SCI Transaction Management in our FPGA–based PCI–SCI Bridge

Mario Trams, Wolfgang Rehm

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

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

Abstract— **Our VIA–capable PCI–SCI bridge was already introduced to the community in several papers (e.g. [3], [4]). While in these publications the focus were set at the advantages of the integration of VIA characteristics into an SCI architecture and the positive impact on message passing libraries, this paper is intended to provide the reader with a couple of implementation details.**

Functionality and cooperation of many of the hardware components are described and deepened by tracing a remote write operation throughout the whole system from PCI bus to PCI bus. This trace also discloses the composition of the latency.

Keywords— **Message Passing, Cluster Computing, Protected User–Level DMA, Distributed Shared Memory, Scalable Coherent Interface, Virtual Interface Architecture, Network Interface, PCI, FPGA**

I. Motivation and Introduction

Since 1996 we put our efforts in SCI technology. While in the beginning this was limited by the use and programming of commercially available hardware, this changed when we received two PCI–SCI bridges developed by the CERN (Switzerland) RD24 project [1]. These bridges were based on reconfigurable FPGAs (Field Programmable Gate Arrays), but were not fully implemented that time. Nevertheless we saw a big potential and continued FPGA implementation. Meanwhile the Virtual Interface Architecture (VIA) came up and some ideas/methods used there seemed to us very useful to have also in an SCI hardware. Although the CERN PCI–SCI bridge was reconfigurable, it was quite old already and there were several architectural reasons that led us to the decision to build up a new FPGA–based PCI–SCI bridge with some changes and latest technologies. In 1999 we had the first prototype of our hardware ready and since that time we are about to implement FPGA functionality step by step.

Although SCI is intended for distributed shared memory (even cache coherent), several groups [11], [16] including our working group [9], [10] use SCI as low–level layer for message passing. This results in very low messaging latencies. Also in the particular case of PCI–SCI interfaces there is no cache coherency which makes this hardware family more suitable for message passing rather than shared memory.

The key points that our design shall realize in addition to features offered by todays commercially available PCI–SCI hardware are:

- Protected User–Level DMA for large block transfers to unload the CPU whenever possible.
- An improved memory management to increase flexibility for exporting local memory to remote nodes and thus making real zero–copy possible.

This paper is intended to provide some concrete information about the internal operation of the hardware, especially the internals of the FPGAs used in our design where the majority of the know–how is concentrated. This includes also some performance data showing that our proof–of–concept design is not too bad compared with other PCI–SCI hardware.

At the beginning, the hardware architecture and particulary the logical structure of the FPGAs are discussed. Based on this information some important details about write operations are stressed and the overall bridge operation is demonstrated by tracing a single remote write operation through the system. The bandwidth for remote write operation based on SCI *dmove* transactions is shown as well.

II. The Basic Hardware Architecture

Figure 1 describes the general architecture of the PCI–SCI bridge. The central units in the design are the both FPGAs — The PCI FPGA and the SCI FPGA. Contents and operation of these reconfigurable programmable logic devices are described in separate sections later.

Other important components are the SCI Link Controller (LC–2 from Dolphin [15]), the Dual–Ported Memory (DPM) and the Static Memory (SRAM). While the DPM is mainly intended for SCI or rather BLINK[1] packet storage, the SRAM contains important information for the control flow: *Upstream Address Translation and Protection Table*, *Downstream Address Translation and Protection Table*, and *Virtual Interface Context Memory.*

The hardware architecture contains also a special PCI–to–PCI bridge. Although this chip doesn't implement any conceptually important function, it simplifies several things that are outside the scope of this paper. As described later, this chip even has some bad impact on la-

[1]BLINK is the *Backside–Link* of the LC–2. SCI packets are encapsulated in BLINK packets which are 8 bytes larger than the actual SCI packet.

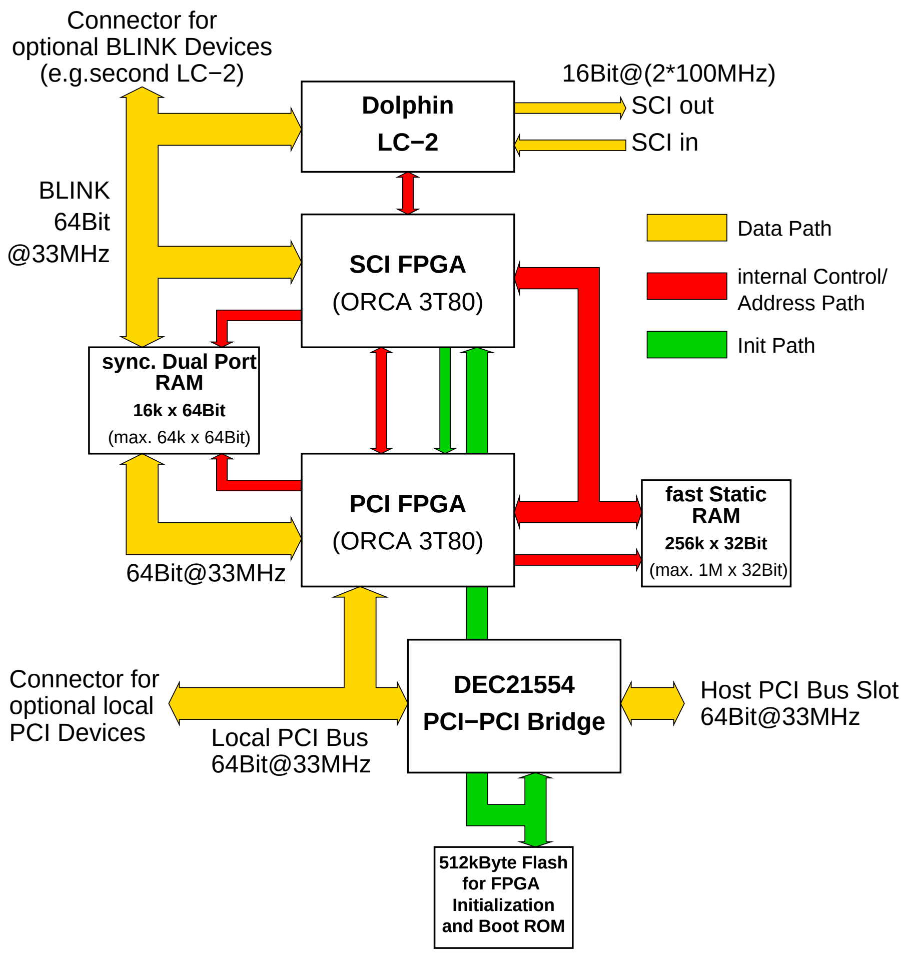

Fig. 1. Hardware Architecture

tency. But nevertheless we decided to use it to not make the design more complicated than it already is. The shown Flash EEPROM and the corresponding Init Path is also important for the initialization and reconfiguration of the PCI–SCI board, but has no meaning during actual operation.

