UNIVERSITÀ DEGLI STUDI DI ROMA "SAPIENZA"
DEPARTMENT OF COMPUTER SCIENCE

Doctorate
Computer Science
Ciclo XXX

SOLUTIONS FOR THE OPTIMIZATION OF THE SOFTWARE INTERFACE ON AN FPGA-BASED NIC

Michele Martinelli

Advisor:
Alessandro Lonardo

Rome, 2017

Thesis Committee

Dott. Alessandro Lonardo (Advisor)
INFN, Sezione di Roma
Rome, Italy

Prof. Piero Vicini
Department of Physics
Sapienza University of Rome
and
INFN, Sezione di Roma
Rome, Italy

Prof. Alessandro Mei
Department of Computer Science
Sapienza University of Rome
Rome, Italy

External Reviewers

Prof. Mark Silberstein
Department of Electrical Engineering
Technion-Israel Institute of Technology
Haifa, Israel

Prof. Roberto Giorgi
Department of Information Engineering
University of Siena
Siena, Italy

Publications and Patents

Parts of the work presented in this thesis have been published in the following conference and journal papers. Furthermore, part of the work has been filed to be patented.

Submitted Journals:

- "Dynamic many-process applications on many-tile embedded systems and HPC clusters: The EURETILE programming environment and execution platforms", *P. S. Paolucci, A. Biagioni, L. G. Murillo, F. Rousseau, L. Schor, L. Tosoratto, I. Bacivarov, R. L. Buecs, C. Deschamps, A. El-Antably, R. Ammendola, N. Fournel, O. Frezza, R. Leupers, F. L. Cicero, A. Lonardo, M. Martinelli, E. Pastorelli, D. Rai, D. Rossetti, F. Simula, L. Thiele, P. Vicini, and J. H. Weinstock*

  Journal of Systems Architecture, vol. 69, pp. 29-53, 2016.

Conferences Participation:

- **TALK** "The next generation of Exascale-class systems: the exanest project", *R. Ammendola, A. Biagioni, P. Cretaro, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, F. Simula, P. Vicini, G. Taffoni, J. Navaridas, J. A. Pascual, N. Chrysos and M. Katevenis*

  Venue: *Euromicro Conference on Digital System Design - Special Session on European Projects in Digital Systems Design (EPDSD), Wien, Austria.*

  Date: august 30 – september 1 2017.

- **POSTER (BEST POSTER AWARD WINNER):** "GPU I/O persistent kernel for latency bound systems".

  M. Martinelli (on behalf of NaNet team).

  Venue: *International Symposium on High Performance Distributed Computing (HPDC2017), Washington DC, USA.*

  Date: june 27 – july 1 2017.

- **POSTER:** "NaNet: FPGA-based Network Interface Cards Implementing Real-time Data Transport for HEP Experiments".

R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, L. Pontisso, D. Rossetti, F. Simula, M. Sozzi, P. Cretaro, P. Vicini

Venue: *Real Time 2016, Padova, Italy.*

Date: june 5–10, 2016.

- **TALK** "GPU for triggering in HEP esperiments". *R. Ammendola, M. Bauce, A. Biagioni, S. Di Lorenzo, R. Fantechi, M. Fiorini, O. Frezza, G. Lamanna, F. Lo Cicero, A. Lonardo, M. Martinelli, A. Messina, I. Neri, P.S. Paolucci, R. Piandani, L. Pontisso, M. Sozzi, P. Vicini*

  Venue: *Real Time 2016, Padova, Italy.*

  Date: june 5–10, 2016.

- **POSTER:** "NaNet: Design of FPGA-Based Network Interface Cards for Real-Time Trigger and Data Acquisition Systems in HEP Experiments".*A. Lonardo, R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, F. Lo Cicero, M. Martinelli, P. S. Paolucci, E. Pastorelli, L. Pontisso, D. Rossetti, F. Simula, M. Sozzi, L. Tosoratto, P. Vicini*

  Venue: *NSS/MIC 2015 - 2015 Nuclear Science Symposium and Medical Imaging Conference, San Diego, USA.*

  Date: october 31, november 7, 2015.

- **TALK:** "Hardware and Software Design of FPGA-based PCIe Gen3 interface for APENet+ network interconnect system". *R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, D. Rossetti, F. Simula, L. Tosoratto, P. Vicini*.

  Venue: *International Conference on Computing in High Energy and Nuclear Physics (CHEP2015), Okinawa, Japan.*

  Date: april 13–17, 2015.

Patents pending:

- "Sistema per accelerare la trasmissione dati di tipo collettivo nelle interconnessioni di rete". Request number 102016000071637, July 2017.

Contents

List of Tables

List of Figures

List of Symbols

DAQ Data Acquisition

FPGA Field Programmable Gate Array

GPU Graphic Processing Unit

HEP High Energy Physics

HPC High Performance Computing

NIC Network Interface Card

NVM Non-Volatile Memory

RDMA Remote Direct Memory Access

Motivations

Many of the scientific challenges we face today are complex and need computational power to be solved, but it is a fact that the ramp up of microprocessors frequency is over. In recent years, fundamental physical limitations have caused the deviation from the path predicted by Moore [1] and the scaling rules set forth by Dennard and coworkers [2]. An emerging solution to this problem in the HPC arena is represented by the so-called "heterogeneous systems", where accelerators like GPUs and FPGAs are integrated into systems with conventional microprocessors.

On November 2017, a total of 91 systems on the latest available TOP500 [3] list are using accelerator and/or co-processor technology, up from 86 on November 2016.

A general *heterogeneous architecture* — to be used as a reference in the rest of the dissertation — is the "computing node" (or *node* for short), composed by one or more microprocessors connected to one or many "programmable accelerators" (*i.e.* FPGAs and GPUs), through a system bus (*i.e.* PCIe, AXI, *etc.*) or shared memory. Within such a computing node, all integrated devices (CPUs, GPUs, FPGAs) share a common virtual address space, which constitutes the node local memory space.

In HPC systems, one of the most important features is being able to scale to a certain number of interconnected computing nodes, while retaining the ability for each node to use remotely attached accelerators on an equal footing compared to the local ones.

From an application programmer perspective, exploiting an accelerator requires delivering raw data into its local memory and then, after the relevant computation is performed, pulling back from it the results.

Different solutions like OpenPower CAPI [4][5] and CCIX [6] are being developed to address accelerators which are physically integrated into the computing node. To the same effect, researchers have also implemented several extensions and variations within the MPI (Message Passing Interface) framework, which is the industry-wide standard as well as one of the most widely used communication libraries for HPC, such as cudaMPI [7], GAMPI [8] or MPI-GDS [9]. In this work, however, the focus will mainly be on the underlying low-level software stack. Indeed, when addressing the memory of a device integrated into a remote node, transit from one of the local memories to remote memory through the network becomes critical. Most of the effort must therefore be spent in designing an optimized route through the communication stack for such transfers. This may require both support in hardware and optimization in software when implementing the communication primitives: a traditional hardware and software architecture imposes a significant load on a computing node CPU and its memory because traffic between the network and the node accelerators memory

must be buffered in CPU memory. A solution to this problem is the RDMA communication paradigm, which empowers the programmer with the semantics to explicitly manage reading/writing data from/into another node memory together with the necessary synchronization; whereas this is supported by an RDMA-supporting NIC, these transfers can proceed with minimal demands on the CPU memory bus and processing resources. This allows high-throughput, low-latency networking, which are mandatory requirements in order to protect the scalability in massively parallel computers.

More specifically, a feature called GPUDirect [10] allows a device on the PCIe bus to autonomously read/write data from/to an NVIDIA GPU memory. If this PCIe device is a NIC, the data can be sent over the network directly to a remote GPU memory and vice-versa, in the so-called "GPUDirect RDMA" paradigm. This mechanism makes a PCIe-based system integrating one or more CPUs, GPUs and FPGAs an implementation of the reference computing node architecture defined above.

The RDMA communication paradigm, FPGA-based networks and accelerators can be combined in many ways, in order to specialize the reference architecture of the computing node for different contexts. In this thesis I will address some of the most challenging issues of RDMA communication in FPGA-based networks in three different application scenarios: GPU-accelerated HPC clusters (Chapter 1), Data Acquisition Systems (DAQ) (Chapter 2) and power efficient Exascale systems (Chapter 3). The focus of the activity has been on the design, development and experimental validation of low-level software stack solutions addressing the network performances, *i.e.* communication latency and bidirectional bandwidth between communication endpoints. All my research activity was carried out at APE Lab of Istituto Nazionale di Fisica Nucleare (INFN), which has been active in the development of custom parallel machines dedicated to computational physics simulations (APE [11], ape100 [12], APEmille [13] and apeNEXT [14]). Leveraging on the acquired know-how in networking and re-employing the gained insights, the group has focused its activities on the design of the already mentioned "heterogeneous" systems.

In the first phase of my PhD, my research has focused on the design and the implementation of a stable PCIe Gen3 GNU/Linux device driver for the APEnet+ V5 [15] FPGA-based NIC. This allowed me to have a firm ground to build further developments along the way of optimizing the network performance. I then studied the various issues related to the design of data acquisition cards, in order to actively contribute to the development of the NaNet [16] DAQ boards family dedicated to HEP experiments. To this purpose, I devised a GPU-controlled device driver to be used in combination with existing user-space software libraries, in order to bypass the kernel-to-user space traversals and further improve the communication latency.

The developed software is currently under test in a prototype system deployed at the NA62 [17] experiment at CERN and the report of its operation will be published in future conference papers. I also started collaborating in the "Horizon2020 FET-HPC ExaNeSt project" [18], which aims at prototyping and developing solutions for some of the crucial problems on the way towards production of Exascale-level supercomputers. My contribution to this project was mainly involved with the design of the prototype node, especially for what regards the network architecture and its software interface.

The projects described above will all be discussed in detail in the rest of this dissertation, highlighting the red thread of my contributions, *i.e.* the optimization in the design of the hardware/software interface and the low-level network software stack techniques adopted. Finally, I will provide the foreseen directions for my future research.

Part I

The APEnet project: hardware and software design of an FPGA-based PCIe Gen3 interface for APEnet+ network interconnect system

1 APEnet project

Contents

1.1 Introduction

The APEnet+ project was born to equip COTS-based clusters with a point-to-point interconnect adapter, derived from custom designed HPC systems optimized for Lattice Quantum Chromo Dynamics (LQCD) simulations, featuring a 3D toroidal network mesh.

APEnet+ has specific support for GPUs for use in heterogeneous clusters, yielding significant performance improvements in the simulation of various HPC applications, such as Spiking Neural Networks and Heisenberg Spin Glass simulations and breadth-first search on large graphs [19].

In HPC clusters, low latency and high bandwidth are key factors for overall efficiency and scalability; to improve them, APEnet+ V5 provides hardware support for the Remote DMA paradigm (RDMA), removing the bottleneck of the operating system managing the network transfers by implementing zero-copy transfers.

In this chapter we present the status of the 3rd generation design for the board (called V5) built around the 28 nm Altera Stratix V FPGA.

Among the upgrades provided by the new platform, we mention that the PCIe Gen3 x8 IP is used in combination with a 3rd party core, controlled by an AXI bus; this gives us the option for future integration in the design of a full-featured ARM processor, which is slated to be one of the most significant additions to the next generation of FPGA platforms.

The remainder of this chapter is organized as follows: after an outlook of the state of the art in Sec. 1.1.1, we describe the hardware design of APEnet+ V5 in Sec. 1.2, with an overview of the architecture and improvements like PCIe Gen3 compliance and the new Altera Stratix V FPGA features.

Sec. 1.3 is about the software architecture; our test results are shown in Sec. 1.4.

1.1.1 Related work

Researchers are turning their attention to improving and optimizing the network between the accelerators like GPUs [20], Xeon PHI co-processor [21] and FPGAs [22] [23].

Beside providing processing acceleration capabilities, FPGA devices are also suitable to implement interconnect networks for HPC systems [24], becoming an important cornerstone of computing systems. Examples are the Microsoft Catapult and the AXIOM projects; the former is a scalable architecture implementing a specialized direct network with 20 Gbps bidirectional links to enable FPGA-to-FPGA communication at submicrosecond latency [25], while the latter is based on a Zynq Ultrascale+ board with four USB-C based connector (15Gbit/s per channel), four A53 ARM cores and up to 32GiB memory per node [26] [27].

