somewhat higher performance then real DMA), it compares very well with reality for message sizes above 128Bytes. It seems that the differences for the smaller message sizes originate from some effects of the PCI–SCI hardware not reflected in the formula above. However, for the following considerations this does not matter.

Now, since we know exact overheads, we can analyze the following hypothetical case:
Which performance values we'll get, if the same DMA engine uses longer PCI bursts than 64Bytes?

Figure 2 shows some graphs of expected DMA performance with longer bursts than 64Bytes. As we can see, even a doubled burst length (128Bytes) results in a massive performance growth of about 20MB/s. So it's a great challenge to do some more work in this area.

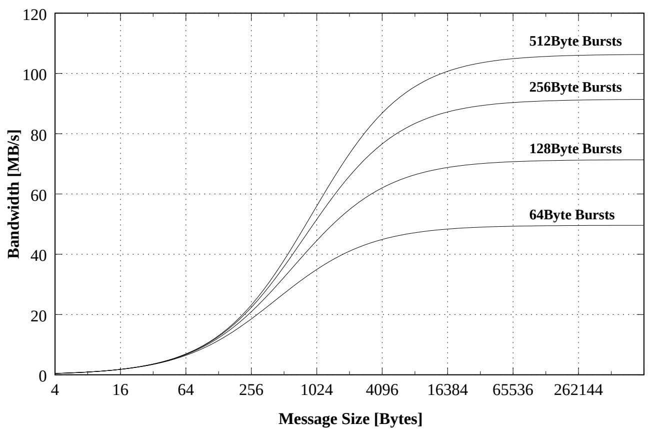


Figure 2: Expected DMA Performance with different PCI Burst Lengths

However, to achieve a good performance it's necessary to use long bursts at source <u>and</u> destination. While it's relatively simple to generate long bursts at the source and to send out multiple SCI packets, there's required some more intelligence at the destination. The receiving PCI–SCI bridge has dynamically to detect consecutive data packets and has to merge these together on the flow. This is a very critical task.
We also don't have to forget that it's unusual that a node receives data from only one sender at a time. This makes it much more complicate to merge incoming SCI writes together. To keep throughput as high as possible the hardware must check a pool of packets to detect possible candidates for merging (out–of–order execution).

On the other hand it's more important to elimiate short bursts at the source, since reading from main memory costs significantly more overhead then writing ([PCI98]). Additionally, SCI Shared Memory shows bandwidth of about 82MB/s without forming longer bursts than 64Bytes at the destination. Therefore a doubled burst length on PCI reads should deliver expected DMA bandwidth (about 72MB/s) without the need for a doubled burst length at the receiving node. However, for longer read bursts than 128Bytes longer write bursts will become necessary.

2.3 Why to use DMA?

Assuming the DMA engine for PCI–SCI bridges can't be improved any more (for whatever reason), there is a serious question why to use a DMA engine at all. When the SCI Shared Memory is always better than DMA, you'll never achieve a better bandwidth with DMA. Although you can enhance system throughput in several cases by using SCI Shared Memory, another fact is often not respected: During Shared Memory data transfer, the CPU does nothing more than stupid copy operations. In contrast, DMA is working in background. Of course, DMA also affects CPU operation due to the limited main memory bandwith, but caches typically help to decouple CPU operations from memory. To quantify CPU or rather memory bandwidth influence during DMA we measured memory bandwidth with and without running SCI DMA. Test hardware was the same as above. Memory bandwidth was calculated by forming a sum over a large 32Bit integer array. We tested three different access methods. The first makes absolutely no use of the cache by accessing memory in steps of 32Bytes (cache line size). This shows the worst case behaviour. The second test uses the cache in a medium manner by performing consecutive memory accesses. And finally, the last test operates completely inside cache. Table 1 shows the results.

	without DMA	with DMA	% available with DMA
no Cache use	51.2MB/s	43.9MB/s	85.7%
medium Cache use	136.4MB/s	122.0MB/s	89.4%
full Cache use	408.8MB/s	402.3MB/s	98.4%

Table 1: Memory Bandwith of BX Dual 350MHz PII with/without running DMA

These results are a bit surprising, since memory performance (and therefore CPU performance too) breaks down only a little bit even when the cache is not used. Another surprise (not shown in the table) is that the performance of the DMA engine is not significantly affected even if the CPU produces heavy memory traffic.

