Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN
Matematica e Informatica

CICLO XXIV

COORDINATORE Prof.ssa Ruggiero Valeria

A Scalable Parallel Architecture with FPGA-Based Network Processor for Scientific Computing

Settore Scientifico Disciplinare INF/01

Dottorando	**Tutori**
Dott. Pivanti Marcello	Dott. Schifano Sebastiano Fabio
	Dott. Simma Hubert

Anni 2009/2011

Contents

List of Figures

List of Tables

Acronyms

ACK Acknowledge

ASIC Application Specific Integrated Circuit

BAR Base Address Register

CplD Completion with Data

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CRDBAR Credit Base Address Register

CRDFIFO Credit FIFO

DATACHK Data Checker

DATAGEN Data Generator

DDR Double Data-Rate

DESCRFIFO Descriptor FIFO

DMA Direct Memory Access

DRV Device Driver

EMI ElectroMagnetic Interference

FPGA Field Programmable Gate Array

ID Identifier

IOC Input/Output Controller

LBM Lattice Boltzmann Methods

LIB Low Level Library

LID Link-ID

LQCD Lattice Quantum-Chromodynamics

LVDS Low-Voltage Differential Signaling

MAC Medium Access Control

MDC Management Data Clock

MDIO Management Data Input/Output

MMAP Memory Mapping

MPI Message Passing Interface

MRd Memory Read Request

MWr Memory Write Request

NAK Not Acknowledge

NTFBAR Notify Base Address Register

NGET Network GET

NID Notify-ID

NPUT Network PUT

NWP Network Processor

PCIe PCI Express

PCI Peripheral Component Interconnect

PCKARB Packet Arbiter

PCKBUFFER Packet Buffer

PGET Processor GET

PHY Physical Layer

PIC PCI Express Input Controller

PIO Programmed Input/Output

PLL Phase Locked Loop

POC	PCI Express Output Controller
PPUT	Processor PUT
RAO	Remote-Address Offset
RC	Register Controller
RCV	Receiver Buffer
RSND	Re-send Buffer
RXCLK	Receive Clock
RXFSM	Reception Finite State Machine
RXLINK	Link Receiver
RX	Receive
RXSYNC	Receiver Synchronizer
SDR	Single Data-Rate
TB	Test-Bench
TLP	Transaction Layer Packet
TNW	Torus Network
TXCLK	Transmit Clock
TXFIFO	Injection Buffer
TXFSM	Transmission Finite State Machine
TXLINK	Link Transmitter
TX	Transmit
VCID	Virtual Channel ID
VC	Virtual Channel
VHDL	VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

WCB Write Combining Buffer

WC Write Combining

XAUI 10 GigaBit Attachment Unit Interface

XGMII 10 GigaBit Media Independent Interface

Chapter 1

Introduction

Several problems in scientific computing require surprisingly large computing power which current HPC commercial systems cannot deliver. In these cases the development of application-driven machines has to be taken into account since it can be the only viable and rewarding approach.

From the computational point of view, one of the most challenging scientific areas is the *Lattice Quantum Chromodynamics* (LQCD), the discretized and computer-friendly version of *Quantum Chromodynamics* (QCD). QCD is the fundamental quantum theory of strong interactions. It describes the behavior of the elementary constituents of matter (the quarks) that are building blocks of stable elementary particles, such as the proton and the neutron, and of a wealth of other particles studied in high-energy physics experiments.

Monte Carlo simulations are a key technique to compute quantitative predictions from LQCD. In LQCD the physical space-time is discretized on a finite 4D lattice (state-of-the-art sizes are $\approx 80^4$ lattice sites). Larger lattices would be welcome, since more accurate prediction would be possible. However algorithmic complexity grows with the $6^{\text{th}} \ldots 10^{\text{th}}$ power of the linear lattice size, so the size of the lattices is in fact limited by available computing power.

The Monte Carlo algorithms used in this field are characterized by a very small set of kernel functions that dominate the computational load. Five or six key critical kernels correspond to $\simeq 95\%$ of the computational load. Within this already small set of kernels, the evaluation of the product of the Dirac operator, a sparse and regular complex-valued matrix, with appropriate state-vector is truly dominant, so approximately $60 \ldots 90\%$ of the computing time of LQCD codes is spent in just this task.

The Monte Carlo algorithms applied in this field (and the Dirac-operator in particular) have many features that make an efficient implementation on massively-parallel machines relatively easy to achieve.

- the computation can be easily partitioned among many processors; basically the whole lattice can be divided in smaller equally-sized sub-lattices and each one can be assigned to a different CPU;
- *Single Instruction Multiple Data* (SIMD) parallelism can be easily exploited; all CPUs perform the same program on a different data set;
- Data access patterns of each CPU to its local memory are regular and predictable, allowing to exploit data-prefetch to prevent stalls of the processors;
- The communication pattern among CPUs is very simple: if processors are assigned at the vertices of a D-dimensional grid, all data communications for the Dirac kernel occur between nearest-neighbor processing elements. This suggests to use a D-dimensional mesh interconnection network, for which it is possible to obtain scalability, high bandwidth and low latency.

The properties of the network are relevant for the overall performance of the program execution, since latencies and low-transfer data rate may negatively affect the execution time, causing stalls of the CPUs. The required features of the interconnection network are strictly related to parameters of the application, performance of the processor, memory bandwidth, and size of the on-chip memory.

In the last 20 years of the previous century, several dedicated LQCD machines used as building-block processing elements which were designed from scratch for this specific application, so all their features were carefully tailored to best meet all requirements. As an example, the main features of the APE series of processors can be summarized as following:

- high performance low-power processors that could be easily packed in a small volume;
- instruction set optimized for LQCD applications. For instance, the so-called *normal* operation $a \times b + c$ with complex operands a, b, c was the implemented in hardware to accelerate the vector-matrix multiplications;
- low-latency high-bandwidth network system, with network interface integrated inside the processor.

These features made it possible to assemble systems that had far better performance (in term of several relevant metrics, such as performance per dissipated power, performance per volume, price per performance) than achievable with commodity building blocks, whose structure was not well matching the requirements.

The advent of *multi-core* processors (including powerful floating-point support) in the last years has radically changed these conditions. Starting from the IBM Cell Broadband engine (CBE), these new processors have dramatically increased the computing performance available on just one computing node. Processors in this class include now the multi-core Intel chips of the Nehalem and Sandy-bridge families.

As an example, the most recent multi-core architecture developed by Intel, the Sandy-Bridge processor, delivers ≈ 200 Gflops in single precision and ≈ 100 Gflops in double precision. New VLSI technologies and architectural trade-offs which are more oriented to scientific computing have allowed to integrate large memory banks on chip, and made the processor much less power-hungry. Again, as example, the Sandy-Bridge integrates up to 20 MB of on-chip cache and dissipates less than ≈ 150 Watt.

These features strongly suggest that multi-core commodity processor are a very good option for being used as a building-block to assemble a new LQCD system. As an additional advantage, high performance and large memory on chip allow to map large partitions of the lattice on each CPU, improving the volume over surface ratio. This reduces the communication requirements and renders the architecture well balanced.

As discussed above, a massively parallel system needs an interconnection harness. From LQCD applications, what we need is:

- scalability up to thousand of CPUs,
- appropriate bandwidth to balance the computing power of the processor, and sufficiently low latency to allow efficient communication of the small-size data packets that occur in the typical communication patterns of the algorithms.
- first-neighbor only data-links since each CPU needs to exchange data only with its nearest neighbors in space

The best network topology meeting all the above described requirements is a 3D-mesh.

In contrast to the processor developments, commercially available interconnections for commodity processors, perform poorly with respect to the requirements discussed above. This has mainly two reasons:

- the network interface is not directly coupled to the processor; the connection goes over some standardized bus, PCI Express or similar, introducing large latencies.
- star topologies are used to interconnect thousand of processors with one or more levels of switches, making scalability technically difficult and expensive, and introducing extra latency in the communication

This discussion suggests that an optimal LQCD engine today should use commodity processors while a custom interconnection network is still a rewarding option. This is for instance the choice made by the QPACE project.

This work was done in the framework of the FPGA Torus Network (FTNW) project, which extends the results achieved within the QPACE project, Its aim was to developing an FPGA-based implementation of a light-weight communication network to tightly interconnect commodity multi-core CPUs in a 3D torus topology.

FTNW is based on a communication core that is largely independent of the technology used for the physical link and of the interface with the processor. Thus it can be adapted to new developments of the link technology and to different I/O architectures of the CPU.

In this context, the current thesis work focuses on the design, implementation and test of a 3D network based on the FTNW architecture. A major part of the work concerned the interface between the communication network and the computing node based on PCI Express, a standard I/O bus supported any several recent commodity processors. I implemented and tested the network processor on commodity systems with dual-socket quad-core Nehalem and six-core Westmere processors.

The main activities of the thesis thesis cover both hardware and software topics:

- extensions of VHDL modules used for the implementation of the Torus Network on the QPACE project
- design and implementation of a new processor interface based on the standard PCI Express protocol to interconnect the CPU and the Torus Network

- design, development and test of a device driver for the Linux operating system to access and manage the network device
- design and development of a system library to access the network device from user-applications in an efficient way and with small overhead
- development of micro- and application-benchmarks to measure both latency and bandwidth of the communications

The thesis is organized as follows:

- in chapter 2, I outline the development and the implementation of the network processor, describing the communication model and the layered structure of the network design. The main part of this chapter describes the physical and logical layer of the torus links, including details of the VHDL modules that implement the custom communication protocol of the network.
- Chapter 3 explains the general mechanisms for moving data between the main memory of the CPU and the I/O devices. I then discuss the best way to map the transactions between CPU and I/O devices on the specific PCI Express Protocol and describe the design and implementation of the specific VHDL modules to interface the network processor with the CPU.
- In chapter 4, I describe the design and the implementation of a device driver for the Linux operating system to manage I/O with the network processor device. Moreover, I describe the implementation of a user-library to access the network device from user-applications, implementing the transaction methods between CPU and I/O devices outlined in chapter 3.
- In chapter 5, I present benchmark programs and their results for the measurement of latency and bandwidth of the communications.
- Conclusions of the development on the Network Processor are discussed in chapter 6.

Chapter 2

Network Processor

The Network Processor (NWP) implements the interface to access a custom high-bandwidth low-latency point-to-point interconnection network with three-dimensional toroidal topology. If we consider the computing-nodes as edges of a 3D lattice, the NWP provides the interconnection between each node and the nearest neighbors along the six directions [1] with periodic re-closings, providing a full-duplex point-to-point reliable link; this interconnection is also known as *3D-torus* or *torus-network*, it physically reproduces the communication pattern of common scientific applications e.g. "Lattice Quantum-Chromodynamics" (LQCD) and "Lattice Boltzmann Methods" (LBM).

The Network Processor takes advantages of hardware and software techniques to achieve the minimal latency (tentatively in the order of 1μsecond) and the maximum bandwidth for the communication by lowering the protocol overhead.

The hardware level consists of three layers as shown in figure 2.1. The highest two are implemented in VHSIC Hardware Description Language (VHDL) language on a reconfigurable Field Programmable Gate Array (FPGA) device, while the lowest is implemented on a commodity device. The choice to implement the Network Processor on a FPGA instead of a custom Application Specific Integrated Circuit (ASIC) has several advantages such as the shorter development time and costs, lower risks, and the possibility to modify the design of the NWP after the machine has been deployed.

The software stack has been developed for the Linux operating system, it consists on two layers, the lowest is a Device Driver (DRV) to provide the basic access to NWP and the higher is a Low Level Library (LIB) of routines accessible

[1]Direction-names are X+, X-, Y+, Y-, Z+ and Z-.

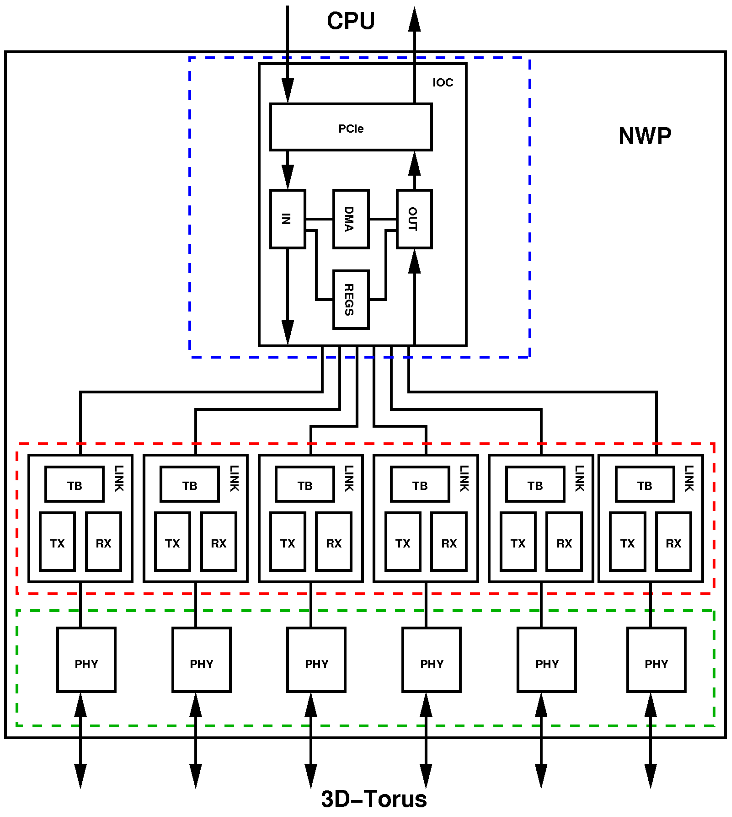

Figure 2.1: *NWP block-diagram, on the green-dotted box are shown the PHY commodity components that manage the physical level of the links such as electrical signals; on the red-dotted box are shown the 6 link managers, together called TNW, that take care of the PHYs configuration and reliable data transfer. On the blue-dotted box is shown the interface to the CPU, called Input/Output Controller (IOC), it manages the PCIe link to the CPU and the DMA engine.*

by applications.

From bottom to top of the NWP stack, the first hardware layer is the Physical Layer (PHY) that takes care of the electrical signals and link establishment, it is based on the commodity component PMC-Sierra PM8358, mainly a XAUI-to-XGMII SERDES [2] device that allows data communication on four PCIe Gen 1 lanes, with a raw aggregate bandwidth of 10 Gbit/s, value estimated as sufficiently large for which application type (mainly LQCD) NWP has been developed for[25][26]. The choice for a commodity device allows to move outside the FPGA the most timing-critical logics, also the use of a well-tested and cheap transceiver based on commercial standard shorten the development time.

The second HW layer, called Torus Network (TNW) is one of the two layers implemented on the FPGA, a "Xilinx Virtex-5 lx110t" for the QPACE machine or an "Altera Stratix IV 230 GX" for AuroraScience, TNW has been written in VHDL language, mainly his purpose is to implement a custom communication protocol optimized for low latencies. This layer exports the injection and reception buffers to exchange data between nodes, as well as the rules to obtain a reliable link among an unreliable electrical path. TNW task is also to act as the Medium Access Control (MAC) for the PM8358 so it implements the modules to configure and check the PHY.

The third and last layer of the hardware stack. is the "Input/Output Controller" (*IOC*) and will be explained in detail in chapter 3.2. The purpose of the IOC is to export to the CPU the injection, reception and control/status interfaces, providing a translation layer between the CPU's Input/Output system and the TNW.

The software layers, are explained in detail in chapter 4. They are tailored to provide a convenient an efficient access to the NWP from thread-applications. For this purpose I developed a driver for the Linux operating system and a Low Level Library (LIB), allowing threads to directly access the injection buffers avoiding time overhead from frequent context switches between user- and kernel-mode.

The NWP provides the hardware control of the data transmission and has *injection* and *reception* buffers for each of the six links, the applications access the torus-network by (i) moving data into the injection buffer of the NWP of sending node, and (ii) enabling data to be moved out of the reception buffer of the NWP of the receiving node. Thus, the data transfer between two nodes proceeds according

[2]XAUI means "10 GigaBit Attachment Unit Interface" while XGMII means "10 GigaBit Media Independent Interface" (compliant to IEEE 802.3ae standard). SERDES means serializer/deserializer, deriving from the conversion from parallel data at one side of the device to serial data at the other side and vice-versa.

to the *two-sided* communication model [3] , where explicit operation of both sender and receiver are required to control the data transmission [29].

Data is split in a hierarchical way into smaller units during the various transmission steps. The size and name of these units depends on the step which is considered. Applications running on CPUs exchange “Messages” with a variable size between a minimum of 128 Bytes and a maximum of 256 KBytes, but always in multiples of 128 Bytes. At the NWP level a message is split into “Packets” with a fixed size of 128 Bytes. They are the basic entity which the network processors exchange. Inside the NWP a packet is split into “Items”, which have a size of 4 or 16 Bytes, depending on the stage of the NWP considered.

Tracing a CPU-to-CPU data transfer over the 3d-torus the following steps occur:
1. The application calls the send operation specifying over which of the six links to send the message. It simply copies the message-items from the user-buffer to the address-space where the injection buffer of the link is mapped. Depending on the architecture and the I/O interface of the CPU, this operation can be implemented according to different schemes.
2. As soon as an injection buffer holds data, the NWP breaks it into fixed-size packets and transfers them in a strictly ordered and reliable way over the corresponding link. Of course, the transfer is stalled when the reception buffer of the destination runs out of space (back-pressure).
3. The receive operation on the destination CPU is initiated by passing a *credit* to the NWP. The credit provides all necessary informations to move the received data to the physical-memory and to *notify* the CPU when the whole message has been delivered.
To allow a tight interconnection of processors with a multi-core architecture, the TNW also supports the concept of virtual channels to multiplex multiple data streams over the same physical link. A virtual channel is identified by an index (or tag) which is transfered over the link together with each data packet. This is needed to support independent message streams between different pairs of sender and receiver threads (or cores) over the same link. The virtual channels can also be used as a tag to distinguish independent messages between the same pair of sender and receiver threads[29].

In the rest of this chapter I explain the PHY and TNW hardware layers in detail.

[3]Two-sided Communication Model, see section 2.1 for more details.

2.1 Two-Sided Communication Protocol

The “Two-Sided Communication Protocol” is based on the classic approach to distributed memory parallel systems programming, the “message passing” model where messages are exchanged using matching pairs of `send` and `receive` function calls from the threads involved into the communication. The basics of this model are widely used in Message Passing Interface (MPI) programming and the most important concept is the notion of matching. Matching means that a `receive` call does not deliver just the next message available from the transport layer, but a message that matches certain criteria. Typical criteria implemented are sender ID, size or the user specified tag.

The NWP implementation of the Two-Sided Communication Protocol is based on three separated operations, one to send data and two to receive them;
the first operation, called `SEND`, allows to move messages from the user-space of the sending thread directly into the injection buffers of the network processor, triggering the message delivery to the peer-entity at the other side of the communication link.
The remaining two operations are both required to receive data, the first operation is called `CREDIT`, the receiving thread must issue it to NWP as soon as possible to provide the basic information to deliver the incoming messages directly to the main memory of the receiving node, once the CREDIT has been issued, NWP can autonomously end-up the message delivery and the thread can spend to any other operation that does not require the incoming data. The second operation to receive a message is `POLL`, this operation is undertaken by the thread when it requires the incoming data to go on with computation, calling POLL is tested the condition of completely delivered message, once the condition is true the thread can proceed using data, otherwise it keeps waiting.

The whole mechanism is based upon the use of tags, each SEND has its own tag that must match with the corresponding CREDIT/POLL tag on the receiver side, subsequent messages with the same tag must be sent in the same order as the CREDIT/POLL are issued to avoid data loss or corruption.

The above mechanism limits the need of temporary buffers to store incoming data and any other complex infrastructure at operating system level to keep care of transactions, leaving the whole communication in user-space under the control of the application.

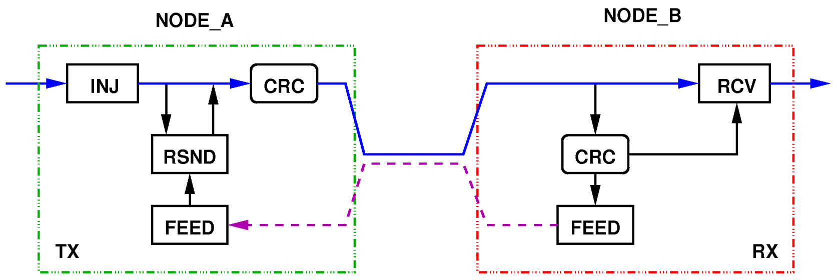

Figure 2.2: *Node-to-node reliable communication diagram, it is based on a ACK/NAK protocol. Each packet is protected by a CRC and a copy of data is stored into the Re-send Buffer (RSND), if the CRCs on TX and RX match the packet is accepted issuing an ACK or otherwise it is discarded issuing a NAK. The ACK feedback removes the packet from RSND while a NAK implies the resend of the packet.*

2.2 TNW Communication Protocol

The entities at the end-points of the Physical Link Layer are the transmitter (TX) and the receiver (RX), the communication among them follows few simple rules, a “Packet-Based ACK/NAK Protocol[2]”, that allow a reliable data exchange. This protocol implements only the detection of possible errors, not their correction, if an error occurs data are simply re-transmitted. The basics of the above mentioned protocol require a data unit called “packet”, with a well-defined format and size, that are received by RX in the strict order they are sent by TX, even despite an error occurred; the entire packet is protected by a Cyclic Redundancy Check (CRC) calculated upon the entire packet and appended to it on transmission, a copy of the packet is stored into a Re-send Buffer (RSND) in the case it must be re-transmitted, on the receiver side the CRC is re-calculated and compared with the appended one, if they match an Acknowledge (ACK) is sent-back by the FEED module and the packet is accepted by RX and stored into the Receiver Buffer (RCV) while dropped by the resend-buffer, on the other hand the packet is dropped by RX and a Not Acknowledge (NAK) feedback is sent-back, at this point TX enters in RESEND mode and re-send the copy stored into the resend-buffer until it is not acknowledged by RX or until a time-out occurs. The above mechanism is shown in figure 2.2.

Acknowledging one packet per time before sending next could be a waste of bandwidth, a way to maximize throughput is to send a continuous packet-flow while waiting for feedbacks, this implies that each packet has his-own sequence number that must be claimed into feedback and the calibration of resend-buffer to store a sufficient number of packets corresponding to in-flight packets. When an error occurs, RX drops the faulted packet and sends-back a NAK with the corresponding sequence number, discarding all the in-flight packets that follow without issuing feedbacks, TX stops sending new packets and issues a “RESTART” command to RX followed by the packets with equal and greater sequence number then the one on the feedback, after RX receives the RESTART it behaves as usual. “receiver-buffer’ **Protocol rules:**

- packet format and size are well-defined
- packet is entirely protected by a CRC and has a sequence number
- RX must receive packets in the same order TX send them, also despite any error occurrence
- TX computes the CRC and appends it to packet
- RX re-computes CRC, if it matches with appended one, send-back an Acknowledge (ACK), if not, send-back a NAK (not acknowledge)
- packet is stored into a *resend-buffer* until not ACK by RX
- resend-buffer must fit in-flight packets number

Required control characters:

- ACK to acknowledge a packet that match with his-own CRC
- NAK to not acknowledge a packet that does not match with his-own CRC
- RESTART to stop RX to discard incoming packets after an error occurs

The “TNW Communication Protocol” also manages the case the receiver-buffer runs-out of space, as explained in section 2.1, the CPU must to issue a CREDIT to RX as soon as possible, ideally before the first packet of a message has been transmitted by TX, in this case data can stream directly from the link to the main memory without lying too much time into RCV. To the contrary, if TX starts to transmit the message before the CREDIT has been issued, RCV can quickly run-out of space, and all the further incoming packets are treated by RX as faulted, discarding them and sending-back to TX a NAK feedback; at this point TX will behaves as usual, entering in RESEND mode until the packets are not acknowledged by RX or a timeout occurs. If a CREDIT is issued to RX before the timeout occurrence, the communication restarts.

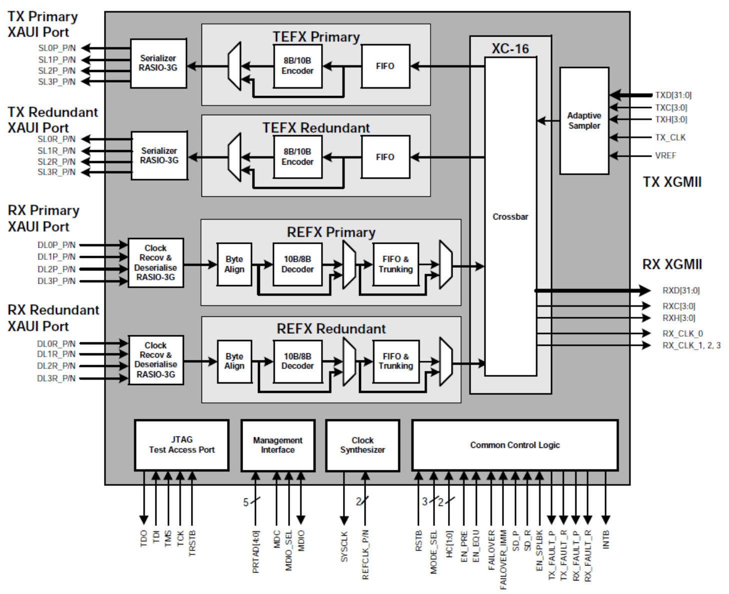

Figure 2.3: *Block-diagram of the commodity PHY PMC-Sierra PM8358. On the left-side are visible the Primary/Redundant serial ports (XAUI) used for the node-to-node communications; on the right-side are visible the parallel ports (XGMII) managed by the TNW module.*

2.3 Physical Link Layer

The Physical Layer (PHY) is the unreliable data path between the network processors. The bandwidth required for the project described in this thesis is 10 Gbit/s. This can be reached, for instance, by using a "PCI Express Generation 1" link with 4 lanes (4x). The link is implemented by using the commodity PHY component "PM8358" manufactured by PMC-Sierra.

The PM8358 is a multi-protocol silicon device for telecommunication and has the block-diagram as shown in figure 2.3. This PHY implements several telecom standards, such as PCIe, Gigabit Ethernet, 10 Gigabit Ethernet, Infiniband, Common Public Radio Interface, High-Definition TV, etc. and supports a wide range of operative frequencies (1.2 to 3.2 GHz).

The PM8358 is mainly a XAUI-to-XGMII serialization/deserialization device (SERDES). While XGMII allows only a limited signal routing (maximum 7 cm) [4], the XAUI ports provide serialized data over LVDS pairs. These allow to drive

[4]See `http://www.10gea.org/xaui-interface-introduction-to-xaui/`

drive signals over a longer distance, for instance over a back-plane or cable for node-to-node communication.