Many technologies based on the PCIe protocol have been proposed, such as the InfiniBand Architecture (IBA) – a fabric which is an industrial standard for high bandwidth/low latency communication – and other protocols also available for FPGA like Serial RapidIO, Fibre Channel, PCI Express Fabric [28] and many more. On the other hand, undertaking the effort of designing a custom interconnection hardware allows the optimization for a specific workload or task. A natural extension of this idea would be using reconfigurable hardware like an FPGA – *e.g.* the TH2 Express interconnect [29], employed in the Tianhe-2 supercomputer currently holding second place in the TOP500 [3] –, simple encapsulation of other protocols in FPGA-to-FPGA interconnects like in Unified PHY Interface (UPI) system [30], or completely custom interconnects as in BlueLink [31] or such as those in Data Acquisition Systems, see for example the NaNet [16] and Felix [32] projects.

The *PCI Express Adaptive Communication Hub* is a project which in its latest version PEACH3 [33] also uses an Altera Stratix V FPGA together with a PCIe Gen3 host interface

to provide a PCIe ring-based network; it focuses more on being dedicated to tightly coupled accelerators (TCA) than to provide a general purpose, modular NIC.

Whatever task they are put to – as a networking system component like in our case or as a computing accelerator in itself – FPGA-based devices also need high throughput towards their own hosts, often requiring non-trivial software infrastructure. In [34] an open source framework is presented which enables easy integration of GPU and FPGA resources and provides direct data transfer between the two platforms with minimal CPU coordination at high data rate and low latency. In [35] it is described an efficient and flexible host-FPGA PCIe communication library whose presented results were obtained on PCIe Gen2 Xilinx FPGAs.

1.2 APEnet+ V5 hardware

Fig. 1.1 shows the APEnet+ V5 architecture; it is split in three main blocks: *Network Interface*, *Router* and *Torus Link*.

- *Network Interface* is in charge of managing the data flow between PCIe bus and APEnet+: on the TX side it gathers data coming from the PCIe port, generates packets and forwards them to the *Router* block, while on the RX side it provides hardware support for the RDMA programming model. The APEnet+ packets have 256 bits-wide header, footer and data words with variable size payload. *Network Interface* uses a 3rd party device, a Gen3-compliant soft core by PLDA (see Sec. 1.2.1).

- *Router* manages dynamic links between APEnet+ ports: it inspects the packet header and translates its destination address into a proper path across the switch according to a dimension-ordered routing algorithm, managing conflicts on shared resources.

- *Torus Link* allows point-to-point, full-duplex connections of each node with its neighbors, encapsulating the APEnet+ packets into a light, low-level, word-stuffing protocol able to detect transmission errors via CRC. It implements 2 virtual channels and proper flow-control logic on each RX link block to guarantee deadlock-free operations.

APEnet+ V5 was developed on the Altera DK-DEV-5SGXEA7N development kit (see fig. 1.2), a complete design environment that features a 28 nm FPGA (*Stratix V*).

This device offers a PCIe connector, which supports the connection speed of Gen3 at 8.0 Gbps/lane with the host machine and supplies power to the board, a 40G QSFP connector and two HSMC ports which allow the interconnection between boards.

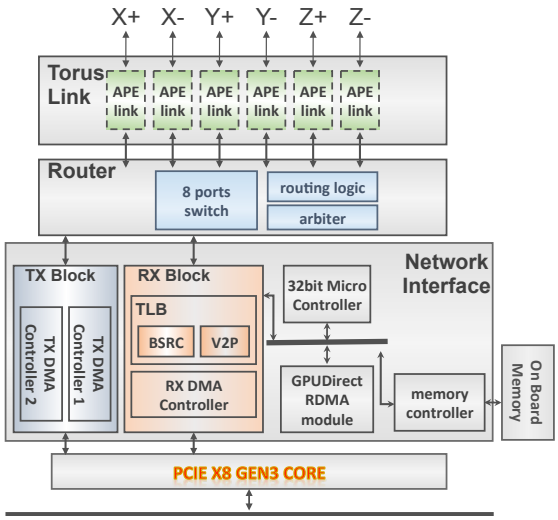


Figure 1.1: Outline of the main logic blocks of APEnet+ architecture.

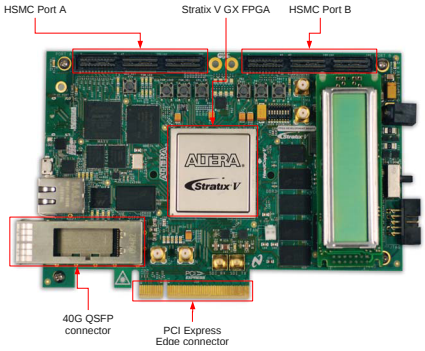


Figure 1.2: Altera DK-DEV-5SGXEA7N Development Board.

Thanks to the modularity of its architecture, a first implementation of APEnet+ V5 with only three links was developed in a straightforward way.

1.2.1 PCIe Gen3 Interface

PCIe Gen3 is based on 8.0 Gbps lanes using a 128b/130b encoding, therefore the protocol overhead is reduced to less than 1% from ∼20% for previous PCIe generations. The upgrade to Gen3 for the host interface allows, with 8 bonded lanes, a total raw bandwidth around 8GT/s. To keep the *Network Interface* clock frequency at 250MHz, the back-end datapath must be widened to 256-bits.

In order to simplify the interface to the PCIe Hard IP, we used a 3rd party device, the PLDA XpressRICH Gen3 soft IP core. The interface of this core with the backend is an AXI4 bus as depicted in fig. 1.3. Inclusion of AXI4 inside our architecture is a useful preparatory exercise for future use of embedded ARM hard IP processors, foreseen only on high-end FPGAs like the future Altera 10 device family.

The XpressRICH core consists of 3 different layers: the *PCIe layer*, the *Bridge layer* and the *AXI layer*. The PCI Express layer contains a PCIe Controller and the interfaces with the Bridge layer – *i.e.* in order to access the PCIe Config Space and manage the low power states and interrupt signals for the IP. The Bridge layer interconnects and arbitrates between input and output flows, implementing up to 8 independent DMA Engine modules and translating between the AXI and PCIe interfaces.

The DMA IF has the task of programming the DMA channels by means of the AXI

memory-mapped interface.

We allotted four DMA channels arranged in a 2 RX/2 TX split to the task of increasing the bandwidth between GPU/CPU and APEnet+ across the PCIe interface. This choice was dictated by the need to mitigate the two latencies that incur first when setting up a DMA and, after the DMA is setup, waiting for data to start flowing. By means of issuing an alternating 'ready' signal between two channels when setting up the DMA, the achieved improvements were remarkable, either in transmission and reception, as shown in Sec. 1.4. Another significant upgrade to our architecture which was made possible by the switch to a 28 nm FPGA was the upsizing of the Look-Up Tables (LUTs) and the CAMs employed within the hardware implementation of the TLB; the effects of this upgrade are apparent in the bandwidth graphs (see Sec. 1.4).

A state machine collects the requests coming from APEnet+ V5 and programs dedicated registers according to a Round Robin algorithm to perform PCIe transactions.

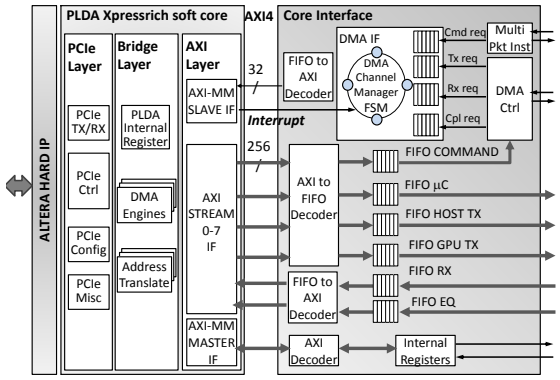


Figure 1.3: Design of PCIe Gen3 interface, based on AXI4 protocol.

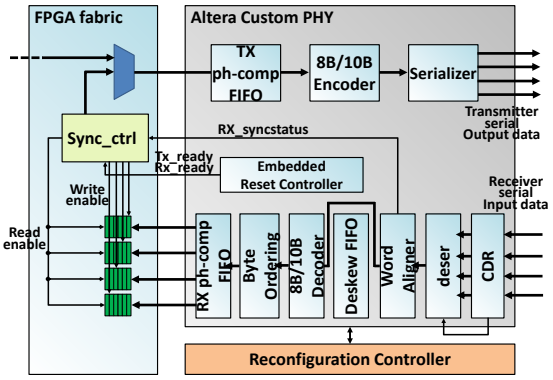


Figure 1.4: APEnet+ V5 Torus Link architecture.

FPGAs of the Stratix V GX family feature full-duplex transceivers with data rates from 600 Mbps to 12.5 Gbps, offering several programmable and adaptive equalization features. The Torus Link block Physical Layer is made up of an Altera Custom IP Core (with the corresponding reconfiguration block) and a proprietary channel control logic (`Sync_ctrl` block), as shown in fig. 1.4. The Altera Custom IP Core is a generic PHY which can be customized in order to meet design requirements. The receiving side consists of a Word Aligner, 8b/10b decoder, Byte Ordering Block and RX phase compensation FIFO. Similarly, on the transmitter side, each transceiver includes a TX phase compensation FIFO, an 8b/10b encoder and a Serializer. The Word Aligner restores the word boundary based on an alignment pattern that must be received during link synchronization. A status register asserted by the Altera Avalon interface triggers the Word Aligner to look for the word alignment pattern

in the received data stream. The Byte Ordering block looks for the byte ordering pattern in the parallel data: if it finds the byte ordering pattern in the MSB position of the data it inserts pad bytes to push the byte ordering pattern to the LSByte(s) position. Finally, the RX phase compensation FIFO compensates for the phase difference between the parallel receiver clock and the FPGA fabric clock. The write and read enable signals of the Deskew FIFOs are managed by the `Sync_ctrl` block: the write signals are asserted for each lane after recognition of 8b/10b keyword /K28.3/, while the read signal, common to all FIFOs, is asserted when all FIFOs are no longer empty. Altera Transceiver Reconfiguration Controller dynamically reconfigures analog settings in Stratix V devices: it is able to compensate for variations due to process, voltage and temperature in 28 nm devices.

Cable	BER	Data Rate
10 m Mellanox optical cable	$<$ 2.36 E-14	11.3 Gbps
1 m Mellanox copper cable	$<$ 1.10 E-13	10.0 Gbps

Table 1.1: APEnet+ BER measurements on Altera 28 nm FPGA.

In conclusion, three bi-directional data channels could be implemented on the development board. The X channel was implemented using the 4 lanes of the QSFP connector; the Y and Z channels were implemented onto the HSMC interface. Very preliminary BER measurements were performed on the X channel of APEnet+ V5 (see table 1.1). The testbed consists of two Stratix V FPGAs connected by InfiniBand cables with different lengths and support media (optical and copper). Copper wires results derive from experiments conducted with 10 Gbps-certified cables; they are expected to improve as soon as they are repeated with commercially available 14 Gbps-certified ones.

1.3 APEnet+ V5 Software Design

In the context of x86/x86_64 GNU/Linux-based hosts, the memory management system implies memory virtualization. The virtual address space is paged in 4 KiB chunks with the Linux kernel providing virtual-to-physical address translation services to the APEnet+ device driver. Using the RDMA communication paradigm, a NIC is required to autonomously handle buffer registration for the application – with all the added complexities of virtual memory management for x86/x86_64 on a GNU/Linux OS – to achieve the so-called zero-copy, off-loaded implementation. We developed the virtual address translation mechanism in two steps: first, a solely μC-based version using a `Nios II` processor (described below); we then added a hardware TLB in such a way that we could measure real advantages with higher

level software tests and perform a comparison between the two versions.

The `Nios II` processor, in Fig. 1.5, is a general-purpose RISC processor core designed for Altera FPGA devices, featuring full 32-bit instruction set, data path, and address space.

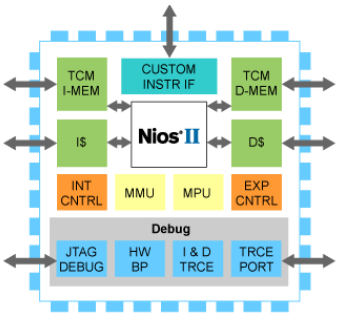


Figure 1.5: Schematic view of `Nios II` processor.

It is a configurable soft-IP core that can be customized on a system-by-system basis adding or removing features to meet the assumed performance goals. A detailed description of the `Nios II` platform is available here [36]. When the application invokes a buffer registration, the CPU-hosted APEnet+ device driver driver scans the pages spanned by the buffer, pins them and registers them individually resolving their virtual and physical addresses and storing them in the RDMA LUT.