Based on information of Shared Memory and DMA performance and DMA influence on CPU operation, we can compare Shared Memory and DMA in view of time available for the CPU to do other stuff than communication. On one hand there is Shared Memory requesting the CPU activity for a certain time. And on the other hand there is DMA taking in worst case about 15% of CPU performance for a longer time. So we can plot two graphs over message size. The first represents the CPU time available during DMA which is 85% of the time used for the transfer. The second graph is the difference between time used for DMA and time used for Shared Memory (this time where the CPU is 100% available for other purposes than communication).

For the Shared Memory performance we can't use the detailed values shown in figure 1. These are Ping–Pong values meaning that there's the complete transmission time across the network included. In reality, writing some bytes into Shared Memory takes the CPU much less time (see also [AmRo98]). Although it produces some inaccuracies for message sizes below 64Bytes we assume that the CPU can write at full speed for simplicity (about 82.5MB/s).

Note that the DMA engine actually also takes not that much time as measured by the Ping–Pong test for short transfer sizes. This is essentially the same problem as in case of Shared Memory discussed just before. But we assume that the CPU performance is affected uniformly over the complete transmission time. Although this seems to discriminate DMA in view of CPU utilization, we must remember that DMA setup (building the descriptor etc.) takes the CPU completely. This fact is not respected here. So oriented to the graphs below, DMA has in one point an advantage and in another point a disadvantage over Shared Memory. We made no detailed measurements about this stuff, but both things should compensate each other. However, even if the result is not completely accurate, it gives a raw idea about the facts we want to demonstrate here.

Figure 3 shows CPU time available during Shared Memory and DMA transfer.

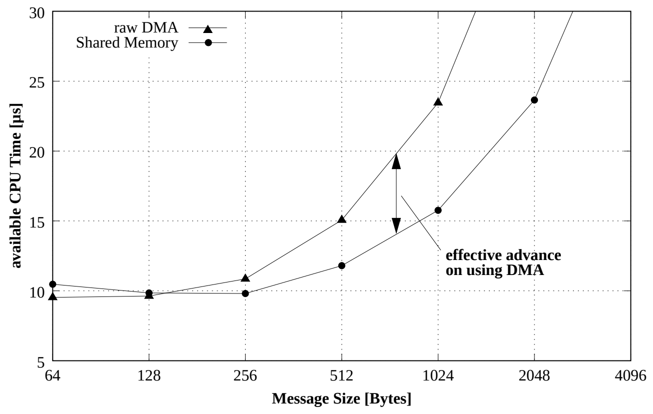


Figure 3: CPU time available during data transfer

The figure shows that from view of CPU utilization up to a message size of about 128Bytes SCI Shared

Memory would be the best choice. For message sizes larger than 128Bytes DMA gives the CPU more time for other computations. Even if CPU performance decreases somewhat more then 15% (e.g. 25%), the switching point increases not dramatically.

Of course, if the CPU has no other work to do (e.g. on blocking sends), then the faster Shared Memory will always be the best way. But even in case of many blocking sends the CPU could be used for more demanding jobs by running more than one computing process or thread per CPU.

2.4 A first Conclusion

Even if a DMA engine may not beat Shared Memory performance in view of bandwidth, it can help to increase general system performance. To use a DMA engine for message sizes as small as possible and to waste at least as possible CPU time a user level DMA mechanism is recommended. In case of very short message sizes SCI Shared Memory is indispensable due to the very short latency.

Currently, Dolphin's PCI–SCI bridges offer a powerful Shared Memory implementation only and a less powerful conventional DMA engine. This is one motivation for us to work on a new PCI–SCI bridge architecture to add a *Protected User–Level DMA* feature.