The XGMII ports are parallel buses connected to the application logic implemented inside the FPGA. Each port has a 32-bits data-path (TXD and RXD) and 4 bits for flow-control (TXC and RXC). These lines operate in Double Data-Rate (DDR) mode with two independent clocks (TXCLK and RXCLK). Each 8 bits of the data-path have one associated bit of flow-control, which flags whether the data is a "data character" or a "control character". The latter are functional to the correctness of the transmission and management of the link.

Data serialization includes the "8b/10b Encoding" encoding stage to reduce ElectroMagnetic Interference (EMI) noise generation in different manners and to reduce data corruption. The 8b/10b Encoding embeds the clock of the data source into a data-stream. This eliminates the need of a separate clock-lane which would generate significant EMI noise and it also keeps the number of '1' and '0' transmitted over the signal lines approximately equal to maintain the DC component balanced and to avoid interference between bits send over the link, see [2] for more details.

The configuration/status of the PHY is accessible via MDIO [5] (Management Data Input/Output). This it is a serial master/slave protocol to read and write the configuration registers on a device. In our case the logical master is the NWP (physically one the FPGA) and the slave is the PHY. The master can access different slaves via a two-signal bus shared by them. One signal is the "Management Data Input/Output/Serial Data Line" (*MDIO*) that actually carries control and status informations. The other is the "Management Data Clock" (*Management Data Clock (MDC)*), which is just a strobe, i.e. the MDIO line is sampled at the rising edge of MDC.

The NWP interconnects the nodes of a machine as edges of a three-dimensional grid with toroidal re-closing. Multiple grids with this topology can be connect together simply "opening" the re-closings and pairing them, resulting in an extended single machine with more nodes. Vice versa, a single machine can be partitioned into multiple independent ones just by "cutting" some connections and re-closing them into the new smaller partition. This allows to assemble a machine with an arbitrary number of $N_x \times N_y \times N_z$ nodes that can be partitioned as independent machines or expanded as needed.

for more details.

[5]For the specific, the MDIO "Clause 45" defined by IEEE 802.3ae that allow to access up to 65536 registers in 32 different devices.

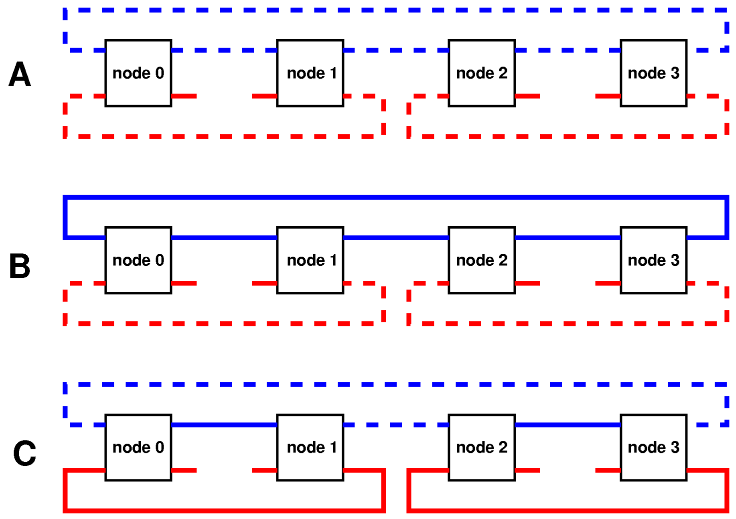

Figure 2.4: *Re-partitioning example of a four-nodes machine. Depending on configuration of PHY ports, primary (blue lines) or redundant (red lines), the machine can be configured as one four-nodes machine (layer "B") either as two separated two-nodes machines (layer "C") without re-cabling the connections.*

Usually such a re-partitioning of a machine would require re-cabling. However, exploit the fact that the PM8358 provides a "Primary" and a "Redundant" XAUI port. They are both connected to XGMII interface via an internal crossbar and are mutually exclusive. This feature allows to physically connect one PHY with two others instead of one. Thus, by selecting at configuration time which of the two physical links will be actually activated, we can re-partition the machine without the need for re-cabling.

An example of re-partitioning is shown in figure 2.4. The layer "A" of the figure shows a four-nodes machine, where each node is connected to its nearest neighbors only along one dimension, the dotted-blue lines are the connections between primary ports of the PHYs while the dotted-red lines are the connections between redundant ports (nothing prevents to connect a primary port to a redundant). The machine can be configured as one four-nodes machine enabling via software all the connections on primary ports (blue lines) and disabling the connections on redundant (red lines), as shown in layer "B". Alternatively the machine can be configured as two separated two-nodes machines, enabling the redundant links among nodes 0 and 1 and among nodes 2 and 3, preserving the primary connections among node-pairs, as shown in layer "C" of the figure.

2.4 Torus Network Layer

The basic idea underlying TNW is to provide a high-bandwidth low-latency point-to-point link among two computing-nodes on a massive parallel machine for scientific computing.

NWP is based on the 3D-torus topology so it must manage six links, one per spatial direction, all those links lie on the Torus Network (TNW) hardware layer and each of them is the Medium Access Control (MAC) for the Physical Layer (PHY) of the link connecting two computing-nodes, implementing a reliable communication among a non-reliable media. To allow a tight interconnection of processors with multi-core architecture, TNW must support an abstraction mechanism to virtually provide a dedicated link among a pair of cores belonging to different computing-nodes, I used the “Virtual Channels” mechanism to multiplex data streams belonging to different core-pairs over the same physical link. TNW implements a “Two-sided Communication Protocol” where data exchange is completed with explicit actions of both sender and receiver, see section 2.1 for a more detailed explanation of the Two-sided Communication approach.

Each TNW link is mainly divided into “transmitter” (*TXLINK*) and “receiver” (*RXLINK*) sub-systems that will be presented in detail in sections 2.4.1 and 2.4.4, each link also implements a Test-Bench (TB) module, explained in section 2.4.5, fully controlled via register interface, that features a data generation/check mechanism to fully test the low-level activities of TNW. On the receiver-side I also implemented a synchronizer (Receiver Synchronizer (RXSYNC)), explained in section 2.4.3, to efficiently synchronize data among the clock-domain of the PHY (125 MHz DDR) and the clock-domain of TNW (250 MHz Single Data-Rate (SDR)) [6]. In figure 2.5 is shown the diagram of the TNW layer.

2.4.1 The “TXLINK” Module

The Link Transmitter (TXLINK) has the purpose to send the data injected by the CPU over the physical link in a reliable manner, it also takes care to manage and configure the physical link driver (PHY) via the MDIO interface.

The main components of TXLINK are the “Injection Buffer” called **TXFIFO**,

[6]The PHY outputs data @125 MHz in Double Data-Rate (DDR) mode, considering two subsequent data, one is produced on the rising-edge of the clock while the other is produced on the falling-edge. The TNW clock-domain is 250 MHz and accepts input data in Single Data-Rate (SDR) mode, in this case input data are expected on the rising-edge of the clock.

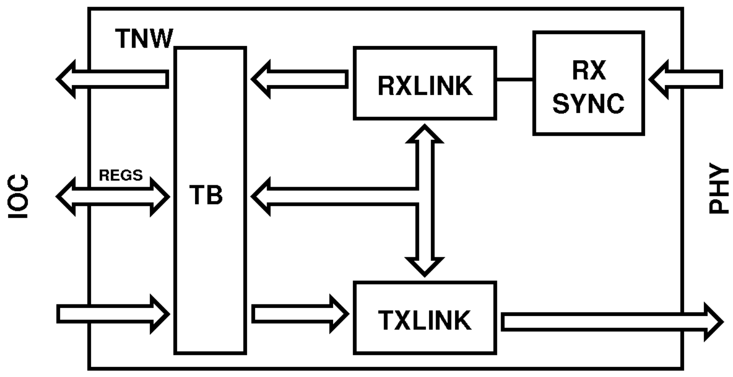

Figure 2.5: *Block-diagram of the TNW layer. The bottom-side shows the transmitter path, data flows from the CPU interface (IOC) to the physical-link (PHY), the transmission is managed by the TXLINK module. The top-side shows the receiver path, data flows from the physical-link (PHY) to the CPU interface (IOC), the RXSYNC module synchronizes the incoming data between the PHY clock-domain and the TNW clock-domain; the RXLINK module manages data reception. The TB module features a data generation/check mechanism to fully test the TNW low-level activities, it's interposed between the CPU interface and the TNW's injection/reception buffers, when not enabled it is completely transparent to the other modules and acts like a pass through without affecting data-flow. Otherwise it separates TNW from the CPU interface, autonomously managing the data-stream.*

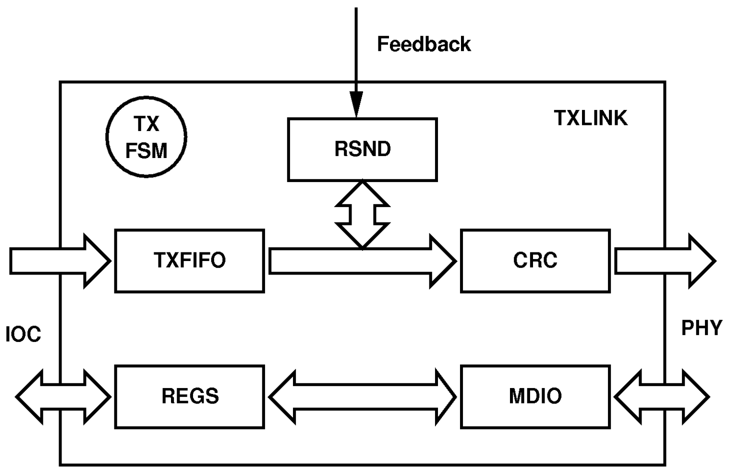

Figure 2.6: *Diagram of the TXLINK module. The upper section is the data transmission pipeline, it includes the injection buffer (TXFIFO) and the modules to implement the custom packet-based ACK/NAK protocol, RSND stores packet-copies in case they must to be resent due to corruption among the link, CRC module computes the protection-code to validate or not the packet at the receiver-side. The lower section is the register access including the MDIO interface to the PHY.*

the "Re-send Buffer" called **RSND**, the "CRC[7]module" and the Transmission Finite State Machine (TXFSM). A diagram of TXLINK is shown in figure 2.6.

Before describing the TXLINK functionalities, it is important to specify the informations contained into the basic entity managed by NWP: the "packet" that is composed by a header and a 128-bytes payload; the header contains all the informations to fully route data from sender to receiver, these informations are: the Link-ID (LID) that specifies over which one of the 6 links data will be sent, the Virtual Channel ID (VCID) that specifies which virtual channel within the same link will be used, the Remote-Address Offset (RAO) that specifies the offset to write data respect to the memory-address of the reception buffer at the receiver side of the link. The payload contains the actual data to be sent.

A hardware module, located into the IOC module so external to TNW, uses the LID field of the packet-header to select in which of the 6 TXFIFOs to inject data, preserving VCID and RAO fields, the TXFSM is sensitive to the injection

[7]The CRC module directly derives from the VHDL code provided in the paper "Parallel CRC Realization[30]" by Giuseppe Campobello, Giuseppe Patanè, Marco Russo. This module implements a parallel method to calculate CRC checksums, allowing it's use in high-speed communication links.

buffer status, when at least one packet is stored there, TXFSM starts to extract it and route it through the transmission pipeline. During the 4-stages transmission pipeline a new header is assembled using VCID and RAO fields and the payload is appended. The payload is stored inside the TXFIFO in 8 16-bytes items while during the pipeline it is extracted as 32 4-bytes items due to the PHY data-interface width. While the packet steps into the pipeline a copy is stored into the re-send buffer where it lies ready to be retransmitted in case the receiver claims a corrupted reception; the copy is removed only if the receiver claims a correct reception of the packet. The above scheme is based on a ACK/NAK protocol where the sender trails to the packet a 4-bytes CRC item calculated over the whole packet and the receiver recalculates the CRC comparing it with the trailed one, if CRCs match, the packet is accepted and a positive feedback (ACK) is sent back, otherwise the packet is discarded and is sent back a negative feedback (NAK). A more accurate description of the protocol used for a reliable communication in TNW is exposed into section 2.2. A remarkable thing to keep in mind is that a TNW link is composed by one TXLINK and one RXLINK modules, the TXLINK module of an end-point is directly connected with the RXLINK of the other end-point, so RXLINK checks the correctness of sent data but only TXLINK can send feedbacks to the peer, at this purpose TXLINK accepts requests from RXLINK to send back feedbacks to the peer; the feedback sending has a highest priority respect to data.

TXLINK contains a set of configuration and status registers to control the configuration of the link and to provide a snapshot of its status.

2.4.2 The MDIO Interface

TXLINK manages the configuration of the PHY device via the MDIO interface, briefly described in section 2.3. The MDIO module is based on a FSM that manages read/write operations to the PHY that are always composed by an ADDRESS frame followed by a READ or WRITE frame, the frame layout is shown in figure 2.7 and follows the explanation, the PREAMBLE, a sequence of 32 “logic 1s”, is at the beginning of each frame to establish synchronization among endpoints;
ST is the “Start of frame”, a pattern of 2 “logic 0s”;
OP is the “Operation code”, `0b00` for ADDRESS frame, `0b11` for READ frame, `0b01` for WRITE frame;
PRTAD is the “Port address” and identifies the PHY to which the operation has been issued;
DEVAD identifies the MDD (Maximum Data Delay) type for the communication,

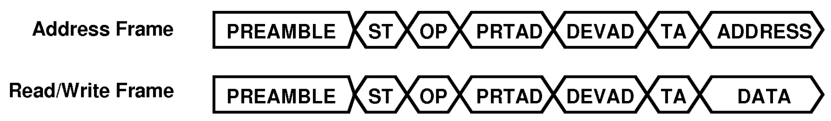

Figure 2.7: *MDIO frames diagram*
MDIO frames diagram.

in this case it is set to DTE-XS (Dumb Terminal Emulator) `0b00101`;
TA is the “Turn-around”, a 2-bit time spacing between DEVAD and DATA/ADDRESS fields to avoid contention during a read transaction, during a read transaction, both the master and the slave remain in a high-impedance state for the first bit time of the turnaround, and the slave drives a “logic 0” during the second bit time of the turnaround, during a write transaction, the master drives a “logic 1” for the first bit time of the turnaround and a 0 bit for the second bit time of the turnaround;
DATA/ADDRESS is a 16-bit field that, depending on the frame, contains the register address to operate to or the read/write data.

I implemented the MDIO module to be accessible via internal register of TXLINK, a write operation addressed to the MDIO_W register triggers the MDIO module for a write transaction to the PHY, the MDIO_W register layout includes the address of the PHY register where to write to and the data to be written, a busy-flag in MDIO_W reflect the status of the MDIO transaction. Same concept is applied to the MDIO_R register, a write operation addressed to this register triggers the MDIO module for a read transaction from the PHY, the MDIO_R register layout includes the address of the PHY register to read and a busy-flag reflect the status of the MDIO transaction, when the transaction is done, the data field of MDIO_R contains the value read from PHY.

2.4.3 The Receiver Synchronizer

The RXLINK module, as whole the TNW layer, works on a 250 MHz clock-domain where data are sampled in Single Data-Rate (SDR) mode, considering two subsequent data both are sampled on the rising-edge of the clock, as shown on the left-side of figure 2.8, while the PHY outputs data @125 MHz in Double Data-Rate (DDR) mode, considering two subsequent data, former is produced on the rising-edge of the clock while the latter is produced on the falling-edge, as shown on the right-side of figure 2.8, these data are in sync with the clock recovered from the 8b/10b data-stream received.

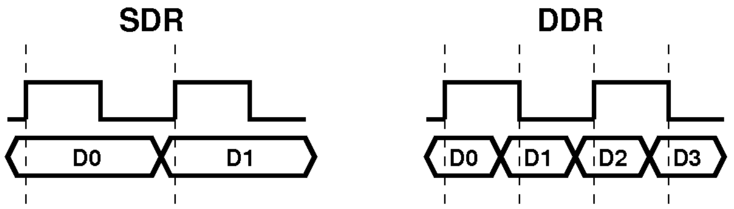

Figure 2.8: *Differences among "Single Data Rate" (SDR) mode and Double Data-Rate (DDR) mode. In SDR data are sampled either on rising or falling edge of the clock, in this example data are sampled on rising edge. In DDR data are sampled on both clock edges.*

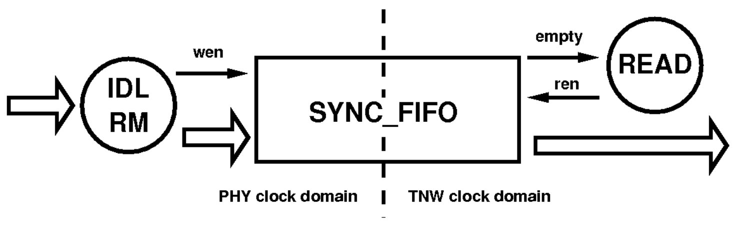

Figure 2.9: *RXSYNC diagram, data are synchronized among PHY's and TNW's clock-domains using a FIFO. To avoid FIFO-full due to the differences among clock-frequencies, not all the incoming data are pushed-in, the control-character not needed by the TNW protocol are removed by the IDL_RM logic. The READ logic extracts data as soon as they are available.*

To deal with the above mentioned characteristics, is required a synchronization layer between PHY and TNW clock-domains that allows to get in sync data in a reliable manner. At this purpose I implemented the Receiver Synchronizer (RXSYNC) module, shown in figure 2.9.

The data clock-domain transition is implemented using a "Synchronization FIFO" (SYNC_FIFO), data are written in sync with PHY's clock and read in sync with TNW's clock. First of all the 125 MHz output from the PHY is doubled to 250 MHz using a Phase Locked Loop (PLL), in this way data can be sampled in SDR mode instead of DDR.
The custom protocol used for TNW-to-TNW communications foreseen the injection of IDLE code (K28.3) every 256 consecutive-packets sent to allow for sufficient clock rate compensation between the end-points of a link; these IDLEs are required just on the earliest stage of the receiver so they are removed from the data stream by the "Idle Removal" (IDL_RM) logic and not written into SYNC_FIFO to avoid FIFO full. To the contrary data-packets are stored into the FIFO and can

proceed to the TNW clock-domain.

The *empty* signal of the FIFO triggers the READ logic to extract data that are then output.

2.4.4 The "RXLINK" Module

The purpose of the Link Receiver (RXLINK) is to receive packets from his peer entity in a reliable manner and deliver data to the CPU.

The basic idea underlying the structure of RXLINK is to implement the two-sided protocol and to provide a set of Virtual Channel (VC) to allow communications of independent thread-pairs running on adjacent computing-nodes sharing the same physical link or to allow different message priorities. Each VC is managed with a credit-based mechanism where the CPU provides to TNW the informations to complete the message delivery, the "credit" precisely, these informations contains the IDs of both link and VC (LID and VCID) the message will come, the main memory-address (CRDADDR) to deliver data and the message size (SIZE), the last information is the Notify-ID (NID), this is the offset respect to a particular main memory-address (Notify Base Address Register (NTFBAR)) where the CPU will wait the notification from TNW that the whole message has been received and CPU can use data. The messages-items, called packets, incoming from the link are stored into TNW buffers until the CPU does not issues the corresponding credit, these buffers can rapidly run out of space and to cause the stall of the communications, this requires that for each send operation must be issued the corresponding credit, and referred to the same link and VC, they must be issued in strictly order; to avoid that the stall of a VC affects the others, each virtual-channel has his own credit-based mechanism explained later in this section.

RXLINK is mainly composed by the Reception Finite State Machine (RXFSM), the "CRC" module and the "Reception Buffer"(**RXBUFF**) as shown in figure 2.10. RXFSM takes in charge to supervise the reception of the packet-items from the PHY and to verify their correctness with the help of the CRC module, for each incoming packet is recalculated the CRC and, if it matches with the trailed one, data are stored into RXBUFF and is triggered TXLINK to send to the peer entity a positive feedback (ACK), otherwise is triggered a negative feedback (NAK), each feedback is accompanied by the sequence-number relative to the packet. The detailed description of reliable communication protocol is in section 2.2.

RXBUFF is the most complex module of the receiver block, it actually implements the logic to manage the two-sided communication protocol and the virtual-

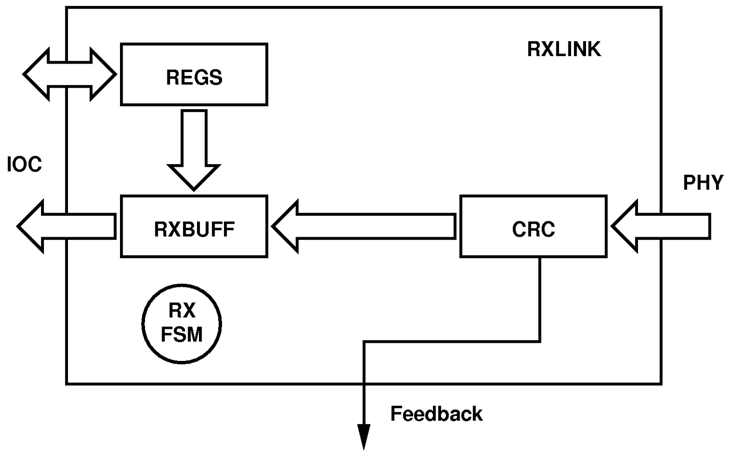

Figure 2.10: *Diagram of the RXLINK module. The lower section includes pipeline where data are checked using the CRC method and the reception buffer (RXBUFF) to store data before sending them to the CPU. The upper section is the register access.*

channels. A diagram of RXBUFF is shown in figure 2.11.

Each Virtual Channel (VC) has his own private Credit FIFO (CRDFIFO) and Descriptor FIFO (DESCRFIFO), the former stores the credits issued by the application while the latter stores the descriptors (actually the headers) of the received packets, when a packet has been received and validated, the payload is stored into the Packet Buffer (PCKBUFFER) while the header is stored into the DESCRFIFO relative to the VC the packet belong, if the Packet Arbiter (PCKARB) finds a match between a DESCR and a CRD for the corresponding VC, it triggers the operations to deliver data to the CPU. When a credit is exhausted, in other words all the packets relative to the same message are delivered to CPU, a particular transaction called "notify" is sent to the CPU, telling that the whole message has been arrived and data are ready to be used by the CPU.

The PCKBUFFER is designed around a single memory block, common to all virtual channels, an internal module called "free-pool" keeps trace of the unused memory-locations, so each incoming packet can be stored into an arbitrary location, independently from the VC it belongs and packets belonging to the same message could be stored into non-adjacent memory-locations, this allows to efficiently manage the TNW storage without reserving a fixed amount of memory to each virtual-channel that could results in a waste of memory-space.

The main memory-addresses where to send data and notifies are a combination of base-addresses stored into internal registers and some fields of both credit and

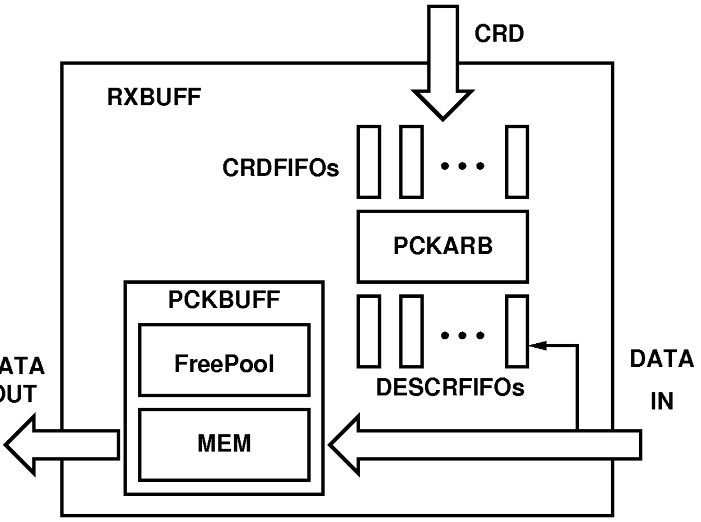

Figure 2.11: *Detailed view of RXBUFF, credits provided by CPU are stored into the related CRDFIFO based on the virtual-channel they has been issued for, the incoming data are stored into MEM while the header (descriptor) is stored into the related DESCR-FIFO depending on the virtual-channel they belong, if the Packet Arbiter (PCKARB) find a match between one descriptor and one credit, the interface with the CPU is triggered to move data into main memory.*

descriptor. For data there is one Credit Base Address Register (CRDBAR) register per virtual-channel, it stores the main memory-address where starts the address-space of the buffer to deliver data relative to the specific VC, at this address is summed the Remote-Address Offset (RAO) field stored into the descriptor of the packet. Same concept is used to retrieve the physical memory-address to notify the CPU that a message has been completely delivered, a register called Notify Base Address Register (NTFBAR) stores the memory-address where starts the address-space of the buffer to deliver the notification relative to the specific message, at this address is summed the Notify-ID (NID) field stored into the credit for the message, it specifies the offset to write the notify respect to NTFBAR.

2.4.5 The "Test-Bench" Module

The Test-Bench (TB) features a data generation/check mechanism to fully test the functionalities of TNW such as the register access and the communications. It's interposed between the CPU interface and the TNW's injection/reception buffers, when not enabled it is completely transparent to the other modules and acts like a pass through without affecting data-flow. Otherwise it separates TNW from the CPU interface and manages autonomously the data-stream. TB is mainly composed by a data generator Data Generator (DATAGEN) and a Data Checker (DATACHK), as shown in figure 2.12,

both can manage a sequential counter or a pseudo-random sequence of data, the former has been used in the earliest phases of the link development where the data transmission has been controlled using a simulation tool and it was more simple to follow sequential data on the simulation window, the latter has been used for all the actual link tests. DATAGEN is controlled by a finite-state machine (FSM) managed via configuration registers and sensitive to the status of the injection fifo of the link to generate data on-the-fly and to inject them only when the fifo is not full, two 64-bits pseudo-random generators has been implemented to fit the 128-bits data-bus of the injection buffer, each generator implements the following equation to generate data,

$$rnd = rnd_{prev} \oplus (C + swap(rnd_{prev}))$$

where the swap() function permutes the 17 lowest bits of nd_{prev} with the 47 highest bits. The sequential number generator is a 32-bits counter replicated 4 times to fit the 128-bit data-bus. DATACHK is similar to DATAGEN it is managed by configuration registers and is controlled by a FSM, it implements both the same

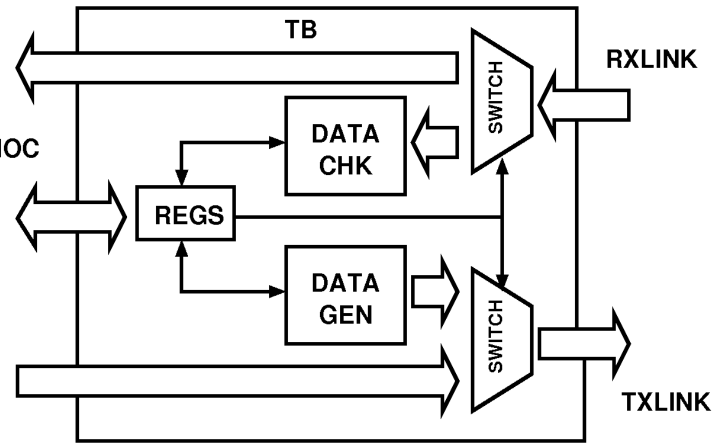

Figure 2.12: *Block-diagram of the TB module. It's interposed between the CPU interface and the TNW's injection/reception buffers, when not enabled, the switches act like a pass through between IOC and TNW, so TB is completely transparent to the other modules, otherwise TB separates TNW from the CPU interface and manages autonomously the data-stream. All the TB's functionalities are fully controlled by register interface.*

sequential- and the pseudo-random number generators, here the incoming data are compared to the internal-generated ones and if they differ an error-counter is incremented.