A. The PCI FPGA

The PCI FPGA implements (or is intended to do so) key features of the whole PCI–SCI bridge:

1. Translation of PCI into SCI transactions and vice–versa.
2. Downstream Address Translation for outgoing transactions.
3. Upstream Address Translation that is required to ”virtualize” the exportable memory for exporting any arbitrary memory page rather than a fixed memory portion.
4. Protected User–Level DMA Engine including Virtual Interface Architecture functionality (Doorbells, Work Queues, ...) to offer a handy mechanism for user processes to use block–moving DMA instead of the processor for data transmission.

While the first two points are known from todays commercially available PCI–SCI hardware, the remaining ones are dedicated only to our hardware solution.

Figure 2 gives a simplified view of the modular structure and the communication paths of the PCI FPGA internals.

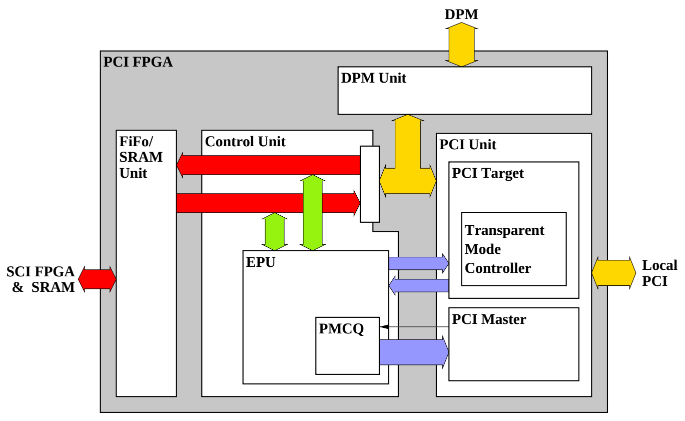

Fig. 2. Simplified internal Structure of PCI FPGA

A.1 PCI Unit

There’s a PCI Unit that is working relatively independent from the rest of the logic and contains both the PCI Target and PCI Master. Independence means here that the PCI Unit is able to transfer data between Dual Ported Memory (DPM) and local PCI bus in parallel to work of remaining units in the FPGA. This is very important in order to achieve a high internal parallelism for large throughput.

The PCI Target is responsible for all accesses initiated by the host processor, such as DPM & SRAM accesses, FiFo interface accesses (see later), and accesses to internal Configuration and Status Registers (CSRs) that are not shown in figure 2. And, of course, the most important function is the handling of accesses to imported remote SCI memory. For the latter one there's a lot of logic encapsulated in a separate module — the Transparent Mode Controller (TMC). The PCI Target and in particular the TMC is working here in strong interaction with the EPU (see later) that performs the difficult management operations.

The PCI Master receives its commands from the EPU and mainly transfers data automatically between DPM and host main memory. Later, the PCI Master shall be able to merge continuous commands (if possible) in order to achieve long PCI burst transfers. Currently it starts a new PCI transaction for every new command and thus limiting the maximal burst length to 64 bytes.

A.2 Control Unit

The Control Unit implements some central control functions inside the PCI FPGA. The most important component contained inside is the so–called EPU. EPU is an acronym for *Embedded Processing Unit* and is comparable with a small microprocessor or rather micro controller. The EPU consists of a 2–stage execution pipeline (fetch and execute) and implements relatively complex operations while keeping a RISC–like instruction coding for simplicity. Some examples of currently implemented operations are command exchange with the SCI FPGA, address translation using the tables stored in the external SRAM, and control of the PCI Master to get access to host main memory. Interfacing with the PCI Master is performed through a command queue (PMCQ — *PCI Master Command Queue*) that can take up a number of commands.

The EPU plays a central role inside the PCI FPGA since all relatively complex operations maintained by this FPGA are performed there. Besides operations needed for remote write/read operations that will be discussed later, the DMA engine functionality that is not implemented yet is a good example for a compound of a bunch of single operations that need to be done:

1. A DMA descriptor has to be fetched from main memory.
2. The right downstream address translation and protection table (DownATPT) entry has to be read to determine the global SCI address as well as to check whether the DMA on this area is allowed.
3. The right upstream address translation and protection table (UpATPT) entry has to be read to determine the local host PCI address as well as to check whether the DMA on this area is allowed.
4. PCI read transactions (or SCI read transactions) have to be initiated.
5. If a read transaction has finished, an appropriate SCI write respectively PCI write transaction has to be performed.
6. When a DMA block transfer has finished, the DMA descriptor must be changed to mark it "ready".

The designers of the PCI–SCI bridge developed at the University of Munich [7], [8] had similar problems to solve. Finally they decided to implement a small and simple, but very effective microcode sequencer.

Another famous example for a processing unit embedded in a communication hardware is Myricom's LANai chip [21] for their Myrinet cards. Besides some other components such as the low–level wire interface, this chip contains a small 32 Bit Load/Store RISC CPU. However, this processor is not very specialized for its purpose and is very similar to general–purpose CPUs that use memory–mapped registers for communication with other units in the system. Although this general "touch" simplifies the design process a lot, it wastes also a lot of performance potential.

