Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Daniel Balkanski Friedrich Seifert Wolfgang Rehm

Proposing a System Software for an SCI–based VIA Hardware

Preprint SFB393/01-15

Preprint-Reihe des Chemnitzer SFB 393

SFB393/01-15 **March 2001**

Contents

Author's addresses:

Daniel Balkanski
Friedrich Seifert
Wolfgang Rehm
TU Chemnitz
Fakultät für Informatik
Professur Rechnerarchitektur
D-09107 Chemnitz

`http://www.tu-chemnitz.de/informatik/RA`

Abstract

In this document we present the architectural design we develop for the system software, which integrates our new architecture PCI–SCI Bridge with VIA support into the LINUX operating system. The development of this system software is a part of a bigger project, the end goal of which is to build a complete interconnect solution from the hardware to communication libraries and to prove in practice our new architectural concepts. This interconnect solution will be suitable for building inexpensive clusters optimized for running MPI based message-passing applications. The work presented here can serve as a good starting point to a broad range of experimental designs ranging from "pure" VIA and "pure" SCI shared memory communication devices to "combined" VIA-SCI designs like ours.

1 Introduction

SCI [4] is a modern communication technology offering extremely low latencies and very high bandwidth. But today's PCI–SCI Bridge architectures still have some disadvantages that make building MPI based message passing applications on top of such hardware difficult and resource wasting. The main disadvantage is that there is no Protected User Level DMA available, which can significantly increase the overall system performance.

On the other hand the Virtual Interface Architecture (VIA) [5] defines a standardized interface between high-performance network hardware and the operating system and is intended to improve performance of distributed applications by reducing the latency associated with critical message passing operations.

This motivated us to develop a new PCI–SCI Bridge Architecture [7], [6] that combines the positive characteristic of SCI with advanced memory management and Protected User-Level DMA of VIA in optimal way. To prove these ideas in practice we built a hardware prototype [8] the core of which is based on reconfigurable logical devices instead of ASICs. This gives us a flexibility to experiment with a wide range of different PCI–SCI Bridge architectures only by redesigning the firmware and system software. That's why we named the prototype Generic PCI–SCI Bridge or GPSB in short.

Currently we are developing the appropriate firmware and system software for the prototype which will realize our new PCI-SCI Bridge architecture concepts. Initially the system software will be targeted to PC and Alpha LINUX platforms considering the most attractive ones for cluster computing.

Until now only one vendor — Dolphin Interconnect Solutions AS produces commercial PCI–SCI Bridges and details for internals of their operation aren't available nor the system software is is Open Source. For that reason we decided that the description of our design would be of interest for other groups that want to experiment with the SCI technology their selves and/or the VI Architecture especially on the basis of our highly reconfigurable hardware prototype. We consider that our design can serve as a good starting point to a broad range of experimental designs ranging from "pure" VIA and "pure" SCI shared memory communication devices to "combined" VIA–SCI designs like ours.

2 SCI Protocol And GPSB Hardware Peculiarities and Their Impact on The System Software Design

In contrast with most of the busses SCI uses only split transactions, which means that they consist of a separate request subaction and an optional response subaction. The send command inside the packet falls into one of four main categories: response-expected request, move request, event request and response.

As the SCI standard states, the combination of initiator *nodeID* and *transactionID* uniquely identifies each of the outstanding response-expected transactions. Since the valid range for *transactionID* values is from 0 to 63 they are limited resource and special care must be taken to not initiate a new response-expected request transactions with the same combination of *nodeID* and *transactionID* before the old ones are not accomplished. Or in other words before their corresponding response subactions with same *transactionID* have arrived.

In contrast with response-expected transactions the move requests doesn't have a response subaction and their *transactionID* can be reused immediately.

The responses can be generated only in answer of response-expected requests.

We will not go into details about architectural advantages nor about principles of operation of the GPSB hardware. They are already described and the interested reader can read more about them in [7], [6] and [8]. We only want to briefly describe one interesting feature of the hardware that is slightly aside from the main functionality but which is from the main importance for the system software. This is the so called manual packet mode, which provides the system software with mechanisms for implementing the communication between different nodes that is required for its normal operation. This manual packet mode is based on one of the two different packet types the hardware distinguishes, named automatic and manual.