A set of internal registers manage and keep track of the test bench activities, such as the control of DATAGEN and DATACHK, e.g. selecting which kind of data to generate (random or counter), as well to introduce voluntary errors inside the data-flow, keep track of the whole amount of data generated/checked and eventually the number of errors detected.

Chapter 3

Network Processor's CPU Interface

In this chapter I explain the general mechanisms for moving data between the main memory of the CPU and the I/O devices. I then discuss the best way to map the transactions between CPU and I/O devices on the specific PCI Express Protocol and describe the design and implementation of the VHDL modules to interface the network processor with the CPU.

3.1 Transaction Models

Data exchange between CPU and Network Processor (NWP) can be implemented in several ways, here follows the descriptions of four of these methods applied to the Intel Architecture, analyzing pros and cons.

To send a message along a TNW link, from the main memory point of view, requires to move data to the injection buffer of the link. In NWP jargon, data movement can be done in two ways, the former, called Processor PUT (PPUT), is shown in the left-side of figure 3.1, it foreseen that the CPU actively loads data from main memory (1) and stores them into the injection buffer (2), one-by-one, being involved for the whole time the data movement requires. Beside is shown the temporal diagram of the data movement between CPU and NWP using the PPUT method. The latter method, called Network GET (NGET), is shown in the right-side of figure 3.1, here the CPU simply gathers informations about the data movement (address and size) and pass them to the network processor (1) that will autonomously retrieve data from main memory (2,3) without any other CPU intervention; CPU will be notified about the completion of the data movement (4). The temporal diagram of NGET method is shown beside.

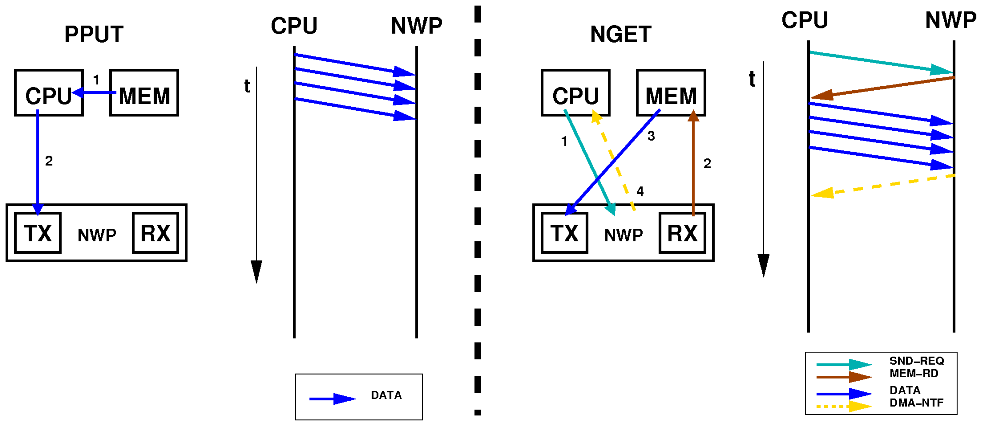

Figure 3.1: *Diagram of PPUT and NGET transaction-methods to send data over the link, targeting the Intel Architecture. With PPUT the CPU directly loads data from memory (MEM) to its internal registers (1), then it stores data (2) to the injection buffer (TX) of the Network Processor (NWP); the CPU is involved to data movement instead of computation for all the transaction-time. With NGET the CPU collects informations about transactions and pass them to the NWP (1) that will independently interact with memory to retrieve data (2,3), at the end of the transaction NWP notifies CPU (4); in NGET computation and data movement can be overlapped but it suffers of higher latency respect to PPUT due to the higher number of CPU-NWP interactions. Beside of each transaction-method is shown the relative temporal diagram of the operations.*

As predictable, in Intel Architecture, the PPUT method offers lower latencies respect to NGET due to the lesser number of interactions required; the PPUT drawback is that it keeps the CPU involved until the end of transaction, preventing it to perform calculations, on the contrary NGET allows the CPU to do not care about the data movement, letting it available for calculations.

On the receiving side, to receive a message from a TNW link requires to move data from the receiver buffer of the link to main memory, and also in this case there are two ways to do it, the former, shown in the left-side of figure 3.2, is called Network PUT (NPUT), in this case, the CPU provides to the network processor the informations about the message it is waiting for (1), such as the address to deliver it and its size, calling these “credit” (CRD); as well as message’s fragments have been received the network processor independently interact with the memory controller to move data to main memory (2) at the address provided by the CPU; then the CPU has to be notified (3) about the whole message reception. Beside is shown the temporal diagram of the data movement between NWP and CPU using the NPUT method. The latter method, shown in the right-side of figure 3.2, is called Processor GET (PGET), here the CPU is actively involved to the receive operations; when the network processor has been received a defined amount of data, it notifies the CPU (1) that will issue read operations (2/3) to the network device to retrieve data from the receiving buffer and then move them into memory. The temporal diagram of PGET method is shown beside.

In NPUT method, once the CPU has issued the credit, it can continue with other tasks, leaving to the network processor the interaction with memory to carry-out the data-reception, in this way computation and communication can be overlapped; in PGET the CPU must to issue read operations to retrieve data from NWP, limiting the overlap of computation and communication.

The Network Processor object of this thesis interfaces with the CPU via the PCI Express bus. In this section I describe how to map the transaction methods described in section 3.1 to the operations implemented by the PCI Express Protocol.

The PCI Protocol, from whom PCIe derives, provides three transaction modes[2] on which can to map PPUT, NGET and NPUT transaction models; to understand the differences among them I will explain the transactions considering the simplified architecture in figure 3.3, here two devices (device0 and device1) are attached to the PCI bus, the CPU can access the PCI via a companion-chip called “ChipSet”, it also integrates the memory-controller to access the main memory modules.

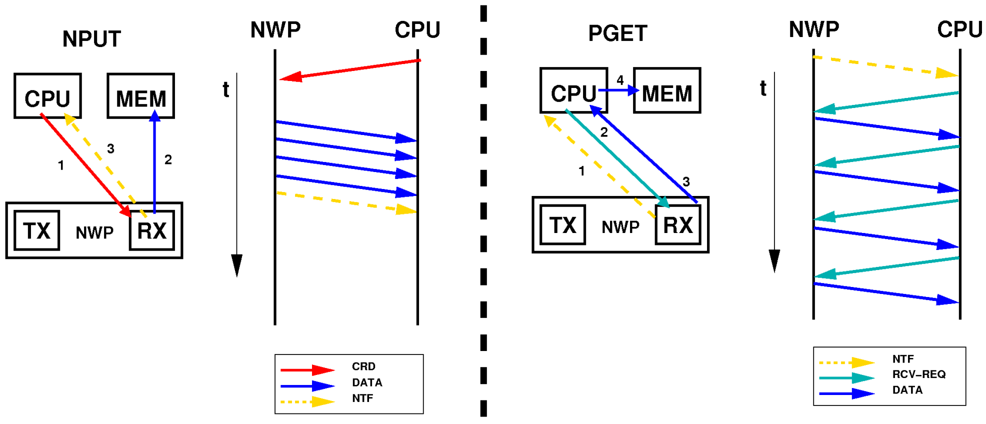

Figure 3.2: *Diagram of NPUT and PGET transaction-methods to receive data from the link. With NPUT the CPU provides a credit (1) to the Network Processor (NWP), incoming data are written to memory (2) according with the credit-information, when the whole message has been received the CPU is notified (3). With PGET the NWP notifies (1) the CPU that some data has been arrived, the CPU issues read operations to retrieve data from RX-buffer (2/3) and moves them into memory (4). Beside of each transaction-method is shown the relative temporal diagram of the operations.*

The first mode is Programmed Input/Output (PIO), it can be directly mapped to the PPUT model, when the CPU wants to start a write transaction to a target device, it must load data from memory into its internal registers and then store them to the target device address-space, in a read transaction the CPU must interrogate the target device and then waiting for the response. The PIO mode involves both the CPU and the target device until the transaction is carried out, keeping busy the CPU on data transferring instead of computation, to transfer small amount of data from memory to the target device this is not an issue, actually has better performance in terms of latency, to the contrary, for large data transfer the PIO mode subtract computational resources.

The second mode is Direct Memory Access (DMA), from the network processor point of view it can be directly mapped to NGET respect to data-read operations and mapped to PPUT respect to write operations. In DMA a PCI device can directly access the main memory both for load and store operations, without the intermediate step of the CPU registers, letting the CPU to continue computing; considering a data transfer from memory to a target device as done previously for PIO mode, the CPU must to gather the informations to carry-out the transaction and pass them to the device, at this point the CPU job is done and can continue

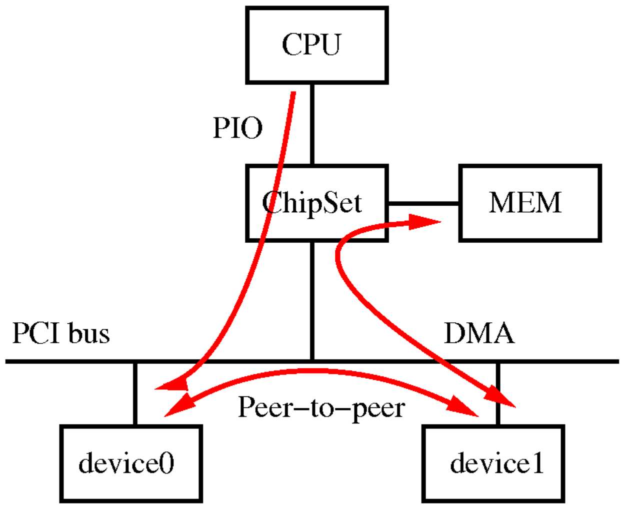

Figure 3.3: *PCI transaction modes, in PIO mode both CPU and target device are involved until the end of the transaction, data are transferred from the internal registers of the CPU to the address-space of the target device. In DMA the device directly access the memory without CPU intervention, just interacting with the memory controller, letting the CPU freely computing. In Peer-to-peer mode a master device can autonomously access a target device.*

the computation, the device can autonomously interact with the memory to transfer data. DMA mode suffers of higher latency compared to PIO due to the initial transaction between CPU and device to start the data transfer but this mode allows to free the CPU from any other intervention, letting it to dedicate to computing instead of data transfer.

The third mode is *Peer-to-peer*, it is not implemented by NWP, and it provides the direct data transfer between two PCI devices, for store operation a master directly move data to the address-space of the target, while for load operation the master interrogates the target device and then waits for the response.

The PCIe Protocols foreseen 4 transaction types, depending on the address-space they are generated. The 4 address-spaces are I) Memory, to transfer data to or from a location in the system memory map; II) IO, to transfer data to or from a location in the system IO map, this address space is kept to compatibility with legacy devices and is not permitted for Native PCI Express devices; III) Configuration, to transfer data to or from a location in the configuration space of PCIe devices, it is used to discover device capabilities, program plug-and-play features and check status; IV) Message, provides in-band messaging and event-reporting without consuming memory or IO address resources.

Of all the above mentioned transaction type, in the NWP design I used only the "Memory Address-Space", it supports both Read and Write transactions called "Memory Read Request" (MRd) and "Memory Write Request" (MWr). Each of the above mentioned operation is mapped into the data transfer unit of the PCIe protocol, the Transaction Layer Packet (TLP). Memory transaction TLP has a well-know format and size, it is composed by a header and a payload (here I do not consider the optional digest), the header includes lots of informations where the most relevant for NWP are shown in figure 3.4, they are the type of transaction, given by the union of "Type" and "Format" fields; the "Traffic class" to provide differentiated services for different packets; the "Attributes", the "Requester ID" that identifies the device initiates the transaction; the "Tag" that identifies each outstanding request issued by a requester; allowing a device to issue more transactions at the same time; the memory-address for which the transaction has been issued; and last the payload size (if any).

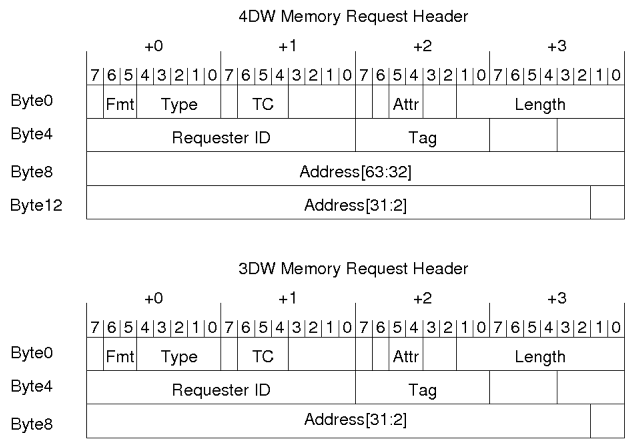

Figure 3.4: *PCI Express (PCIe) memory request headers, top-side shows the 4DW header used to operate on a 64-bit address-space, bottom-side shows the 3DW header used to operate on a 32-bit address-space*

The Memory Read Request (MRd) must be completed returning-back to the requester a Completion with Data (CplD), where the payload contains the read value while the header requires some fields coming from the request, such as the "Traffic Class", the "Attributes", the "Requester ID", the "Tag" to match the outstanding request with the completion, the "Byte Count" that specifies the remaining bytes count until a read request is satisfied, and the "Lower Address" that are the lowest 7 bits of address for the first byte of data returned with a read.

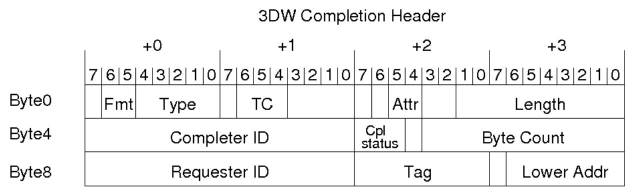

Figure 3.5: *PCI Express (PCIe) completion header.*

Considering the above mentioned memory operations, I can map the PPUT, NGET, NPUT and PGET transaction-methods in PCIe TLPs. PPUT simply translates into a Memory Write Request (MWr) TLP originating from CPU to NWP. NGET is split into four TLPs, the first is a MWr from the CPU to the NWP containing all the informations to trigger the DMA transfer from memory to Network Processor, in the specific a MRd from NWP to MEM and a CplD in the opposite direction, finally NWP notifies the CPU via a MWr. NPUT requires three

TLPs to carry-out the transaction, the first is a MWr from CPU to NWP to issue the CREDIT, then NWP deliver data to memory and notifies the CPU using two MWr. PGET is mapped to three TLPs, the first is a MWr from NWP to CPU to notify that a message has been received, the CPU then issues one or more MRd to retrieve data from NWP via CplD. A summary of the above operations is shown in table 3.1.

Method	Source	Dest	Operation	TLP
PPUT	CPU	NWP	data	MWr
NGET	CPU	NWP	snd_req	MWr
	NWP	MEM	mem_rd	MRd
	MEM	NWP	data	CplD
	NWP	CPU	dma_ntf	MWr
NPUT	CPU	NWP	crd	MWr
	NWP	MEM	data	MWr
	NWP	CPU	ntf	MWr
PGET	NWP	CPU	ntf	MWr
	CPU	NWP	rcv_req	MRd
	NWP	CPU	data	CplD

Table 3.1: *Mapping of the PPUT, NGET, NPUT and PGET transaction-methods in PCIe TLPs.*

3.2 Input/Output Controller

The Input/Output Controller (IOC), is the interface exported to the CPU by the Network Processor (NWP), providing a translation layer between the CPU's Input/Output System and the Torus Network (TNW) module; it supports three of the transaction models described in section 3.1, PPUT and NGET for transmission, and NPUT for reception. IOC layer is present only in the AuroraScience version of NWP, for the QPACE machine I've been worked only on PHY and TNW layers.

IOC has been designed to interface with the "Intel Nehalem Processor Family" whom I/O subsystem is managed by the "Tylersburg Chipset" that interfaces with the FPGA, where NWP has been implemented, via PCIe Gen 2 protocol. IOC accesses the PCIe link using two main subsystems, the former manages the transac-

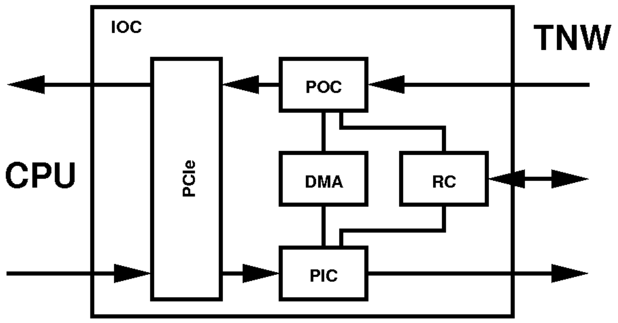

Figure 3.6: *Conceptual layout of the IOC module.*

tions from the CPU to the NWP and is called PCI Express Input Controller (PIC), the latter manages the transactions from the NWP to the CPU and is called PCI Express Output Controller (POC), these two subsystems cooperate to send/receive data among the end-points of the torus-link and to access the configuration/status registers of the NWP.

The Input/Output Controller (IOC) is mainly composed by four modules as shown in figure 3.6:

- PCI Express Input Controller (PIC)
- PCI Express Output Controller (POC)
- Register Controller (RC)
- Direct Memory Access (DMA) Controller

further each module will be explain in details, follows a brief overview of their functionalities.

PIC manages the transactions from the CPU to the NWP, it's purpose is to properly recognize the incoming transaction type analyzing the packet header and deliver it to the proper IOC module to manage that kind of transaction. POC manages the transactions in the opposite direction respect to PIC, it takes care to deliver to the CPU the transactions generated by the other IOC modules, correctly formatting the out-coming packets based on the information they pass to it. Both PIC and POC interface with the PCIe IP hardwired into the FPGA.

The purpose of the RC module is to manage the configuration and status registers of both IOC and TNW, accepting transactions from PIC both for read and write operations, and interacting with POC in case of reads, sending register-values to CPU.

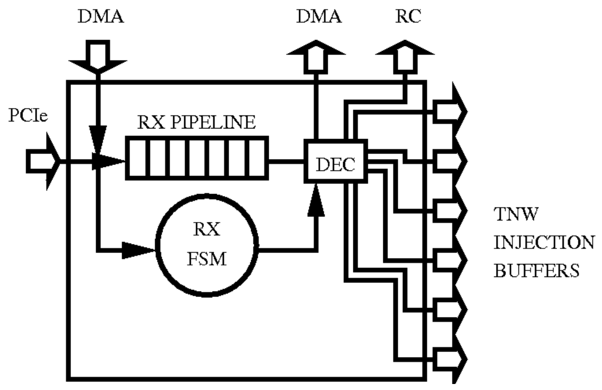

Figure 3.7: *Conceptual layout of the PIC module, incoming packets are analyzed by the RX_FSM and DEC to route the transaction to the right interface.*

The DMA engine receives from PIC the CPU instructions to fetch data to be sent over the TNW links and triggers POC to generate the corresponding memory-reads. When data are sent back from memory, DMA interacts with PIC to properly encapsulate them into TNW packets.

3.2.1 PCIe Input Controller

The PCI Express Input Controller (PIC), by itself, implements the PPUT transaction models exposed in section 3.1, in cooperation with the DMA engine exposed further, it allows also to implement the NGET method.

The PIC module is mainly a stream-pipeline where the transactions generated by the CPU are properly recognized by a finite-state machine (*RX_FSM*) and a transaction decoder (DEC), to be then delivered to the appropriate NWP subsystem, could it be an injection-buffer, the register-controller or the DMA engine. The conceptual layout of PIC is shown in figure 3.7.

The RX_FSM is controlled by two signals exported by the PCIe Macro, the first is the "Start of Packet" (SOP) that asserts, as its name explains, when the first 16-bytes item of a packet has been received, actually this word is the header of the transaction; the second signal is the "End of Packet" (SOP) that asserts to denote that the last packet item has been received. Each transaction is composed by a fixed size header followed by a payload of arbitrary length (but not exceeding the PCI protocol limit of 4 KBytes), this rules the RX_FSM layout, as shown in figure 3.8, after initialization the FSM is in the idle state (RXIDLE), waiting for packets, the SOP signal changes the state to HDR where the packet-header is

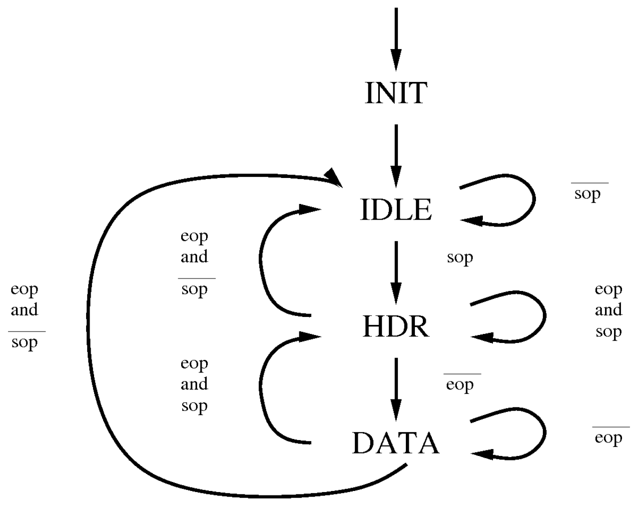

Figure 3.8: *Diagram of the finite state machine RX_FSM of the PIC module, it is controlled by the SOP and EOP signals exported by the PCIe Macro and allows to manage transactions with arbitrary payload size.*

evaluated to recognize the transaction type; in case of register-read the packet is only composed by the header and the EOP signal is asserted together with SOP; if a payload is present, the state changes to DATA where the FSM stills until the assertion of EOP when the last packet item has been received.

The header fields, particularly the memory address, is used to define the NWP sub-system to deliver the transaction, in case of data packet to be sent over a link, data will be delivered to one of the TNW injection buffers depending on the memory address stored into header; going through the pipeline some header fields are analyzed to define the destination link while others are extracted and re-elaborated to match the TNW header format; each payload item simply goes through the pipeline as it is (no data buffering is applied at IOC level of the datapath) and is flanked by his own destination address before to be stored into the injection buffer.

When a register access transaction arrives, it is simply forwarded to the RC module that will take in charge to carry out the operation.

PIC also cooperates with the DMA engine to properly deliver the incoming data resulting by a memory-read transaction (NGET model). Each memory-read issued to CPU is marked with a “tag” that will identifies the corresponding Completion with Data (CplD) are sent back to NWP. The informations to deliver data among the Torus Network are stored by the DMA module into registers addressable by the tag of the CplD, letting PIC to format the destination address of each line of the payload to adhere to the TNW format.

3.2.2 Design Optimization, TXFIFO as Re-order Buffer

An efficient implementation of the PPUT transaction model on the latest Intel processors requires to use of the “non-temporal store instructions” treating data with the Write Combining (WC) semantics, see [7] section 10.4.6.2 and [10], in this case the store transactions will be weakly ordered, meaning that the data may not be written to memory in program-order. A problem arise due to the fact that injection buffer of each link is implemented by the TNW module called Injection Buffer (TXFIFO), it is a queue where data are stored waiting to be sent over the link, the rules implemented by TNW layer imply that data must to be stored there in the strict order as they will be sent. One solution to the above problem is to re-implement the TXFIFO as a “re-order buffer” where to store and re-assemble packet-fragments sent by the CPU before being sent over the link.

The re-order buffer is subdivided into four modules:

- "DATA_MEM" is the memory where data are stored waiting for completion
- "PCK_CNT" is an array of registers that count how many items of each packet are arrived
- "RDY_FIFO" is the queue where the base addresses of the completed packets are stored
- "READ_MNG" is the FSM that pops a ready address from RDY_FIFO and extracts the corresponding data items from DATA_MEM when requested by TNW.

To follow the below description see figure 3.9. A packet is composed by 8 16-Bytes items and each item is joined with the corresponding remote address offset. When a packet-item reaches the re-order buffer (1), lowest bits of his remote address offset are used to address the DATA_MEM location where to store it;
each item stored increments the counter inside PCK_CNT (1), corresponding to the packet the item belongs, when this counter reaches the maximum value, all the items of the packet has been arrived, and his base address is pushed into RDY_FIFO (2).
When RDY_FIFO is not empty, READ_MNG pops an address and informs the

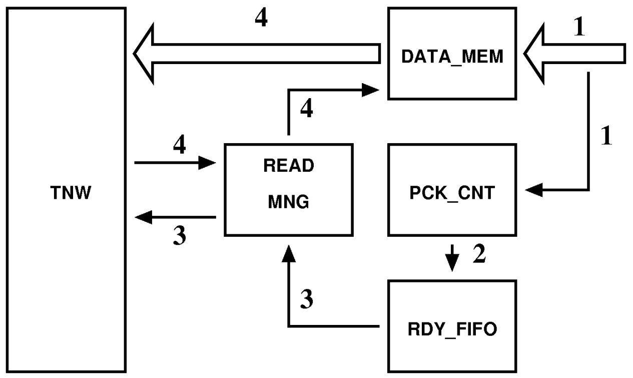

Figure 3.9: *Re-order Buffer scheme, data are stored into DATA_MEM until completion (1), PCK_CNT counts how many items of a packet are arrived (1), when done, the packet address is pushed into RDY_FIFO (2); READ_MNG interacts with TNW to send packets over the link (3,4).*

TNW (3) that a packet is ready to be sent, TNW fetches data from DATA_MEM via READ_MNG (4).

The DATA_MEM addressing limits the number of packets that can be sent in a single transaction, if the size of the message to be sent exceed DATA_MEM size, the software must to split the transaction in more of them, to avoid that more packets with the same remote address offset interfere, causing data-loss. The remote address offset interference does not affect the implementation of TXFIFO as a queue but the software-controls required to grant the packet-order limits the inbound bandwidth of TXFIFO by a factor of 5 as explained in section 5.3.

The implementation of TXFIFO as a queue has a traversal time of 17 clock-cycles, from the moment data are written into the FIFO and the moment they are extracted, while the implementation as re-order buffer has a traversal time of 20 clock-cycle, adding a penalty of just 3 extra clock-cycles but it allows to maximize the throughput of the CPU, doubling the bandwidth over the TNW link, as will be explained in section 5.3.