3 An advanced PCI–SCI bridge

Although we think that it's impossible for us to improve raw bandwidth of Dolphin's PCI–SCI bridge Shared Memory, there are several basic architectural facts to work on. As previously mentioned, the most important architectural improvement is to add *Protected User–Level DMA*. But there are also other facts influencing the design. As an example, Dolphins bridges spend a lot resources to speed up read transactions by implementing separate read stream buffers [Dolph1] and related stuff. However, our focus is on message passing applications, not on shared memory applications. Therefore it's enough for us to write into remote memory except for some things which are difficult to realize using write only. So we can simplify the hardware in this point and implement restricted read operations only.

3.1 Virtual Interface Architecture and SCI

Following projects like SHRIMP [SHRIMP] and U–Net [UNET], the Virtual Interface Architecture (VIA) is the latest initiative to define a hardware communication layer for message passing. And, of course, VIA incorporates protected user level DMA.

3.1.1 A short VIA Overview

We want to show only some basic things here. For detailed information refer to [VIA], [BuGe98], or [Tra98].

In figure 4 the communication stack principle of the VIA is shown.

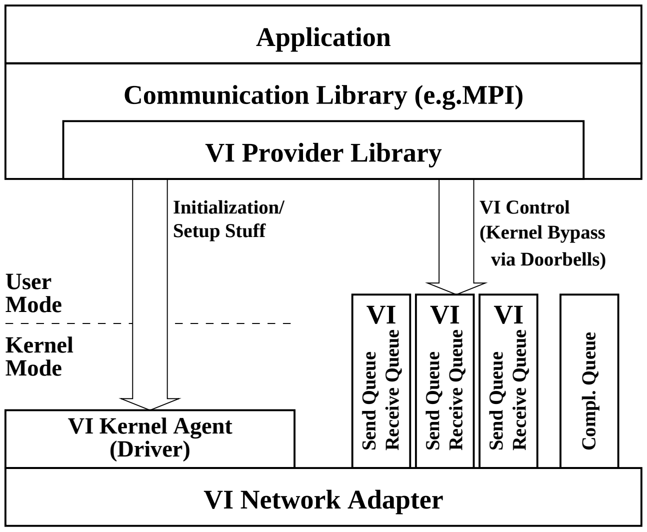


Figure 4: VIA Communication Stack

The VIA defines a hardware offering a set of so called *Virtual Interfaces* (VIs). Every VI represents an autonomous (virtual) hardware providing things like queues for send and receive descriptors. Registers to control the virtual hardware are mapped via *Doorbell* pages into the user process' address space. Prior to execution of a descriptor the hardware checks if the VI the descriptor belongs to is allowed to access the specified addresses. To achieve this check the hardware makes use of *Protection Tags*. Every local memory page the VIA hardware may access is protected by such a protection tag. And every VI has also assigned a protection tag. So the hardware simply compares the VI's protection tag with the memory page's one. This mechanism guarantees that a VI can only access dedicated memory regions.