In the early stages of our design we planned to use such mechanism as it is used in the Munich PCI–SCI bridge. However, the problem with microcode sequencers is that operations that need N cycles to complete also need N microcode instruction words. The microcode sequencer in the Munich bridge can at least handle idle or wait cycles without special wait–instructions. This saves many microcode instructions.

Nevertheless, the tasks to be performed by our processing unit are much more complicated and expensive. Apart from remote memory access operations such as PIO or RDMA, the doorbell and work queue handling implied by the VIA functionality will need support by the processing unit to a great extend (lots of small different tasks). This would substantially increase the hardware expense for a conventional microcode sequencer — especially the depth of the instruction memory had to be increased which would cause probably the largest problems. Therefore we decided to use a real processor model rather than a microcode sequencer.

In comparison with the processor inside the Myrinet LANai chip, this processor architecture is very specialized and practically not usable as general–purpose CPU — There are even no general–purpose registers. Further details of the EPU operation are outside the scope of this paper.

A.3 FiFo/SRAM and DPM Unit

These are other semi–autonomous units that are responsible for executing commands such as reads/writes to Dual Ported Memory or Static RAM, or pushes and pops to/from the several command FiFos contained in the SCI FPGA.

B. The SCI FPGA

The job of the SCI FPGA is concentrated on dealing with SCI packets and talking with the SCI Link Controller. A very raw overview of the logical structure of the SCI FPGA is shown by figure 3.

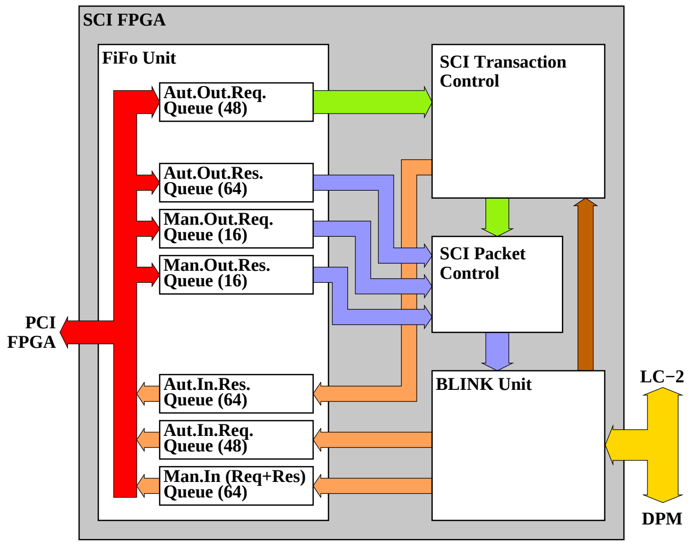

Fig. 3. Simplified internal Structure of SCI FPGA

B.1 FiFo Unit

The FiFo Unit of the SCI FPGA interface consists of a set of outgoing and incoming queues for SCI packets that can be accessed by the PCI FPGA. Also shown by the figure is the nominal depth of each queue.

Since SCI is based on a split–transaction protocol and SCI transactions are usually divided into requests and responses, the SCI FPGA has to handle both packet types separately to guarantee deadlock–free operation.

Note that only control information such as buffer IDs, transaction IDs, addresses, or status bits are physically stored in all these queues. The actual data body of SCI packets is exchanged between the PCI FPGA and the SCI Link Controller via the Dual Ported Memory.

Inside the FiFo Unit there is one queue that stores both manual incoming response and request packets. For automatic incoming packets two queues are used — one for requests and another for responses.

Queues for outgoing packets are always split into separate queues for requests and responses.

The sizes of the queues are well selected so that neither deadlocks nor queue overflows can occur.

Note: The hardware supports two basic packet modes — a manual packet mode and an automatic packet mode. The manual packet mode is intended for sideband communication and debugging purposes where SCI packets have to be assembled and sent out explicitly by software. The automatic packet mode covers all other transactions managed completely autonomous by the hardware.

B.2 SCI Transaction Control

This component is responsible for dealing with automatically generated requests coming from the PCI FPGA via the *Automatic Outgoing Request Queue*. These packets need some special care. This includes for instance:

- Transaction ordering must be ensured in some cases.
- Appropriate actions have to be done when a response doesn't come back after a certain amount of time (Time–Out).

In order to achieve such functionality, incoming SCI responses are forwarded from the BLINK Unit to the SCI Transaction Controller.

B.3 SCI Packet Control

Another component shown in figure 3 is the so–called SCI Packet Control component. This unit more or less multiplexes packets to be sent out from different sources and hands them over to the BLINK Unit

B.4 BLINK Unit

The BLINK Unit represents the interface for the LC–2 and sends out or receives packets. Incoming packets are decoded and forwarded into the right incoming queue. Is there no free incoming buffer available, the packet transfer is rejected and the LC–2 has to re–send it later. In case of incoming automatic request packets the BLINK Unit also keeps in mind some data originating from the request. This information is later needed in order to generate the right response packet. An example is the source node ID of the request that has to be used as destination node ID for the response when it is sent back later (in case of non–responseless transactions).

While the whole BLINK packet is taken out of the DPM when an outgoing manual packet is being handed over to the Link Controller, this is not true for automatically generated packets. In this case only the data body is taken from DPM and both header and trailer of the BLINK packet are injected by the SCI FPGA on–the–fly.

Incoming response packets that belong to automatically generated transactions (such as transparent read or write) must be feed to the SCI Transaction Controller that will put them into the *Automatic Incoming Response Queue*.

Incoming packets are always completely stored inside the DPM, even if this is not really needed since required header information of automatic packets is stripped by the SCI FPGA for further processing.

III. Basics of Remote Write Operation

For remote memory write accesses the PCI Target supports similar to Dolphins PSB chip (PCI–SCI Bridge chip) a set of write buffers — four pieces. These write buffers can take up a consecutive block of up to 64 bytes each. More correctly said, they can take up to 16 consecutive 32 bit words.

A. Write Buffer States

For each of the four write buffers there's a small state machine as illustrated by figure 4.