Typical APEnet+ data packet is formed by a 128-bit header and footer plus a payload of maximum size equal to 4 KiB, *i.e.* the atomic Linux page size, thus a single packet buffer can span one or two pages, depending on its size and offset. We remark that the 4 KiB size limit does not weigh on the application: the APEnet+ device driver transparently resolves its chunks at registration time.

APEnet+ implements network transfers by means of RDMA PUT semantics (see Fig.1.6); in this way, the data-receiving side of the application initiates the *rendezvous* by advertising the virtual address of its registered buffer to its data-transmitting peer. This means that the packets emitted by a RDMA PUT operation posted by the transmitting node to its own APEnet+ are signed with the virtual address of the receiving node buffer that the companion APEnet+ on this latter, if it is not to interfere with computation, must autonomously resolve into physical addresses.

On the receiving node, the buffer virtual address is therefore used by APEnet+ as key to perform a sort of address translation retrieving all needed buffer information from the table, primarily ID of the owner process and its physical address.

The first task of the receiving logic is thus scanning the buffers pool registered by a running application to determine whether the destination virtual address actually belongs to any of them – this is Buffer Search (BSRC).

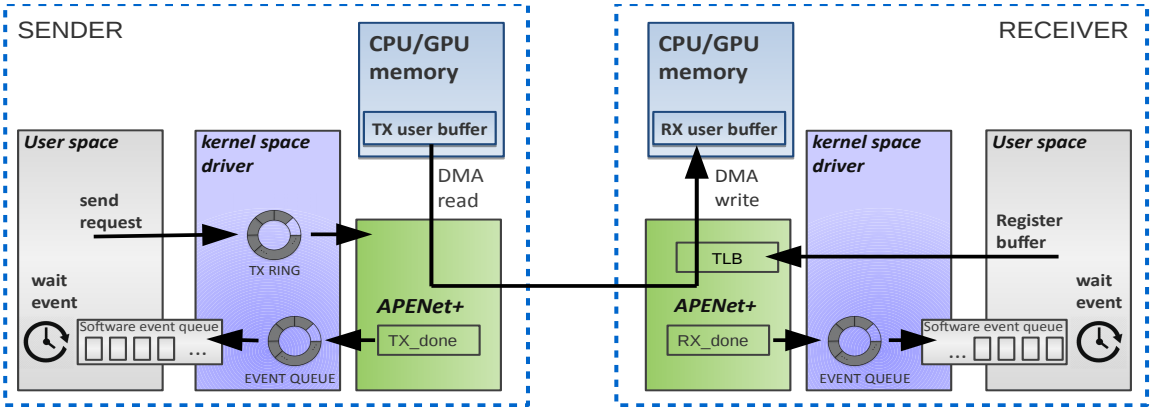


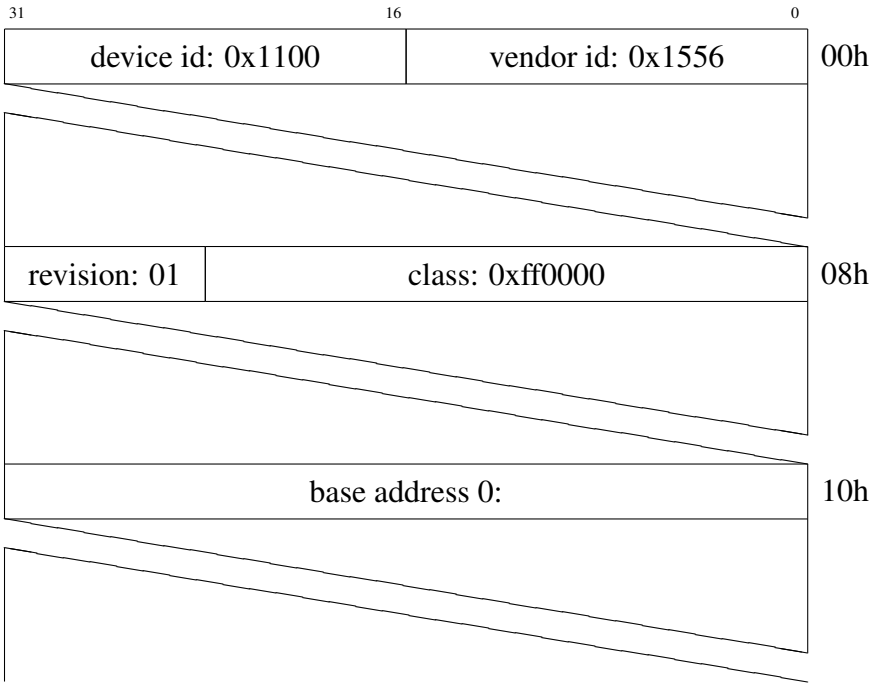
Figure 1.6: Software architecture of APEnet+ V5 NIC.

The correct Page Table is then accessed and its walkthrough is performed – this is virtual-to-physical (V2P) translation – to resolve PCIe addresses of the pages where a number of DMA transactions are finally issued to.

Finally, a completion transaction is performed to fill an event queue that can notify the code managing the transaction – be it in the host OS or the application – that data has been delivered, typically after the last received packet of the exchanged message.

1.3.1 PCIe Gen3 Driver Design

PCIe specification expects, for each device, a software driver initialization and configuration via a separate Configuration Address Space. Almost all PCIe devices are required to provide configuration registers for this purpose. With respect to the standard PCI configuration space, APEnet+ configuration as seen by the Linux Device Driver is:



For the sake of simplicity this section is divided into a transmission phase (TX) and a reception phase (RX) to better explain how the driver works.

Transmission phase

The process of data transmission is composed by a sequence of steps, each of which will be discussed in detail later in this section. The idea is:

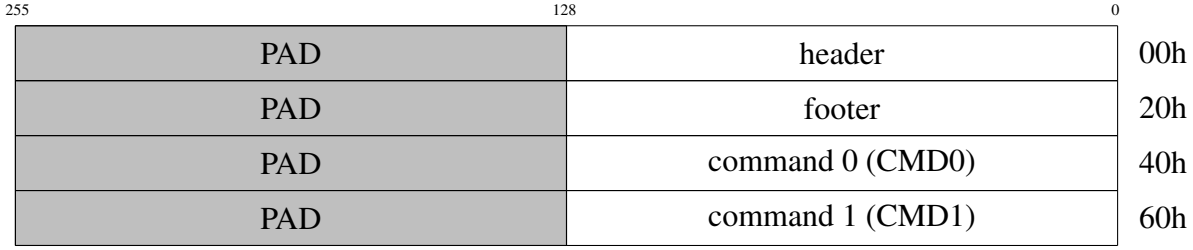
- the user-space process calls a library function to send data from a certain source buffer (TX buffer) to a certain remote buffer (RX buffer);
- in kernel-space, the device driver processes the request, pinning and locking the source buffer in memory;
- the device driver fills a 'descriptor' with references to the source buffer physical address and to the destination buffer virtual address;
- the device driver notifies the board that a new 'descriptor' is ready;
- the hardware DMA-reads the memory and starts processing the new descriptor.

All the relevant information to program APEnet+ for a data transfer (*i.e.* where the data is located and where it must be sent) are stored in a **descriptor**, filled by the device driver and inserted in a circular list called **TX ring**. The DMA-capable memory region

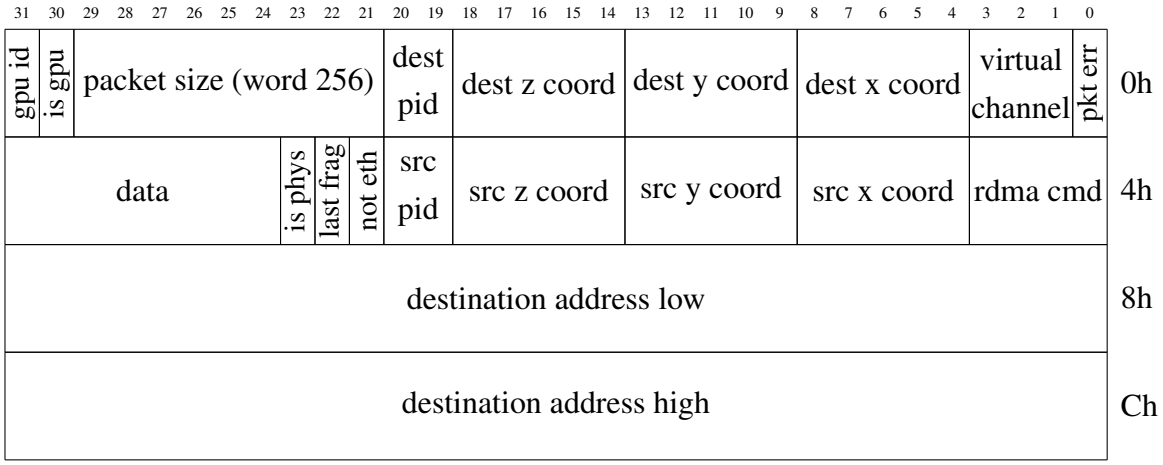
where the **TX ring** will be placed is allocated in the init phase of the driver and the memory start address is sent to the board by writing it in registers 96 (TX_RB_ADDR_HI) and 97 (TX_RB_ADDR_LOW): as the name suggests, the first is for the lower 32-bit of the address while the latter is for the upper 32-bit part.

Since **TX ring** is a circular collection of descriptors, it is important for the driver to not overwrite memory that was not read by the board, while the hardware must pay attention not to read inconsistent 'unfilled' data. The driver uses the high part (16 bit) of the APEnet+ register 98 (TX_RB_NEXT_PTR) to update the counter of the ready descriptors in the **TX ring**. The board updates the same register in the lower 16 bit to notify the driver about the number of read events. In this way the driver is always aware of the 'pointer' where the board is reading the events from (pointer to the next read event or `nr_ptr`) and where it must write the next ready descriptor (pointer to the next write event or `nw_ptr`).

Each descriptor consists of four 256-bit words, each of which is padded to zero in the upper (most significant) part, which is not used at the moment.



Descriptor - header This is the packet header structure:



Note that in the header there is a reference to the destination buffer address, which changes based on the type of address used:

- remote virtual address - streaming buffer

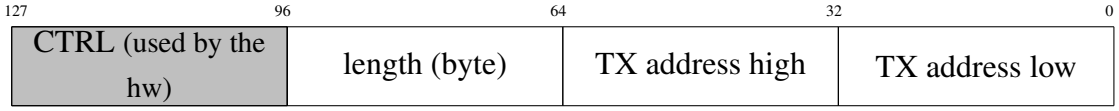
- remote virtual address - persistent buffer
- local physical address

When a peer initializes the communication, it is not aware of the address of the buffer allocated by the other peer, so it is not possible to fill the header address field with a 'valid' address. To solve this problem, is it possible to use a 'streaming buffer'. The header 'destination address' field for a streaming buffer is always filled with '0's. Usually, streaming buffers are used just to exchange the address of a 'persistent' buffer where data is going to be stored. Local physical addresses are used only for debugging purposes.

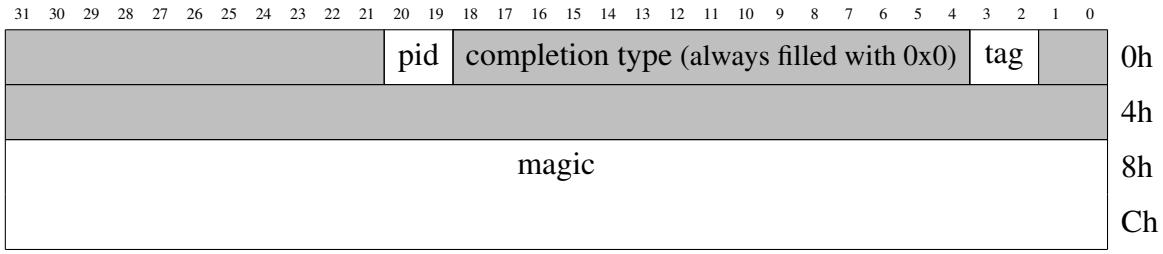
Descriptor - footer The footer is filled in the TX phase whith just 2 control words used by the hardware for check purposes. The format is as follows:

Descriptor - command 0 (cmd0)

The command 0 is used mainly by the hardware to program a DMA read with the source buffer memory address and its length.