3.2.3 Register Access Controller

When an incoming transaction is a register access, PIC routes the packet to the RC module. The RC modules, which diagram is shown in figure 3.10,

takes care of all the configuration/status registers transactions, it is mainly composed by a command FIFO (CMD_FIFO) written by PIC and a Finite-state

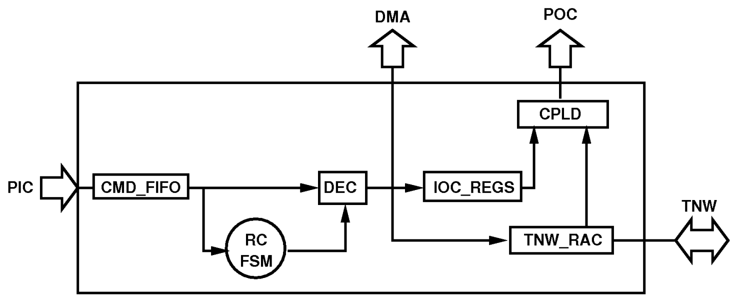

Figure 3.10: *Diagram of the Register Access Controller.*

Machine (RC_FSM) that extracts commands from the FIFO and routes the transactions to the right port, waiting for the completion of the operation before extracting another transaction from the CMD_FIFO.

The modules which RC interfaces are: I) the IOC registers, they stores all the configurations and status of the interface to the CPU of the network processor, bit-stream (firmware) that will be loaded into the FPGA at power-up, II) the TNW register interface to setup and control all the links facilities, III) the DMA engine to autonomously retrieve data from main memory, IV) the POC module to return the register values in case of register read. ENDREAD The whole TNW lies on a different clock-domain respect to IOC where RC belongs, so all the transactions to TNW registers must be synchronized to the IOC clock-domain, I implemented the synchronization into different ways respect to data and commands, both in read and write transactions, data simply falls through a "Metastability Sync." [37] [34] [38] where data outputs from one register of the out-coming clock-domain and are synchronized using two registers into the incoming clock-domain, avoiding data loss due to metastability of the registers. For the command the path is different, I used a "Pulse Sync." where the command is converted to a toggler [1] into the origin clock-domain and passes through a metastability synchronizer and then is re-converted to a pulse signal in the destination clock-domain. To be sure that the transaction has been correctly delivered, the command signals are also used as a probe, when a write command traverses all the synchronizers from IOC clock-domain to TNW domain and came back, I'm sure that the write transaction has been delivered, while in case of a read transaction the round-trip among synchronizers takes also in charge the time to respond of the TNW, granting that

[1]A toggler is a signal that flips each time a condition changes, it also known as "Non-return to zero" (NRZ) signal, because different to the normal signals, it does not pulse going to logical 1 and then to logical 0 but it just inverts his present state to the opposite.

when the command come-back to IOC domain, also the value read from TNW is already synchronized into IOC domain.

A read transaction of a NWP register is performed via a “Memory Read request” according to the PCIe protocol, this kind of transaction initiated by the CPU must end with a Completion with Data (CplD) sent back by NWP, in this case, in addition to the read register-value that figures as payload of the CplD, RC must extract all the informations from the Memory Read request to properly format the CplD header to send back, informations are: the “Traffic Class”, the “Attributes”, the “Requester Identifier (ID)”, the “Tag” and the “Lower Address Bits”, all these fields are explained in section 3.1. Once RC has collected all the informations. it triggers to the PCI Express Output Controller (POC) module to send back the completion.

3.2.4 DMA Engine

The DMA engine interacts with PIC and POC modules to implement the NGET transaction model exposed in section 3.1, DMA transactions are triggered via register-access, writing informations about the data transfer to carry-out, such informations must include all the coordinates to retrieve data from main memory and to notify the CPU about the operation has been carried-out.

Data to be retrieved are moved by the CPU in a reserved buffer allocated in main memory, the DMA engine stores the base-address of that buffer in a register called DMA_BAR, all the DMA-requests issued by CPU refer to this address, providing to the engine the offset respect to it and the size of the data to send. As well as the transaction has been carried-out, the DMA engine notifies the CPU about the transfer completion. The DMA engine diagram is shown in figure 3.11.

The DMA request is queued into the “Requests-FIFO” (REQ_FIFO) waiting for a time slot to be issued to the memory controller by the POC module (explained further in this chapter), the DMA engine provides to POC all the informations to prepare a Memory Read Request (MRd) addressed to the main memory controller. See section 3.1 for more details about memory transacions over PCIe protocol. Each memory read has his own TAG field in the header that will identify the transaction among all the ones issued by the network processor, the TAG will be kept in the Completion with Data (CplD) sent-back by the memory controller to identify at which memory read it is associated. The TAG provided by POC is used to address the DMA_STORE, a set of register-locations that keep track of the outstanding memory reads, it stores all the destination-attributes of the transactions; the destination-attributes are: the address to deliver data, the SIZE of the trans-

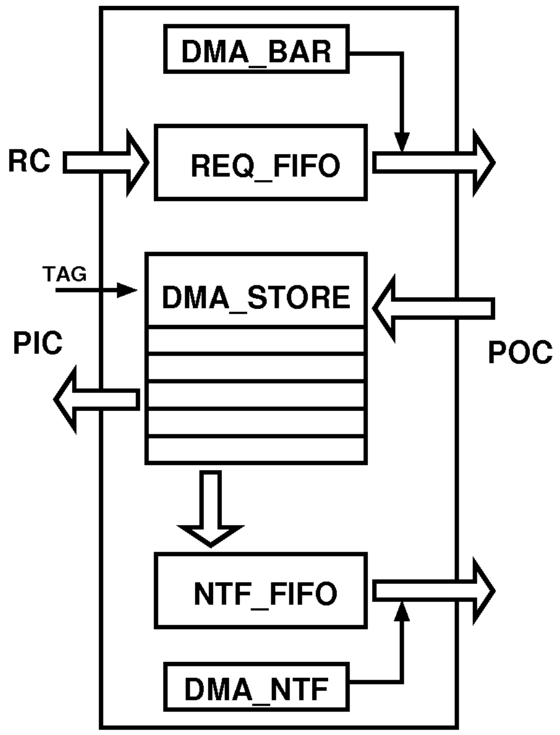

Figure 3.11: *Diagram of the DMA engine.*

action and the the Notify-ID (NID) to inform the CPU. When a CplD has been received by PIC, its TAG extracts from DMA_STORE the informations to reconstruct the destination-address of data in the format managed by the TXLINK module of the TNW layer, and the SIZE field is decremented according to the CplD size, as well as SIZE reaches the value "zero", it means that all the data has been sent over the link, at this point the NID is extracted and written into the "Notify FIFO" (NTF_FIFO) and the DMA_STORE location is freed for another transaction. The NTF_FIFO is required for the same reason as the REQ_FIFO: waiting for a time slot by the POC module but at the same time to free the DMA_STORE resources to allow new incoming transactions. The notify is generated using the same concepts explained in section 2.4.4, the DMA_NTF_BAR register stores stores the memory-address where starts the address-space of the buffer to deliver the notification relative to the specific DMA request, at this address is summed the Notify-ID (NID) field that specifies the offset to write the notify respect to NTFBAR. NTF_FIFO asserts a transaction request to POC that will carry-it-out as well as is available a time slot to send the notify to the CPU.

3.2.5 PCIe Output Controller

The PCI Express Output Controller (POC) module manages the NWP-to-CPU transactions, managing the accesses to the PCIe link issued by RC, DMA and TNW modules; apart the CplD requests issued by RC, all the transactions to the main memory are issued via the DMA mode called NPUT model in section 3.1. POC implements a separated auxiliary Finite-State Machine for each NWP module that must send data to main-memory, all controlled by the main state machine (MAIN_FSM) that catch the requests asserted by each module and, based on a priority system, triggers the corresponding FSM when a slot is available, allowing the state-machines to access the PCIe interface to carry-out its transaction. The MAIN_FSM layout is shown in figure 3.12, after a request to access the PCIe interface has been received and the PCIe interface is free to send data, MAIN_FSM goes to the START_* state triggering the corresponding FSM and moves to the BUSY_* state waiting for the end of the transaction that is achieved when the triggered FSM returns to the IDLE state, at this point MAIN_FSM is able to serve another request.

As previously said, POC implements more auxiliary FSM, follows a brief description of their functionalities.

The TNW_FSM is the only finite-state-machine that directly interacts with a module external to the IOC layer, it's purpose is to deliver to the CPU the data received

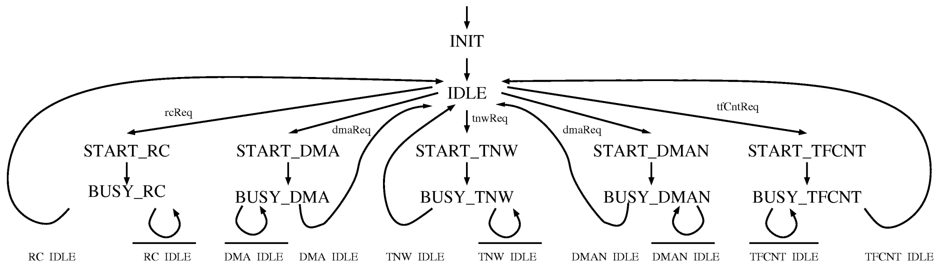

Figure 3.12: *Diagram of the finite state machine MAIN_FSM of the POC module, it is controlled by the requests issued by the modules that must access the PCIe interface to send data to the CPU. The FMS goes to the START_* state when the corresponding request has been received and this transaction triggers the corresponding FSM to access the PCIe, then MAIN_FSM stalls in the BUSY_* state until the triggered FSM does not returns to the IDLE state, meaning that the transaction has been successfully completed, at this point MAIN_FSM can return to his own IDLE state, ready to serve another transaction.*

by the 6 TNW links; an arbitration stage, based on round-robin schedule, decide which link-request to satisfy, at this point the finite-state-machine interacts with the receiver-buffer of the TNW link to extract data to be delivered to the CPU. When all the packets related to a message has been sent, the TNW link also asserts the request to send the NOTIFY to the CPU.

The RC_FSM satisfies the requests issued by the RC module, the module that interacts with the registers of the TNW layers and the IOC internal registers, when a register-read-request is issued to the RC module, it fetches data to send-back to the CPU and gathers from the read-request the informations to format the completion header then this data are presented to POC and the MAIN_FSM is triggered with a request to access the PCIe link; when a slot is available the MAIN_FSM triggers the RC_FSM to start, which took control of the PCIe interface and acts to send-back the completion with data to the CPU.

The purpose of DMA_FSM and DMA_NTF_FSM is to carry-out the NGET transactions issued by the CPU, the former issues the memory-reads required to retrieve data from main memory to be sent over the TNW links, the latter issues the NPUT operations to write in main memory the NOTIFY location related to the NGET transactions as well as they are carried-out to inform the CPU about the send completion.

The TFCNT_FSM implements the back-pressure of the TXFIFO-modules on the TNW layer writing into a main-memory location their status; at each TXFIFO is associated one counter that keeps track of how many packets had been extracted, these counters are written into main-memory (NWP_CNT) and are writable only by the network processor, they make pair with another set of counters (CPU_CNT) writable only by the CPU, each time the CPU inject packets into one TXFIFO, it updates the related CPU_CNT, while when NWP_CNT are updated by NWP when packets are extracted from TXFIFO, the difference between CPU_CNT and NWP_CNT reflects the number of packets still into the injection buffer and the CPU can throttle his transactions avoiding data loss due to FIFO full. A software-programmable threshold controls the refresh rate of NWP_CNT in main-memory. A more accurate description about this mechanism is in section 4.3.3.

Chapter 4

Software Layers

The Network Processor (NWP) has been design to provide a high-bandwidth low-latency link between computing node of a massive parallel machine, at hardware level this has been done implementing a lightweight custom communication protocol among point-to-point links. The same requirements must to be honored also at software level, at this purpose I implemented a software-stack to allow applications to directly access the Network Processor without the intervention of the operating system, avoiding more of the overheads introduced by it.

The software layers of the Network Processor (NWP) has been designed for the Linux Operating System and optimized for the "Intel Nehalem Processor Family" who interfaces with the hardware layers of NWP via PCIe Gen 2 protocol. At the lowest layer of the software-stack, shown in figure 4.1, lies the Device Driver (DRV), it implements the kernel module to provide an entry-point to NWP, integrated into the operating system environment, accessible by the applications. DRV takes care to instantiate in the kernel-space the data structures and buffers access the Network Processor. Upon the Device Driver (DRV) services has been implemented a Low Level Library (LIB) to provide to threaded-applications a set of functions and routines to configure and access NWP directly from user-space. LIB is based on the following assumptions:

- only one application can access NWP at a time
- NWP is configured directly by the application owning it at start-up
- other applications can own NWP only after his release

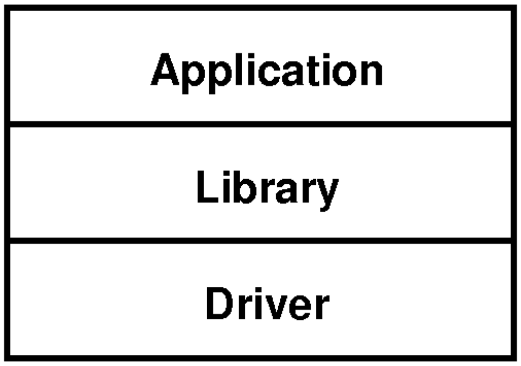

Figure 4.1: *NWP software stack. The lowest layer is the Device Driver (DRV) who configures the network processors and allocates in main memory the required buffers. Over the Device Driver (DRV) is implemented the Low Level Library (LIB) who exports to the application all the basic routines to configure/monitor the network processor as well as the functions to send/receive data over links.*

4.1 Communication Model

The Network Processor's communication model is based on the "Two-Sided Communication Protocol" as explained in section 2.1, two applications at the end-points of a link that want to exchange a message, must to cooperate to carry-out the transaction. The receiver must to provide to his own Network Processor the informations to deliver the incoming message to the receiving buffers in main memory (CREDIT), then to be waiting for the notification about the complete delivery of the message (POLL), while the sender must to move the message into the injection buffer of his own NWP (SEND).

As explained in section 3.1, there are two methods to implement the SEND operation: PPUT and NGET, one way to efficiently implement the **PPUT** method is to use the "Memory Mapped I/O", this mechanism allows to associate a range of memory addresses to device memory, whenever an application reads or writes in the assigned address range, it is actually accessing the device[43]. So mapping the injection buffers in the main memory address space (TX-BUFF) allows to move data to be sent just performing a write operation to a memory address, then the CPU will convert this transaction into a write operation to the device. NWP has been designed to support communications among multi- and many-core CPUs, providing a set of "Virtual Channels" (VC) to allow to access the link at different core-pairs site in adjacent computing-node, at this purpose each link address space

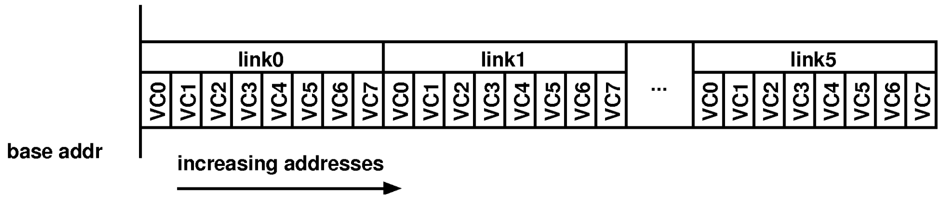

Figure 4.2: *Mapping of the injection buffers (links) and related virtual channels (VCs) into main memory using the "Memory Mapped I/O" to implement the PPUT method. To send a message along the desired link using a given VC is just a matter to calculate the corresponding offset respect to the base address where the first injection buffer has been mapped.*

has been sub-divided into memory areas corresponding to different VCs, to send data along a defined link using a given VC is just a matter to move data at the correct offset respect to the base address where the first injection buffer has been mapped, as shown in figure 4.2.

The efficiency of "Memory Mapped I/O" mechanism can be improved marking the memory area as Write Combining (WC)[10], more I/O transactions to contiguous memory addresses can be accumulated into the Write Combining Buffer (WCB) and delivered to the Network Processor as a single I/O transaction, minimizing the overheads related to the transaction-setup exploiting the bandwidth between the CPU and the Network Processor.

To implement the SEND operation with the **NGET** method, one way is to allocate in main memory a buffer (NGET-BUFF) where the CPU copies data to be sent and triggers the Network Processor to retrieve them via Direct Memory Access (DMA), at this point the communication is completely asynchronous respect to the CPU. After triggering the SEND operation, the CPU can dedicate to other tasks such as computation, and can ignore the status of NGET-BUFF as long as it must perform a new communication. One mechanism to keep track of the communication status is to allocate in main memory a "notify buffer" (NGET-NTF-BUFF) where the CPU sets a value when performs the SEND operation and the Network Processor modifies it when the communication has been carried-out, to know the communication status, the CPU must to check this memory location (NGET-POLL).

Respect to the receiving operations, the Network Processor requires a buffer (RX-BUFF) where to deliver the messages incoming from the links via DMA transaction, as previously said the Network Processor manages more virtual-channels per link and can be adopted the same philosophy as TX-BUFF: the destination address of data inside RX-BUFF depends from the link they income and the virtual-channel they belong. As further explained the actual implementation of RX-BUFF is quite different, due to the problem to obtain large contiguous areas of physical-memory from the system, has been allocated a different memory area per virtual-channel. The informations to the Network Processor about the memory address where to deliver data are provided by the application issuing a CREDIT, it also contains the size of the message the application is waiting for and the "notify-index" (NID); as for the notify-system of the NGET method, the application can keeps track of the receiving-status checking the "notify buffer" (NTF-BUFF), this is a set of memory locations addressable by the NID where the application sets a value when performs the CREDIT operation to receive a message and the Network Processor modifies it when all the packets related to a message has been received, to know the reception status, the application must to check this memory location performing the POLL operation, when the POLL operation detects that the message has been received, it frees the RX-BUFF moving data to a user-defined buffer where the application can use them.

The access to the NWP's configuration/status registers is performed via "Memory Mapped I/O", the whole register address-space is mapped into kernel-space (RC-BUFF) where read/write operations are converted into I/O transactions.

For performance-critical applications like the communications among the Network Processor, Memory Mapping (MMAP) can be implemented to provide user programs with direct access to device memory[43], all the above mentioned buffers, allocated in kernel-space, are re-mapped into the application-space, limiting the overheads given by frequent context switches between user- and kernel-mode.

4.2 Driver

The Network Processor is seen by the operating system as a PCI character device, each PCI device has a unique identifier given by the number of the PCI bus, the slot it is plugged and the function number; after the device has been discovered by the system, it can be accessed by the applications via two methods, the former foreseen to associate the PCI identifier to a file descriptor, letting the applications to access it via read and write system-calls managed by the operating system; the latter maps the device to one or more memory address spaces, setting the Base Address Register (BAR), and then re-mapping these addresses into the user-space letting the applications to access the peripheral performing read/write operations directly to these memory addresses, avoiding all the overheads given by the previous method. The accesses to NWP are performed using the latter method described above.

The *int init_module(void)* function initializes all the kernel structures and the required buffers, see section 4.1. The network processor is accessible by the CPU basically via 2 address-spaces, the former is a 64-bit wide address-space to access the injection buffers, the latter, 32-bit wide, allows to access the configuration/status register. The PCI devices stores into registers the physical addresses assigned to these address-spaces by the system at boot-time, calling these registers Base Address Register (BAR). The *init_module()* function reads these BARs and using the sys-call

```
void __iomem * ioremap (ulong offset, ulong size);
```

it re-maps the corresponding address-spaces into kernel-space. The injection buffers are re-mapped to TX-BUFF while the register access is re-mapped to RC-BUFF. To perform efficient transactions to TX-BUFF, this address-space must to be marked by the Device Driver (DRV) as Write Combining (WC) setting the "Memory Type Range Registers" (MTRRs) of the CPU[10], the operating system provides the sys-call

```
int mtrr_add (ulong base, ulong size, uint type, char increment);
```

The reception buffer (RX-BUFF) requires to allocate in kernel-space a significant amount of contiguous memory, as explained in section 4.1, due to the problem to obtain large contiguous areas of physical-memory from the system, the reception buffer has been allocated as separated memory areas, one per virtual-channel, using the sys-call

```
ulong __get_free_page(int flags);
```

to allocate big chunks of memory.

All the other buffers such as NGET-BUFF, NGET-NTF-BUFF and NTF-BUFF, has been allocated using the sys-call

```
void * kmalloc (size_t size, int priority);
```

due to they do not require a large amount of memory as RX-BUFF.

The Device Driver (DRV) implements the Memory Mapping (MMAP) method, it is part of the *file_operations* structure and is invoked when the *mmap* sys-call is issued. This method allows the mapping of device memory directly into a user process's address space, letting the application to direct access the buffers associated to the device, instead of issuing system calls, limiting the overheads due to continuous context switchings. The buffers allocated in kernel-space by the *init_module()* function are flagged with "PG_reserved"[44] meaning that they are accessible to kernel-only, the Device Driver (DRV) exports the function

```
static int ftnw_mmap ( struct file * filp, struct vm_area_struct * vma );
```

that implements MMAP and resetting the PG_reserved flag by calling the sys-call

```
int remap_pfn_range ( struct vm_area_struct * vma, ulong addr, ulong pfn,
ulong size, pgprot_t prot)
```

to re-map in user-space the buffers allocated in kernel-space.

4.3 Low-level Library

The Low Level Library (LIB), called **libftnw**, has been designed upon the services provided by the Device Driver (DRV); it provides to the applications a set of functions to directly access the Network Processor, a sub-set of these functions, to send/receive data and register access, are listed in the following sections, divided in to category by their purpose. The full set of functions is listed in section B.

4.3.1 Device Initialization and Release

The LIB provides to the applications the *int ftnwInit(void)* routine to configures the Network Processors and all the software structures to properly access the device.

This routine calls the *int open(const char *pathname, int flags)* sys-call to open the file descriptor associated to the entry point of the Network Processor, then it use the sys-call *mmap()* to re-map in user-space all the buffers allocated by the Device Driver (DRV) in kernel-space such as TX-BUFF, RC-BUFF, NGET-BUFF, NGET-NTF-BUFF, RX-BUFF, NTF-BUFF and TFCNT-BUFF to allow the application to directly access them without the intervention of the operating system. This routine returns 0 in case all the operations has been successfully executed, -1 otherwise.

The LIB also provides the function *int ftnwOpen(void)* to the applications that just require to open the device without re-mapping the buffers in user-space. Returns the NWP file descriptor.

To release the device at the end of the application, the LIB provides *int ftnwFinalize (void)* and *ftnwClose(void)*, both close the NWP file descriptor and always return 0.

4.3.2 Register Access

The LIB provides the register access to NWP performing read and write operations to RC-BUFF where the registers are mapped, offsets respect to the base of RC-BUFF are the register-addresses. NWP's registers are all 32-bit-wide except the requests for DMA Engine that are 64-bit-wide.

The application accesses a register in write-mode by calling the function

int ftnwPokeReg (uint regaddr, uint regval);

the parameter *regaddr* is the address of the register to write to while *regval* is the 32-bits value to write. The version of this function for 64-bit write is

int ftnwPokeReg64 (uint regaddr, ulong regval);

To access the registers in read-mode I implemented the function

*int ftnwPeekReg (uint regaddr, uint * regval);*

as for the write functions, the parameter *regaddr* is the register-address to read while the 32-bit value returned by the read operation is stored into *regval*. I do not implemented a 64-bit version of this function due to the fact that the only 64-bit-wide register in NWP is the request-FIFO of the DMA Engine and it is write-only.

4.3.3 PPUT Send

The LIB provides two sets of functions that implements the SEND operation, depending on the transaction model used: **PPUT** or **NGET**, see section 3.1 for more details.

The PPUT model is implemented by the function

*int ftnwSend (uint lid, uint vcid, void * txbuf, uint txoff, uint msglen);*

its purpose is to send a given amount of data (*msglen*) stored into the buffer *txbuf* at the offset *txoff* along a given link *lid* using the virtual-channel *vcid*; both msglen and txoff are in units of 128 Bytes, the size of the single packet that can be moved along the link.

In this section I explain the most performing implementation of *ftnwSend()*, the one that properly works if the TXFIFO has been implemented as a re-order buffer, in section 5.3 will be presented the comparison with other implementations.

This function is sensitive to the status of the injection buffer to avoid data-loss due to packet-overwrite given by to the implementation of the re-order buffer, see section 3.2.2, so *ftnwSend()* must to check the emptiness of the injection buffer and to obtain the exclusiveness to move data in it, the last condition has been attained using a *mutex*, while the other has been attained with the back-pressure method explained as follows. I implemented the back-pressure using a set of two counters per link, at each injection buffer is associated one counter that keeps track of how many packets had been injected, these counters (CPU_CNT) are writable only by the *ftnwsend()* function, they make pair with another set of counters (NWP_CNT) writable only by the Network Processor, each time *ftnwsend()* moves data to an injection buffer, it updates the related CPU_CNT, while NWP_CNT are updated by NWP when packets are extracted to be sent over the link, the difference between CPU_CNT and NWP_CNT reflects the number of packets still into the injection buffer and *ftnwsend()* can throttle his transactions avoiding data loss.

If the comparison of the above mentioned counters reveal the emptiness of the injection buffer, *ftnwsend()* moves data into it. I implemented the data-moving using the "Intel Intrinsic"[15], an API extension built into the compiler that allows to use a set of functions written in C language that the compiler directly maps to assembly instructions. Data moving, shown in figure 4.3, is basically divided into three steps:
1) Data load from main memory to CPU's registers
2) Data stream to "memory mapped I/O" addresses
3) Fence operation to ensure flush of remaining WCB at the end of the message transmission.