The automatic request-send type packets are generated in result of access to the special memory mapped I/O regions in which the imported Distributed Shared Memory resides. The GPSB hardware on the remote side automatically processes them by accessing the requested exported memory region and, if the transaction is response-expected, returns the required automatic response packets. When these automatic response packets come back to the importer node they are also automatically processed by the GPSB hardware, which closes the transaction after that.

The manual type packets are explicitly generated by the system software and they are used to carry control protocols between the nodes. This enables the system software to implement the functionality required for SCI and VI Architecture such as establishing connections, exporting, importing, etc. Additionally the System software has also got a possibility to generate packets manually which can be recognized and interpreted like automatic by the peer side.

SCI is a highly reliable technology and packet loss is considered a very rare and catastrophic event. But anyway our design must deal with such events although they are highly improbable. As the SCI standard requires the GPSB hardware uses response timeouts to detect errors that result in the loss of automatic request-send or response-send packets. This is accomplished by calculating the time interval within which a response is expected, and when that time limit is exceeded it signals the system software by rising an interrupt. Because not responded response-expected transactions can be generated, for instance, as a result of a PIO access from the CPU to imported shared memory regions only asynchronous error notification is possible. The VI Architecture suggests the VipErrorCallback function to be implemented in the VI User Agent for such purposes. It can be used by the VI application to register an error handling function.

In contrast to automatic packets manual request packets are not automatically responded by the GPSB hardware nor timeout detection is performed for them. They carry control protocol data of the System Software and must be responded by the corresponding software layer on the peer node. If this software is not operational for some reason (e.g. loaded) there is no one to generate responses. This blocks the *transactionID*'s on the requester side and will finally lead to deadlocks.

To resolve this problem a special Transaction Tracking Subsystem is implemented in the system software, which maintains information for the *transactionID*'s of all manual requests in progress. This enables a special watchdog timer to cleanup split response-expected transactions periodically avoiding deadlocks in this way.

The synthesized reconfigurable core logic inside the prototype realizes practically the whole basic functionality of the hardware. To avoid deadlocks it maintains separate buffers (Fig. 1) for incoming packets, for outgoing manual and automatic request-send packets, and for outgoing manual and automatic response-send packets. We should mention that packets are stored if the buffers in Blink rather than in SCI format. (The Blink format is composed of an SCI core and Blink extension header and trailer.)

After issuing the send command for a given packet buffer it can't be considered free for reuse immediately, because the hardware needs some time to accomplish the actual send of the packet. System software must take the information for accomplished sent request and response commands from the hardware. Due to hardware expense considerations, this information contains only the number of accomplished sends after the last check relative to the order of issuing of the commands. This means that special care must be taken on SMP machines to keep track of the command issue order correctly.

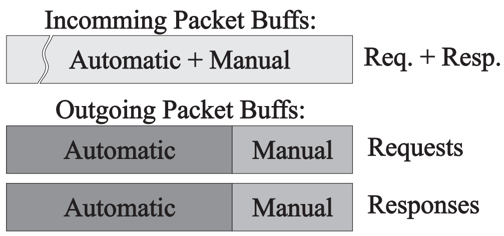

Figure 1: GPSB Packet Buffers

3 Layered Architecture of System Software

We decide to implement a layered architecture of the system software splitting it into two main modules - low level device driver *gpsb*, and Kernel Agent *gpsbka* (Fig. 2) This approach gives us flexibility in an early stage of the project to create software emulation of the GPSB hardware and *gpsb* low-level driver. We avoid in this way lagging of the upper levels of system software and application development behind the hardware development, which is a common problem in such projects.

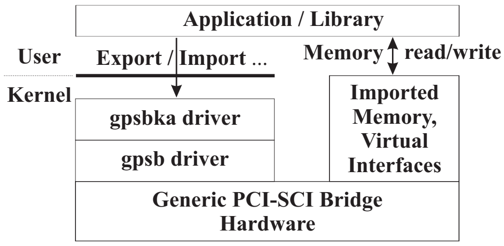

Figure 2: Layered architecture of System Software

From other side separating the operating system dependent management policies inside the Kernel Agent from the mechanisms, which the low-level driver *gpsb* must provide, makes our design more structured and easy to handle and therefore latter porting of the System software to other platforms will be more easy and painless.