Descriptor - command 1 (cmd1) The 'cmd1' word is also called 'completion' because the hardware sends it directly to the event queue controller to generate the 'sent event' completion (see Sec. 1.3.1).



tag

Name	Length (bit)	Value	Description
tag	2	0x0	pass CMD1 to the event queue controller (APEDEV_COMP_EQ)
		0x2	discard CMD1 (APEDEV_COMP_NONE)

magic The most important field is the word 'magic', which is actually a reference for the transmitting buffer. In fact, it is filled with the virtual address of the structure that represents a user buffer in the kernel. This structure contains all the relevant information:

- virtual and physical addresses of the user buffer
- length
- a pointer to the list of pages associated to the buffer
- the 'context' of the buffer, a word used by the user to check the data consistency

So, when the board finishes transmitting the data and the completion is written to memory, it is possible to know what buffer this is referring to by simply dereferencing the virtual address contained in the 'magic' field.

For efficiency reasons, the driver can fill multiple descriptors before updating next write pointer accordingly. In this way, the board will program a single DMA for all the descriptors.

Since virtual memory can map a non-contiguous memory region in a contiguous virtual address space, the buffer could be fragmented all over physical memory. Every transfer bigger than one page is managed by a scatter-gather (SG) list, where every element of the list is a page of the buffer. In this case, each page will be mapped in one **TX ring** descriptor, *i.e.* every descriptor owns the associated physical address of one page of the buffer.

The function that fills the descriptors is called fragment_and_send because it divides the source buffer in chunks of maximum 'PAGE_SIZE' bytes. The pseudo code is as follows (error detection and details are left out):

```
int fragment_and_send(header, tx_buffer, dest_address, CMD1)
for_each page in tx_buffer
        tx_dma_addr = virtual_to_physical_address(page)

        if(is_last_chunk)
                header.flags |= APEDEV_HDR_FLAG_LAST_FRAG
                CMD1.tag = APEDEV_COMP_EQ
```

```
else
        header.flags |= APEDEV_HDR_FLAG_NOT_LAST_FRAG
        CMD1.tag = APEDEV_COMP_NONE

header.address = destination_address
header.size = chunk_size_convert_to_128b_words

CMD0 = fill_cmd0(chunk_size, tx_dma_addr)
footer = 0

txring_wait_for_slots()
txring_fill_descriptor(header, footer, CMD0, CMD1)

ring.nw = (1+ring.nw) % ring.size
destination_address += chunk_size
```

Listing 1.1: Initial pseudo-code implementation of `fragment_and_send`

Note that the completion is usually generated only on the last frag, so for the other descriptors the IS_LAST_FRAG header bit is cleared.

It is also important to check for available slots in the **TX ring** before writing the descriptor, otherwise the driver could overwrite a descriptor that was not yet read by the board. The function fill_descriptor simply fills the header, footer, cmd0 and cmd1 fields in the **TX ring** element data structure.

The above function computes the address by adding 'chunk size' (which is at maximum PAGE_SIZE) to the virtual address. In this way, the NIOS on the receiving side can compute the correct physical address to program the DMA engine.

If the transmission uses a streaming buffer, the destination address is always set to '0' in its header field. Because of this, it is not advisable to try sending more than one page: the second page would be at address 0x1000 (*i.e.* address 0x0 + PAGE_SIZE) which could be a 'valid' virtual address.

Local physical addressing is used mostly for debugging purposes, but if the transmission uses this kind of addressing, the send function has to translate the virtual address, so the pseudo code changes as follows (the important differences with respect to the previous code are highlighted):

```
int fragment_and_send(header, tx_buff, dest_addr, CMD1)
for_each page in transmission_buffer
```

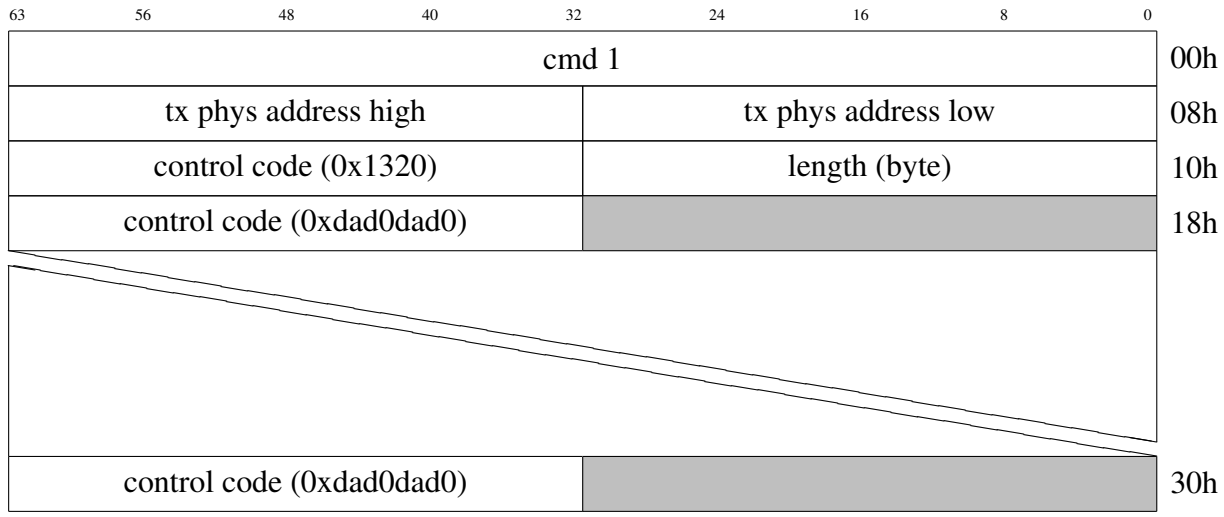
```
tx_sg = tx_buffer[page]
rx_sg = rx_buffer[page]
rx_dma_addr = sg_dma_address(rx_sg)
tx_dma_addr = sg_dma_address(tx_sg)

if(is_last_chunk)
        header.flags |= APEDEV_HDR_FLAG_LAST_FRAG
        CMD1.tag = APEDEV_COMP_EQ
else
        header.flags |= APEDEV_HDR_FLAG_NOT_LAST_FRAG
        CMD1.tag = APEDEV_COMP_NONE

header.flags |= APEDEV_HDR_FLAG_PHYS_ADDR

header.destination_address_low = u64_lo(rx_dma_addr)
header.destination_address_high = u64_hi(rx_dma_addr)
header.size = chunk_size_convert_to_128b_words

CMD0 = fill_cmd0(chunk_size, tx_dma_addr)

txring_wait_for_slots()
txring_fill_descriptor(header, footer, CMD0, CMD1)

ring.nw = (1+ring.nw) % ring.size
```

Listing 1.2: Initial c-code implementation of `fragment_and_and_send phys`

At this point, the board processes the transfer as programmed in the **TX ring** descriptor and sends a completion back to the host, which signals the driver that the transfer has ended or an error occurred.

The completion is maintained in a cyclic queue called **event queue** that is the same for the 'sent event' and for the 'receive event'. Therefore, after the start of a transfer, the driver starts polling the event queue and when a new completion is found, it is put into a queue, ready to be caught by the process that invokes the 'wait_event' library function.

The format of a 'sent event' completion is:



Receiving phase

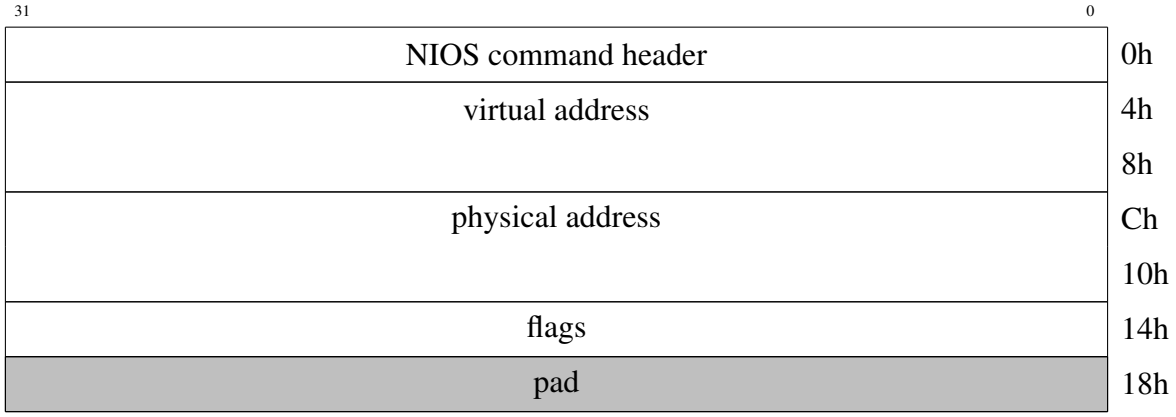
To send data, a process must 'register' a buffer, *i.e.* it must signal the driver that an allocated memory area must be pinned to permit to the APEnet+ board to DMA-access the data in memory.

Since pinning and unpinning memory are expensive operations, the driver should be as lazy as possible in performing such operations. For this reason, the driver implements a simple cache for registered buffers: before pinning new pages of memory, it checks if they were already registered, skipping altogether a new registration if the memory is the same. The idea is to use the previously registered buffer instead of re-registering a new set of pages.

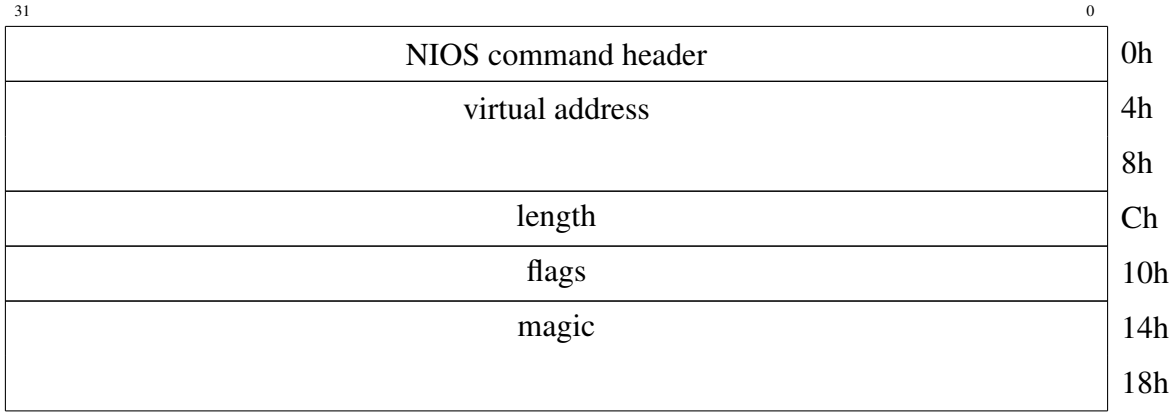
In the RX part of the transmissions, a key role is played by the `Nios II` microprocessor. When a new packet arrives, the `Nios II` translates the virtual address into a physical address that the hardware uses to DMA-write the data at the correct location. If the newly arrived data must be placed into a persistent buffer, the virtual address is relative to a pre-registered buffer and so the `Nios II` simply has to find the associated physical address. If the buffer is a streaming one, the virtual address is null and the microprocessor has to find the first buffer that can contain the new data.

To allow the `Nios II` to translate the addresses, the user must register the association between virtual and physical addresses. To do so, the hardware exposes a set of registers called **CMD FIFO** where the driver writes 'commands' to register pages and buffers.

Registering a page means telling the `Nios II` that a certain page of memory is mapped at a certain physical memory location. The format of the command to register a page is as follows:



This is also needed to 'register' the buffer on the `Nios II` memory, because the microcontroller needs information about the buffer (*i.e.* the length, the context, *etc.*). The command format to register buffers on `Nios II` is as follows:



To summarize, when the user calls the 'register_buffer' function, the driver must:

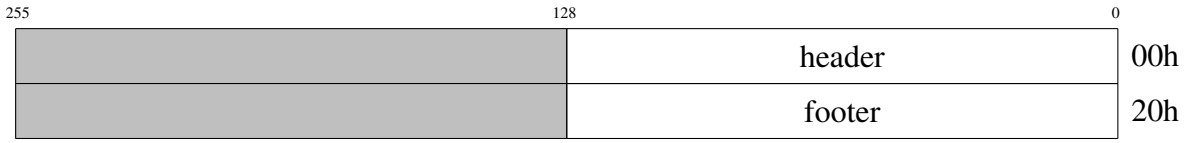
- retrieve the pages that make up the user buffer and put them into a scatter-gather list;
- lock and pin the pages in memory;
- register the pages and the buffer on the `Nios II`.