To fully exploit the bandwidth of the link between the CPU and the Network Processor, the memory area, where the injection buffers has been mapped, has been marked by the Device Driver (DRV) as Write Combining (WC), in this case data to be sent to contiguous memory addresses can be accumulated into the Write Combining Buffer (WCB) and delivered to the Network Processor as a single I/O transaction, minimizing the overheads related to the transaction-setup, WCB size in Intel processors is not architecturally defined, in the Nehalem/Westmere processors available on the AuroraScience machine, the WCB size is 64-bytes corre-

```
if (isempty_txfifo()) {

  // Lock the mutex to send over a link
  pthread_mutex_lock (&mutex[lid]);

  // Send packets
  while (there_are_pck_to_send) {

    // Point (1) move 64-bytes from memory to registers
    xmm0 = _mm_load_si128((__m128i *)  ubuf+0 );
    xmm1 = _mm_load_si128((__m128i *)  ubuf+1 );
    xmm2 = _mm_load_si128((__m128i *)  ubuf+2 );
    xmm3 = _mm_load_si128((__m128i *)  ubuf+3 );

    // Point (2) move 64-bytes from registers to combining buffer
    _mm_stream_si128((__m128i *) kbuf+0, xmm0 );
    _mm_stream_si128((__m128i *) kbuf+1, xmm1 );
    _mm_stream_si128((__m128i *) kbuf+2, xmm2 );
    _mm_stream_si128((__m128i *) kbuf+3, xmm3 );

    // Point (1) move 64-bytes from memory to registers
    xmm0 = _mm_load_si128((__m128i *)  ubuf+4 );
    xmm1 = _mm_load_si128((__m128i *)  ubuf+5 );
    xmm2 = _mm_load_si128((__m128i *)  ubuf+6 );
    xmm3 = _mm_load_si128((__m128i *)  ubuf+7 );

    // Point (2) move 64-bytes from registers to combining buffer
    _mm_stream_si128((__m128i *) kbuf+4, xmm0 );
    _mm_stream_si128((__m128i *) kbuf+5, xmm1 );
    _mm_stream_si128((__m128i *) kbuf+6, xmm2 );
    _mm_stream_si128((__m128i *) kbuf+7, xmm3 );

    increase(ubuf);
    increase(kbuf);
  }
  // Point (3) grant flush of combining buffers
  _mm_sfence();

  // Release the mutex
  pthread_mutex_unlock (&mutex[lid]);

}
```

Figure 4.3: *Conceptual extract of the ftnwSend() function, if the injection buffer is empty and the thread obtain the exclusiveness to send data over the link (mutex), data are loaded from main-memory (1) to CPU internal registers and then are moved to WCB (2) to be sent to the Network Processor. The "store fence" (3) should to grant the flush of the WCBs before to release the mutex.*

sponding to the "cache-line" size. Referring to code in figure 4.3, I implemented point (1) using the intrinsic function
*__m128i _mm_load_si128(__m128i const*p)*,
it is directly mapped to the assembly instruction **MOVDQA**[8] that loads 128 bits of data from the user-buffer *ubuf* into one of the XMM registers, executing 4 of these instructions half of the packet to be sent over the link has been loaded into the CPU registers and can be executed the point (2). I based point (2) on the "non-temporal move instructions"[10] of the Intel Processors, they allow to move data from the processor's registers directly into system memory without being also written into the L1, L2, and/or L3 caches. For each of the 4 XMM registers loaded at point (1) is called the intrinsic function
void _mm_stream_pd(double p, __m128d a)*,
mapped on the assembly instruction **MOVNTDQ** that moves the 128 bit item to the address space where is mapped the injection buffer, as previously said, that address space has been set as WC by the Device Driver (DRV) so the items are actually stored into a WCB that when it has been full-filled, it triggers a 64-Bytes-wide transaction to the PCIe link. The "fence" operation at point (3) grants that all the WCBs has been flushed before *ftnwSend()* returns, I implemented this operation using the intrinsic function *_mm_sfence()* that is mapped on the assembly instruction **SFENCE**, it serializes load and store operations from/to memory

4.3.4 NGET Send

The NGET transaction model explained in section 3.1 is implemented by the function

> *int ftnwNgetSend (uint lid, uint vcid, void * txbuf, uint txoff,*
> *uint nid, uint msglen);*

its purpose is to send a given amount of data (*msglen*) stored into the buffer *txbuf* at the offset *txoff* along a given link *lid* using the virtual-channel *vcid*; both msglen and txoff are in units of 128 Bytes, the size of the single packet that can be moved along the link. The *nid* parameter is explained further.

This function copies data from the application-space to the NGET-BUFF where NWP will retrieve them to be sent; the PCIe transactions are limited in the size of 4KB so *ftnwNgetSend()* splits the whole size of the message in sub-requests with a maximum size of 4KB and issues them to the DMA engine of the Network Processor using register access interface function *ftnwPokeReg64()*. Due to the split of SEND transaction in sub-requests, I provided a way to the application to keep track of all of them (if required), each request has his own notify-index that is calculated incrementing the *nid* parameter provided to the function. To simplify *ftnwNgetSend()* usage, this function returns the last notify-index issued for the whole SEND operation, that the application can check to ensure about the communication has been carried-out.

The LIB provides the function

int ftnwNgetTest (uint nid);

to check if a NGET transaction has been carried-out, due to the implementation of *ftnwNgetSend()* the nid parameter could be any index between the one provided to *ftnwNgetSend()* as parameter and the last nid returned by that function. Generally *ftnwNgetTest()* is called to check only the last nid returned by *ftnwNgetSend()*. Non-blocking function, immediately returns in both cases if nid has been set or not; the blocking version of this function is

int ftnwNgetPoll (uint nid);

that polls the notify-index nid until it is not set, meaning that all the sub-requests of the NGET SEND has been satisfied.

4.3.5 Receive

The LIB provides a set of functions for the receive operations, as explained in section 4.1, the data incoming from the Network Processor must to be stored into main memory as well as they arrive, the informations where to store them must to be provided by the application issuing a CREDIT. I implemented the CREDIT operation with the function

int ftnwCredit (uint lid, uint vcid, uint rxoff, uint msglen, uint nid);

The above function provides the coordinates where to store data in main memory via the parameters link-id (lid), virtual-channel-ID (vcid) and the offset inside the RX-BUFF (rxoff); the size of the message is provided by the parameter msglen, while the nid parameter is the notify-index the Network Processor must to set to inform the application that the whole message has been received.

As previously said in section 4.1, to check if the whole message has been received, the application must to check the notify-index performing a POLL operation, I implemented that operation with the function

> *int ftnwTest (uint lid, uint vcid, uint rxoff, uint msglen, void * rxbuf, uint nid);*

the parameters *lid*, *vcid* and *nid* are combined to obtain the offset inside NTF-BUFF to check for the notify about message receiving, if this location has not yet been set by the Network Processor, the function immediately returns and the application must to further check the notify-index. If the notify-location has been already set, the function combines the parameters *lid*, *vcid* and *rxoff* to obtain the offset inside RX-BUFF where the message has been stored by NWP and moves the amount of data given by the parameter *msglen* to the user-defined buffer pointed by *rxbuf*.

The blocking version of this function is

> *int ftnwPoll (uint lid, uint vcid, uint rxoff, uint msglen, void * rxbuf, uint nid);*

that continue to check the notify-index until it's not set.

4.4 Application Examples

Follows some application examples based on the **libftnw** library.

4.4.1 Ping Example

The simplest application example is "Ping", it is useful to understand the basis of the Network Processor communication protocol, the SENDER, pictured as a red-node in figure 4.4, sends a message using the *ftnwSend()* function, while the RECEIVER, the blue-node, issues the credit to receive the message with *ftnw-Credit()* and waits for the message delivery polling for the notify with *ftnwPoll()*, as well as the message has been arrived the latter function moves data to rxbuf where the receiver can use them for further computations.

The access to the Network Processor is granted to the application by calling *ftnwInit()*. At the end, the application must to release the Network Processor by calling *ftnwFinalize()*.

The source-code is listed in figure 4.5, it does not include all the variable declaration and others operations to save space.

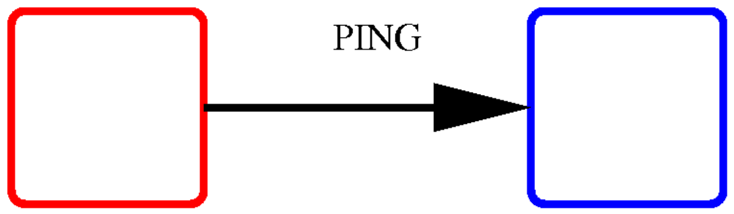

Figure 4.4: *Ping Example, the red node sends a message to the blue one.*

4.4.2 Ping-Pong Example

The “Ping-Pong” example is similar to Ping but in this case, one node called INITIATOR, sends a message to the other node called COMPLETER, that must send-back the message to INITIATOR.

The INITIATOR, pictured as a red-node in figure 4.6, first of all issues the credit for the PONG that will be sent back by the REPEATER, it is good practice to issue the credit as soon as possible to avoid dead-locks or delays. Then the INITIATOR sends a message using the *ftnwSend()* function. The REPEATER, the blue-node, issues the credit as first operation with *ftnwCredit()* and waits for the message delivery polling for the notify with *ftnwPoll()*. As well as the message has been receive, *ftnwPoll()* moves data to rxbuf as temporary data-location, then data are immediately sent to the INITIATOR using *ftnwSend()*. The INITIATOR waits for the PONG to come-back using *ftnwPoll()*, once this function returns, the PING-PONG data exchange has been correctly done.

The access to the Network Processor is granted to the application by calling *ftnwInit()*. At the end the application must release the Network Processor calling *ftnwFinalize*.

The source-code is listed in figure 4.7, it does not include all the variable declaration and others operations to save space.

```
/************* SENDER *************/

#include <libftnw.h>

int main (...) {

  ulong txbuf __attribute__((aligned(16)));

  ...

  ftnwInit();

  // Send PING
  ftnwSend( XPLUS, VC0, &txbuf, 0, msgsize );

  ...

  ftnwFinalize();
}


/************* RECEIVER *************/

#include <libftnw.h>

int main (...) {

  ulong rxbuf __attribute__((aligned(16)));

  ...

  ftnwInit();

  // Issue CREDIT for PING
  ftnwCredit( XMINUS, VC0, 0, msgsize, nid );

  ...

  // Polling for PING NOTIFY
  ftnwPoll( XMINUS, VC0, 0, msgsize,
           &rxbuf, nid );

  ...

  ftnwFinalize();
}
```

Figure 4.5: *Ping source-code example.*

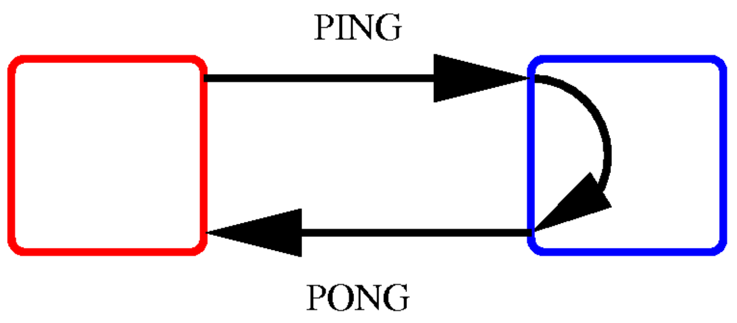

Figure 4.6: *Ping-pong Example, the red node sends a message (PING) to the blue one that, immediately send-it-back (PONG).*

```
/************* INITIATOR *************/

#include <libftnw.h>

int main (...) {

  ulong txbuf __attribute__((aligned(16)));
  ulong rxbuf __attribute__((aligned(16)));

  ...

  ftnwInit();

  // Issue CREDIT for PONG
  ftnwCredit( XMINUS, VC0, 0, msgsize, nid );

  // Send PING
  ftnwSend( XPLUS, VC0, &txbuf, 0, msgsize );
  ...

  // Polling for PONG NOTIFY
  ftnwPoll( XMINUS, VC0, 0, msgsize,
            &rxbuf, nid );

  ftnwFinalize();
}


/************* COMPLETER *************/

#include <libftnw.h>

int main (...) {

  ulong rxbuf __attribute__((aligned(16)));

  ...

  ftnwInit();

  // Issue CREDIT for PING
  ftnwCredit( XMINUS, VC0, 0, msgsize, nid );

  ...

  // Polling for PING NOTIFY
  ftnwPoll( XMINUS, VC0, 0, msgsize,
            &rxbuf, nid );

  // Send-back PONG
  ftnwSend( XPLUS, VC0, &rxbuf, 0, msgsize );

  ...

  ftnwFinalize();
}
```

Figure 4.7: *Ping-Pong source-code example.*

Chapter 5

Results and Benchmarks

5.1 FPGA Synthesis Report

Table 5.1 shows the synthesis-report of the Network Processor targeting the FPGA Altera Stratix IV 230 GX, this synthesis includes the following main-features:

- 1 Altera PCI Express Hard-IP block
- 1 IOC interface to CPU including the DMA engine
- 6 Torus Network links equipped with the re-order buffers

The whole design has been properly mapped and routed into the FPGA without any timing-slack among logic-elements; it has been synthesized at a frequency of 250 MHz, using the 18% of the available logic and the 17% of the memory embedded into device. The present implementation of NWP uses only one out of two of the PCI Express hard-IP blocks embedded inside the FPGA fabric.

5.2 PHY Bit Error Rate Test

The Bit Error Rate (BER) defines a characteristic for the communication quality between end-points, it is related to the number of faulty bits among the whole amount of bit transmitted. The PCIe protocol defines a tolerance of 1 faulty bit among a total of 10^{12} bits transmitted. To test the actual BER of the PM8354 PHY, I implemented a test-bench where two PHYs communicate via cables of different length (0.5, 1 and 3 meters). I used the "test-bench" firmware described

Fitter Summary	
Fitter Status	Successful
Quartus II 64-Bit Version	9.1 Build 222 10/21/2009 SJ Full Version
Family	Stratix IV
Device	EP4SGX230KF40C2
Logic utilization	18 %
Combinational ALUTs	17,308 / 182,400 (9 %)
Memory ALUTs	3 / 91,200 (< 1 %)
Dedicated logic registers	25,130 / 182,400 (14 %)
Total registers	25631
Total pins	612 / 888 (69 %)
Total block memory bits	2,504,480 / 14,625,792 (17 %)
Total GXB Receiver Channel PCS	8 / 24 (33 %)
Total GXB Receiver Channel PMA	8 / 36 (22 %)
Total GXB Transmitter Channel PCS	8 / 24 (33 %)
Total GXB Transmitter Channel PMA	8 / 36 (22 %)
Total PLLs	7 / 8 (88 %)

Table 5.1: *Synthesis report for the Network Processor targeting the FPGA Altera Stratix IV 230 GX.*

in section 2.4.5, in an extensive campaign to test the PM8354 link, the firmware has been deployed into a Xilinx Virtex-5 Development kit where has been plugged a PM8358 test-bed board developed by Karl-Heinz Sulanke from DESY-Zeuthen (DE), the boards are shown in figure 5.1. Two of the above mentioned setup has been connected using cable of different length (0.5, 1 and 3 meters) and run the tests for several days up to 21 of bidirectional transmissions without any error detected; in figure 5.2 is shown the setup with 0.5 m cable.

I done an equivalent test on the AuroraScience version of the NWP, running for one week without errors.

A most significant test has been run on the QPACE machines installed at the Jülich Forschungszentrum and Wuppertal University in Germany, in this case has been not used the test-bench firmware but a statistical-physics application running for 24 hours on both installations using a total amount of 512 node for a grand-total of 1536 active links, detecting a total amount of 39 code-violation/disparity errors. Considering the number of errors detected by the two machines (ERR_NUM), the full overlap among computation and data transmission over the 24 hours (RUN-

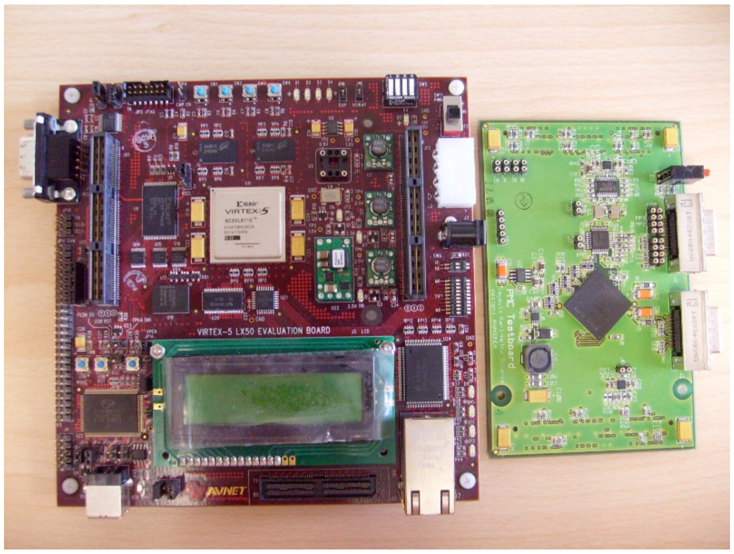

Figure 5.1: *Hardware components for the BER test of the PM8354 PHY, on the left-side is shown the FPGA development kit while on the right-side is shown the PHY test-bed.*

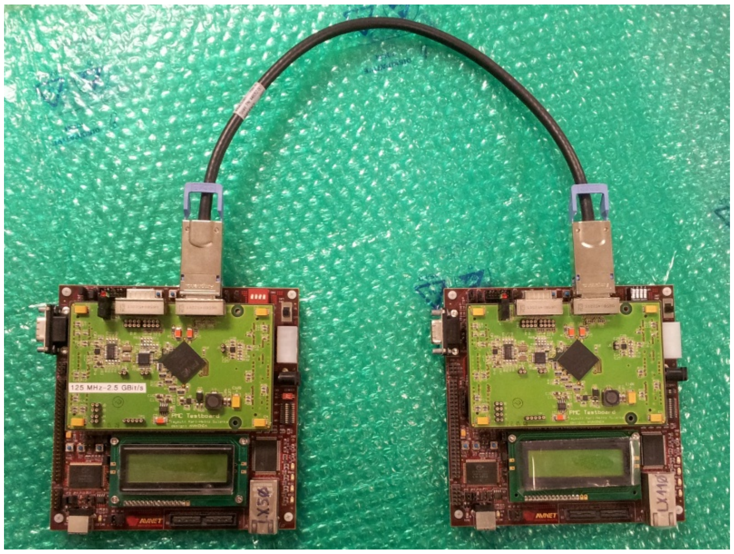

Figure 5.2: *The dual end-point setup for the BER test. Here two FPGA development kits are equipped with the PHY test-beds and connected by a 50cm cable. Each FPGA has been programmed with the test bench firmware for the full-duplex communications. This setup has been used for an extensive campaign with many runs up to 21 days of bidirectional transmissions without any error detected.*

TIME), the number of active links (ACT_LINKS), the bandwidth of 1 GB/s per link (BW_LINK), and assuming that for each error only 1 bit has been flipped, the Bit-Error Rate has been

$$\begin{aligned} BER &= ERR\_NUM/((RUNTIME * 60 * 60) * ACT\_LINKS * BW\_LINK \\ &= 39/((24 * 60 * 60) * 1536 * (1 * 10^9)) = 2.939 * 10^{-16} = \\ &\approx 3 * 10^{-16} \end{aligned}$$

5.3 CPU-to-NWP Transmission with PPUT Model

In this section I discuss several ways to realize the *ftnwSend()*[1] function of the library which implements the software layer of the PPUT model as explained in section 3.1. In particular I focus on techniques to gain best performances on the PCI Express bus between CPU and NWP.

The PPUT operation is based on the Programmed Input/Output (PIO) scheme. During the boot phase the injection buffers of the Network Processor (NWP) are memory-mapped into the memory-space of the CPU. Data can then moved from the main memory to the injection buffers performing store operations on specific memory addresses.

The communication-bus between CPU and NWP is a 8x PCI Express bus. Each line runs @5 Gbit/s, then the over-all maximum raw bandwidth is 40Gbit/s. Due to the 8b/10b encoding required to reconstruct the transmission clock on the receiving side, each Byte (8 bits) sent is encoded using 10 bits. Taking into account this overhead, the maximum achievable bandwidth is

$$(8/10) * 40Gbit/s = 32Gbit/s = 4GByte/s$$

As mentioned in section 3.1, the data transfer unit of the PCIe protocol is the Transaction Layer Packet (TLP), which includes the following items:

- 1 Byte START character to mark the begin of the packet;
- 2 Bytes encoding the index number of the packet used to detect missed or dropped packets;

[1]The *ftnwSend()* function is explained in section 4.3.3.

- 12 or 16 Bytes depending on the TLP type (3DW or 4DW) to encode the header ; [2]
- a payload which size can range from 0 to 4096 Bytes;
- 4 optional Bytes of ECRC for end-to-end error detection;
- 4 Bytes of LCRC for the error detection on the TLP;
- 1 Byte of END character.

In the computing-system I took into account, described in section 3.2, the injection- and receiver-buffers of the network processor are mapped into a 64-bit address-space, and the ECRC is not used. In this configuration TLPs use a 4DW (16 Bytes size) header and the PIO mode of the Intel CPU achieves a maximum burst payload of 64-Bytes. The overall TLP-size results to

$$1 + 2 + 16 + 64 + 4 + 1 = \mathbf{88Bytes}$$

The payload/TLP efficiency-ratio of $64/88 = 0.727$, and the maximum achievable bandwidth over the link is

$$0.727 * 4GBytes/s = 2.909GBytes/s$$

The above result is an upper-bound of the achievable bandwidth and will be used as reference to compare the efficiency of several implementations of the PPUT method that is described in the following.

Figure 5.3 shows a first implementation of the PPUT operation. Data are first loaded from the user memory area pointed by *uoff* into the 128-bit variables mapped on the SSE registers (XMMx) of the CPU. Then, contents of the registers are stored on consecutive memory-addresses of the kernel pointed by the *koff* variable, previously mapped into user space by standard Memory Mapping (MMAP) system-call.

Using the code of fig. 5.3 each _mm_stream_si128() is translated to a pcie-write of 16-Bytes performed every 120 ns (30 cycles), see fig. 5.4. This achieves a CPU-to-FPGA bandwidth of 0.133 GB/s.

$$BW = \frac{16Bytes}{120ns} = 0.133GB/s$$

```
xmm0 = _mm_load_si128((__m128i *)  uoff+0 );
xmm1 = _mm_load_si128((__m128i *)  uoff+1 );
xmm2 = _mm_load_si128((__m128i *)  uoff+2 );
xmm3 = _mm_load_si128((__m128i *)  uoff+3 );
xmm4 = _mm_load_si128((__m128i *)  uoff+4 );
xmm5 = _mm_load_si128((__m128i *)  uoff+5 );
xmm6 = _mm_load_si128((__m128i *)  uoff+6 );
xmm7 = _mm_load_si128((__m128i *)  uoff+7 );

_mm_stream_si128((__m128i *) koff+0, xmm0 );
_mm_stream_si128((__m128i *) koff+1, xmm1 );
_mm_stream_si128((__m128i *) koff+2, xmm2 );
_mm_stream_si128((__m128i *) koff+3, xmm3 );
_mm_stream_si128((__m128i *) koff+4, xmm4 );
_mm_stream_si128((__m128i *) koff+5, xmm5 );
_mm_stream_si128((__m128i *) koff+6, xmm6 );
_mm_stream_si128((__m128i *) koff+7, xmm7 );
```

Figure 5.3: *Scheme of the PPUT method implemented using Intel intrinsics functions, the _mm_load_si128() load 128 bits of data from main memory to CPU registers _mm_stream_si128() writes data to memory-addresses where injection buffers have been mapped resulting in PCI Express transactions.*

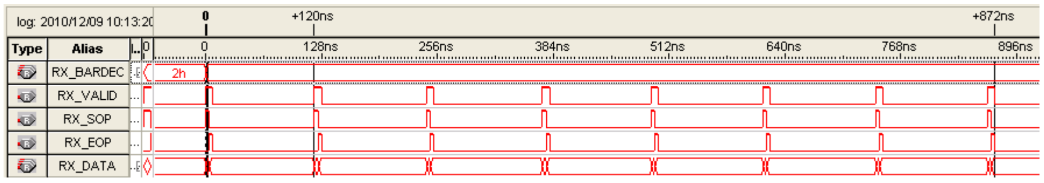

Figure 5.4: *Tracing at the FPGA boundaries of the PCI Express transactions issued by the code in figure 5.3, each _mm_stream_si128() results in a 16 Bytes TLP every 120 ns.*

Arranging the sequence of load and store in a different way (e.g. interleaving loads and stores) does not result in better performance.

Stream operations are issued to consecutive memory addresses and a way to limit the setup overhead of the pcie transactions is to configure the memory areas of injection buffers as Write Combining (WC). This operation can be done by modifying the Memory Type Range Registers (MTRR) of the CPU. This is a technique supported by Intel processors which use Write Combining Buffer (WCB) available on the CPUs to enable burst transfer from CPU to IO sub-system, both memory and IO devices. Store operations are staged on internal buffers of the

[2]In PCI Express Protocol the 3DW (Double Word) header is used when the device has been mapped in a 32 bits address space, while the 4DW header is used in case of 64 bits address space.

CPU. As a Write Combining Buffer (WCB) is full, it is flushed to the memory system generating a unique burst transfer with a payload of 64 Byte (the size of a cache-line). Write Combining Buffer (WCB)s may be flushed also for other reasons, like de-scheduling of threads or because of an interrupt. This may cause out-of-order issuing of store-operations, and mixing items of two different WC-buffers, which are then written into the FPGA with a different oder, corrupting the integrity of the message. To control flushing of WC buffers, and enforce ordering I use memory-fence instructions, see code in figure 5.5. Two blocks of four 16-Bytes items are first moved from memory to SSE registers, and then stored to memory. At end of each store block, a memory fence is issued to force the flush of the WC buffer and keep the order of stores. This requires also a mutex to avoid interleaving of items issued by different threads.

```
  pthread_mutex_lock();

  xmm0 = _mm_load_si128((__m128i *)  uoff+0 );
  xmm1 = _mm_load_si128((__m128i *)  uoff+1 );
  xmm2 = _mm_load_si128((__m128i *)  uoff+2 );
  xmm3 = _mm_load_si128((__m128i *)  uoff+3 );

  _mm_stream_si128((__m128i *) koff+0, xmm0 );
  _mm_stream_si128((__m128i *) koff+1, xmm1 );
  _mm_stream_si128((__m128i *) koff+2, xmm2 );
  _mm_stream_si128((__m128i *) koff+3, xmm3 );

  _mm_sfence();

  xmm0 = _mm_load_si128((__m128i *)  uoff+4 );
  xmm1 = _mm_load_si128((__m128i *)  uoff+5 );
  xmm2 = _mm_load_si128((__m128i *)  uoff+6 );
  xmm3 = _mm_load_si128((__m128i *)  uoff+7 );

  _mm_stream_si128((__m128i *) koff+4, xmm0 );
  _mm_stream_si128((__m128i *) koff+5, xmm1 );
  _mm_stream_si128((__m128i *) koff+6, xmm2 );
  _mm_stream_si128((__m128i *) koff+7, xmm3 );

 _mm_sfence();

 pthread_mutex_unlock();
```

Figure 5.5: *Implementation of PPUT method to exploit the Write Combining Buffer (WCB). 64 Bytes of data are loaded from memory to registers with using _mm_load_si128() intrinsics and then stored to WCB using the _mm_stream_si128() intrinsic. The _mm_sfence() intrinsics serializes the use of WCB, granting the order of the transaction to the PCI Express link between CPU and FPGA.*

Using this approach, and neglecting the time to check the status of the transmission fifos, I write 64 Bytes every ≈ 124 ns, see figure 5.6, which translates to a CPU-to-FPGA bandwidth of ≈ 0.516 GB/s.

$$BW = \frac{64Bytes}{124ns} = 0.516GB/s$$

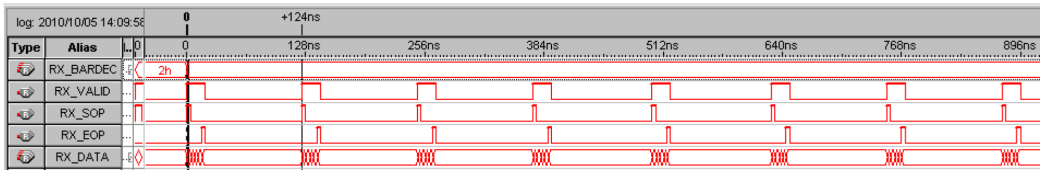

Figure 5.6: *Tracing at the FPGA boundaries of the PCI Express transactions issued by the code in figure 5.5, four _mm_stream_si128() result in a 64 Bytes TLP every 124 ns.*

To overcome the problem of unpredictability of flush of WC-buffers, the txFifo has been implemented as a *reorder-buffer*, see section 3.2.2, an addressable memory capable to keep order of the 16-Bytes data-items issued by the CPU. The reorder buffer module is implemented using a memory called DATA_MEM and a fifo called RDY_FIFO. The lower three bits of the PCI-e address are used to address the data memory to stores the 16-Bytes data-items into lines of 128-Bytes. As one memory-line is filled, i.e. one TNW-packet is ready, the address is pushed into the address-ready fifo. As txLink would pop a TNW-packet, an address is popped-out from the address-ready fifo, and it is used to read one row of the data memory. Use of reorder buffer eliminates the overhead of fence operations, WC-buffers can be fully or partially flushed with any order, and the CPU can use all WC-buffers available. Using this approach the CPU issues in average, a PCI-e write of 64-Bytes every 24 ns, see fig 5.7, corresponding to a CPU-to-FPGA bandwidth of 2.667 GB/s.

$$BW = \frac{64Bytes}{24ns} = 2.667GB/s$$

Taking into account this limit I obtain:

The traversal-time of the Network Processor, the time needed to move a packet of 128B from the PCIE-RX interface to the PCI-TX interface of the peer, including the traversal time of the PHYs and the cable, is 596 ns, see figure 5.8.