3.1 GPSB Kernel Agent driver - *gpsbka*

The Kernel Agent in conjunction with the low level driver and the hardware forms the VI Provider. It is accessed by VI applications through the VI User Agent. As it is part of the operating system it can accomplish those privileged operations that are needed for administrative purposes and for ensuring protection between several processes. The Kernel Agent's major tasks are:

- Creation and destruction of Virtual Interfaces
- Memory registration and protection
- Connection setup and breakdown.
- SCI Shared memory administration

The former two involve only local operations while the latter require communication with other nodes. In section 4 we will take a closer look at the implementation of the Kernel Agent.

3.2 GPSB low level driver - *gpsb*

The *gpsb* layer provides an abstraction of a network interface controller which supports a special Distributed Shared Memory Mode, and VIA functionality. It provides the upper software levels with a well-defined portable programming interface to its functionality. The interface functions can be divided into several different groups, which we describe briefly in the following.

The first and most important group of interface functions provides the basis for the Kernel Agent operation. The functions inside this group can be further subdivided to functions for control of Upstream and Downstream Address Translation and Protection Tables, VI Context Memory, Protection Tags and a Control Message Transport. We will discuss the purpose and implementation of this important group of functions in bigger details in subsequent sections.

Development of the firmware for such a design is a long and complicated process, which requires lots of complex tests proving the correct operation of the synthesized logic under different conditions. Other kinds of tests that we often need to perform are to prove out and tune some algorithms, which must be later built inside the system software. These tests are very difficult and time wasting to be programmed and debugged in a kernel space and very often a small mistakes leads to hangs of the whole system. So we decided to provide an additional programming interface which offers possibilities for obtaining the specific status and configuration information and to establish user space mappings of most of the CSR's and memory regions of the GPSB hardware. For similar purposes another two sets of interface functions to the manual packet send-receive functionality of the hardware are provided. One of them makes it possible to generate all important types of SCI packets which are useful for such non-cache coherent PCI hardware. The second set of functions can be used to obtain received and buffered manual packets, separated on requests and responses.

All these additional sets of interface functions plus the main set of interface functions are available to the user space by means of I/O controls (IOCTLS's), which gives a basis for writing user-level diagnostics and tests and speeds up the development. Another possible application can be to analyze other SCI systems by writing applications that inject or collect SCI packets.

Because the GPSB hardware is based on many programmable and reconfigurable logical devices we must ensure solutions for their programming and configuration from the very beginning. Instead of designing lots of system configuration utilities that directly interact with the hardware and require root privileges (like for example in a CERN PCI-SCI Bridge design [3]) we decided to integrate access functions to programmable devices inside the driver and provide appropriate programming interfaces to the configuration utilities. This gives us the advantage of concentrating the lot's of similar code inside the driver, which makes improvements and support easier. The only small drawback is the slightly increased size of the driver. However, that is not a problem concerning the configurations which our design is targeted to.

In addition the *gpsb* driver that has been designed to support an unlimited number of boards from the very beginning is also responsible for initialization of the GPSB hardware. Because very often some parts of the synthesized logic of our experimental design fail from one firmware version to another we decided to integrate more advanced diagnostic functions inside the *gpsb* than are normally typical for a device driver. But instead of totally rejecting support for a given failed device we signal the type of failure and enable only partial driver functionality which can be more or less restricted depending on the failed subsystem. This gives us a chance to diagnosing the problem and to reprogram the programmable logic devices.

4 Implementation of the VI Kernel Agent and Demands on the Low Level Driver

Following we will discuss what mechanisms the low level driver must provide in order to implement the required VIA functionality on.

4.1 VI Creation and Destruction

Virtual Interfaces (VIs in short) are the access points from a user process to the VIA hardware. A VI comprises some main memory resources for the descriptor queues and a so called *VI context*, which is a memory structure located on the NIC. It contains status information about the VI, attributes and FIFOs for the posted descriptors. VI contexts are a limited resource of the VI NIC and their number directly determines the maximal number of supported VIs.

When a new VI is to be created a free context must be allocated and set up appropriately. Upon destruction of a VI the associated context must be released again.

So the low level driver should provide a functions to allocate and free a context of a specific NIC. This administration must be done separately for each NIC installed. Further, functions are needed to alter a given context, i.e. to change the status or the attributes of the VI, and to read out the current state.

4.2 Memory Registration and Protection

Memory management is an essential part of the VI Architecture. It is responsible for locking communication buffers into physical memory. This is done during *memory registration*. Moreover it must ensure that several processes cannot interfere with each other unintentionally. This Protection is achieved by so called *protection tags*. Every registered memory region is assigned a certain protection tag, likewise each VI. The hardware ensures that a memory region can only be used for communication on VIs with the same tag.