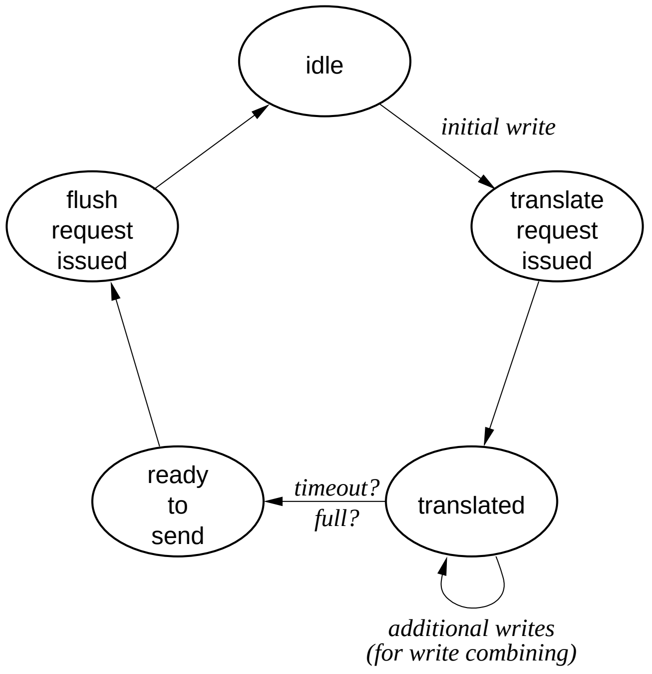

Fig. 4. Simplified Write Buffer State Machine

In the two states surrounding the *idle* state the state machine basically waits for the execution of EPU routines. Stays a write buffer in *translate request issued*, an appropriate command was given to the EPU to perform a downstream address translation and to fetch other attributes of the accessed SCI page. The last state — *flush request issued* — has a similar meaning since the state machine has to wait there until the EPU has handed over the right command into the Automatic Outgoing Request Queue of the SCI FPGA.

A write buffer can accept further data transferred by a later, but not necessarily subsequent PCI write transaction when the buffer stays in the *translated* state. The only allowed write–combining strategy for PCI write transactions is append–at–top (growing addresses). This functionality is currently implemented in VHDL code, but has not yet been synthesized and tested. Therefore the four write buffers have never been used concurrently yet.

The state *ready to send* is entered whenever there's no more data to add to the write buffer. This state causes a command sent to the EPU for handing over the write buffer to the SCI FPGA for further processing.

B. Write Buffers vs. Outstanding Transactions

Although there are only four write buffers available, the hardware design supports up to 48 outstanding remote write transactions[2]. This is achieved by decoupling write buffers from outstanding transactions as illustrated by figure 5.

[2]To save hardware ressources, only 32 are enabled yet

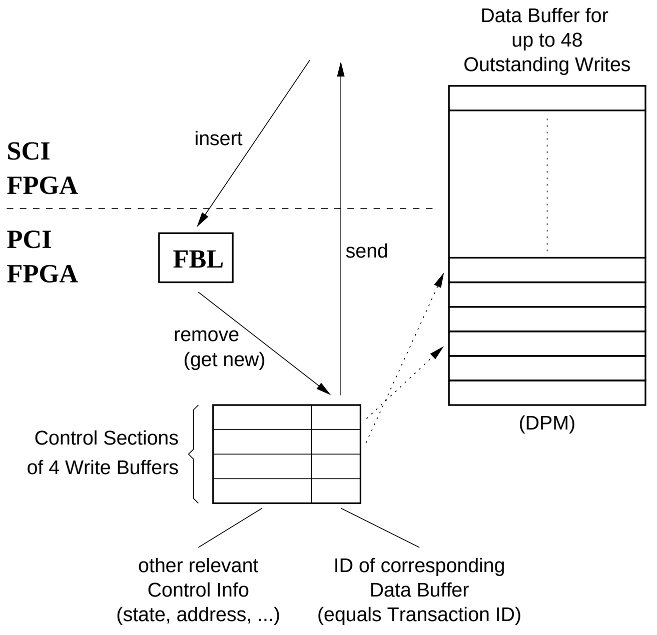
Data Buffer for up to 48 Outstanding Writes
insert
SCI FPGA
PCI FPGA
FBL
send
remove (get new)
Control Sections of 4 Write Buffers
(DPM)
other relevant Control Info (state, address, ...)
ID of corresponding Data Buffer (equals Transaction ID)

Fig. 5. Transaction Buffer Assignment for Write Buffers

The mechanism is working quite simple. There is a Free Buffer List (FBL) that contains all 48 (or 32) buffer identificators that have a direct correlation to the 64 available SCI transaction IDs. When a new write buffer is being opened, the next free transaction ID is fetched from the FBL. When the write buffer content (or rather relevant control information of the write buffer) is handed over to the SCI FPGA, this write buffer becomes free and can be reused immediately instead of waiting until the transaction has completed finally. Once it has completed anytime in future, the transaction ID is brought back into the FBL.

The advantage and intention of this strategy is to keep the amount of relatively expensive write buffers at small numbers while allowing as much outstanding transactions as possible. The handling of lots of outstanding transactions inside the SCI FPGA requires a comparatively small amount of hardware ressources than the same number of "real" write buffers which would occupy a large portion of the PCI FPGA.

The transaction IDs stored inside the FBL are not used exclusively for remote write operations. All transactions except manually created ones take transaction IDs and hence data buffer storage from the same pool.

C. Supported data types and data encapsulation

Since the design is primarily intended as optimized communication hardware for message passing libraries, we decided to support aligned 32 bit words as smallest unit (in the following text only referred as *word*). This simplifies the logic at some points and thus reduces the required hardware resources and can help to speed up some things.

In which way are these single words encapsulated inside SCI packets? As described by the SCI standard [17], a chunk of consecutive data up to an amount of 16 bytes within a 16–byte aligned block can be transferred by using *writesb* transactions. However, to avoid too many different types of SCI packets that each require a completely different handling we have "misused" standard *nwrite16/nwrite64* packets (*dmove* respectively) in order to transfer less than 16 or 64 bytes of data. Actually, the used mechanism is very simple and similar to this one used in *writesb* transactions. Figure 6 shows an example of a partly filled 64–byte–payload packet.