When new data arrives:

- the board raises an interrupt;

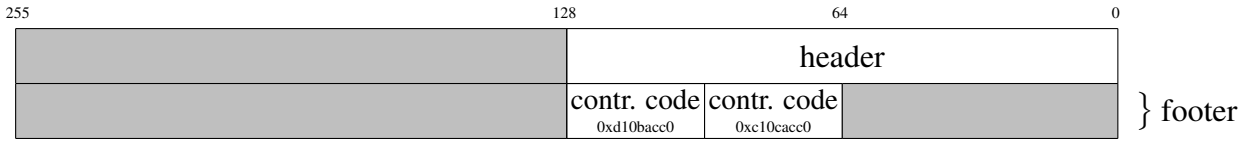
- meanwhile the `Nios II` microprocessor translates the virtual address contained in the packet into the corresponding physical address;
- the board DMA-writes the received data and a completion to memory;
- meanwhile the driver starts polling for the RX completion and checks for errors;
- when the user polls for new events, the driver will return the completions found in memory (DMA-written by the board);
- the user buffer is filled with the incoming data by the board in DMA mode.

The format of the 'receive event' completion changes based on the type of addressing mode used (see below); the format is the following:

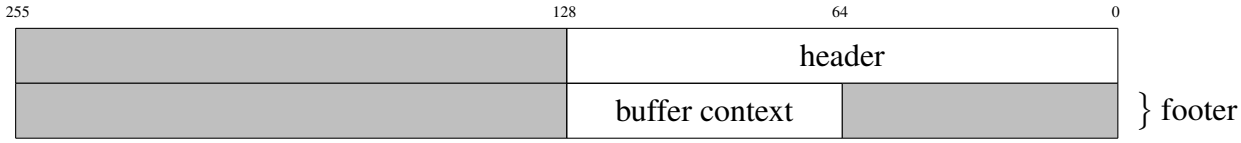


physical address: the header contains the physical address of the destination memory region. Note that the header is the same as written by the transmitter.

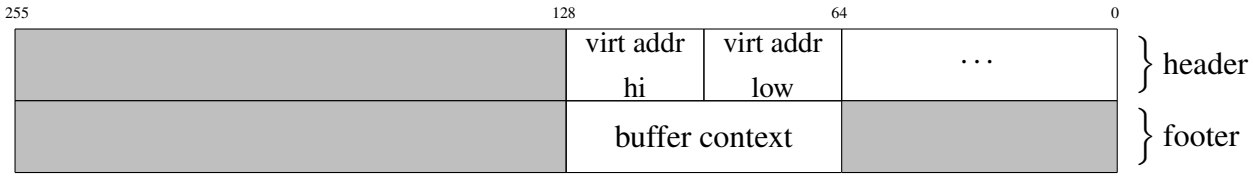
The footer contains 2 control words (d10bacc0:c10cacc0).



persistent buffer: the header contains the same virtual address as written by the transmitter. The footer contains the context of the destination buffer.



streaming buffer: The header contains the *real* virtual address. Since the sender filled it with 0s, the hardware on the receiver side finds a buffer that can contain data and uses the virtual address to fill the header field. The footer contains the context of the destination buffer.



1.4 Test results

All development was performed on the Test and Development (T&D) platform which is composed by two X10DRG-Q SuperMicro servers, connected by the cables as in table 1.1, populated with Haswell E5-2620@2.40 GHz CPUs, hosting the DK-DEV-5SGXEA7N Development Board as APEnet+ V5 and NVIDIA Tesla K40m GPUs; debugging was aided by a Tektronix TLA-7012 Gen3 Logic Analyzer. Both systems use the NVIDIA driver version 367.57 and CUDA version 7.5.

We performed two sets of tests to measure latency and bandwidth.
In both tests the receiver starts by sending the *receiving buffer* virtual address to the sender using two alternatively different methods. The first is an out-of-band Ethernet connection (the simpler way) while the second uses a list of "streaming" buffers. With the latter option, the sender uses the fictitious 0X00000000 virtual address as destination, then the receiving side puts the new data into the first unused preregistered buffer.

Latency tests are carried out by sending a packet in a ping-pong fashion: when a “ping” packet from the sender is received on the receiver side, a “pong” answer of fixed size is sent back. Time elapsed since the “sent” event until the “receive” one of the “pong” packet is measured by using the Intel x86 internal cycles counters accessed by the *Time Stamp Counter (TSC)* register. As expected, results (see Fig. 1.8 and Fig. 1.7) are barely influenced by data size for small packets.

In the bandwidth tests, multiple packets are sent at once but only the last one is ’ACK’ed by the receiving side. Again, time since the “sent” until the “receive” event is measured. We made some preliminary tests on a single machine with one APEnet+ V5 in loop-back configuration to measure the improvement in performance gained by the use of two DMA engines. Results are shown in Fig. 1.9 and Fig. 1.10.

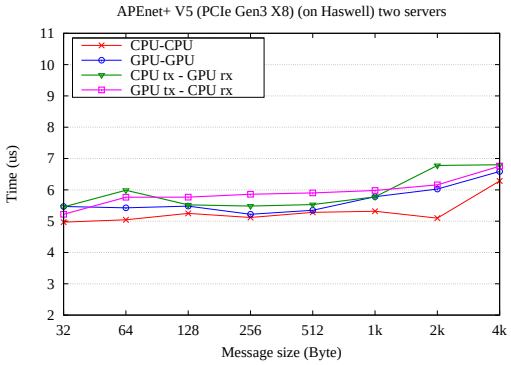


Figure 1.7: Software latency including OS jitter.

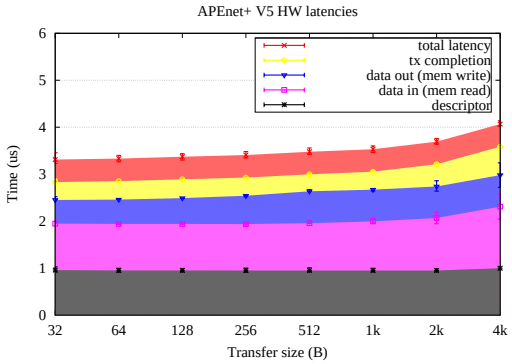


Figure 1.8: Hardware latency for small packet is barely influenced by data size.

Furthermore, we used two different servers to take the same measurement; results for every combination of host/GPU memory and TX/RX modes are shown in Fig. 1.11. As it can be seen, the GPU TX path is slower than the host one, due to the asymmetric performance of the upstream and downstream PCIe, as also addressed in [37], GPU memory read performance is poor on most PCIe switches/root complexes besides very few ones (*e.g.* PLX); better performance is achieved by using the host memory as source.

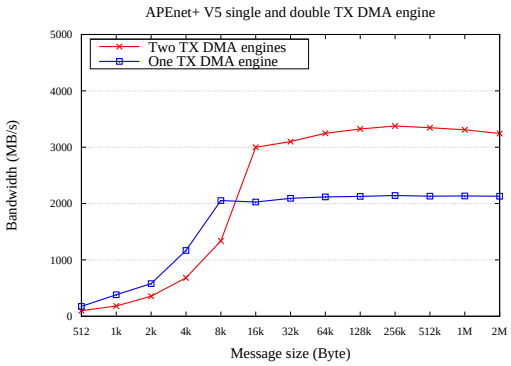


Figure 1.9: In this loop-back benchmark, physical addresses is used for both source and destination, bypassing the virtual-to-physical translation.

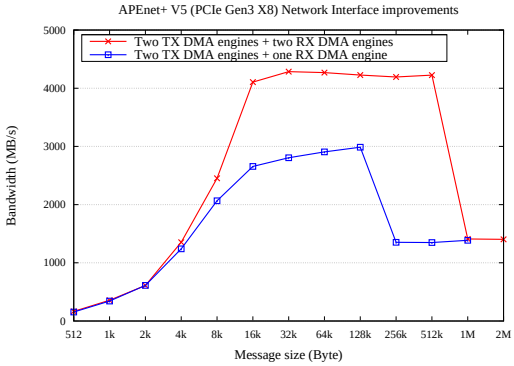


Figure 1.10: In this loop-back benchmark, we doubled the engines on the RX side and used virtual addresses, as can be seen by the TLB miss, which starts around 1M for the new TLB and around 256kB for the old version.

The comparison of the results to Fig. 1.12 shows an improvement in performance by using a second DMA engine, even if it only changes the point in which the application starts hitting the performance degradation. As regards the TLB improvements, see Fig. 1.11 and Fig. 1.12 for a comparison between the old and the new version bandwidths. We are working on the improvement of the receiving side since, as can be seen in Fig. 1.13, it currently

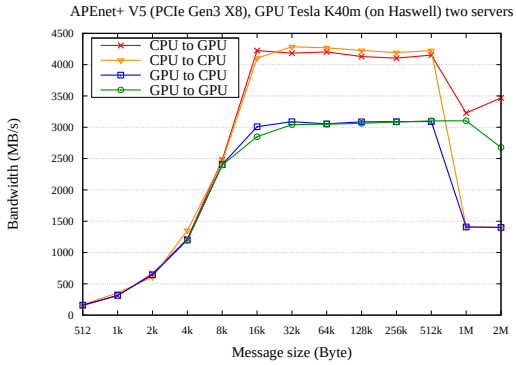


Figure 1.11: Bandwidth measured using the new TLB and double DMA engines on the RX side.

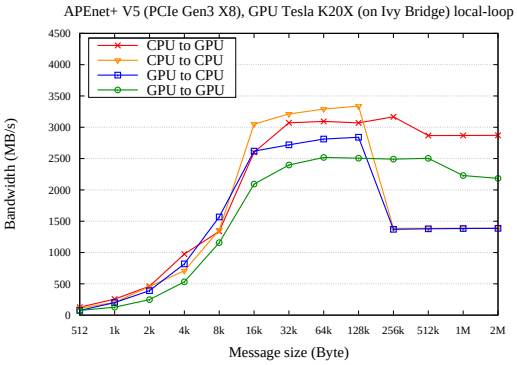


Figure 1.12: Bandwidth measured using the old version of the TLB. Miss-in-cache starts around 256kB for CPU communication. This effect is absent in the GPU plots because of the larger memory pages.

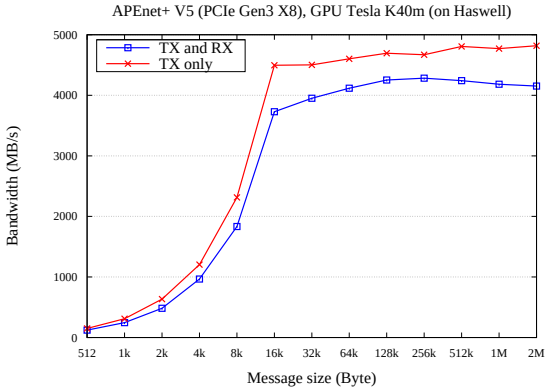


Figure 1.13: Performance gain flushing away the RX part of the communication spots the need for RX optimization.

represents the limiting factor of the overall communication performance.

In fig. 1.14 and in fig. 1.15 we report the bandwidth/latency comparison for the APEnet+ V5 board with respect to the old APEnet+ Gen2.

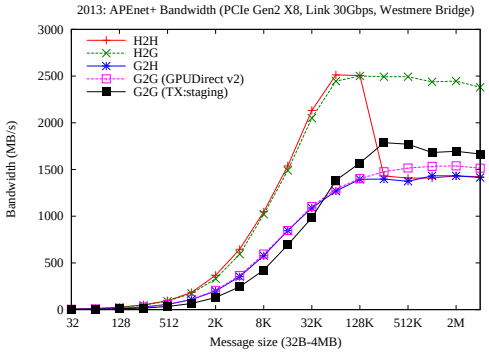


Figure 1.14: Bandwidth results in the old APEnet+ V4 NIC.

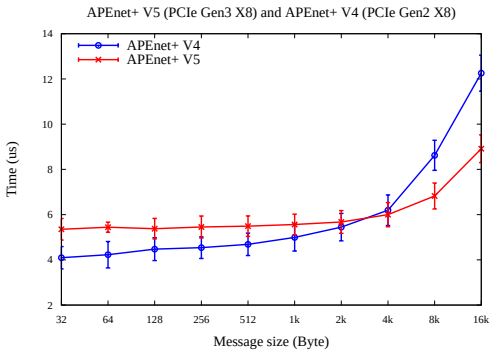


Figure 1.15: APEnet+ V5 vs APEnet+ V4 CPU-to-CPU latency results.

Part II

NaNet: a family of FPGA-based PCIe Network Interface Cards for High Energy Physics

2

NaNet project

Contents

2.1 Introduction

The NaNet project goal is the design and implementation of a family of FPGA-based PCIe Network Interface Cards for High Energy Physics to bridge the front-end electronics and the software trigger computing nodes.