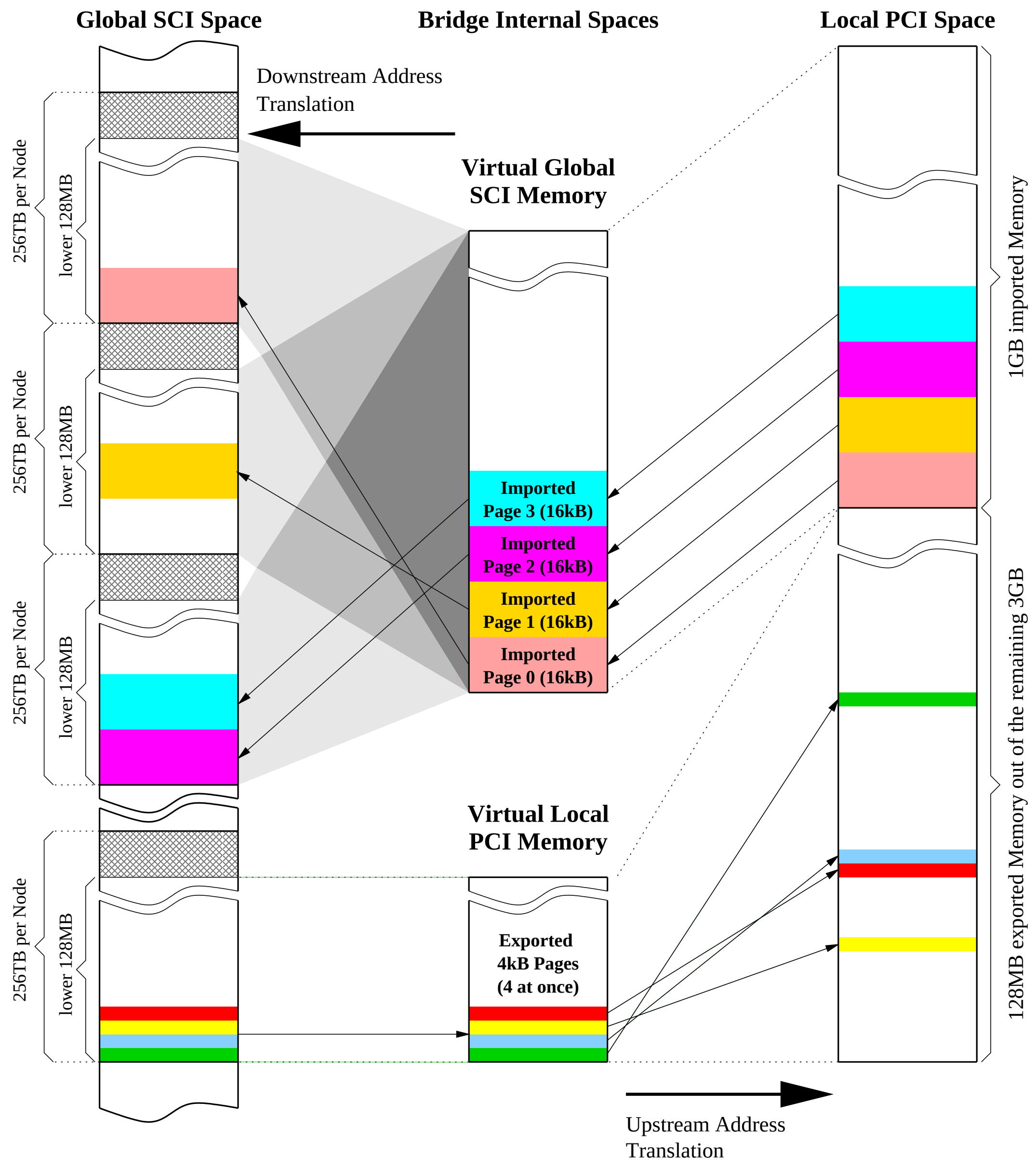


Figure 5: Address Translation Model

Every local VI is logically connected with exactly one remote VI. Every send descriptor inside the local send queue directly corresponds with a receive descriptor in the remote VI's receive queue. This offers a simple, but powerful send/receive mechansim with zero–copy capability. The sender specifies the location of data to send in its send descriptor without the need to know where the data is finally placed in remote memory. This information is taken at the remote node out of the receive descriptor.
Besides send/receive communication, VIA specifies also a *Remote DMA* (RDMA) mechanism. This mechanism is similar to DMA mechanisms on SCI. Hence, the sender has to specify both source– and destination addresses of data to be transfered. This eliminates the need of a receive descriptor.
To improve descriptor completion detection, VIA suggests so called *Completion Queues* (CQ). Every VI may post a completion notification into a CQ after a descriptor is processed. A process typically using some more than only one VI may direct completion notifications into one CQ. This avoids the software to look at a lot of places for the next completed descriptor and finally decreases time of completion to time of completion detection.

As already mentioned, VIA is a pure messaging model. Hence there is nothing like shared memory as we know it from SCI. Although small messages profit from VIA due to the operating system bypass, SCI Shared Memory is even faster for these

message sizes. In addition, there are several things much easier to implement using Shared Memory instead of sending messages (think of mutexes or similar).
Therefore it would be a great advance to have a hardware implementing both techniques. The VIA specification [VIA] allows a wide range for implementations and is therefore suitable for an integration into the SCI architecture (refer also to [Tra98]).