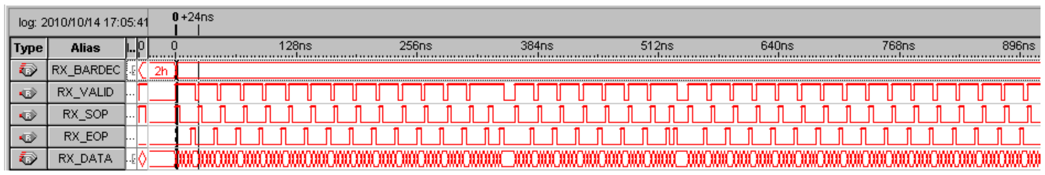

Figure 5.7: *Tracing at the FPGA boundaries of the PCI Express transactions using Write Combining (WC) without fences among packets and the re-order buffer inside the FPGA, four _mm_stream_si128() result in a 64 Bytes TLP every 24 ns.*

Firmware	Inbound Bandwidth	Efficiency
no-WC	0.133 GB/s	4.6 %
with-WC	0.516 GB/s	17.7 %
WC+ROB	2.667 GB/s	91.7 %

Table 5.2: *CPU to FPGA bandwidth achieved by the three PPUT approaches.*

5.4 CPU-to-NWP Transmission with NGET Model

In this section I present the results of my implementation of the NGET model, explained in section 3.1, focusing on latency and bandwidth of the PCI Express link among CPU and NWP.

NGET operation is based on Direct Memory Access (DMA). The CPU passes to NWP the memory-addresses of data to be sent that will interact with the memory controller to retrieve data to be sent independently from the CPU. Data to be sent must to be moved from the user-buffer into the NGET-buffer, a memory area allocated by the kernel and re-mapped into user-space to be accessible by the application, then one or more DMA requests are issued to the Network Processor until the whole message has not been sent.

The DMA module supporting NGET operations with the following features:

- 8-command fifo with a maximum length of 1024 DW each (4 Kbytes)
- pci-read requests are popped-out from the command-fifo and issued by the FPGA one after the other
- a notify item is issued after all completion transactions have been received
- no check is implemented on txFifo status before issuing a command

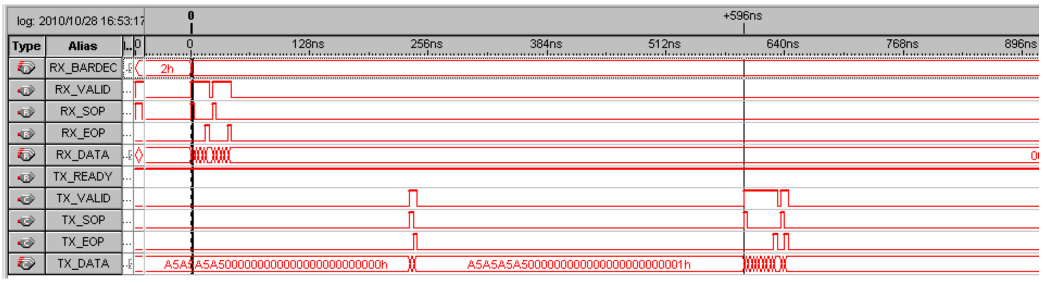

Figure 5.8: *596 ns is the traversal-time of the Network Processor (NWP) including the CPU-interface, the Torus Network link and two PHYs.*

- no support for back-pressure is implemented
- max of 8 tags are used

In figure 5.9 is shown the behavior of a NGET operation:

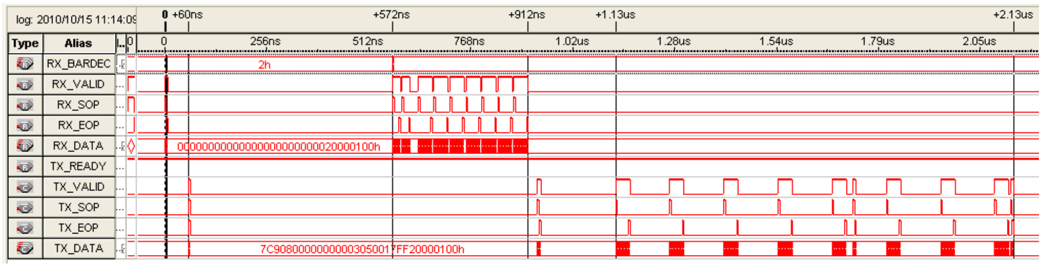

Figure 5.9: *Timing of the NGET operations traced inside the FPGA.*

1. the CPU issues a NGET command at time 0
2. after 60 ns POC issues a read request to the memory controller
3. after 512 ns completion packets arrive
4. at time 936 the notify for NGET is sent to the CPU

The latency of NGET has been measured issuing 1000 NGET transactions with the minimum data-size for the Network Processor, 128 Bytes, and calculating the average time among them, see the code in figure 5.10, the time has been calculated considering:

```
for (iter=0; iter < 1000; iter++ ) {

  rdtscpll(startTick, aux);

  // Move data from user-space to NGET-buffer and
  // issue the transaction to \ac{NWP}
  nid = ftnwNgetSend(lid, vcid, &txbuf, txoff, nid, msglen);;

  // poll for NGET notification
  ftnwNgetPoll(nid);

  rdtscpll(endTick, aux);

}
```

Figure 5.10: *Code to benchmark NGET latency.*

- the data movement from user-buffer to kernel-buffer
- the issue of the NGET command to NWP
- wait for the notification about the transaction has been carried-out

The function ftnwNgetPoll() features the _mydelay() routine used to introduce a delay (as active wait of CPU) in the loop waiting for the notification, to avoid/reduce conflicts accessing the memory.

In the following table I summarized the results of the tests with various values of delay and payload of 128 Bytes:

delay (μsec)	min (μsec)	max (μsec)	avg (μsec)
0	1.68	2.83	1.71
0.125	1.68	3.18	1.71
0.250	1.68	2.30	1.69
0.500	1.60	2.17	1.61
1.000	2.13	3.14	2.14
1.500	1.59	3.12	1.61
1.700	1.80	3.49	1.81
2.000	2.09	3.90	2.10

In figure 5.11 I report the latencies measured during 1000 tests with delay set to zero.

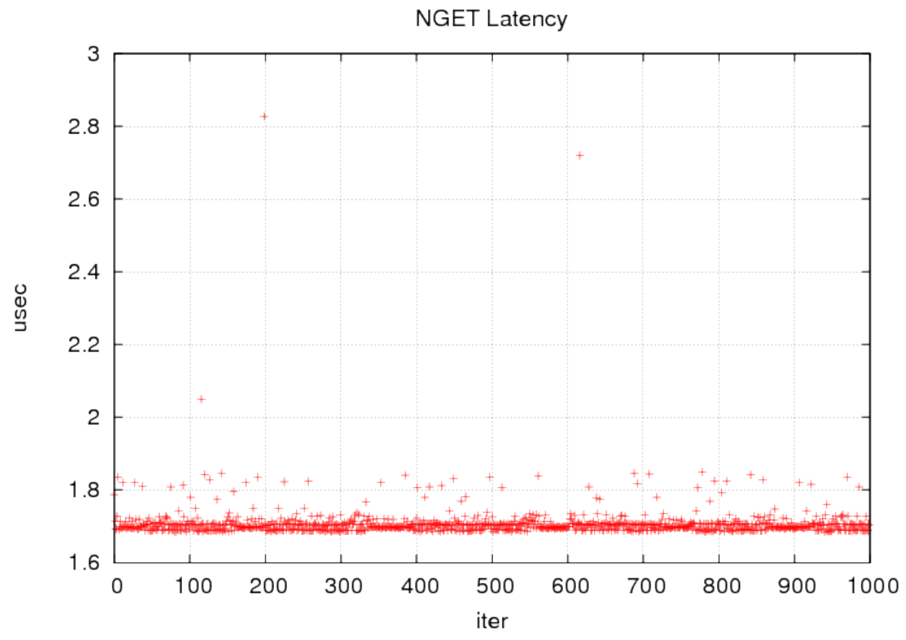

Figure 5.11: *NGET latencies over 1000 measurements with delay=0 and payload=128B.*

In the table 5.3 I report the time of NGET as function of payloads. In the last column I report the corresponding bandwidth.

In figure 5.12 is shown a snapshot of a NGET operation with a payload size of 1024 Bytes:

- at time 0 (corresponding to the bold black bar) the NGET command arrives
- at time $0 + 60$ ns the FPGA issues the memory read
- at time $0 + 584$ ns the data arrives
- at time $0 + 948$ ns the notify is sent to the CPU

The FPGA inbound bandwidth is $(924 - 584)/1024 = 3.01$ GB/s.

size (Bytes)	min (μsec)	max (μsec)	avg (μsec)	Bw (GB/s)
128	1.69	1.83	1.71	0.075
256	1.74	1.81	1.76	0.146
512	1.83	1.90	1.85	0.277
1024	2.05	3.55	2.09	0.489
2048	2.56	3.28	2.59	0.791
4096	3.21	3.44	3.27	1.253

Table 5.3: NGET *time as function of payload (100 samples).*

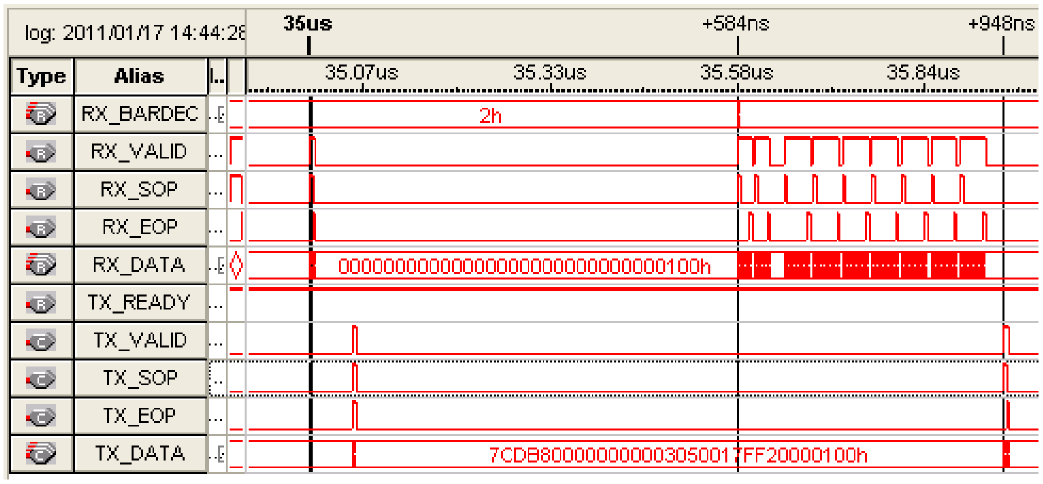

Figure 5.12: *Snapshot of NGET with payload=1024 B.*

To benchmark the bandwidth achieved with the NGET model, I implemented a "dense transactions" scheme to hide the latency effects, see code in figure 5.13, transactions are started in a loop such that during each iteration are started $N/2$ new transactions and waiting for $N/2$ other transactions to be completed. The maximum number of transactions in flight is equal to N. I run this scheme for several message-size from 128 Bytes to 4 Kbytes.

In the following tables I report the results for N=4 and N=7:

The link between CPU and NWP is composed by 8 lanes PCI Express Gen. 2 for a maximum raw bandwidth of 40Gbit/s; due to the 8b/10b encoding required to reconstruct the transmission clock on the receiving side, each Byte (8 bits) sent

```
rdtscpll(startTick, aux);

for ( 1 .. N ) {
  copy_data_toNget_dmaBuffer(nid);
  issue_Nget_request(nid);
  nid = nid + 1;
}

for ( iter=0; iter < 1000; iter++ ) {
  for ( 0 .. 8-N ) {
    copy_data_toNget_dmaBuffer(nid);
    issue_Nget_request(nid);
    nid = nid + 1;
  }

  for ( 0 .. 8-N ) {
    wait_nget_notify(ngetWaitId);
    ngetWaitId = ngetWaitId + 1;
  }
  nid = nid % 8; ngetWaitId = ngetWaitId % 8;
}

for ( 0 .. 8-N ) {
  wait_nget_notify(ngetWaitId);
  ngetWaitId = ngetWaitId + 1;
}

rdtscpll(endTick, aux);
```

Figure 5.13: *Code to benchmark NGET bandwidth.*

Size (Bytes)	BW (GB/s)
128	0.420
256	0.862
512	1.439
1024	2.603
2048	3.136
4096	3.147

Table 5.4: *NGET Bandwidth N=4, 1000 samples.*

Size (Bytes)	BW (GB/s)
128	0.450
256	0.869
512	1.596
1024	2.734
2048	3.122
4096	3.143

Table 5.5: *NGET Bandwidth N=7, 1000 samples.*

is encoded on the lanes with 10 bits, for a maximum bandwidth of

$$(8/10) * 40Gbit/s = 32Gbit/s = 4GByte/s$$

The data transfer unit of the PCIe protocol is the TLP (Transaction Layer Packet) that is composed by

- 1 Byte of START character
- 2 Bytes of sequence number to detect missed or dropped packets
- 12 or 16 Bytes of header depending on the TLP type (3DW or 4DW) [3]
- from 0 to 4096 Bytes of payload
- optionally 4 Bytes of ECRC for end-to-end error detection

[3]In PCI Express Protocol the 3DW (Double Word) header is used when the device has been mapped in a 32 bits address space, while the 4DW header is used in case of 64 bits address space.

- 4 Bytes of LCRC for the error detection on the TLP
- 1 Byte of END character

in the computing-system I consider, I mapped the nget-buffer in a 64 bits address space, resulting in a 4DW TLP-header (16 Bytes size), the ECRC has not been used, and the DMA mode of the Intel CPU in my setup, has been set for a maximum burst payload of 128-Bytes, so the overall TLP-size is $1+2+16+128+4+1=\mathbf{152Bytes}$ with a payload/TLP efficiency-ratio of $128/152=0.842$, the maximum usable bandwidth over the link results in

$$0.842 * 4GBytes/s = 3.386GBytes/s$$

comparing this upper-bound with the bandwidths in tables 5.4 and 5.5 with a payload of 4 KBytes, the efficiency of my implementation of the NGET method reaches $\approx 93\%$ of the bandwidth available between the CPU and the Network Processor.

5.5 CPU-to-CPU Transmission Benchmarks

In this section I will present the benchmarks on the link among Network Processors, comparing my implementations of PPUT and NGET models. Both models has been benchmarked using code based on the "libftnw" library and then using code that directly re-implements the libftnw features reducing software overheads. The CPU-to-CPU bandwidths and transmission-times for PPUT and NGET have been measured using a single threaded program that sends/receives data over TNW, which links are physically looped-back.

5.5.1 Benchmarks Using libftnw

The transmission time is measured at code level between the function-call to send data, **ftnwSend** for PPUT method and **ftnwNgetSend/ftnwNgetPoll** for the NGET's, and the function-call that polls for data arrival, **ftnwPoll**.

Table 5.6 and 5.7 show the transmission times and bandwidth depending on packet size respectively for PPUT and NGET methods.

The transmission time for the PPUT method is fitted by

$$f(x) = 1.561 + 0.001 * x$$

```
  for (iter = 0; iter < niter; iter++) {
    ftnwCredit( );                // set credit

    rdtscll(startTick);

#ifdef PPUT
      ftnwSend    ( );            // send using PPUT method
#endif

#ifdef NGET
      ftnwNgetSend( );            // send using NGET method
      ftnwNgetPoll( );            // poll for DMA completion
#endif

    ftnwPoll( );                  // poll for data arrival

    rdtscll(endTick);
  }
```

Figure 5.14: *Code for CPU-to-CPU transmission time and bandwidth measurement.*

PPUT									
Pckt size (B)	128	256	512	1024	2048	4096	8192	16384	32768
time (μsec)	1.674	1.815	2.178	2.782	3.891	6.299	11.012	20.675	39.406
BW (GB/s)	0.076	0.141	0.235	0.368	0.526	0.650	0.744	0.792	0.832

Table 5.6: *PPUT transmission times and bandwidths depending on packet size.*

while the transmission time for the NGET method is fitted by

$$f(x) = 2.103 + 0.001 * x$$

resulting in a latency of 1.561 microseconds for the PPUT and 2.103 for the NGET.

The Torus Network (TNW) implements a custom communication protocol, over 4 PCIe Gen1 lanes as physical layer. The custom protocol allows to send only packets with a fixed size payload of 128 Bytes, preceded by a 4 Bytes header and followed by a 4 Bytes Cyclic Redundancy Check (CRC) for the error detection; totally a packet is composed by $4+128+4 = 136 Bytes$, for a payload per packet ratio of $128/136 = 0.941$.

As previously told, the physical link is composed by 4 PCIe Gen1 lanes, for a maximum raw bandwidth of 10Gbit/s; due to the 8b/10b encoding necessary to

NGET									
Pckt size (B)	128	256	512	1024	2048	4096	8192	16384	32768
time (μsec)	2.612	3.538	3.806	4.109	4.997	8.051	13.162	24.172	141.113
BW (GB/s)	0.059	0.105	0.188	0.292	0.438	0.570	0.670	0.728	0.764

Table 5.7: *NGET transmission times and bandwidths depending on packet size.*

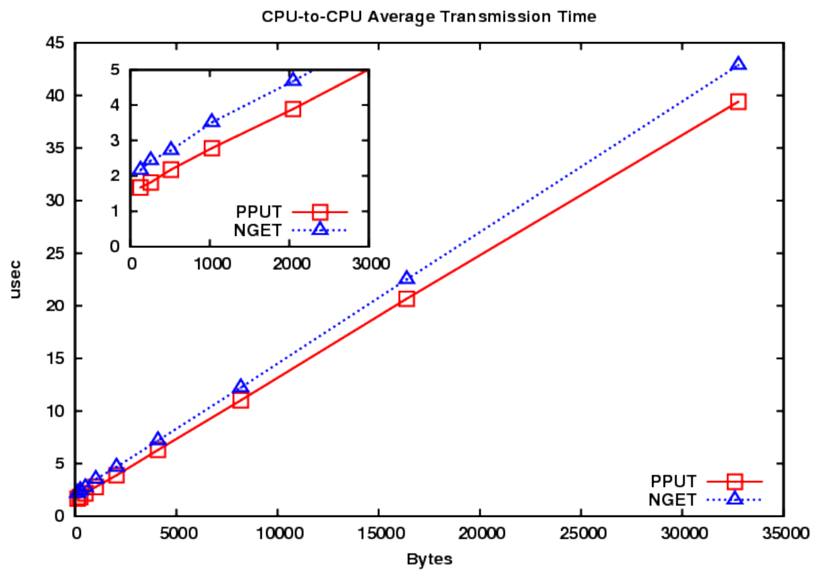

Figure 5.15: *CPU-to-CPU transmission time measurement of PPUT and NGET methods.*

reconstruct the transmission clock on the receiving side, each Byte (8 bits) sent is encoded on the lanes with 10 bits, for a maximum bandwidth of

$$(8/10) * 10Gbit/s = 8Gbit/s = 1GByte/s$$

Considering the maximum raw bandwidth and the payload per packet ratio, the maximum usable bandwidth over the link results in

$$0.941 * 1GBytes/s = 0.941GBytes/s$$

this upper-bound of the bandwidth will be used as reference to compare the efficiency of my implementations of the PPUT and NGET at TNW layer.

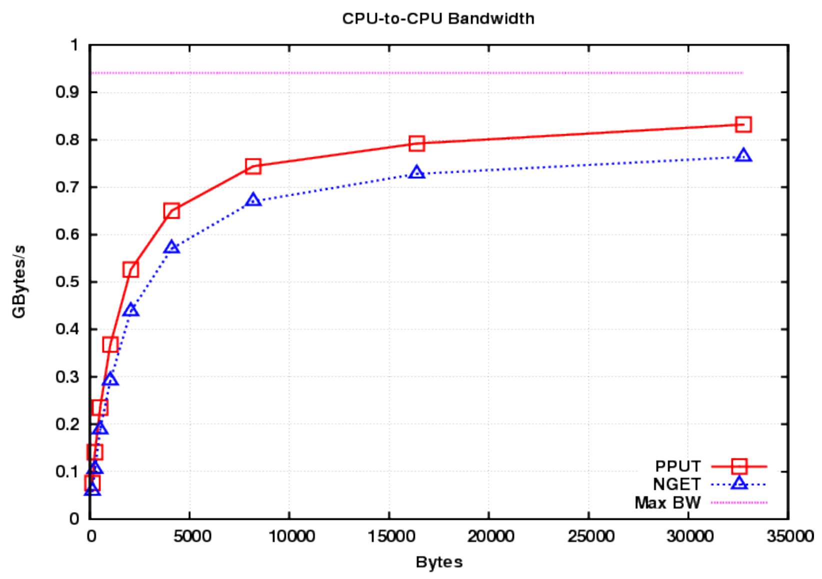

Figure 5.16: *CPU-to-CPU bandwidth measurements of PPUT and NGET methods, the purple line at 0.941 GBytes/s indicates the max usable bandwidth of the TNW link.*

5.5.2 Benchmarks Not Using libftnw

The CPU-to-CPU bandwidth and transmission time are measured using a single threaded program (testDMA8) that sends/receives data over TNW, which links are physically looped-back. Differently from testDMA6, testDMA8 do not use the functions exported by libftnw to send/receive data, but re-implements the features directly into the code, to reduce software overhead. The CPU-to-CPU transmission path is split into 3/4 sections, depending on the transmission method, and each time must to be considered as the elapsed from a common start.

PPUT time is subdivided as follows:

- SEND consider the transfer time between CPU-registers to PCIe
- POLL consider SEND time plus
  - transfer time between CPU and FPGA
  - transfer time between FPGA and FPGA
  - transfer time between FPGA and CPU
  - polling time

Method	BW (GB/s)	max%
PPUT	0.832	88
NGET	0.764	81

Table 5.8: *Comparison between PPUT and NGET methods, the max bandwidth is measured with a packet size of 32KB, compared in percentile with the max bandwidth (0.941 GBytes/s) achievable in the TNW link.*

- PCPY consider POLL time plus the copy of the data between DMA buffer to USR buffer

NGET time is subdivided as follows:

- DCPY consider the copy of the data between USR buffer to DMA buffer
- SEND consider DCPY time plus
    - time to format and transfer the request to the FPGA
    - transfer time between FPGA and CPU of the memRead
    - transfer time between CPU and FPGA of the completion
    - transfer time between FPGA and CPU of the notify
- POLL consider SEND time plus
    - transfer time between CPU and FPGA
    - transfer time between FPGA and FPGA
    - transfer time between FPGA and CPU
    - polling time
- PCPY consider POLL time plus the copy of the data between DMA buffer to USR buffer

The firmware used for the tests is the **ftnww-0421-A1B0**, running on the board **anode033**.