Protection tags must be requested from the Kernel Agent by the application, and the Kernel Agent must ensure that no two processes get the same tag at any time. Protection tags are another limited resource of a VI NIC. By analogy to VI contexts they are administered by the low level driver, which provides functions to allocate and free a tag for a specific NIC installed.

During memory registration all memory pages concerned are forced into physical memory and locked down. Then the NIC must be informed about the physical addresses so that it can access the main memory later on by DMA operations. The address information together with the protection tags is stored in the so called Upstream Address Translation and Protection Table (UpATPT) on the NIC. For every registered or exported page a separate ATPT entry is needed. The VI Architecture requires that a contiguous part of the table is allocated for each registered memory region. There is no such restriction for exported memory, however, it simplifies the implementation. In order to realize distributed shared memory our hardware possesses another table, the Downstream Address Translation and Protection Table (DnATPT).

The low level driver administers both tables and provides access to them through a set of functions to allocate and free a number of block of entries, and to change their contents.

4.3 Connection setup and breakdown

The VI Architecture embodies a point–to–point connection model. Before any communication between two processes can take place their VIs must be connected to each other. Managing these connections is the most complex task of the Kernel Agent, as it requires communication with other nodes.

In order to be independent of the existence of another network we decided to use only he SCI fabric to exchange control information. Following we sketch the connection setup for the Client/Server model and analyze what transport mechanisms are needed for its realization.

In the Client/Server model a connection is initiated by the server waiting for a connection. After that the client can issue a connection request. It comprises the address and attributes of the local VI, the remote address and some additional information, i.e., the packed size is fixed and in our current implementation won't exceed 64 bytes. Connection request packets are always sent in the context of the calling process, which means that the packet send function is allowed to block. When the connection request arrives at the server node the Kernel Agent must check if there is a process waiting for the request and return an appropriate answer, which can be encoded using not more than three bits. This happens in the context of processing incoming packets. In order to allow all send functions to block, we decided to install a kernel thread for each block that dispatched incoming control messages.

Upon arrival of a matching connection request the server process returns from the wait function with the address and attributes of the remote VI. Now it can check the properties of the remote VI and decide whether to accept or to reject the connection. In order to accept it, it calls the appropriate Kernel Agent function, which sends an accept packet to the client node. It contains the server VI attributes and a few more fields so that it's even smaller than a connect request packet. To make sure that the client's request has not timed out meanwhile the server waits for a positive answer from the remote Kernel Agent. That Kernel Agent checks if the request is still pending and in case it is, it sets up the client VI and returns a positive status. When the Kernel Agent on the server side receives the answer it sets up the local VI, but it needs another handshake with the client before the connection can be considered established since the client's request could have timed out just now. For this purpose it sends an accept confirmation packet which only identifies the remote request by an integer value. This is done in the context of the server process again. When the client Kernel Agent receives the confirmation and the request is still pending it wakes up the waiting client process and returns a positive status. Finally, when this answer arrives the server process resumes execution and the connection is established.

Summarizing this protocol we can state that the low level driver must provide a facility to send packets of a fixed size. Further, there must be a mechanism to pass incoming packets to the Kernel Agent in an asynchronous fashion.

4.4 SCI Shared Memory Administration

A unique feature of our hardware is that it allows to share memory between several nodes. It is the Kernel Agent's task to set up the hardware properly. This means that the Upstream Translation Table on the exporting node must be filled with the physical addresses of the exported area. After that the indexes of the table entries used must be transferred to the importing node to set up the Downstream Translation Table.

In principal it is possible to use any arbitrary set of entries for a single area. The other extreme is to use one consecutive block. Both methods have got advantages. While the former avoids fragmentation of the Upstream table the latter minimizes the amount of data to be exchanged. As a trade off we decided to use a set of blocks for each exported area. We allocate as many consecutive entries as possible for each block and only transfer their start index and length. However, the size of such messages depends on the size of the exported ares. Thus, the low level driver must be able to transfer messages of arbitrary length.

5 Implementation of Low Level System Software Main Functional Subsystems

As we mention in section 3.2 the main task of the low-level system software is to provide the required mechanisms for the Kernel Agent operation, which are control of Upstream and Downstream Address Translation and Protection Tables, VI Context Memory management, Protection Tag management and efficient Control Message Transport.