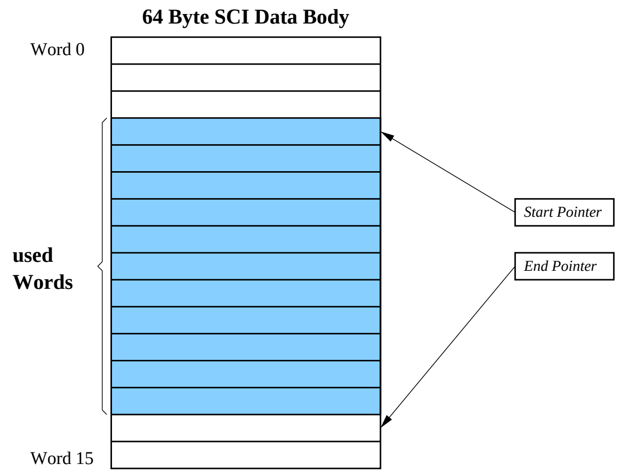

Fig. 6. Example of partly utilized data body of an SCI write packet

There are two small pointers encapsulated in otherwise unused high–order SCI address bits that specify start and end (+1) of the valid range. These pointers are finally evaluated by the PCI Master in order to start and finish the PCI write burst at the right offsets.

D. dmove or nwrite?

As mentioned previously at several occurances, the hardware is able to generate either *dmove* or *nwrite* transactions. Advantages and disadvantages of both shall be well known and are not to be discussed here. The decision whether to generate *dmove* or *nwrite* is very simple as this is just one attribute of each SCI page. That is, each *Downstream Address Translation and Protection Table* entry contains a single bit that specifies whether *dmove* or *nwrite* transactions shall be used. This bit has no meaning for the PCI FPGA. It is only the SCI Transaction Controller inside the SCI FPGA that is concerned about this issue.

IV. Remote Write Execution

This section gives some detailed information of the flow of a write transaction through the system. The values given here are partly measured in reality and partly derived from the design (simulation). Real–world measurements were made on a 667MHz Alpha 21264 (EV67) system based on the Tsunami chipset with 64Bit/33MHz PCI bus.

Figure 7 illustrates the composition of the remote write latency. As it can be seen there, it takes a total amount of 89 PCI cycles (33MHz; 30ns cycle time) that is required for the transmission of one word from start of the PCI bus transaction on the initiating node to end of the PCI transaction on the destination node. This results in a hardware latency of 2.67μs.

The latency components and actions performed by each hardware module for a single one–word write operation are explained step by step as following:

A: During the first step of a remote memory write the host chipset places the write transaction onto the host PCI bus where it is forwarded through the 21554 PCI–PCI bridge. This takes 5 cycles.

B: In phase B the write transaction appears on the local PCI bus and is taken by the PCI FPGA. There it is further processed and handed over to the SCI FPGA (inserted into the *Automatic Outgoing Request Queue*). All single steps included by the "further processing" are not visible in figure 7. However, here they are:

1. When the PCI Target detects the remote write operation it is immediately assigned to a write buffer and written data is forwarded to the right place in the DPM. According the data buffer assignment mechanism described earlier by figure 5 the right SCI transaction ID (and hence, DPM location) is fetched from the corresponding FBL. However, this does not consume extra time since there is always an ID prefetched for immediate use.
2. The write transaction has caused the execution of an EPU routine and forced a transition in the write buffer state (refer to figure 4). The EPU routine consists of two instructions: perform a downstream address translation including a fetch of some other attributes related to the accessed SCI page (two read cycles from the on–board static RAM) and inform the Transparent Mode Controller about the completion of the translation operation.
3. The completion of the translate request forces a write buffer state transition to *translated* where the buffer is able to accept other write transactions to append them, if possible.
4. Since currently there's no write combining synthesized, the buffer state is immediately set to *ready to send* and again an appropriate command is given to the EPU for flushing the write buffer (flushing in this context means to hand it over to the SCI FPGA). The corresponding EPU routine consists of two instructions again, where the first is writing the right command into the *Automatic Outgoing Request Queue* and the second informing the Transparent Mode Controller as done previously.
5. The completion of the flush command causes a state transition back to *idle* and the write buffer can be used for new operations.

Some additional notes:

– Currently there are always write requests based on a 64 byte data body generated (nwrite64 or dmove64). If possible, 16 byte packets shall be generated later in order to reduce SCI traffic.

– In case of a PCI write burst, the PCI Target is able to forward the whole burst into the DPM in parallel to

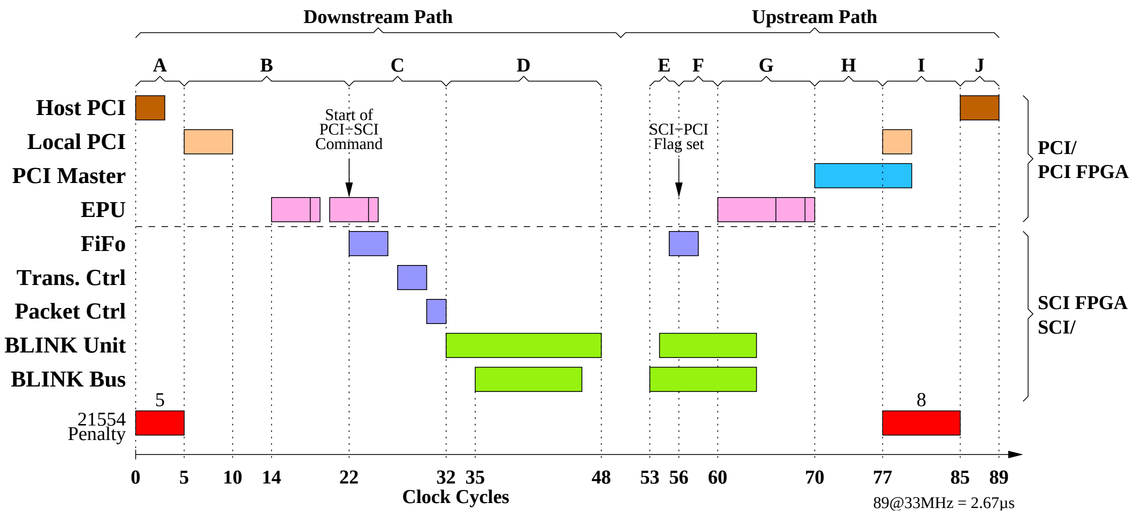

Fig. 7. Latency portions for Remote Write (1 word encapsulated in dmove64 or nwrite64)

EPU operation. If the burst attempts to cross a 64 byte boundary the PCI Target issues a *Target Disconnect.*

– Of course, the PCI Target can accept new write transactions into other write buffers while the EPU is serving some requests — everything is made highly parallel.