3.2 Major Changes on current PCI–SCI Architecture

The most affecting change is the introduction of a second page based address translation from local memory space seen by the PCI–SCI bridge into real physical local memory space. Figure 5 shows the extended address translation model. While a PCI–SCI bridge needs at least a *Downstream Address Translation* to translate 32Bit PCI addresses into 64Bit SCI address space, VIA needs only an *Upstream Address Translation* for accesses to local memory. This upstream address translation is required to achieve that a virtual contiguous memory region of the user process' address space is also virtual contiguous on the hardware. In addition, the page table for upstream address translation contains the protection tags to perform access protection on a per–page base. Although there are also PCI hostbridges available performing address translation from PCI space into memory space (e.g. the 21174 for Alpha systems [Sam98]), it's at least required to implement the access right checking. Hence there's in either case a page table required. But since it is uncertain whether mainstream PC techniques will also implement such a feature in near future, it is required to offer a complete page table including both address translation and address protection.

As we'll discuss in the next section, the downstream address translation table must also include protection tags when the hardware shall offer a similar DMA mechanism as known from todays' PCI–SCI bridges.

3.3 Remote DMA with SCI/VIA

The following section illustrates some advances of a combined SCI/VIA architecture.
Messaging applications based on *Distributed Shared Memory* (DSM) can profit a lot from VIA functionality. Remember that pure VIA doesn't offer a feature for DSM. But the VIA control path provides an excellent way to reduce communication overhead.

Figure 6 shows an example where four processes are involved in a typical SCI scenario based on a conventional PCI–SCI bridge. There are two independent pairs of processes (A–C and B–D). Each process has own memory which is exported to SCI and imported by the corresponding remote process (compare with colors and/or labels). As an example, process A has imported segment 2 which is actually located in the physical memory of the remote host and owned by process C. When A wants to send some data to C using segment 2, A has the choice whether to do this by simply writing into sement 2 (in case of a short message) or by using the DMA engine (in case of a larger message). But the DMA engine can only be used through the operating system. Neither it's guaranteed that A specified in its DMA command only this local memory owned by A, nor it's guaranteed that A specified the right remote memory location. Although the imported SCI memory could be protected by protection tags (assuming a VIA–like DMA control), this is impossible for local memory. As it can be seen, the DMA engine has direct access to local memory.

Figure 7 shows the same scenario as in figure 6, but with a combined SCI/VIA architecture hardware.

The communication hardware follows the address translation model as shown in figure 5. That is, there are two address translations inside the hardware.
Correlating to the example above, where A wants to send C some data, A has the same choice for communication (Shared Memory or DMA). But now the DMA mechanism can't only be used through the OS kernel. When both address translations incorporate an access right mechanism, the DMA engine is only able to access dedicated address ranges. To achieve this, the translation tables of the hardware inside host 1 have to be initialized so that for segments 1 and 2 the same protection tag is used as it is assigned to the Virtual Interface used by process A.