Tables 5.9 and 5.10 show the transmission times and bandwidth depending on packet size respectively for PPUT and NGET methods.

```
for (iter = 0; iter < niter; iter++) {
  ftnwCredit( ); // set credit
  rdtscp(startlow, starthigh, startaux); // START TIME COUNT
  ...
    for ( i = 0; i < len; i++ ) { // send using PPUT method
      vtmp0 = _mm_load_si128((__m128i *)  uoff_tx+(i<<3)+0 );
      ...
      _mm_stream_si128((__m128i *) koff_tx+(i<<3)+7, vtmp3 );
    }
    _mm_sfence();
    // GET SEND TIME FOR PPUT
    rdtscp(endlow[SEC_SEND], endhigh[SEC_SEND], endaux[SEC_SEND]);
  ...
  do { // poll for data arrival
    pollcnt--;
  } while ( (*notify_a != POLLMAGIC) && (pollcnt > 0) );
  // GET POLL TIME
  rdtscp(endlow[SEC_POLL], endhigh[SEC_POLL], endaux[SEC_POLL]);

  for ( i = 0; i < len; i++ ) // copy dmaBuffer to rxBuffer
  ...
  // GET PCPY TIME
  rdtscp(endlow[SEC_PCPY], endhigh[SEC_PCPY], endaux[SEC_PCPY]);
}
```

Figure 5.17: *Code for PPUT method CPU-to-CPU transmission times and bandwidths measurement.*

```
for (iter = 0; iter < niter; iter++) {
  ftnwCredit( ); // set credit
  rdtscp(startlow, starthigh, startaux); // START TIME COUNT
  ...
  // send using NGET method
    for ( i = 0; i < len; i++ ) // copy txBuffer to dmaBuffer
    ...
    // GET DCPY TIME
    rdtscp(endlow[SEC_DCPY], endhigh[SEC_DCPY], endaux[SEC_DCPY]);
    ...
    while ( i < lendw ) {
      ...
      _mm_stream_si128((__m128i *)(ftnw_descr->racbuf + DMAREQ),
          _mm_set_epi32(0, 0, (int)(regval >> 32), (int)(regval & 0xFFFFFFFF)));
      ...
    }
    do { // poll for DMA completion
      dmapollcnt--;
    } while ( (*dmanotify_a != POLLMAGIC) && (dmapollcnt > 0) );
    // GET SEND TIME FOR NGET
    rdtscp(endlow[SEC_SEND], endhigh[SEC_SEND], endaux[SEC_SEND]);
  }

  do { // poll for data arrival
    pollcnt--;
  } while ( (*notify_a != POLLMAGIC) && (pollcnt > 0) );
  // GET POLL TIME
  rdtscp(endlow[SEC_POLL], endhigh[SEC_POLL], endaux[SEC_POLL]);

  for ( i = 0; i < len; i++ ) // copy dmaBuffer to rxBuffer
  ...
  // GET PCPY TIME
  rdtscp(endlow[SEC_PCPY], endhigh[SEC_PCPY], endaux[SEC_PCPY]);
}
```

Figure 5.18: *Code for NGET method CPU-to-CPU transmission times and bandwidths measurement.*

Pckt size (B)	128	256	512	1024	2048	4096	8192	16384	32768
SEND time	0.015	0.018	0.127	0.143	0.489	1.200	2.742	5.849	12.198
SEND BW	8.389	12.979	4.045	7.157	4.192	3.413	2.988	2.801	2.686
POLL time	1.656	1.792	2.063	2.603	3.690	5.959	10.221	18.886	36.253
POLL BW	0.077	0.143	0.248	0.393	0.555	0.687	0.801	0.868	0.904
PCPY time	1.734	1.876	2.160	2.769	3.917	6.471	11.222	20.354	39.424
PCPY BW	0.074	0.136	0.237	0.370	0.523	0.633	0.730	0.805	0.831

Table 5.9: *PPUT Method, Transmission times and bandwidth per each section, depending on packet size. Times in μsec, bandwidth in GB/s*

Pckt size (B)	128	256	512	1024	2048	4096	8192	16384	32768
DCPY time	0.016	0.020	0.030	0.050	0.164	0.352	0.671	1.332	2.808
DCPY BW	8.191	12.531	16.834	20.324	12.513	11.645	12.202	12.304	11.668
SEND time	1.649	1.704	1.874	2.013	2.389	3.186	4.795	8.073	14.732
SEND BW	0.078	0.150	0.273	0.509	0.857	1.286	1.708	2.030	2.224
POLL time	2.232	2.364	2.724	3.262	4.387	6.707	11.368	20.713	39.552
POLL BW	0.057	0.108	0.188	0.314	0.467	0.611	0.721	0.791	0.828
PCPY time	2.311	2.449	2.892	3.433	4.601	7.099	12.148	22.270	42.630
PCPY BW	0.055	0.105	0.177	0.298	0.445	0.577	0.674	0.736	0.769

Table 5.10: *NGET Method, Transmission times and bandwidth per each section, depending on packet size. Times in μsec, bandwidth in GB/s*

The transmission time for the PPUT method, excluding PCPY time, is fitted by

$$f(x) = 1.538 + 0.001 * x$$

while the transmission time for the NGET method, excluding PCPY time, is fitted by

$$f(x) = 2.065 + 0.001 * x$$

resulting in a latency of 1.538 microseconds for the PPUT and 2.065 for the NGET.

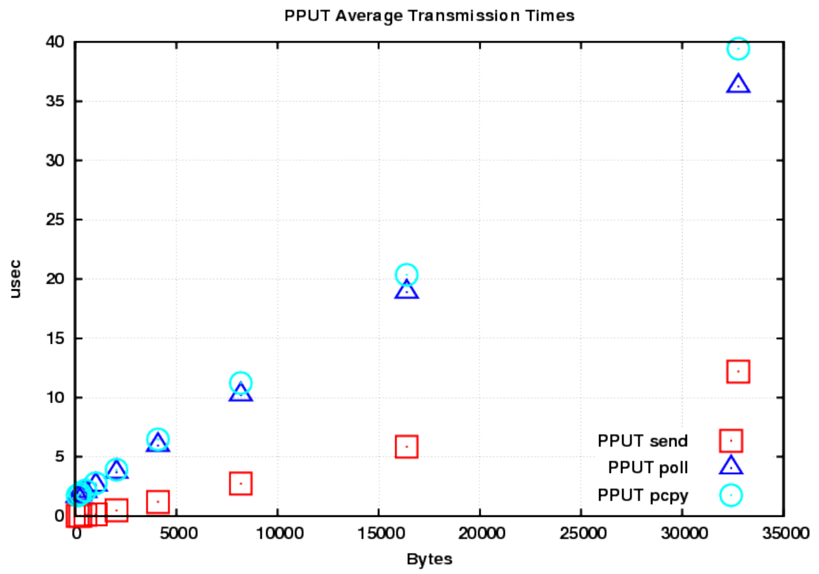

Figure 5.19: *CPU-to-CPU transmission times measurement of PPUT method, the red spots are the time to send data from CPU registers to PCIe, the blue ones include the time to send data over TNWs and DMA time to deliver to system memory, the light-blue include also the time to copy data from DMA buffer to USR buffer.*

5.6 Transmission Test on the AuroraScience Machine

In this section will be shown the transmission test done on the AuroraScience machine installed in Trento, reporting the measures of transmission time and bandwidth on up to four links contemporary. The test has been limited to four links out of six of the Torus Network because, in the days I run the test, the connections on the Z directions of the AuroraScience machine were not yet available. This test has been done using the Network-Processor-firmware marked as "Production" that implements the TXFIFO as a queue and not as a re-order buffer, see section 5.3.

The hardware setup for the test has been the following:

- Nodecard revision: E
- CPU: X5680 @ 3.33GHz
- Nodecards: anode248, anode247, anode240
- Firmware revision: ftnww-0410-41A0-E

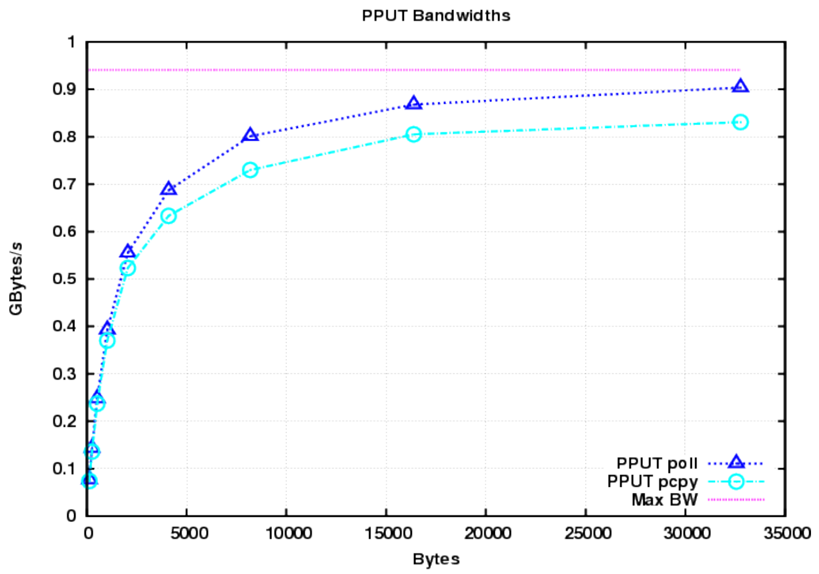

Figure 5.20: *CPU-to-CPU bandwidth measurement of PPUT method, the blue line shows the bandwidth achievable without considering the time to copy data from DMA buffer to USR buffer, that time is considered in the light-blue one. The purple line at 0.941 GBytes/s indicates the max usable bandwidth of the TNW link.*

This test is based on the communication concept called *"ping-pong"*, the basic idea is that two entities, called INITIATOR and REPEATER, exchange a message; INITIATOR sends a message to the REPEATER that sends it back, see figure 5.23. Calculating the *"round-trip time"* (the time the message takes to be sent and come-back), is possible to know the *"transmission time"*, that is with good approximation the half of the round-trip. Also known the message-size I can calculate the *"bandwidth"* as the message-size divided by the transmission time.

Replicating the ping-pong at the same time among all the links that I want to test, I can calculate the *"aggregate bandwidth"* that is the total amount of data sent among all links divided by the transmission time.

I run the ping-pong test using one, two, three and four links at the same time, sending messages of variable length, from 128 Bytes up to 256 KBytes, iterating each transmission thousand times to limit the fluctuations. The resulting transmission times, calculated by the INITIATOR, is then used to calculate the aggregate bandwidth.

The test on one link is done running one thread on the INITIATOR node that communicates with another thread on the REPEATER node, both sending mes-

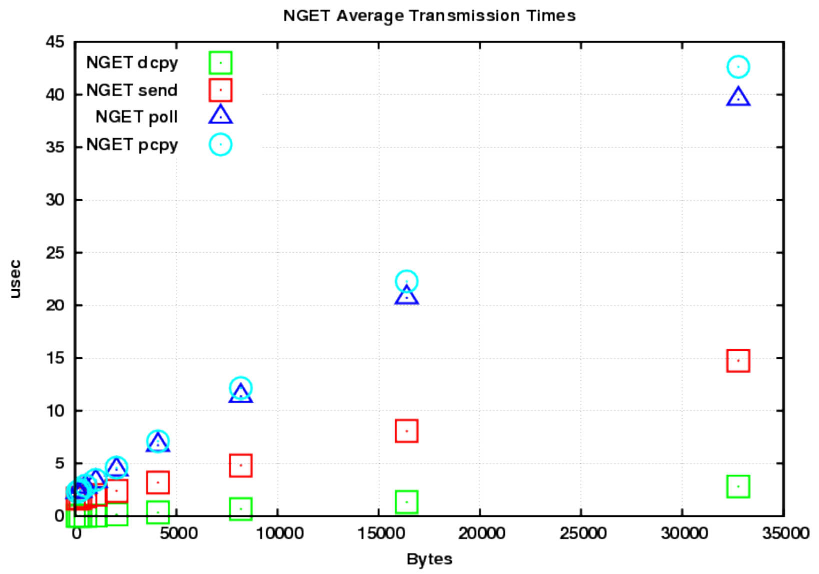

Figure 5.21: *CPU-to-CPU bandwidth measurement of NGET method, the green line is the time to copy data from USR buffer to DMA buffer, the red lines includes time to send data from CPU registers to PCIe, the blue one includes the time to send data over TNWs and DMA time to deliver them to system memory, the light-blue includes also the time to copy data from DMA buffer to USR buffer.*

sages on X+ and receiving on X-, for this reason the REPEATER is called REP_X; refer to figure 5.24, red path.

The test on two links is done involving three nodes; the INITIATOR node instantiates two threads, one communicating on X direction as the same as one-link-test, while the other communicating on Y direction with a different node respect to REP_X; the communications on Y occur sending messages on Y+ and receiving on Y-, for this reason the REPEATER is called REP_Y; refer to figure 5.24, red and green paths.

Consequently the tests on three and four links are done instantiating respectively three and four INITIATOR-threads on the same node, communicating with the peer thread on one of the repeater. In the three-links case the REP_X instantiates two REPEATER-threads, one receiving on X- and sending-back on X+, while the other is receiving on X+ and sending-back on X-; REP_Y instantiates one REPEATER-thread receiving on Y+ and sending-back on Y-; refer to figure 5.24, red, green and blue paths. In the four-links case the REP_X instantiates two REPEATER-threads, one receiving on X- and sending-back on X+, while

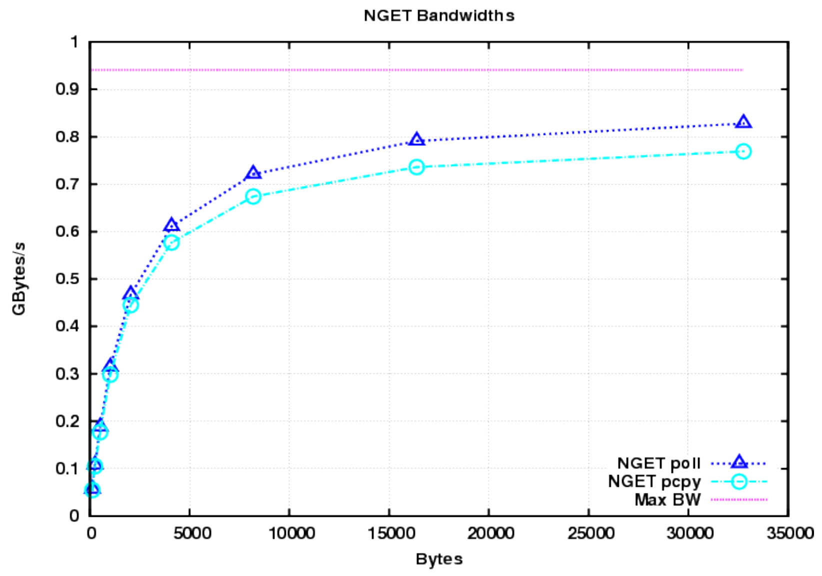

Figure 5.22: *CPU-to-CPU bandwidth measurement of NGET method, the blue line shows the bandwidth achievable without considering the time to copy data from DMA buffer to USR buffer, that time is considered in the light-blue one. The purple line at 0.941 GBytes/s indicates the max usable bandwidth of the TNW link.*

the other is receiving on X+ and sending-back on X-; REP_Y instantiates also two REPEATER-threads, one receiving on Y+ and sending-back on Y-. while the other is receiving on Y+ and sending-back on Y-; refer to figure 5.24, red, green, blue and magenta paths.

The configuration of the nodes for this test has been the following:

- anode248 as INITIATOR (INIT)
- anode247 as REPEATER in X (REP_X)
- anode240 as REPEATER in Y (REP_Y)

In the cases where more threads communicate over different links at the same time:

1. each thread detects his own transmission time that is, at least little variations, the same for all the threads
2. the communications are fully overlapped

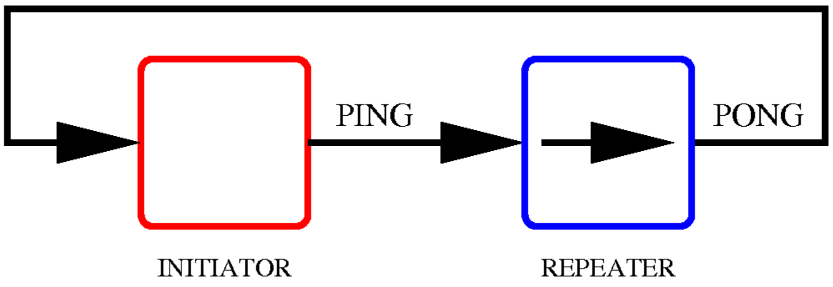

Figure 5.23: *Ping-Pong: INITIATOR sends a message to the REPEATER that sends it back.*

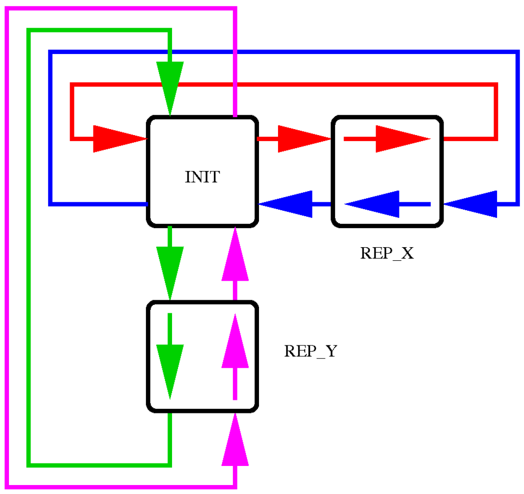

Figure 5.24: *Conceptual configuration for the tests on the four links.*

considering point 1 I calculate the minimum and the maximum transmission time per message-length.

The bandwidth on each link is calculated as the "total amount of data sent" on a link divided by the "transmission time".

In this test the total amount of data sent (totData) is calculated as the double of the length of the message sent (round-trip = 2 * msgLen), multiplied by the number of iterations done (nIter) and also multiplied by the number of active links (nLinks):

totData = 2 * msgLen * nIter * nLink

while the transmission time, due to point 2, can be considered in a conservative manner as the maximum of the transmission times measured by all INITIATOR-threads.

Table 5.11 shows the minimum and the maximum transmission times per message-length, subdivided by number of active links.

msg len KB	1 link	2 links		3 links		4 links	
		min	max	min	max	min	max
0.12	1.69	1.72	1.74	1.75	1.82	1.78	1.83
0.25	2.04	2.10	2.12	2.14	2.16	2.16	2.17
0.50	2.53	2.63	2.64	2.67	2.69	2.69	2.72
1.00	3.68	3.67	3.69	3.76	4.26	4.11	4.18
2.00	5.58	5.72	5.77	5.86	6.01	7.48	7.53
4.00	9.45	9.86	9.87	10.88	11.39	12.14	12.18
8.00	17.36	17.56	18.09	18.96	19.61	20.77	20.86
16.00	33.82	33.99	35.15	34.63	37.15	37.42	37.95
32.00	68.32	69.08	69.12	68.92	72.22	71.40	74.24
64.00	131.03	131.96	136.25	134.82	142.71	143.62	144.12
128.00	271.45	265.48	273.80	269.66	285.79	286.48	288.06
256.00	528.60	531.75	547.92	540.66	572.25	568.43	568.99

Table 5.11: *Transmission times in micro-seconds (μsec) for one, two, three and four links active at the same time.*

Plots of the max transmission time are fitted by the following lines:

- 1-link : $f(x) = 1.45 + 2.07x$
- 2-links: $f(x) = 1.19 + 2.13x$
- 3-links: $f(x) = 1.53 + 2.23x$
- 4-links: $f(x) = 2.52 + 2.22x$

Table 5.12 shows the aggregate bandwidths per message-length, subdivided by number of active links. Remember that the aggregate bandwidth is calculated conservatively using the maximum transmission time.

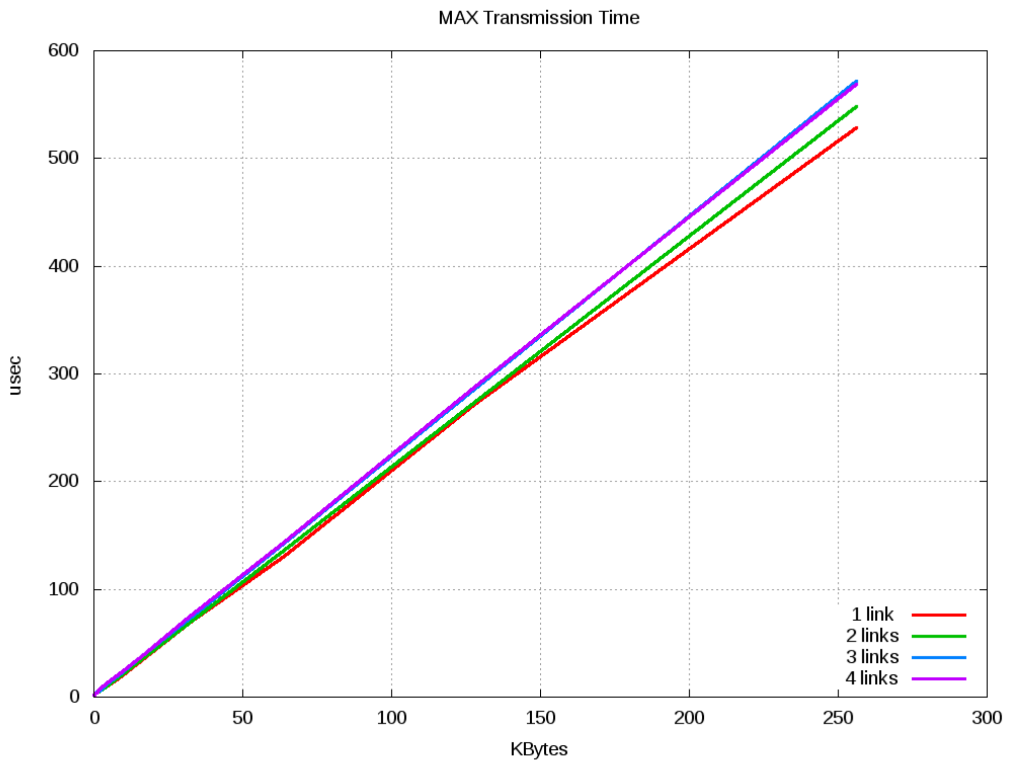

Figure 5.25: *Max Transmission time in micro-seconds (µsec) for one, two, three and four links active at the same time.*

Scalability over multi-nodes

msg len KB	1 link	2 links	3 links	4 links
0.12	0.08	0.15	0.21	0.28
0.25	0.13	0.24	0.36	0.47
0.50	0.20	0.39	0.57	0.75
1.00	0.28	0.55	0.72	0.98
2.00	0.37	0.71	1.02	1.09
4.00	0.43	0.83	1.08	1.35
8.00	0.47	0.91	1.25	1.57
16.00	0.48	0.93	1.32	1.73
32.00	0.48	0.95	1.36	1.77
64.00	0.50	0.96	1.38	1.82
128.00	0.48	0.96	1.38	1.82
256.00	0.50	0.96	1.37	1.84

Table 5.12: *Aggregate Bandwidth in GB/s for one, two, three and four links active at the same time.*

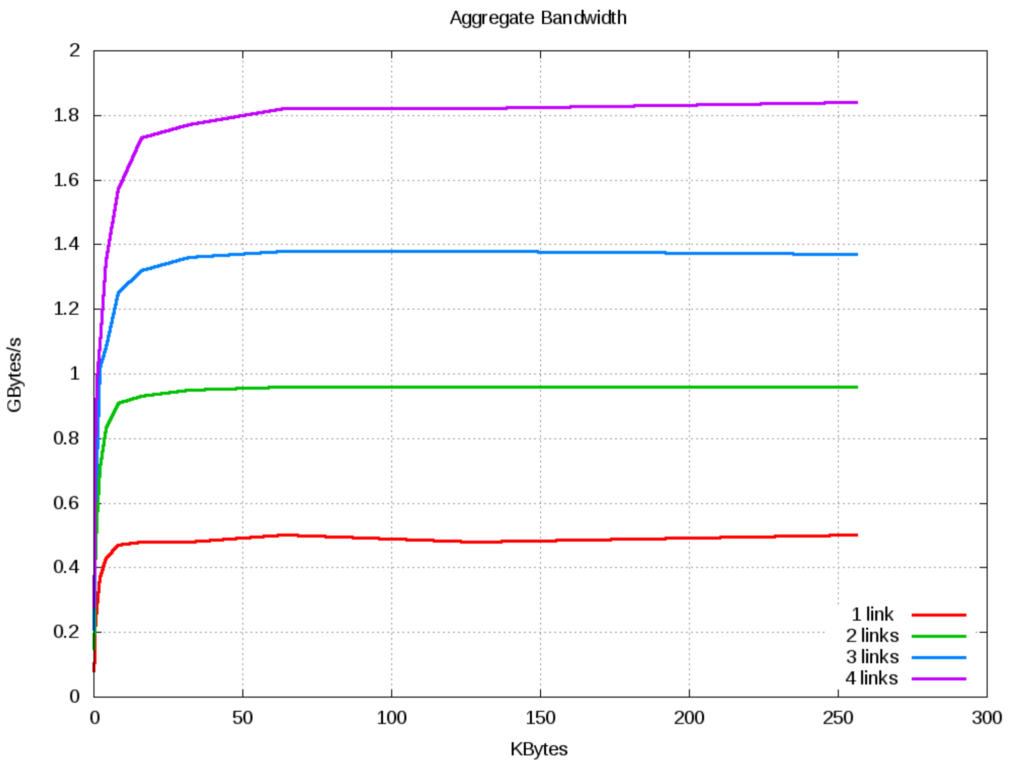

Figure 5.26: *Aggregate Bandwidth for one, two, three and four links active at the same time.*

Chapter 6

Conclusions

In this thesis I have discussed the design and the implementation of an FPGA-Based Network Processor for scientific computing, specifically designed for applications in theoretical physics such as Lattice Quantum ChromoDinamycs (LQCD) and fluid-dynamics based on the Lattice Boltzmann (LBM) approach; state-of-the-art programs in this (and other similar) applications have a large degree of available parallelism, that can be easily exploited on massively parallel systems, provided the underlying communication network has not only high-bandwidth but also low-latency.

The overall architecture of the Network Processor (NWP) has been divided into three independent layers: the physical layer (PHY),The "Torus Network" layer (TNW) and the "Input Output Controller" layer (IOC). This modularity allows to implement separately each of the layers, so the NWP can be more easily tailored to specific processor architectures or signalling technologies; equally important, each of the layers can be independently.

Building on previous work, I have designed in details, built and tested in hardware, firmware and software an implementation of this structure, tailored for the most recent families of multi-core processors manufactured by Intel. The physical layer (PHY) has been implemented using a commercial silicon-component, the PMC-Sierra PM8354, a serializer/de-serializer device (SERDES) who manages the electrical signaling of the link connecting adjacent Network Processors. I tested the reliability of the physical link both on to recent experimental massively parallel machines (QPACE and AuroraScience) developed by the LQCD community. The "Torus Network" layer (TNW) is the main logical network engine, providing access to the three-dimensional toroidal network. The present version of TNW is based on Field Programmable Gate Array (FPGA) technology, to reduce development costs and time, and is fully written in a VHDL code.

This layer is based on the "Two-Sided Communication" model, in which both entities at the end-points of a link cooperate to carry-out the communications; each SEND operation issued by one node, must be matched by an equivalent RECEIVE request at the other side of the link, otherwise the communication is not possible; this model – while lacking full generality – allows a significant reduction in communication latency in all cases in which the pattern of communication is fully known to both communication partners.

The TNW contains the buffers to send and receive data as well as a custom protocol to grant reliable communications over an unreliable physical link. TNW has been designed to work with multi-core architectures, so it implements the virtual-channels mechanism to allow independent communications among pairs of cores at the end-points of the link, sharing the same physical connection. I extended or re-engineered some VHDL modules of this layer, starting from the implementation that had been developed by the QPACE groups and extensively tested the functionalities and the reliability of this layer.

The third element of the NWP, is "Input Output Controller" layer (IOC). This is the interface between the CPU I/O sub-system and the Torus Network. One of the key contribution of the present work is the design, implementation and test of a version of the IOC able to manage the PCI Express transaction-methods adopted by the Intel architectures, specifically the "Programmed Input Output" (PIO) and the "Direct Memory Access" (DMA). The former method requires that the CPU actively moves data from memory to devices; this prevents the CPU from performing other useful work, while the IO transaction is in progress. DMA on the other hand let the communication devices to independently retrieve data from memory after being instructed by the CPU; in this case the CPU can perform other tasks without being involved in data movement. A key feature of this IOC design is the presence of a a re-order buffer that allows to take advantage of the "Write Combining" capabilities of the Intel architecture; this increase substantially bandwidth on the link.

On the software side, I developed and test a device driver for the Linux operating system to access the NWP device, as well as a system library to access the network device from user-applications in an efficient way and with small overhead.