The Control Message Transport is based on the Manual Request-Send Packet Subsystem, the Manual Response-Send Packet Subsystem and Manual Packet Receive Subsystem. Before discussing the Control Message Transport implementation we will briefly describe their design and way of operation.

5.1 Address Translation and Protection Tables Control

The interface functions for Upstream Address Translation Tables control enable the Kernel Agent to implement exporting of the shared memory areas. All pages of the host memory that are registered inside the UpATPT are accessible from remote nodes. The introduction of this UpATPT gives a possibility to export every arbitrary region from processes virtual address space, because exported memory regions don't need to consisted physically consecutive pages. This is one of the main advantages in contrast with conventional PCI–SCI designs.

A similar set of functions for Downstream Address Translation Tables controls the management of imported memory areas. By properly adjusting of the DnATPT entries for the physical pages inside the imported memory area accesses to them can be forwarded to corresponding pages of exported memory areas on remote nodes.

The low-level driver provides only basic functionality of the Address Translation and Protection Tables entries management. The interface includes functions for obtaining information about the size of the tables, for allocating a range of consecutive free entries, freeing the range of entries, setting attributes of given entry and freeing all possible entries. The allocation function simply returns the first available region with the requested size or the biggest possible one if the request can't be satisfied.

5.2 Manual Request-Send Packet and Manual Response-Send Packet Subsystems

Manual request-send and manual response-send packet send functionality is accessible through two sets of interface functions—one for requests and one for responses. Every function of these sets offers the generation of specific type request-send or respectively response-send packet and must always be called from process context. Both request and response types of functions accept parameters for GPSB device number, Target SCI *nodeID*, Source SCI *nodeID*, pointers to data body if the packet type permits that, a pointer to an optional extended header data and a flag indicating whether the send operation must "block on send" or not. The requests-send functions return the *transactionID* under which the packet is sent, while the response functions take the *transactionID* under which packet must be send as parameter. Besides some other specific fields, which are packet type specific, the other difference between the manual request-send and manual response-send functions is that the former ones accept an optional pointer to the response buffer and a flag whether to "block on response" or not.

To avoid redundant data copying pointers to a buffer for the expected response is recommended to be supplied to the manual request-send functions. This makes it possible for the corresponding response to be fetched by the Manual Packet Receive Subsystem directly to the caller buffer. Furthermore because actually three pointers can be supplied for parameters—for the header, for the packet body and for the trailer, packet data will be separated from the protocol information on receiving. This saves this extra job for the caller, which otherwise may need to perform additional data copying.

The implementation of all request-send functions, which offer generation of different types of request-send packets, is similar and follow the generic scheme shown in Fig. 3. At the beginning a free buffer must be allocated. If allocation fails and the "blocking send" flag of the caller process is on it is put to sleep until request-send buffers are freed. After successful request-send buffer allocation the corresponding type of packet is prepared inside the buffer except the header, the creation of which the *transactionID* is required for. If the following attempt to obtain a free *transactionID* fails and the blocking send flag of caller process is on it is put to sleep until *transactionID*'s are freed. After successful obtainment of a *transactionID* the header is also created and packet becomes ready for sending. If the caller has the "block on response" flag turned on, after issuing the send command for the allocated request-send buffer it is put to sleep until the corresponding response arrives or a timeout is detected.

We allocate the *transactionID* just before the send of the packet because we want to de-couple send-buffer allocation from *transactionID* allocation allowing to start the packet preparation even if there are currently no free transactions available.

Response send functions implementation can be illustrated with similar but slightly simplified diagram. That is because no *transactionID* needs to be allocated. Responses send packets