C: The write operation ripples through the major control sections of the SCI FPGA in phase C.
The SCI Transaction Controller removes the transaction out of the queue and takes further care for it. This means, for instance, to watch for possible SCI–response timeouts and to take appropriate actions in such case. Other functionality that is planned here is to ensure transaction ordering for up to 16 different ordering contexts [6]. However, this is not implemented yet and outside the scope of this paper.
Basically, the SCI Transaction Controller prepares everything that is needed by the BLINK Unit to hand–over the right packet to the Link Controller (nwrite64 or dmove64 in this example). However, before the final command arrives at the BLINK Unit, it has to pass the Packet Controller that acts as a simple multiplexer scheduling different sources of outgoing packets.

D: The BLINK Unit needs three cycles before it starts driving the BLINK packet onto the bus. After the packet has been transferred it takes the BLINK Unit two more cycles for some post–processing. Nevertheless, the BLINK Unit can overlap the processing of consequtive transactions that is important for bandwidth considerations (meaning it doesn't need 16 cycles for sustained operation).

The remaining time needed until the request arrives on the other BLINK bus depends on some SCI settings and ring configuration. The value of 18 cycles BLINK–to–BLINK latency is based on a loop–back connection, a 100MHz SCI link frequency and some special kind of cut–through settings for BLINK–to–SCI and SCI–to–BLINK transfers. Actually, 18 cycles are quite a lot of time here. Maybe this is caused by the loop–back transfer and it takes some less amount of time when the destination is another LC–2.

E: As soon as the BLINK Unit receives the write packet, a notification is inserted into the *Automatic Incoming Request Queue.* The FiFo Unit immediately sets a flag to inform the PCI FPGA about the new packet. The cycle count from the beginning of packet arrival to information of the PCI FPGA via this special flag is with 3 cycles very fast.
The BLINK Unit also allocates an incoming buffer and controls the Dual Ported Memory so that the BLINK packet is stored at the right place there.

F: It takes another 4 cycles until the EPU starts to process the incoming request. This delay is mostly caused by some EPU–specific mechanisms.

G: The EPU–routine for processing incoming request packets (remote read/write) consists of three instructions. The first one loads the command from the *Automatic Incoming Request Queue* and branches to another routine depending on the type of the request. In case of remote read/write the second instruction determines the host physical PCI address by performing a lookup in the *Upstream Address Translation and Protection Table* that is stored in the static RAM. Finally, the third instruction issues the right command to the PCI Master by writing it into the *PCI Master Command Queue.*

H: It takes the PCI Master 7 cycles to arbitrate for the local PCI bus and prepare for the transaction until it is started. Preparation means here especially to issue the right transaction to the DPM Unit (remember that the data to be written is stored inside the DPM).

I: The DEC bridge steals us another 8 cycles before it starts the PCI transaction on the host PCI bus. It is not completely clear why it takes 8 cycles here, since a write from host to local PCI takes only 5 cycles (see portion A).

J: The write operation onto the host PCI bus has a duration of 4 cycles and after a total delay of 89 cycles the remote write operation of the single word is finished.

Actually the operation has not completely finished at this moment, since for a total completion the incoming buffer has to be made free again, and in case of *nwrite* a response packet has to be sent back.

A. *Valuation of the Latency*

With a hardware latency of 2.67μs for remote write operations we have achieved a value that we expected based on a so–called half–automatic transfer where the BLINK request packet was pre–prepared by software and sent out manually at the sending node [5].

A few tenths of a microsecond (dependent on the host system) have to be added to this hardware latency to get the software latency (CPU–to–CPU) that is slightly more than 3μs. As comparison, Dolphins ASIC solutions achieve software latencies of 2.3μs in case of the PSB32 chip together with the LC–2. The latest hardware solution, the Dolphin D330 (PSB66 along with LC–3) pushes the latency even more down to 1.5μs.

However, we think that our achieved latency is not too bad for a proof–of–concept and actually we were a bit surprised that we could manage it to keep the latency that low. It is also not completely clear to what extend Dolphins hardware benefits from the higher frequency. Even in case of the PSB32, although it has a 33MHz PCI interface, the BLINK side is clocked at 50MHz and it is not known how many percent of the transaction handling are performed in the BLINK and PCI clock domains.

B. *What about Bandwidth?*

Figure 8 shows the bandwidth we could obtain so far on Alpha UP2000 systems with 64Bit/33MHz PCI Bus.

It is important to note that all involved SCI transactions were *dmove* operations (*dmove64*). That is, no responses were generated at the receiver, and thus the packet throughput is not affected by this additional SCI traffic. Of course, *nwrite* transactions are supported as well. However, there is currently the solution of a logical hardware problem pending, that causes control flow confusions when response packets have to be handled.

Because of the *nwrite*–problem, the shown bandwidth curves should not be put into a 1:1 comparision together with other SCI remote write measurements that usually use *nwrite*.

The most important one of the three bandwidth curves is the lower one, the *SCI Ping–Pong* curve where blocks of data were written into main memory of another node which copied the same data back. 64bit write operations were used in all cases. At a block size of approx. 8kB the highest performance is reached with a bandwidth of about 116MB/s.