I used these system-level software components to develop a suite of micro-benchmarks and application-benchmarks and measured accurately latency and bandwidth of this communication link.

This thesis demonstrates the feasibility of a network infrastructure that saturates the maximum bandwidth of the I/O sub-systems available on recent CPUs, and reduces communication latencies to values very close to those needed by the processor to move data across the chip boundary.

I have experimented both at the firmware and software layer, trying several options allowed by the Intel architecture to move data between memory and devices. The combination of “Write Combining” memory-setting, “Non-Temporal” instructions and re-mapping of kernel-buffers in user-space has given the best results in terms of both bandwidth and latency. These results are close to the architectural limits of state-of-the-art CPUs, that are optimized for bandwidth and throughput, but are far from being the optimal solution for latency-bounded applications. Today the latency problem is clearly the most critical efficiency bottleneck in large multi-processor-systems, and one is led to believe that communication latency is not among the top priorities of silicon companies; any improvement in this parameter that may become available in future processors will have a strong impact on the structure of networks and on the overall performance of large parallel systems.

Future developments on NWP will lead to a more general layout, making its layers fully autonomous of each other, and not dependent on a specific technology. The FTNW project plans to work exactly in this direction, developing a a communication core not directly dependent on any specific technology, and easily portable onto several computing architectures. A future development of the Network Processor can be the interconnection on a 3D-torus topology of computing-accelerators such as the “Graphics Processing Units” (GPUs) or the emerging “Many Integrated Core” (MIC). A further development to improve the NWP capabilities will be the support to “General Routing”, to allow communications between each node of the network, over nearest-neighbor links.

Appendix A

NWP Registers Mapping

A.1 RX-Link Registers

0x0<l>00 RX_EXC RX exceptions			
[00]	RW	00000001	RX_FAULT_P from PHY
[01]	RW	00000002	RX_FAULT_R from PHY
[02]	RW	00000004	INTB from PHY
[03]	RW	00000008	Timeout
[04]	RW	00000010	Credit Fifo overrun, write while full
[05]	RW	00000020	Credit size zero
[06]	RW	00000040	Credit base address written while used
[07]	RW	00000080	Notify base address written while used
[08]	RW	00000100	Data received while offline
[09]	RW	00000200	POC protocol error: unexpected done
[09]	RW	00000400	POC protocol error: unexpected first
[12]	RW	00001000	Invalid rCba arithmetics: carry into bit 32
[13]	RW	00002000	Invalid rNba arithmetics: non-zero LSB in rNba
[16]	RW	00010000	Last NAK: failed CRC (wErrC)
[17]	RW	00020000	Last NAK: dirty data decoding (wErrD)
[18]	RW	00040000	Last NAK: missed header, pending commit (wErrH)
[19]	RW	00080000	Last NAK: failed write, data buffer full (wErrW)
[20]	RW	00100000	Last NAK: timeout (wErrT)

Default: 0x00000003

0x0<l>01 RX_EXEN RX exceptions enable (RW)
Analog bits RX_EXC

Default: 0x000008FF

0x0<l>02 RX_CMD		
[07:00] RW	000000ff	dirty decode ACK
[15:08] RW	0000ff00	dirty decode NAK
[23:16] RW	00ff0000	dirty decode RESTART
[31:24] RW	ff000000	dirty decode IDLE

Default: 0x00000000

0x0<l>03 RX_DATA		
[15:00] RW	0000ffff	packets with CRC errors
[23:16] RW	00ff0000	packets with dirty data
[31:24] RW	ff000000	packets with timeouts

Default: 0x00000000

0x0<l>04 RX_FLOW		
[15:00] RW	0000ffff	packets rejects for header commit
[31:16] RW	ffff0000	packets rejected for failed data write

Default: 0x00000000

0x0<l>05 RX_RXH		
[07:00] RW	000000ff	counter of cycles with RXH(0)
[15:08] RW	0000ff00	counter of cycles with RXH(1)
[23:16] RW	00ff0000	counter of cycles with RXH(2)
[31:24] RW	ff000000	counter of cycles with RXH(3)

Default: 0x00000000

0x0<l>06 Rx Fifo Status		
[00] RO	00000001	Packet Buffer Not Empty
[01] RO	00000002	Packet Buffer $\geq$ half filled
[02] RO	00000004	Packet Buffer Full
[15:08] RO	0000ff00	Descriptor Fifos Not Empty
[23:16] RO	00ff0000	Credit Fifo Not Empty
[31:24] RO	ff000000	Credit Fifos Full

Default: 0x00000000

0x0<l>07 Rx Control		
[00] RW	00000001	Credit Fifo Hold
[01] RW	00000002	Credit Fifo Reset

Default: 0x00000000

0x0<l>1<c> Notify Base Address		
[00] RW	ffffffff	Notify Base Address B_n of channel <c>

Default: 0x0fffffff

0x0<l>2<c> Credit Fifo channel <c>			
[15:00]	WO	0000ffff	Credit Address C_d of channel <c>
[23:16]	WO	0fff0000	Credit Size (aligned to 128B) of channel <c>
[31:24]	WO	f0000000	Notify Address C_n

Default: empty

Note: this register is WO

0x0<l>2<c> Credit Base Address			
[00]	RW	ffffffff	Credit Base Address B_d of Channel <c>

Default: 0x0fffffff

A.2 TX-Link Registers

REG0 Tx Exceptions Status			
[00]	RW	00000001	TX_FAULT_P or TX_FAULTR from PHY
[01]	RW	00000002	NAK counter overflow
[02]	RW	00000004	restart counter overflow
[03]	RW	00000008	feedback mismatch
[04]	RW	00000010	invalid header (missing "first")
[05]	RW	00000020	txFifo written while full
[06]	RW	00000040	txFifo empty while packet send
[07]	RW	00000080	txBuffer full
[08]	RW	00000100	PIC data write while offline
[09]	RW	00000200	fbIn accumulation
[10]	RW	00000400	fbOut accumulation
[23:16]	RW	00FF0000	last restart counter
[28]	RO	10000000	txFifo not empty
[29]	RO	20000000	txFifo not almost empty ($\geq$ 8 pckts)
[30]	RO	40000000	txFifo almost full

Default: 0x00000000

REG1 TX exceptions enable			
[08:00]	RW	000001FF	enable of exceptions
[23:16]	RW	00FF0000	restart limit

Default: 0x00ff01ff

REG2 TX_CTRL			
[00]	RW	00000001	txLink offline
[01]	RW	00000002	txLink reset
[02]	RW	00000004	txLink reset Done
[04]	RW	00000010	rxLink offline
[05]	RW	00000020	rxLink reset
[06]	RW	00000040	rxLink reset Done
[08]	RW	00000100	rxClk DCM reset
[09]	RW	00000200	rxClk DCM reset Done
[10]	RW	00000400	rxClk DCM locked
[12]	RW	00001000	PHY reset
[13]	RW	00002000	PHY reset Done
[20]	RO	00100000	txLink FSM not IDLE
[21]	RO	00200000	txFifo not empty
[22]	RO	00400000	txBuffer not empty
[24]	RO	01000000	rxLink FSM not IDLE
[25]	RO	02000000	rxBuffer FSM not IDLE
[26]	RO	04000000	packet buffer not empty
[27]	RO	08000000	descriptor fifo not empty
[28]	RO	10000000	credit fifo not empty

Default: 0x00000011

REG3 TX_MDIO_R			
[15:00]	RW	0000FFFF	Data (read)
[28:16]	RW	1FFF0000	Address bits [12:0]
[29]	RO	20000000	Busy
[31:30]	RW	C0000000	Address bits [15:14]

Default: 0x00000000

REG4 TX_MDIO_W			
[15:00]	RW	0000FFFF	Data (to write)
[28:16]	RW	1FFF0000	Address bits [12:0]
[29]	RO	20000000	Busy
[31:30]	RW	C0000000	Address bits [15:14]

Default: 0x00000000

REG5 TX_CNT_NAK			
[31:00]	RW	FFFFFFFF	Total count of received NAK

Default: 0x00000000

REG6 TX_CNT_RESEND			
[31:00]	RW	FFFFFFFF	Total count of entries into resend mode

Default: 0x00000000

A.3 TB Registers

REG0 TXCTRL				
[00]	RW	00000001	DGEN_EN	endless data-gen enable
[01]	RW	00000002	DGEN_FLAG	data-gen flag: 0=RND, 1=SEQ
[02]	RW	00000004	DGEN_CLEAR	data-gen clear: 1= re-init data generator
[04]	RW	00000010	PICSWITCH	switch between dataGen and PIC
[08]	RW	00000100	BACATXD	change value of TXD bus to X"12345678"
[09]	RW	00000200	BACATXC	force value of TXC bus to X"A"
[14:12]	RW	00007000	CHADDR	channel address
[29:24]	RW	1F000000	ONESHOTCNT	one-shot data-gen counter and enable

DEFAULT VALUE = 0x00000000

Notes:

- writing the one-shot counter triggers send of the number of specified packets
- data sequence is reset after the end of one one-shot run and as the endless enable-bit is de-asserted.

REG 1 RXCTRL				
[00]	RW	00000001	CHKEN	data-check enable
[01]	RW	00000002	CHKFLAG	data-check flag : 0=RND, 1=SEQ
[04]	RW	00000010	POCSWITCH	switch between dataChk and POC

DEFAULT VALUE = 0x00000000

Note: CHKEN = '0' clear register RXERRCNT

REG2 TXFRAMECNT				
[31:00]	RO	FFFFFFFF	TXFRAMECNT	TX Frame Counter

DEFAULT VALUE = XXXXXXXX

REG3 RXERRCNT				
[31:00]	RO	FFFFFFFF	ERRCNT	RX Data-Check Error Counter

DEFAULT VALUE = XXXXXXXX

Note: Register RXERRCNT is valid only if CHKEN = '1'

REG4 RXFRAMECNT				
[31:00]	RO	FFFFFFFF	RXFRAMECNT	RX Frame Counter

DEFAULT VALUE = XXXXXXXX

REG 5 LASTDATACHECKED				
[31:00]	RO	FFFFFFFF	LASTDATACHK	LAST DATA CHECKED

DEFAULT VALUE = ??????

A.4 BAR0 and BAR1: Tx Fifos Address Space

Virtual channels (VC) are mapped on a 64-bit prefetchable base address defined as combination of BAR0 and BAR1 of the PCIe macro, and referred as BAR1 by software.

Currently each packet (128B) is written to BAR0 using following memory address format:

```
lllc.ccoo.o000.0000
```

where `l` are the bit for the link identifier, and `c` are the bits for the channel identifier, and `o` is the offset of the packet aligned to 128B.

Each VC has an address space of 256KB corresponding to 2048 packets (256KB $=$ $2048 \times$ 128B).

The PCI address of each VC is computed as:

```
(l « 21) + (c « 18)
```

where `l` is the link identifier, `c` is the channel identifier.

Link 0	
VC0	BAR0 + 0x00.0000
VC1	BAR0 + 0x04.0000
VC2	BAR0 + 0x08.0000
VC3	BAR0 + 0x0C.0000
VC4	BAR0 + 0x10.0000
VC5	BAR0 + 0x14.0000
VC6	BAR0 + 0x18.0000
VC7	BAR0 + 0x1C.0000
Link 1	
VC0	BAR0 + 0x20.0000
VC1	BAR0 + 0x24.0000
VC2	BAR0 + 0x28.0000
VC3	BAR0 + 0x2C.0000
VC4	BAR0 + 0x20.0000
VC5	BAR0 + 0x24.0000
VC6	BAR0 + 0x28.0000
VC7	BAR0 + 0x2C.0000
⋮	
Link 5	
VC0	BAR0 + 0xE0.0000
VC1	BAR0 + 0xE4.0000
VC2	BAR0 + 0xE8.0000
VC3	BAR0 + 0xEC.0000
VC4	BAR0 + 0xE0.0000
VC5	BAR0 + 0xE4.0000
VC6	BAR0 + 0xe8.0000
VC7	BAR0 + 0xeC.0000

Table A.1: *Mapping of BAR0*

A.5 BAR2 Address Space

A.5.1 TNW Register Address Space

The BAR2 manages LINKs registers, each link has a set of configuration, status and debug registers. The link number is noted as <l> in the below table. Registers are 32-bit wide, but since they are mapped on 128-bit PCI address-space, the address should be 128-bit aligned. All registers are mapped on 32-bit non-prefetchable physical BAR2.

A.5.2 IOC Register Address Space

The BAR2 manages IOC registers, they are mapped on 128-bit PCI address-space so the address should be 128-bit aligned. All registers are mapped on 32-bit non-prefetchable physical BAR2.

A.5.3 TXFIFO Counters Registers

The registers to manage the TXFIFO counters (TFCNT) are mapped on BAR2, the mapping of these registers is:

0x1001 TFCNT main memory address register		
[48:00] RW	FFFFFFFFFFFF	Counters memory address

DEFAULT VALUE = 0000000000000000

0x1002 TFCNT reset register			
[31:00] WO	FFFFFFFF	TFCNTRST	Counters reset

DEFAULT VALUE = NONE

A.5.4 DMA Request

Issues a DMA transaction (NGET).

0x1005 DMA Request			
[10:00] RO	0000007FF	DMALENGTH	DMA length in units of 32bits (DW)
[27:11] RO	00FFFF800	DMAPICATTR	Attributes for PIC header
[30:28] RO	070000000	DMANOTIDX	DMA Notify index
[33:31] RO	380000000	DMABUFOFF	DMA Buffer offset in units of 4KB

DEFAULT VALUE = NONE

A.5.5 DMA Buffer BAR

DMA buffer base address register.

0x1006 DMA buffer BAR			
[47:00] RW	FFFFFFFFFFFF	DMABUFADDR	Base Memory Address to be sent

DEFAULT VALUE = NONE

A.5.6 DMA Notify BAR

DMA notifies base address register.

0x1006 DMA notify BAR			
[47:00] RW	FFFFFFFFFFFF	DMANOTADDR	Base Memory Address for notifies

DEFAULT VALUE = NONE

Register Offset (16B unit)	Description	PCI Address (16B aligned)
	Rx Registers	
`0x0<l>00`	Rx Exceptions and Errors	`(l « 12) + 0x000`
`0x0<l>01`	Rx Exception Enable	`(l « 12) + 0x010`
`0x0<l>02`	Rx Commands Error Count	`(l « 12) + 0x020`
`0x0<l>03`	Rx Data Error Count	`(l « 12) + 0x030`
`0x0<l>04`	Rx Flow Error Count	`(l « 12) + 0x030`
`0x0<l>05`	Rx PHY Error Count	`(l « 12) + 0x050`
`0x0<l>06`	Rx Fifo Status	`(l « 12) + 0x060`
`0x0<l>07`	Rx Control	`(l « 12) + 0x070`
`0x0<l>1<c>`	Notify Base Address	`(l « 12) + 0x100+<c«4>`
`0x0<l>2<c>`	Credit Fifos	`(l « 12) + 0x200+<c«4>`
`0x0<l>3<c>`	Credit Base Address	`(l « 12) + 0x300+<c«4>`
	Tx Registers	
`0x0<l>40`	Tx Exception Status	`(l « 12) + 0x400`
`0x0<l>41`	Tx Exception Enable	`(l « 12) + 0x410`
`0x0<l>42`	Tx Control and Status	`(l « 12) + 0x420`
`0x0<l>43`	MDIO Read Access	`(l « 12) + 0x430`
`0x0<l>44`	MDIO Write Access	`(l « 12) + 0x440`
`0x0<l>45`	Tx NAK Counter	`(l « 12) + 0x450`
`0x0<l>46`	Tx Resend Counter	`(l « 12) + 0x460`
`0x0<l>4f`	TNW Revision	`(l « 12) + 0x4f0`
	Tb Registers	
`0x0<l>50`	TxCtrl	`(l « 12) + 0x500`
`0x0<l>51`	RxCtrl	`(l « 12) + 0x510`
`0x0<l>52`	TxFrameCnt	`(l « 12) + 0x520`
`0x0<l>53`	RxErrCnt	`(l « 12) + 0x530`
`0x0<l>54`	RxFrameCnt	`(l « 12) + 0x540`
`0x0<l>55`	LastDataChecked	`(l « 12) + 0x550`
`0x0<l>56`	UNUSED	`(l « 12) + 0x560`
`0x0<l>57`	UNUSED	`(l « 12) + 0x570`

Table A.2: *LINKs Register Mapping on BAR2. Addresses are 16B aligned, and in unit of byte.*

Register Offset (16B unit)	Description	PCI Address (16B aligned)
IOC Registers		
`0x1000`	Flash Control Register	`0x1.0000`
`0x1001`	TXFIFO cnt main memory address	`0x1.0010`
`0x1002`	TXFIFO cnt reset	`0x1.0020`
`0x1003`	Slot Address	`0x1.0030`
`0x1004`	Links Exceptions	`0x1.0040`
`0x1005`	DMA Request	`0x1.0050`
`0x1006`	DMA Buffer BAR	`0x1.0060`
`0x1007`	DMA Notify BAR	`0x1.0070`

Table A.3: *IOC Register Mapping on BAR2. Addresses are 16B aligned, and in unit of byte.*

Appendix B

libftnw Functions Summary

Summary of the most important functions exported to the applications by **libftnw**.

B.1 Device Initialization and Release

`int ftnwOpen (void);`
Opens the NWP file descriptor. Returns the file descriptor.

`int ftnwClose(void);`
Closes the NWP file descriptor. Always Returns 0.

`int ftnwInit (void);`
Opens the NWP file descriptor and re-maps in user-space all the buffers allocated in kernel-space. Returns 0 in case of success, -1 otherwise.

`int ftnwFinalize (void);`
Closes the NWP file descriptor. Always returns 0.

B.2 Send

```
int ftnwSend (uint lid, uint vcid, void * txbuf, uint txoff,
uint msglen);
```
Implements the SEND operation using the PPUT transaction model.
Parameters:

- lid : link ID [XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS]
- vcid : virtual-channel ID [0-7]
- txbuf : source buffer
- txoff : offset inside the source buffer, in unit of 128 Bytes
- msglen : message size, in unit of 128 Bytes

`int ftnwNgetSend (uint lid, uint vcid, void * txbuf, uint txoff, uint nid, uint msglen);`
Implements the SEND operation using the NGET transaction model.
Parameters:

- lid : link ID [XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS]
- vcid : virtual-channel ID [0-7]
- txbuf : source buffer
- txoff : offset inside the source buffer, in unit of 128 Bytes
- nid : notify index
- msglen : message size, in unit of 128 Bytes

`int ftnwNgetTest (uint nid);`
Tests the notify-index nid to discover if the NGET SEND has been carried-out. Non-blocking function, immediately returns in both cases if nid has been set or not.

`int ftnwNgetPoll (uint nid);`
Blocking version of ftnwNgetTest, it polls the notify-index nid until it is not set, meaning that the NGET SEND has been carried-out.

B.3 Receive

```
int ftnwCredit (uint lid, uint vcid, uint rxoff, uint msglen,
uint nid);
```
Parameters:

- lid : link ID [XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS]
- vcid : virtual-channel ID [0-7]
- rxoff : offset inside the destination buffer, in unit of 128 Bytes
- msglen : message size, in unit of 128 Bytes
- nid : notify index

```
int ftnwTest (uint lid, uint vcid, uint rxoff, uint msglen,
void * rxbuf, uint nid);
```
Tests the notify-index nid to discover if the message has been received. Non-blocking function, immediately returns in both cases if nid has been set or not. If nid is set moves data from kernel-space to user-space.
Parameters:

- lid : link ID [XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS]
- vcid : virtual-channel ID [0-7]
- rxoff : offset inside the buffer in kernel-space, in unit of 128 Bytes
- msglen : message size, in unit of 128 Bytes
- rxbuf : destination buffer in user-space
- nid : notify index

```
int ftnwPoll (uint lid, uint vcid, uint rxoff, uint msglen,
void * rxbuf, uint nid);
```
Blocking version of ftnwTest, it polls the notify-index nid until it is not set.

B.4 Register Access

`int ftnwPokeReg (uint regaddr, uint regval);`
Parameters:

- regaddr : register address to write
- regval : 32-bits value to write

`int ftnwPokeReg64 (uint regaddr, uint regval);`
Parameters:

- regaddr : register address to write
- regval : 64-bits value to write

`int ftnwPeekReg (uint regaddr, uint * regval);`
Parameters:

- regaddr : register address to read
- regval : 32-bits value read

Bibliography

[1] N.R. Adiga et al., *Blue Gene/L torus interconnection network*, March/May 2005, IBM J. RES. & DEV. VOL. 49 NO. 2/3

[2] Mindshare, Inc., R. Budruk et al., *PCI Express System Architecture*, 2003, Addison-Wesley

[3] 2nd Generation Intel® Core Processor Family Mobile, *Datasheet - Volume 1*, "Supporting Intel Core i7 Mobile Extreme Edition Processor Series and Intel Core i5 and i7 Mobile Processor Series", February 2011, `http://download.intel.com/design/processor/datashts/324692.pdf`

[4] 2nd Generation Intel® Core Processor Family Desktop, *Datasheet, Volume 1*, "Supporting Intel Core i7, i5 and i3 Desktop Processor Series", February 2011, `http://download.intel.com/design/processor/datashts/324641.pdf`

[5] Intel® 6 Series Chipset, *Datasheet*, February 2011, `http://www.intel.com/Assets/PDF/datasheet/324645.pdf`

[6] Intel Embedded Site, `http://edc.intel.com`

[7] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, September 2010

[8] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*, September 2010

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, September 2010

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, September 2010

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, September 2010

[12] *Intel X58 Express Chipset, Datasheet*, November 2009

[13] *Intel Processor Identification and the CPUID Instruction, Application Note 485*, August 2009

[14] *Intel Write Combining Implemetation Guidelines*, November 1998

[15] *Intel C++ Intrinsics Reference*

[16] J. Coleman, P. Taylor, *Hardware Level IO Benchmarking of PCI Express*, Intel White Paper, December, 2008

[17] Mellanox Site, ConnectX InfiniBand Adapter Devices, `http://www.mellanox.com/content/pages.php?pg=products_dyn&product_family=3&menu_section=32`

[18] Mellanox Site, ConnectX InfiniBand Adapter Devices, Product Brief, `http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX_Silicon.pdf`

[19] QLogic Site, InfiniBand Adapters, Product Series, `http://www.qlogic.com/Products/adapters/Pages/InfiniBandAdapters.aspx`

[20] P.A. Boyle et al., *Overview of the QCDSP and QCDOC computers*, March/-May 2005, IBM J. RES. & DEV. VOL. 49 NO. 351

[21] F. Belletti et al., *Computing for LQCD: apeNEXT*, 2006, Computing in Science & Engineering 8

[22] G. Bilardi et al., *The potential of On-Chip Multiprocessing for QCD Machines*, 2005, Springer Lecture Notes in Computer Science 3769, 386

[23] J. Makino et al., *A 1.349 Tflops Simulation of Black Holes in a Galactic Center on GRAPE-6*, Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Article n. 43, (2000).

[24] F. Belletti et al., *Simulating an Ising spin-glass for 0.1 seconds with Janus Computing in Science and Engineering* in press, also arXiv:0710.3535v2, (2008).

[25] F. Belletti et al., *QCD on the Cell Broadband Engine*, 2007, LATTICE 2007, Proceedings of Science, arXiv:0710.2442

[26] G. Goldrian et al., *QPACE: Quantum Chromodynamics Parallel Computing on the Cell Broadband Engine*, 2008, Computing in Science & Engineering, Vol. 10, Issue 6

[27] H. Baier, M. Pivanti et al., *Status of the QPACE Project*, 9 October 2008, LATTICE 2008, Proceedings of Science, arXiv:0810.1559v1

[28] H. Baier, M. Pivanti et al., *QPACE - a QCD parallel computer based on Cell processor*, 24 December 2009, LATTICE 2009, Proceedings of Science, arXiv:0911.2174v3

[29] M. Pivanti, S.F. Schifano, H.Simma, *An FPGA-based Torus Communication Network*, 11 February 2011, LATTICE 2010, Proceedings of Science, arXiv:1102.2346v1

[30] G. Campobello, G. Patané, M. Russo, *Parallel CRC Realization*, October 2003, IEEE Transactions on Computers Vol. 52 Issue 10

[31] Clifford E. Cummings, *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*, “SNUG 2001 (Synopsys Users Group Conference, San Jose, CA 2001) User Papers”, March 2001, also available at : www.sunburst-design.com/papers

[32] Clifford E. Cummings, *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, “SNUG 2002 (Synopsys Users Group Conference, San Jose, CA 2002) User Papers”, March 2002, also available at : www.sunburst-design.com/papers

[33] Clifford E. Cummings, *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparison*, “SNUG 2002 (Synopsys Users Group Conference, San Jose, CA 2002) User Papers”, March 2002, also available at : www.sunburst-design.com/papers

[34] Mike Stein, *Crossing the abyss: asynchronous signals in a synchronous world*, July 24, 2003, also available at : www.edn.com/contents/images/310388.pdf

[35] Vijay A. Nebhrajani, *Asynchronous FIFO Architectures (Part 1)*, also available at : www.geocities.com/deepakgeorge2000/vlsi_book/Asynch1.pdf

[36] Vijay A. Nebhrajani, *Asynchronous FIFO Architectures (Part 2)*, also available at : www.geocities.com/deepakgeorge2000/vlsi_book/asynch_fifo2.pdf

[37] Eilahard Haseloff, *Metastable Response in 5-V Logic Circuits*, February 1997, also available at : focus.ti.com/lit/an/sdya006/sdya006.pdf

[38] Ran Ginosar, *Fourteen Ways to Fool Your Synchronizer*, “Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC’03)”, February 2003, also available at : www.ee.technion.ac.il/ ran/papers/Sync_Errors_Feb03.pdf

[39] Cadence Design Systems Inc, *Clock Domain Crossing*, “Closing the loop on clock domain functional implementation problems”, 2004, also available at : www.cadence.com/whitepapers/cdc_wp.pdf

[40] Chris Wellheuser, *Metastability Performance of Clocked FIFOs*, “First-In, First-Out Technology”, 1997, also available at : focus.ti.com/lit/an/scza004a/scza004a.pdf

[41] Frank Gray, *Pulse Code Communication*, Application November 13 1947 Serial No.785697, Patented March 17 1953 No.2632058, also available at : www.freepatentsonline.com/2632058.pdf

[42] J. Corbet, A. Rubini, G. Kroah-Hartman *Linux Device Drivers, Third Edition*, January 25, 2005, O’Reilly Media, Inc.

[43] W. R. Stevens, S. A. Rago *Advanced Programming in the UNIX Environment, Second Edition*, 2005, Addison-Wesley

[44] D. P. Bovet, M. Cesati, *Understanding the Linux Kernel, From I/O Ports to Process Management, First Edition*, October, 2000, O’Reilly Media, Inc.