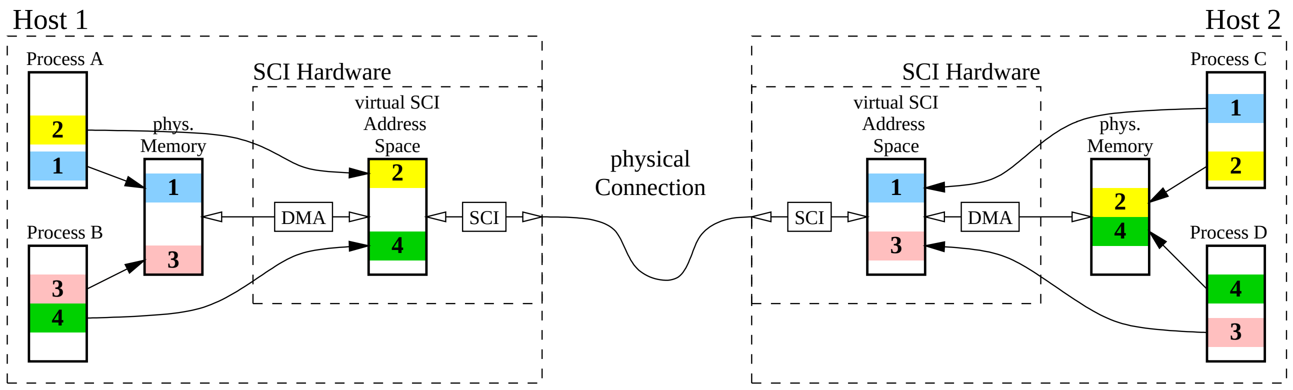


Figure 6: Conventional SCI Model

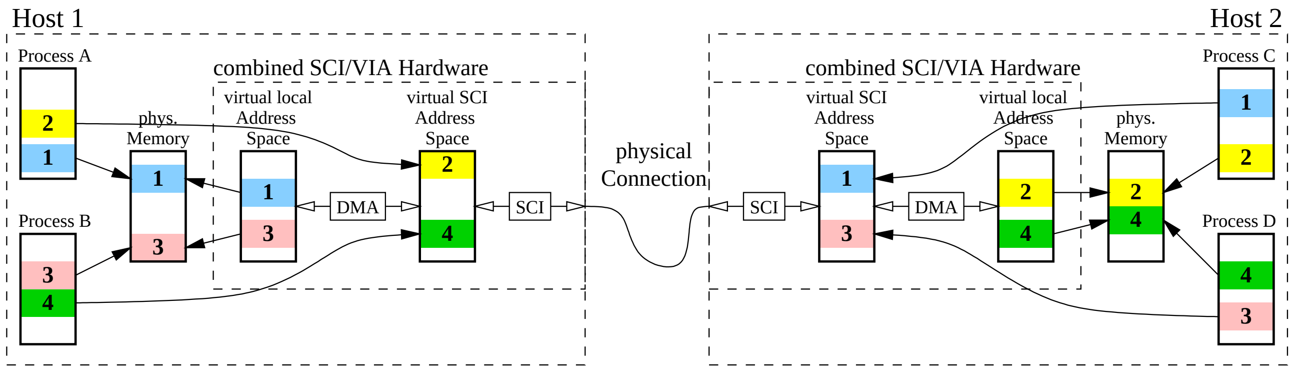

Host 1
Host 2
Process A
Process B
Process C
Process D
combined SCI/VIA Hardware
combined SCI/VIA Hardware
phys. Memory
virtual local Address Space
virtual SCI Address Space
DMA
SCI
physical Connection
1
2
3
4

Figure 7: Combined SCI/VIA Model

3.4 Handling Advances over conventional PCI–SCI

Although the upstream address translation table is no need for SCI Shared Memory (apart from the fact that it makes protected user–level DMA possible), it can help to increase system flexibility a lot. It's no longer necessary to use reserved memory regions to be exported. In case of Dolphin's current PCI–SCI bridge it's in general required to allow incoming memory requests only to a dedicated memory region. The reasons for this restriction are:

- troubles with SCI page size of 512kB handled by the bridge
  A node can export only pages with size of 512kB covering a contiguous range and starting at a 512kB boundary.

- system security
  Dolphin's PCI–SCI bridge potentially exports the whole local physical address range. To avoid illegal accesses to sensible ranges, the bridge allows in practice only accesses to a dedicated region.

These restrictions have a large impact on handling. As an example, it's impossible to implement a zero–copy protocol for MPI. Zero–copy on MPI would mean, that a MPI process must place data to be transfered inside the dedicated memory for SCI communication. But this memory can only be allocated using a **special** `malloc()` function. But the MPI user uses only the **standard** `malloc()` function when he decides to get new memory dynamically. Replacing all standard `mallocs` by special `mallocs` using a special preprocessor is no solution, since exportable memory is limited and not all `mallocs` will be used for communication data. Letting the user the choice of either `malloc` is also not a real solution since it violates a major goal of the MPI standard: **Architecture Independence** [MPI98]. The upstream address translation eliminates all these troubles. Now a node (or a process) may export any physical memory page up to a maximum number of pages.

Btw., troubles and difficulties with the large SCI pages of Dolphin's PCI–SCI bridges were also reported from other working groups concentrating more on shared memory applications than on message passing (e.g. [IBSCH+98],[ScHell98]). So this is a serious task to work on.