The *SCI Write–only* curve shows the bandwidth for 64bit write operations where the receiver did not sent any data back. However, it was ensured that the whole written block has left at least the host chipset. This can be achieved by a PCI read operation (e.g. from DPM) that is

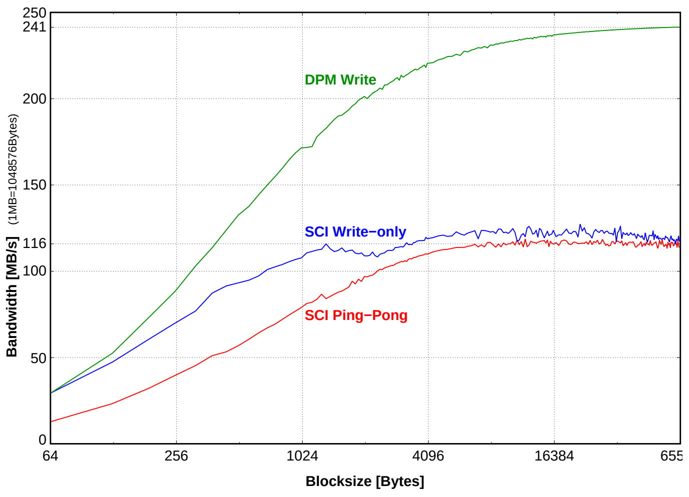

Fig. 8. Write Bandwidth measurements

surrounded by memory barriers and placed after the write loop.

Finally, the third curve (*DPM Write*) represents the write performance into the on–board Dual–Ported Memory and includes the chipset–flush as well. This one can be considered as upper limit for the remote write performance since basically it represents chipset and PCI capabilities.

The gap that opens between *DPM Write* and *SCI Write–only* curves for block sizes up to slightly more than 1kB is likely caused by the fact that the PCI target disconnects PCI bursts at 64 byte boundaries for remote write operations while it accepts virtually endless bursts into DPM. The hardware reaches its maximal remote write capabilities at a block size of about 4kBytes.

Two further effects seen in figure 8 concerning the *SCI Write–only* curve can be noted:

- The local bandwidth minimum at around 2kBytes that is not completely allegeable yet. It may be caused by some interferences that hurt the performance in this region.
- The bandwidth for larger block sizes is with about 120MB/s somewhat higher than the ping–pong bandwith and approaches slowly 116MB/s. This effect is likely caused by the large input buffer capabilities at the receiver (64 packets, or 4kBytes). Further it raises the assumption that the sender is able to produce data faster than it can be processed by the receiver. Indeed, this assumption could already be verified by simulations.

Generally there is to say that the whole internal architecture of the PCI–SCI bridge is highly pipelined. The maximal bandwidth can be calculated by

$$Bandwidth = \frac{Blocksize}{N \cdot Cycletime}$$

where N is the cycle count of the longest pipeline stage in the flow. *Cycletime* is 30ns in our design (33MHz) and the *Blocksize* is 64Bytes.

Assuming that there is no stage in the pipeline that needs a total amount of more than 16 clock cycles per 64 bytes of data, the bandwidth should be around 127MB/s (1MB=1048576 Bytes). This value is more or less reflected by the current status of the implementation, taking into account that the PCI chipset does not always generate full 64 Byte bursts. However, there is still some space for a few improvements and actually we hope to reach a value somewhere around 150MB/s (13–14 cycles worst case per stage) for remote write operations. A migration to the third generation Link Controller, the LC–3 that supports 128 byte SCI packets would further relax this situation greatly, since the cycles needed in each pipeline stage are usually fixed and independent of packet size (except in case of PCI and BLINK bus transfers).

C. Potential for Improvements

Besides increasing the frequency by using faster building blocks (FPGAs, etc.) there's still potential to save a few more cycles. However, for big steps a redesign of the hardware would be required. For instance the extra–penalty caused by the DEC 21554 PCI–PCI bridge is with 13 cycles not less and could be eliminated by using a better FPGA where a PCI–PCI bridge is not needed any more. Another issue is the separation of the whole logic into two FPGAs. If all these things could be placed in one FPGA, the handshaking between PCI– and BLINK–related functional units could be made much more efficiently and the latency could be reduced by maybe 10–15 cycles without a complete redesign of the general logical structure.

And as mentioned just before, the LC–3 and the use of 128 byte packets will lead to a much better bandwidth.

D. What about Read Operations?

Indeed, remote read operations are supported as well and the required functionality is nearly completely implemented. However, we have to mention that it is not our plan to support this functionality to a great extend. The reason for this is again our focus on message passing libraries where remote write operations are much more important than remote reads.

V. Available Software

Apart from a low–level driver and a couple of configuration and test programs for x86 as well as for Alpha Linux systems there is an extended Virtual Interface Provider Library (VIPL) available [20], [12]. The VIPL can be considered as a low–level communication API for hardware based on the Virtual Interface Architecture [19] comparable with SISCI [18] for SCI. It wraps driver and hardware functions and special memory operations provided by a a dedicated memory management module [13] and delivers all this functionality in a more or less standardized way (ExtVIPL) to upper software layers. The phrase *"more or less"* means that the original VIPL had to be extended by a few functions in order to include the Distributed Shared Memory functionality that is not part of the VIA specification.

Although there are made no deeper investigations yet, there are also plans to provide a SISCI API [18] for this special hardware. However, the original SISCI API is closely bound to Dolphins SCI hardware architecture features and therefore there are some doubts that the whole API can be supported 1:1. Nevertheless essential functions such as `Create/Connect/MapSegemt` should be easily adaptable.

Concerning higher software layers there is an ongoing work to develop an MPI library [9], [10] that is able to take the full advantages of the underlying hardware. An interesting piece of software is the so–called *VIA Registration Management*, or in short VRM [14], that is able to hide the limited hardware ressources (amount of exportable and importable memory). Basically, this functionality is achieved by automatic deregistration of areas that have not been used for a longer period and automatic registration of areas that are needed but not registered at that moment (similar to memory caching strategies). A more detailed description is outside the scope of this paper.

VI. To Do

The completion of the full intended hardware functionality still requires a lot of work. In particular this includes the DMA engine and the associated VIA Doorbell functionality that is needed to support true protected user–level DMA within an SCI environment. As previously mentioned, most of this functionality requires the integration of new instructions to the embedded processing unit.