The design includes a network stack protocol offload engine yielding a very stable communication latency, a feature making NaNet suitable for use in real-time contexts; NaNet GPUDirect RDMA capability, inherited from the APEnet+ 3D torus NIC dedicated to HPC systems [38], extends its realtime-ness into the world of GPGPU heterogeneous computing. NaNet features multiple link technologies to increase the scalability of the entire system

allowing for lowering the numerosity of PC farm clusters. The key characteristic is the management of custom and standard network protocols in hardware, in order to avoid OS jitter effects and guarantee a deterministic behaviour of communication latency while achieving maximum capability of the adopted channel. Furthermore, it integrates a processing stage which is able to reorganize data coming from detectors on the fly, in order to improve the efficiency of applications running on computing nodes. Ad hoc solutions can be implemented according to the needs of the experiment (data decompression, reformatting, merge of event fragments).

Finally, data transfers to or from application memory are directly managed avoiding bounce buffers. NaNet accomplishes this zero-copy networking by means of an hardware implemented memory copy engine that follows the RDMA paradigm for both CPU and GPU — this latter supporting the GPUDirect V2/RDMA by NVIDIA to minimize the I/O latency in communicating with GPU accelerators. The quirks in the interactions of this engine with the bulky virtual memory management of the GNU/Linux host are smoothed out by adopting a proprietary Translation Look-aside Buffer based on Content Addressable Memory [39].

My contribution on this project was mainly on the software side (Sec. 2.4), where I worked on the send phase needed to implement a real ”smart trigger” (see Sec. 2.2) and on a persistent CUDA kernel to handle the NIC (see Sec.2.4.1).

2.1.1 Related Works

Data acquisition and high-throughput network mechanisms interfacing the detectors readout are currently under development in several CERN experiments in order to face the increase of luminosity planned for the next years. In [40] the FELIX (Front End LInk eXchange) is presented, a PC-based device to route data from and to multiple GBT links via a high-performance general purpose network capable of a total throughput up to O(20 Tbps). The new data acquisition system under development for the next upgrade of the LHCb experiment at CERN is presented in [41] focusing on the PCIe board PCIe40 aiming to achieve a data throughput of 100 Gbps. The firmware design and implementation of Common Readout Unit (CRU) for data concentration, multiplexing and trigger distribution at ALICE experiment is described in [42].

The integration of GPUs in high-level trigger systems to achieve the computing power to cope with the LHC luminosity upgrade has been extensively investigated in several CERN experiments in recent years [43, 44].

The NaNet project is, to the best of our knowledge, the first effort towards the integration of GPU devices in a low-level (*i.e.* real-time) trigger system of a High Energy Physics

experiment.

2.2 NA62 CERN experiment

The NA62 particle physics experiment at CERN SPS aims at measuring the ultra rare kaon decay $K^+ \to \pi^+ \nu\overline{\nu}$ as a highly sensitive test of the Standard Model and in search for hints of New Physics.

To make beams rich in kaons, the NA62 team fires high-energy protons from the Super Proton Synchrotron (SPS) into a stationary beryllium target. The collision creates a beam which transmits almost one billion particles per second, about 6% of which are kaons. Before entering a large vacuum tank, each particle in the beam is measured by a silicon-pixel detector. A detector called CEDAR determines the types of particle from their Čerenkov radiation. Further detectors inside the tank look for decay particles: a magnetic spectrometer measures charged tracks from kaon decays and a ring imaging Cherenkov (RICH) detector tells the nature of each decay particle. In this particular detector, the particles generate a radiation beam that impinges with a circular footprint onto the light-sensitive tubes of a photo-multipliers array (see Fig. 2.1). Using the information of this "rings" (radius, number of rings, etc.) we can infer what kind of particles have passed through the detector.

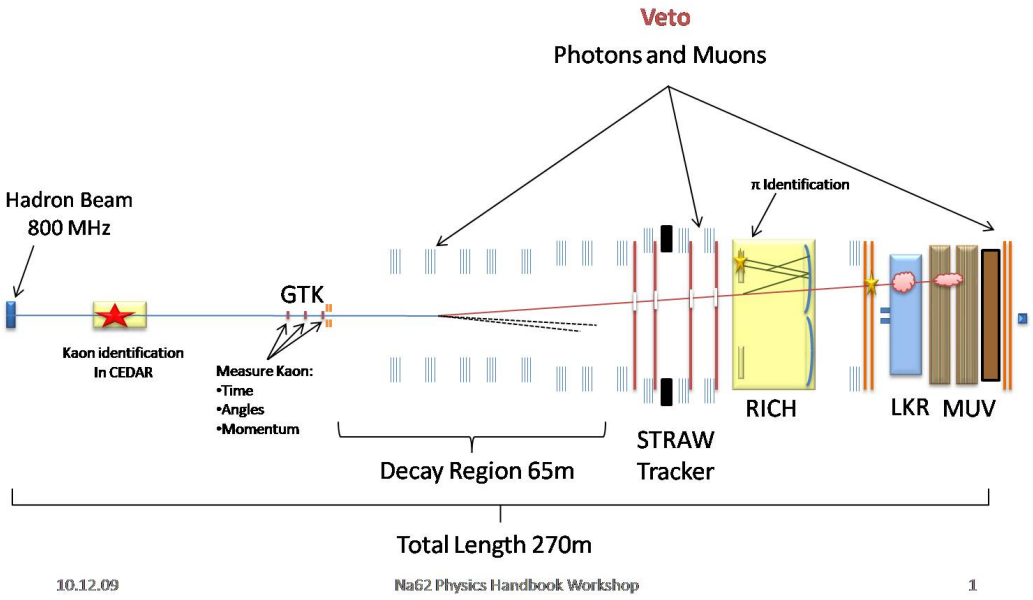


Figure 2.1: NA62 CERN experiment overview.

A multi-level trigger is designed to manage the high rate of data required by the experiment. The first level (L0) is implemented in hardware (FPGAs) on the readout boards

and performs rather crude and simple cuts on the fastest detectors, reducing the high-rate data stream by a factor 10 to cope with the maximum design rate for event readout of 1 MHz. Events passing L0 are transferred to the upper trigger levels (L1 and L2) which are software-implemented on a commodity PC farm. In the standard implementation, the readout boards FPGAs compute simple trigger primitives on the fly, then time-stamp and send them to a central processor for matching and trigger decision. Thus, the maximum latency allowed for the synchronous L0 trigger is related to the maximum data storage time available on the DAQ boards. For NA62 this value is up to 1 ms, in principle allowing use of more compute demanding implementations at this level, *i.e.* the GPUs.

As a first example of GPU application in the NA62 trigger system we studied the possibility to reconstruct rings in the RICH. The center and the radius of the Čerenkov rings in the detector are related to particle angle/velocity. This information can be employed at trigger level to increase the purity and the rejection power for many triggers of interest. The ring reconstruction could be useful both at L0 and L1. In both cases, because of the high rate of 10 and 1 MHz respectively, the computing power required is significant. The GPUs can offers a simple solution of the problem. The use of video cards in the L1 is straightforward: the GPU can act as "coprocessor" to speed up the processing. On the other hand, the L0 is a low latency synchronous level and feasibility of GPU usage must be verified. Data communication between the TEL62 readout boards and the L0 trigger processor (L0TP) happens over multiple GbE links using UDP streams. The main requisite for the communication system comes from the request for <1 ms and deterministic response latency of the L0TP: communication latency and its fluctuations are to be kept under control. The requisite on bandwidth is $400 \div 700$ MB/s, depending on the final choice of the primitives data protocol which in turn depends on the amount of preprocessing actually be implemented in the TEL62 FPGA. So in the final system, $4 \div 6$ GbE links will be used to extract primitives data from the readout board towards the L0TP. To tackle the real-time requirement of the GPU-based L0TP, we considered reusing the GPUDirect RDMA technology that we already implemented in the APEnet+ project for 3D-torus network card. This led to the design and implementation of the NaNet-1 FPGA-based NIC featuring, besides GPUDirect RDMA capability, a UDP offloading engine.

2.3 NaNet design overview

NaNet design is partitioned into 4 main modules: *I/O Interface*, *Router*, *Network Interface* and *PCIe Core* (see Fig. 2.2).

I/O Interface module performs a 4-stages processing on the data stream: following the

OSI Model, the Physical Link Coding stage implements, as the name suggests, the channel physical layer (*e.g.* 1000BASE-T) while the Protocol Manager stage handles, depending on the kind of channel, data/network/transport layers (*e.g.* Time Division Multiplexing or UDP); the Data Processing stage implements application dependent transformations on data streams (*e.g.* performing compression/decompression) while the APEnet Protocol Encoder performs protocol adaptation, encapsulating inbound payload data in APElink packet protocol, used in the inner NaNet logic, and decapsulating outbound APElink packets before re-encapsulating their payload in output channel transport protocol (*e.g.* UDP).

The Router module supports a configurable number of ports implementing a full crossbar switch responsible for data routing and dispatch. Number and bit-width of the switch ports and the routing algorithm can each be defined by the user to automatically achieve a desired configuration. The Router block dynamically interconnects the ports and comprises a fully connected switch, plus routing and arbitration blocks managing multiple data flows @2.8 GB/s

The *Network Interface* block acts on the transmitting side by gathering data coming in from the PCIe port and forwarding them to the Router destination ports while on the receiving side it provides support for RDMA in communications involving both the host and the GPU (via the dedicated *GPU I/O Accelerator* module). A `Nios II` μcontroller in included to support configuration and runtime operations. *Network Interface* block consists of 4 elements: Physical Link Coding, Protocol Manager, Data Processing, APEnet Protocol Encoder. The Physical Link Coding covers the Physical and DataLink Layers of the OSI model, managing transmission of data frames over a common local media and performing error detection. The Protocol Manager covers the Network and Transport Layers, managing the reliability of communication data flow control. A processing stage, Data Processing, that applies some function to the data stream in order to ease the work for the applications running on the computing node, can be enabled on the fly. Finally, the APEnet Protocol Encoder performs a protocol translation to a format more suited for PCIe DMA memory transaction.

Allocation of time-consuming RDMA related tasks has been moved from the `Nios II` μcontroller to dedicated logic blocks. The Virtual Address Generator (VAG) included in the NaNet Controller module is in charge of generating memory addresses of the receiving buffers for incoming data while a Translation Lookaside Buffer (TLB) module, implemented as an associative cache, performs fast virtual-to-physical address translations: a single mapping operation takes only ~ 200 ns [45].

Finally, *the PCIe Core module* is built upon a powerful commercial core from PLDA that sports a simplified but efficient backend interface and multiple DMA engines.

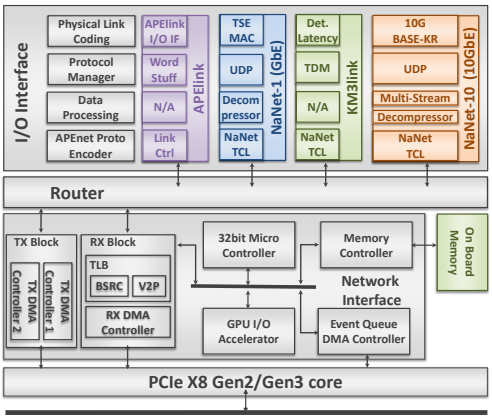


Figure 2.2: NaNet architecture block diagram.

2.3.1 NaNet-1 design

NaNet-1 is a PCIe Gen 2 x8 NIC featuring GPUDirect RDMA over 1 GbE and optionally 3 APElink channels. The NaNet-1 board employs the Altera Stratix IV `EP4SGX230KF40C2` FPGA (see Fig. 2.3); a custom mezzanine was designed to be optionally mounted on top of the Altera board. The mezzanine mounts 3 QSFP+ connectors, thus making NaNet able to manage 3 bi-directional APElink channels with switching capabilities up to 34 Gbps. APElink adopts a proprietary data transmission word stuffing protocol; this is pulled for free into NaNet-1.

For what concerns the implementation of the GbE transmission system we follow the general I/O interface architecture description of Fig. 2.2.