Figure 5 not only shows the address translation scheme, but also some quantitative facts planned to implement. In detail, this is a maximum range of 1GB for imported SCI memory and a maximum of 128MB for exported memory. While exported memory is based on a 4kB page granularity, SCI pages are 16kB in size. The larger SCI pages were chosen to reduce the number of page table entries. Of course, this reduces flexibility a bit. But as discussed previously, major problems with SCI pages larger than CPU page size are occuring on the remote node. But due to the upstream address translation based on 4kB pages these troubles won't be there with this PCI–SCI bridge architecture.

4 Summary

We showed that a DMA mechanism for communication can be more suitable than SCI Shared Memory, even if the total bandwidth of DMA is not that much as in case of Shared Memory.
Further we showed that there's a way to improve actual SCI DMA performance.
In addition, several problems with current PCI–SCI implementations were discussed. These facts demonstrate that there's still a lot of work to do in the area of PCI–SCI bridge technology.
In view of the Virtual Interface Architecture, there are currently only a few hardware implementations available, e.g. [GigaNet]. While this implementation is limited to VIA only, our goal is to bring both ones (SCI and VIA) together. In case of the Remote DMA transfer model we showed that SCI and VIA can be efficiently brought together to form a new communication hardware architecture.

With an advanced PCI–SCI bridge architecture optimized for message passing and incorporating VIA functionality, we want to bring our ideas into the hardware layer of the widely spreaded Cluster Computing research area.

References

[AmRo98] 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

[BIP] Loic Prylli and Bernard Tourancheau: *BIP: a new protocol designed for high performance networking on Myrinet.* In: Workshop PC-NOW, IPPS/SPDP98, Orlando, USA, 1998.
See also `http://lhpca.univ-lyon1.fr/`

[BuGe98] Philip Buonadonna, Andrew Geweke: *An Implementation and Analysis of the Virtual Interface Architecture.* University of California at Berkeley, Computer Science Department, Berkeley, 1998. See also
`http://www.cs.berkeley.edu/~philipb/via/`

[Dolph1] Dolphin Interconnect Solutions AS: *PCI–SCI Bridge Specification Rev. 4.01.* 1997.

[GigaNet] GigaNet Homepage
`http://www.giganet.com`

[IBSCH+98] Maximilian Ibel, Michael Schmitt, Klaus Schauser and Anurag Acharya: *An Efficient Global Address Space Model with SCI.* In: Proceedings of SCI Europe '98, Pages 69–79, Bordeaux, 1998

[KuSt98] Ch. Kurmann and T. Stricker: *A Comparison of two Gigabit SAN/LAN technologies: Scalable Coherent Interface versus Myrinet.* In: Proceedings of SCI Europe '98, Pages 29–40, Bordeaux, 1998

[MPI98] Message Passing Interface Forum: *The MPI message–passing interface standard Rev. 2.0*, May 1998
`http://www-unix.mcs.anl.gov/mpi/`

[MYRI] Collected Myrinet Performance Measurements.
`http://www.myri.com/myrinet/performance`

[PCI98] Wolfgang Rehm, Friedrich Seifert, Mario Trams: *PCI–Analysator entlarvt lahme Chipstze.* In: Elektronik 11/1998, Pages 88-94, WEKA Fachzeitschriften-Verlag GmbH Poing

[Sam98] Samsung Electronics: *AlphaPC 164 UX/BX Motherboard Technical Reference Manual.*
`http://www.usa.samsungsemi.com/products/browse/alphaboards.htm`

[ScHell98] Martin Schulz and Hermann Hellwagner: *Global Virtual Memory based on SCI–DSM.* In: Proceedings of SCI Europe '98, Pages 59–67, Bordeaux, 1998

[SHRIMP] The SHRIMP Project: *Scalable High–performance Really Inexpensive Multi–Processor.*
`http://www.cs.princeton.edu/shrimp/`

[Tra98] Mario Trams: *Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface.* Diploma Thesis, Dept. of Computer Science, University of Technology Chemnitz, 1998. See also
`http://www.tu-chemnitz.de/informatik/RA/themes/works.html`

[UNET] The U-Net Project: *A User–Level Network Interface Architecture.*
`http://www2.cs.cornell.edu/U-Net`

[VIA] Intel, Compaq and Microsoft. *Virtual Interface Architecture Specification V1.0.*
Virtual Interface Architecture Homepage
`http://www.viarch.org`