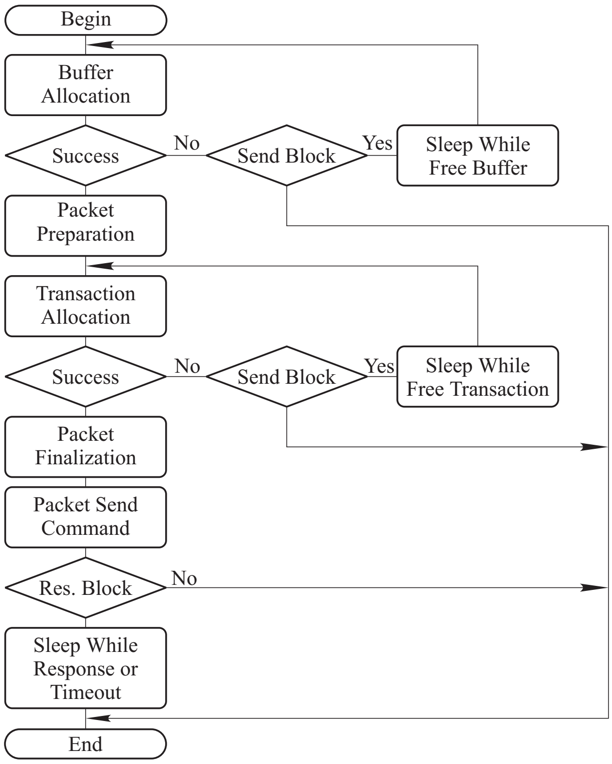

Figure 3: Generic block diagram of the request-send functions

typically are generated in response of request-send packets and for this reason the *transactionID* is known and it is supplied as parameter to the response-send function. Of course, in such generic block diagram for the response-send functions the "block on response" part must also be missing.

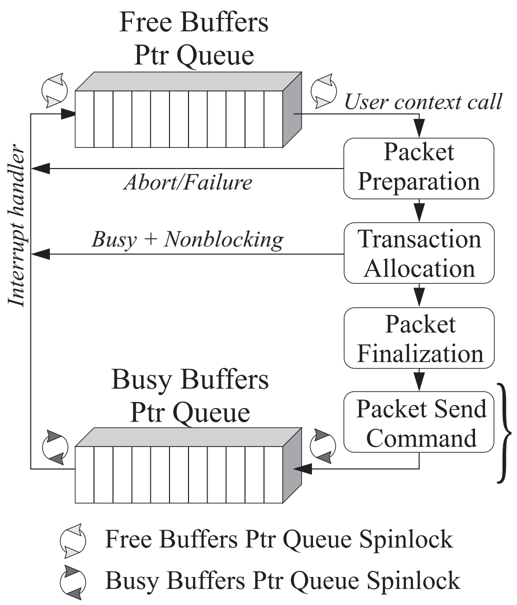

Figure 4: Request-Send Buffers Management

Fig. 4 shows how the request-send buffers management organization is implemented. Pointers to the free request-send buffers are stored in a Free Buffers Pointer Queue. When the request-send function is invoked it takes a buffer from Free Buffers Pointer Queue and starts preparing the packet. If the function execution must be aborted for some reason (e.g. invalid parameters, missing free *transactionID*'s while non- blocking send is requested or signal processing is required) the allocated request-send buffer must be put back to the Free Buffers Pointer Queue. In case of successful packet preparation a send command is issued to the hardware and the pointer to the request-send buffer used is appended to the Busy Buffers Pointer Queue. After accomplishing the send command (or rather accomplishing the number of send commands equal to the number of half of the request-send buffers) hardware generates interrupt which is serviced by the system software. This causes that all pointers to already sent request-send buffers are put back to the Free Buffers Pointer Queue. Spinlocks guarantee the integrity of data structures because they are accessed concurrently from the interrupt handler at interrupt context and from several request-send functions running in user context, which can run simultaneously on different CPUs on SMP machines. The Busy Buffers Queue Spinlock also ensures that the order of send commands issued matches the order of pointers to the request-send buffers inside Busy Buffers Pointer Queue supplied to this commands.

5.3 Manual Packet Receive Subsystem

Receive functionality is accessible also through two sets of functions—one for obtaining the received manual request-send packets and the other for obtaining received manual response-send packets. In practice they offer access to received manual packets buffered by the system software separately in the Manual Request Buffer Queue resp. the Manual Response Buffer Queue in host memory. Both sets of the functions accept a flag as parameter that indicates whether to block until packet is received or not.

When manual packets arrive the GPSB hardware generates an interrupt, which is serviced by the system software. The interrupt servicing routine identifies in which Manual Incoming Packet Buffers received manual packets have received. Because Manual Incoming Packet Buffers are common for both manual requests and responses the packet header must be prefetched in order to determine if the packet is of request or response type.

If the packet is a response the *transactionID* is used to obtain information from the Transaction Tracking Subsystem for this transaction. If the request-send function has registered a buffer for the response, the packet is stored there. Otherwise it is stored in the general Manual Response Buffer Queue. If the transaction tracking information shows that a process is sleeping on the response for this transaction it is woken up.

In case a manual request packet is received it is stored in Manual Request Buffer Queue.