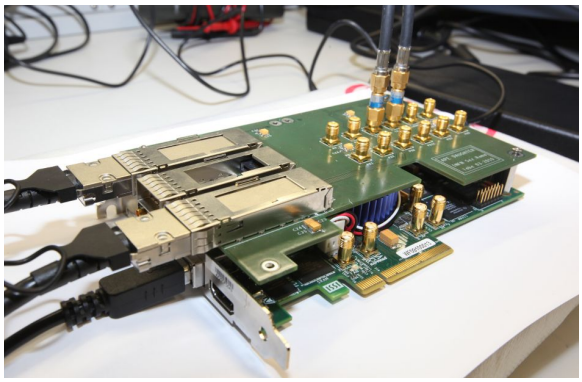


Figure 2.3: NaNet-1 on Altera Stratix IV dev. board EP4SGX230KF40C2 with custom mezzanine card + 3 APElink channels.

We exploit the Altera Triple Speed Ethernet Megacore (TSE MAC) as Physical Link Coding, providing complete 10/100/1000 Mbps Ethernet IP modules. The design employs SGMII standard interface to connect the MAC to the PHY including Management Data I/O

(MDIO); the MAC is a single module in FIFO mode for both the receive and the transmit sides (2048x32 bits).

The data protocol manager tasks are carried out by the *UDP Offloader* dealing with UDP packets payload extraction and providing a 32-bit wide channel achieving 6.4 Gbps (6 times greater than the standard GbE requirements). The UDP Offloader component collects data coming from the Avalon Streaming Interface of the Altera Triple Speed Ethernet Megacore and redirects UDP packets into a hardware processing data path. In this way, the FPGA on-board μcontroller (`Nios II`) is totally discharged from UDP packet traffic management.

The I/O interface data flow control logic is managed by the *NaNet Controller*, a hardware component able to encapsulate data packets in the APEnet+ protocol formed by a *header*, a *footer* (128-bit word) and a *payload* of maximum size equal to 4096 bytes. NaNet Controller implements an Avalon-ST Sink Interface collecting the GbE data flow from the UDP offloader, parallelizing incoming 32-bit data words into 128-bit APEnet+ data ones.

Data coming from the I/O interface are managed by the *Router* component; it supports a configurable number of channels, acting as a multiplexer for a customizable number of ports.

Finally, the *Network Interface* comprises the PCIe X8 Gen2 link to the host system for a maximum data rate of 4+4 GB/s, the packet injection processing logic, the *RX block* and *GPU I/O accelerator* providing hardware support for the RDMA protocol for CPU and GPU, managed by the `Nios II` μcontroller operating at 200 MHz.

2.3.2 NaNet-1 performances

NaNet-1 performances were assessed on a Supermicro SuperServer 6016GT-TF. The setup comprised a X8DTG-DF (Tylersburg chipset — Intel 5520) dual socket motherboard, 2 Intel 82576 GbE ports and NVIDIA M2070 GPU; sockets were populated with Intel Xeon X5570 @2.93 GHz.

The host simulates the RO board by sending UDP packets containing primitives data from the host system GbE port to the GbE port hosted by NaNet-1, which in turn streams data directly towards CLOPS in GPU memory that are sequentially consumed by the CUDA kernel implementing the ring reconstruction algorithm. This measurement setup is called "system loopback".

Exploiting the x86 Time Stamp Counter (TSC) register as a common time reference, it was possible in a single process test application to measure latency as time difference between when a received buffer is signaled to the application and the moment before the first UDP packet of a bunch (needed to fill the receive buffer) is sent through the host GbE port. Communication and kernel processing tasks were serialized in order to perform the

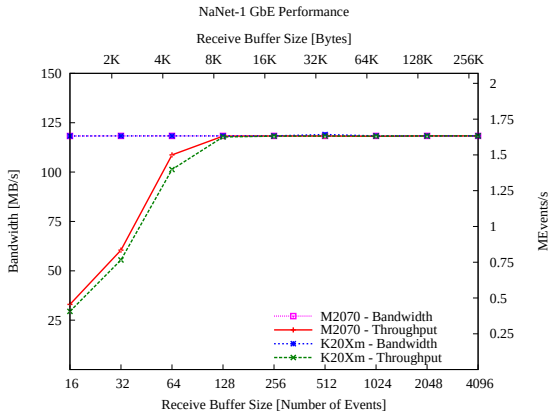


Figure 2.4: NaNet-1 GbE I/O bandwidth and GbE-fed RICH L0TP throughput.

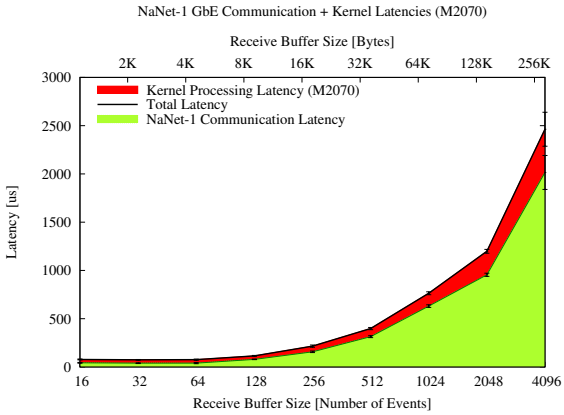


Figure 2.5: Latency of NaNet-1 GbE data transfer and of ring reconstruction CUDA kernel processing.

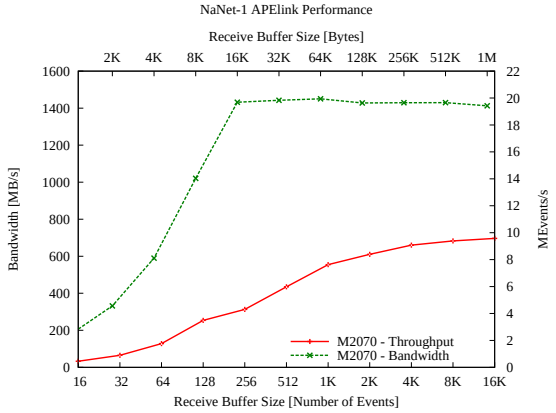


Figure 2.6: NaNet-1 APElink bandwidth and APElink-fed RICH L0TP throughput.

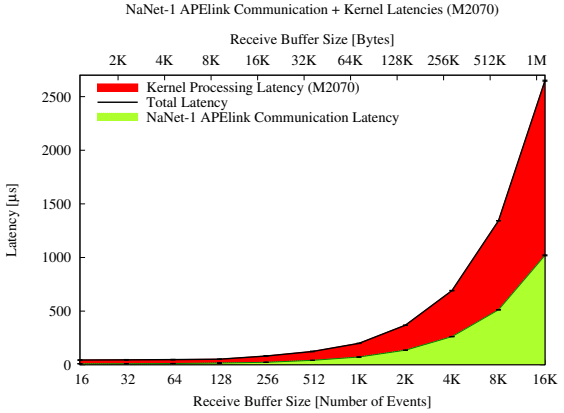


Figure 2.7: Latency of NaNet-1 APElink data transfer and of ring reconstruction CUDA kernel processing.

measure; This represents a worst-case situation: given NaNet-1 RDMA capabilities, during normal operation this serialization does not occur and kernel processing seamlessly overlaps with data transfer. Similarly, we closed in a loopback configuration two of the three available APElink ports and performed the same measurement.

Benchmark results for GbE link bandwidth, varying the size of GPU memory receiving buffers, is shown in Fig. 2.4; it remains practically constant in the region of interest for the reference application and at maximum value for the link.

In Fig. 2.5 latencies for varying size buffer transfers in GPU memory using the GbE link are represented. Besides the smooth behaviour increasing receive buffer sizes, fluctuations are minimal, matching both constraints for real-time and, compatibly with link bandwidth, low-latency on data transfers; for a more detailed performance analysis, see [46].

It is clear that the system remains within the 1 ms time budget with GPU receive buffer

sizes in the $128 \div 1024$ events range. Although real system physical link and data protocol were used to show the real-time behaviour on NaNet-1, we measured on a reduced bandwidth single GbE port system that could not match the 10 MEvents/s experiment requirement for the GRL0TP.

To demonstrate the suitability of NaNet-1 design for the full-fledged RICH L0TP, we decided to perform equivalent benchmarks using one of its APElink ports instead of the GbE one. Bandwidth and latency performances for NaNet-1 APElink channel are in Fig. 2.6 and Fig. 2.7.

Current implementation of APElink is able to sustain a data flow up to $\sim$ 20 Gbps. Results for latency of the APElink-fed RICH L0TP are shown in Fig. 2.7: a single NaNet-1 APElink data channel between RICH RO and GRL0TP systems roughly matches trigger throughput and latency requirements for receiving buffer size in the $4 \div 5$ Kevents range.

2.3.3 NaNet-10 design

NaNet-10 is the evolution of NaNet-1, based on a PCIe Gen2 x8 network adapter implemented on Terasic DE5-net board equipped with an Altera Stratix V FPGA featuring 10GbE SFP+.

The Physical Link Coding is implemented by the Altera 10GBASE-KR PHY and the 10Gbps MAC: the former delivers serialized data to a module driving optical fiber at a line rate of 10.3125 Gbps, the latter supports operating modes starting from 10 Mbps up to 10 Gbps with an Avalon-Streaming interface of 64-bit wide interface running at 156.25 MHz and MII/GMII/SDR XGMII on the network side. A 10 Gbps UDP/IP Core providing full UDP, IPv4 and ARP protocols is in charge of Protocol Manager task. The module offers an AXI-based 64-bit data interface. UDP header settings — *e.g.* source/destination port and destination IP address — are exposed in both transmit and receive sides. Zero-latency between the Protocol Manager and the Physical Link Coding is guaranteed avoiding internal buffer but packet segmentation and reassembly could not be supported. A multi-stream hardware module inspects the received stream and separates the packets according to the destination port to redirect data to different GPU memory buffers to satisfy the application requirements. The decompressor and NaNet TCL units are mostly shared with those in NaNet-1.

2.3.4 NaNet-10 results

To assess the real-time capability of the GPU-RICH, that is, the fundamental requirement for its integration in the NA62 L0 trigger, we started by characterizing separately the compo-

nents involved in the data transport tasks, i.e. NaNet-10 board hardware and software stack, and those involved in processing, i.e. the GPU software application performing the ring reconstruction algorithm. Using a dedicated testbench we measured with a 250 MHz counter the NaNet-10 traversal time of a 1 kB UDP datagram, i.e. the time interval between the first word of the incoming datagram exiting from the 10GbE MAC and the completion of the PCIe DMA of datagram payload towards GPU memory, Fig. 2.8. We identified the reason for the bimodal distribution in the search operation performed in the FPGA to map virtual memory addresses of the receive buffers to physical ones before performing the DMA. We are developing an alternative implementation that will allow this latency to be kept within 1 μs.

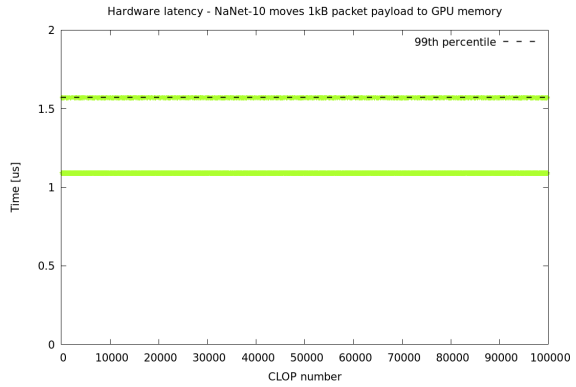


Figure 2.8: NaNet-10 1kB datagram hardware traversal time.

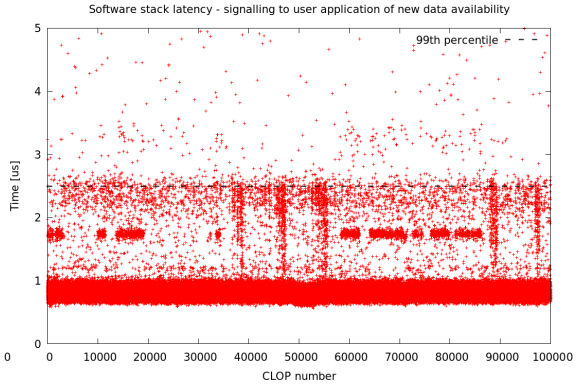


Figure 2.9: Delay of the NaNet-10 hw/sw stack in signaling to the userspace application the filling of a receive buffer in GPU memory.