Another important functionality that has not been implemented yet but is needed in junction with remote write operations is a mechanism to guarantee write transaction ordering. In case of Dolphin's hardware so–called *Store Barriers* are used for this purpose. However, we have planned to use another mechanism were write operations made onto SCI pages with a special attribute (so–called *General Buffer/Transaction Ordering Attribute*) are to be used for that purpose. This eliminates explicit store barriers and thus can help to improve performance. For further details please have a look at [6].

VII. Summary

In this paper we provided some detailed information about the basic operation of our hardware. The raw hardware architecture was briefly reviewed as well as a zoom into both FPGAs. The essentials of write buffer handling policies were described as well as a relatively detailed view of a remote write execution. Finally we could conclude that our hardware solution does not show a fatal lack of performance in comparison to commercial hardware. Nevertheless our PCI–SCI bridge will keep its status as a proof–of–concept and will not go in competition with other solutions.

Latest information about the hardware project described within this paper can be obtained from our web page that can be found at

`http://www.tu-chemnitz.de/~mtr/VIA_SCI/`

References

[1] Hans Müller, A. Bogaerts, C. Fernandes, L. McCulloch, P. Werner and Y. Ermoline: *PCI–SCI Bridge for high rate Data Aquisition Architectures at Large Hadron Collider.* PCI'95 Week, St.Clara, March 1995.

[2] Mario Trams, Wolfgang Rehm, and Friedrich Seifert: *An advanced PCI–SCI bridge with VIA support.* In: Proceedings of 2nd Cluster–Computing Workshop held in Karlsruhe, Pages 35–44, March 1999. See also:
`www.tu-chemnitz.de/informatik/RA/CC99/`

[3] Mario Trams and Wolfgang Rehm: *A new generic and reconfigurable PCI–SCI bridge.* In: Proceedings of SCI–Europe'99 held on 2nd/3rd of September 1999 in Toulouse, Pages 113–120. See also:
`wwwbode.in.tum.de/events/sci-europe99/`

[4] Mario Trams, Wolfgang Rehm, Daniel Balkanski, Stanislav Simeonov: *Memory Management in a combined VIA/SCI Hardware.* In: Proceedings of PC–NOW 2000, Intl. Workshop on Personal Computer based Networks of Workstations held on 5th of May 2000 in Cancun, Mexico. Springer LNCS, ISBN 3-540-67442-X, Pages 4-15.

[5] Mario Trams, Ralph Schlosser and Wolfgang Rehm: *Design Choices and First Results of Our VIA–Capable PCI–SCI Bridge.* In: Proceedings of CLUSTER2000 — IEEE International Conference on Cluster Computing, November 28th – December 1st 2000, Chemnitz, Germany. IEEE Press, ISBN 0-7695-0896-0, Page 349f.

[6] Mario Trams: *Design of a system–friendly PCI–SCI Bridge with an optimized User–Interface.* Diploma Thesis, TU–Chemnitz, 1998.
`www.tu-chemnitz.de/informatik/RA/themes/works.html`

[7] Georg Acher, Hermann Hellwagner, Wolfgang Karl, Markus Leberecht: *A PCI-SCI Bridge for Building a PC-Cluster with Distributed Shared Memory.* In: Sixth International Workshop on SCI-based High-Performance Low-Cost Computing, SCIzzL, Santa Clara, CA , September 1996

[8] Georg Acher, Wolfgang Karl, and Markus Leberecht: *Development of an PCI to SCI Card with FPGAs.* Scalable Coherent Interface / SCI, Architecture and Software for High-Performance Compute Clusters, LNCS State-of-the-Art Survey, October 1999, Springer Lecture Notes in Computer Science Volume 1734

[9] *A new MPI–2–Standard MPI Implementation with support for the VIA.*
`www.tu-chemnitz.de/informatik/RA/projects/chempi-html/`

[10] Sven Schindler, Wolfgang Rehm, Carsten Dinkelmann. *An optimized MPI library for VIA/SCI cards.* In: Proceedings of the Asia-Pacific International Symposium on Cluster Computing (APSCC'2000) held in conjunction with the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPCAsia2000), May 14-17, 2000, Beijing, China.

[11] 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/`

[12] Friedrich Seifert: *Development of system software to integrate the Virtual Interface Architecture (VIA) into the Linux operating system kernel for optimized message passing*, Diplomarbeit, Technische Universit"at Chemnitz, Fakultät für Informatik, 1999.
`www.tu-chemnitz.de/informatik/RA/themes/works.html`

[13] Friedrich Seifert and Wolfgang Rehm: *Proposing a Mechanism for Reliably Locking VIA Communication Memory in Linux.* In: Proceedings of the CLUSTER2000 — IEEE International Conference on Cluster Computing, Nov. 28th – Dec. 1st 2000, Chemnitz, Germany. IEEE Press, ISBN 0-7695-0896-0, Pages 225–232.

[14] Lars Jordan: *Entwicklung eines effizienten Speichermanagementes für das CHEMPI VIA/SCI Device*, Studienarbeit, Technische Universit"at Chemnitz, Fakultät für Informatik, Oktober 2000.
`www.tu-chemnitz.de/informatik/RA/themes/works.html`

[15] Dolphin Interconnect Solutions AS:
`www.dolphinics.com`

[16] Scali AS — Scalable Linux Solutions:
`www.scali.com`

[17] *IEEE Standard for Scalable Coherent Interface (SCI).* IEEE Std. 1596-1992. SCI Homepage:
`www.SCIzzL.com`

[18] Esprit Project 23174 — Software Infrastructure for SCI (SISCI). *Low–level SCI software functional specification* Rev.2.1.1
`www.dolphinics.com/downloads.html`

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

[20] Intel Corp. *Intel Virtual Interface (VI) Architecture Developer's Guide.* Rev. 1.0
`developer.intel.com/design/servers/vi/`

[21] Myricom, Inc.: *LANai Documentation*
`www.myri.com/scs/L3/documentation.html`