Whenever a packet is stored in the Manual Request or Response Buffer Queues the processes sleeping on the corresponding queue are woken up.

5.4 Control Message Transport Subsystem

As we mentioned above the Control Message Transport Subsystem gives the upper level software the possibility to exchange control protocol information. It must provide a high level of reliability and mechanisms for synchronization because the SCI technology does not guarantee that the order in which packets arrive at the receiving side will match the order in which they were sent. At the same time acceptable latency and bandwidth must be provided because it can have impact on the overall performance of the whole communication solution.

We want to emphasize here that overall performance delivered to the applications mainly depends on speed of the automatic packet mode of the GPSB hardware. But if the application code for example forces frequent import and unimport of shared memory regions the speed of underlying System Software and therefore speed of operation of Control Message Transport can have a big impact on the overall performance.

Unfortunately interrupt driven send–receive of manual packets can't offer very big bandwidth and very low latencies. That's because on the receiving side interrupts and therefore kernel call is required. Another reason is that received packets must be transferred from on-board buffers to system memory by the system software using the read PCI transactions, which are typically more than 10 times slower than PCI write transactions.

We decided to overcome the reasons for this bad performance by using some mixture of the manual and automatic mode approach. In our Control Message Remote Write Function we are using manually generated automatic write packets. When such packets are received by the peer node they are interpreted by GPSB hardware like automatic and their data body is written directly to system memory. After this the receiving side automatically generates the corresponding responses for these write request transactions. This in combination with SCI CRC error detection mechanisms gives to us very fast and reliable way for sending unlimitedly long control messages. The function automatically creates the required automatic packets on the basis of the accepted parameters which are GPSB device number, remote *nodeID*, remote buffer virtual address, pointer to control message to be sent and control message size.

Additionally other slower interface function for control message exchange is provided, which is based on ”pure” interrupt-driven manual packet send-receive and can be used to transfer short messages (no longer than the longest currently supported non cache-coherent write transactions which is currently 64 bytes). This function is required to initially transfer the virtual address of the remote buffer needed to the Control Message Remote Write function.

In our future plans for improving the system software we have the idea to minimize the use of manually generated packets for Control Message Transport as much as possible. They shall be used only for initial establishing of certain system software dedicated shared memory areas. These areas will be used to transfer the Control Messages by automatic SCI packets.

6 Summary and Future Work

In this paper we have presented an architectural design of system software for our high speed communication hardware that uniquely combines the Virtual Interface Architecture with SCI distributed shared memory capabilities. It contrast to other solutions no additional low speed connections are needed for the implementation of shared memory and VIA functionality. Besides the main functions the software provides an advanced interface for further firmware development and for debugging. These functions can also be used for studying and analyzing other SCI systems.

Future tasks are the development of an fragmentation free algorithms for the translation table management and a mechanism to exploit the shared memory capabilities of the hardware for Control Message Transport. Moreover, the protocols for connection management and exporting and importing of memory should be improved further.

References

[1] DOLPHIN INTERCONNECT SOLUTIONS AS: *PCI–SCI Bridge Specification Rev. 4.01. 1997*

[2] DOLPHIN INTERCONNECT SOLUTIONS AS: *PSB-64/66, Features and Benefits.* `http://www.dolphinics.no`

[3] 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.

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

[5] *The Virtual Interface Architecture Specification. Version 1.0.* Compaq, Intel and Microsoft Corporations. Dec 16, 1997. Available at: `http://www.viarch.org`

[6] M. TRAMS, W. REHM, D. BALKANSKI, S. SIMEONOV. *Memory Management in a combined VIA/SCI Hardware.* In proceedings to PC–NOW 2000, International Workshop on Personal Computer based Networks of Workstations held in conjunction with the IPDPS 2000, May 2000, Cancun/Mexico.

[7] 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: `http://www.tu-chemnitz.de/informatik/RA/CC99/`

[8] MARIO TRAMS AND WOLFGANG REHM: *A new generic and reconfigurable PCI–SCI bridge.* Proceedings of SCI Europe'99. Toulouse, September 1999.

[9] *M-VIA: A High Performance Modular VIA for Linux.* Project Homepage: `http://www.nersc.gov/research/FTG/via`

[10] P. BUONADONNA, A. GEWEKE, AND D. CULLER: *An Implementation and Analysis of the Virtual Interface Architecture.* Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1998. See also: `http://www.cs.berkeley.edu/ philipb/via/`