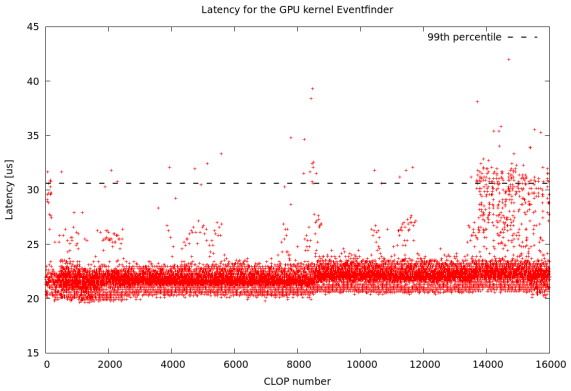


Figure 2.10: Latency of the Event Finder CUDA Kernel performing the indexing of events in the GPU receive buffer.

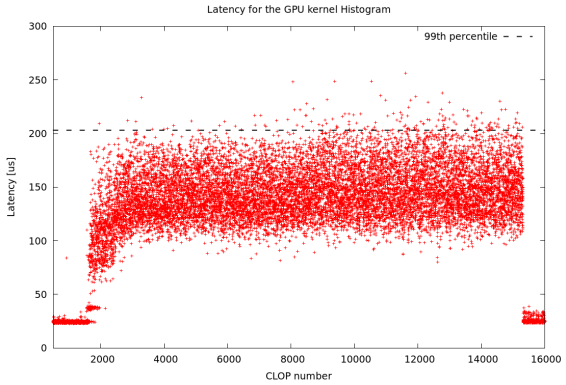


Figure 2.11: Latency of the Histogram CUDA Kernel performing ring reconstruction.

With the same testbench, we measured the delay of the NaNet-10 hw/sw stack when

signaling to the userspace application the filling of a receive buffer in GPU memory, enabling the launch of the CUDA kernel consuming the received data, Fig. 2.8.

The 99th-percentile for this mostly software-related latency is below 3 μs, accounting for a total NaNet-10 hw/sw latency smaller than 5 μs to process a 1 kB UDP datagram. These results validate the real-time features of NaNet-10 as data transport system for the GPU-RICH.

Moving to the processing task, we measured the latencies of the two GPU processing stages, Event Finder and Histogram Ring Reconstruction, in a NA62 setup at $\sim 1/3$ of the nominal event rate with data coming from the RICH detector readout. The experiment Timing, Trigger and Control (TTC) 40 MHz clock has been used to measure the latency in order to have the same time reference with respect to the data source. Results are reported in Fig. 2.10 for the Event Finder CUDA kernel (performing event indexing) and, in Fig. 2.11, for the Histogram CUDA kernel (performing ring reconstruction). Both CUDA kernels are time constrained, with execution time 99th-percentile values well within the available time budget.

Putting all together, Fig. 2.12 and Fig. 2.13 show the overall latency of operations during a single burst, from the receiving of the first UDP packet containing the RICH PMs hit primitives for a given receive buffer (CLOP) to the end of the ring reconstruction on the GPU.

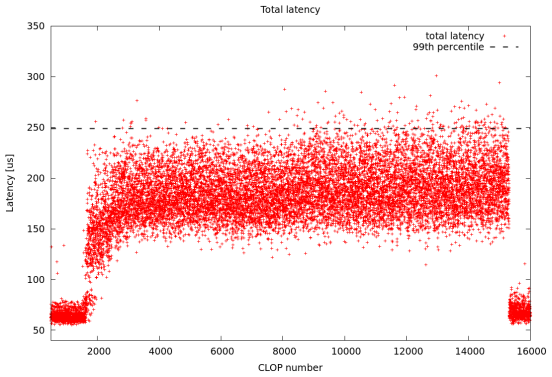


Figure 2.12: Total latency including data transport from RICH readout to GPU memory and GPU processing for ring reconstrution. Data for a whole burst.

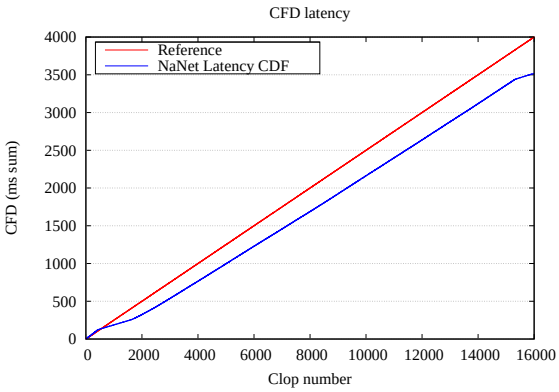


Figure 2.13: Latency Cumulative Distribution Function (CDF).

The plot mimics the particle beam spill shape, with a pedestal at the beginning and at the end of the burst corresponding to RICH PMs hit primitives generated by detector noise and discarded by the reconstruction kernel because of their low number of hits. The $\sim$ 250 μs 99th-percentile value for the total latency confirms the validity of choosing the NaNet

framework to design the GPU-RICH system.

2.4 Software Stack

The NaNet software stack (see Fig.2.14) runs partly on the x86 host and partly on the `Nios II` FPGA-embedded μcontroller (see Sec.1.3). On the host side a GNU/Linux kernel driver controls the device and an application level library provides an API to: open/close the NaNet device; inject commands to register and de-register circular lists of persistent receiving buffers (CLOPs) in GPU and/or host memory, necessary to allocate, pin and return the virtual address of these buffers to the application; manage events generated by the device when receiving packets on the registered buffers in order to promptly invoke the GPU kernel that processes the data just received. On the μcontroller, a single process C program is in charge of device configuration and TLB management.

A CUDA kernel implementation of a fast rings finding algorithm on GPU is another key component of the software stack to keep processing time small and stable enough to cope with input data rate and real-time requirement for the GPU-RICH system.

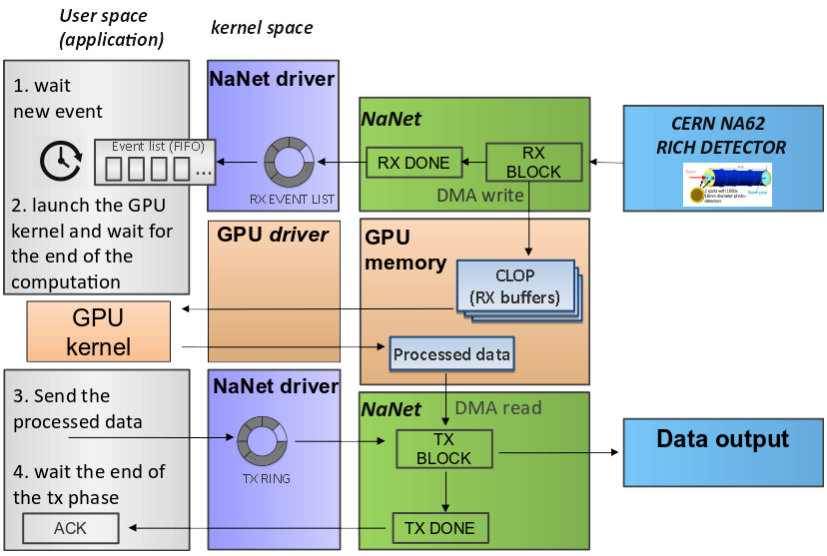


Figure 2.14: NaNet software overview.

2.4.1 GPU-controlling the NIC

One of the bottlenecks in "hybrid" HPC systems is the need of the host CPU to initiate the communication by triggering the NIC, commonly using a kernel-space device driver, even when the data to be transferred resides on the GPU memory. Different solutions have tried to

overcome this problem by using directly the GPU to control the NIC. Such kind of approach is used in GPUnet [37] project, where the networking APIs are invoked from the CUDA kernel itself, but all the calls to the NIC are still performed by the CPU. Other works like GPUrdma [47] and GGAS [20], required partially modification of the NVidia driver to access I/O memory, where "memory-mapped" control registers of the NIC are located. Starting from version 8.0 of CUDA APIs [48], however, it became possible to access the I/O memory without any patch to the driver code, using the GPU itself to drive the NIC, relieving the host CPUs from this task and speeding up the process of receiving, processing and sending data. Disappointingly, this approach has initially demonstrated to yield no advantage using InfiniBand VERBs [49], while in recent work (see [47]) RDMA-based communication bandwidth and latency on a setup equipped with an InfiniBand NIC has shown encouraging results.

In the following, we perform similar investigation on the advantages of a GPU-driven NIC within the peculiar latency-bound NaNet environment.

As already said, in NA62 experiment, a stream of physics events at a 10 MHz rate has to be decimated to 1 MHz before it is passed to subsequent stages. In order to do this, a cascade of detectors is used; we have focused our work on the Ring-Imaging Cherenkov (RICH) detector.

The problem translates to identifying ring patterns in a cloud of points, that is well suited for a GPU.

In the setup of the NA62 experiment, incoming data are DMA-copied into the GPU memory through the PCIe bus by the NaNet NIC which is directly connected to the readout boards of the detector (see Fig.2.15). When data lands in GPU memory, the NIC DMA-write a "Receiving done" completion event in a memory region called "event queue", where is trapped by a kernel-space device driver that is in charge of notifying it to the user application, which in turn launches a CUDA kernel to process the data using the GPU (rings-reconstruction).

Results of this processing, i.e. number and kind (electron, pion, kaon) of rings, is eventually sent via NaNet board to a FPGA performing the first stage trigger, collecting data from all detectors.

The whole process, from data receive in NaNet to results delivered to the FPGA trigger system, has to be completed strictly within 1 ms to avoid data loss.

In the transmission phase, data is to be pulled directly from GPU memory, with the kernel device driver (invoked by the user application) instructing the NIC by filling a "descriptor" with all the relevant information for the transfer (destination IP address, source data memory address, etc.), then dropping it into a dedicated, DMA-accessible memory region called "TX ring". The presence of new descriptors is notified to NaNet by writing on a *doorbell register* over PCIe bus. After the sending phase, where data to be transferred is actually DMA-read

by the NaNet NIC, this latter issues a "tx done" completion event, which is pushed into the "event queue", where the kernel-space driver acknowledges it. As can be seen, the software stack is continuously switching between user-space and kernel-space in either the receive and the sending phase. Our idea is to have a "persistent" CUDA kernel to handle ingress of data, their processing and then egressing the results away without intervention of the host CPU.

To this end, we need to remap in GPU memory both the "TX ring" (for the sending phase) and the "event queue" (for both the receiving and sending phase, in order to catch the completions) to access them directly from the GPU process.

Notice that standard behaviour for a PCIe device driver should be allocating I/O memory via the *pci_alloc_coherent()* function which returns physically contiguous memory; memory of this kind should be remapped by the *cudaHostRegister()* CUDA API with the *cudaHostRegisterIoMemory* flag, but the disadvantage of this approach is that it increases the total latency due to the need of accessing the host memory through the PCIe bus.

So we proceed the other way round, by allocating sufficient GPU memory via *cudaMalloc()* and then translating its virtual memory address into the corresponding physical one via *p2p_get_pages()* GPUDirect kernel API; this physical address can be used directly by the NaNet DMA engines. With this approach, we are able to map both the "TX ring" and the "event queue" in GPU memory to access them directly from a CUDA kernel.

Although a consistency problem may arise when a GPU memory peer-to-peer DMA is performed while a GPU kernel is running, as discussed in [47], we did not experience any in our particular setup. Specifically, a third party PCI device (like NaNet) cannot guarantee consistent in-order updates of GPU memory via peer-to-peer DMA: *e.g.* two sequential RDMA writes can be observed in *reverse order* by a GPU kernel, so that we cannot rely only on the arrival of the "Receiving done" completion event to start the data processing. For this reason, after detecting a completion event, a three-stages processing is performed on the received buffer. In the first stage, the CUDA kernel indexes the starting address of the variable-sized "physics events" contained in the buffer by searching for a 32 bits signature. The indexed events are then used to perform the ring reconstruction. Finally, a third stage poisons the consumed buffer with its index number in the circular list of receive buffers. While not specifically trying to patch the consistency issue, this approach — in conjunction with the limited GPU memory write load generated by the 10 GbE channel — at least vouched that our results were unaffected. Nevertheless, considering the planned development path for the NaNet board featuring more capable I/O channels, this aspect will be carefully analyzed in our future work. The GPUDirect Async [50] is a promising solution that will be investigated to address this potential issue.

Another important aspect is represented by the time spent in the PCIe transactions: as a PCIe device, NaNet is mainly driven through accessing memory-mapped registers. Even taking advantage of the aforementioned *cudaHostRegister* CUDA API to handle I/O memory from inside a CUDA Kernel, the PCIe memory read/write operations are still time consuming. Therefore, we strove to keep the PCIe transactions to as few as possible. We decided to use a kernel-space device driver for the first initialization and remapping phase, we then mainly need to write just the *doorbell register* in the sending phase to inform the NIC about the presence of fresh data to move.

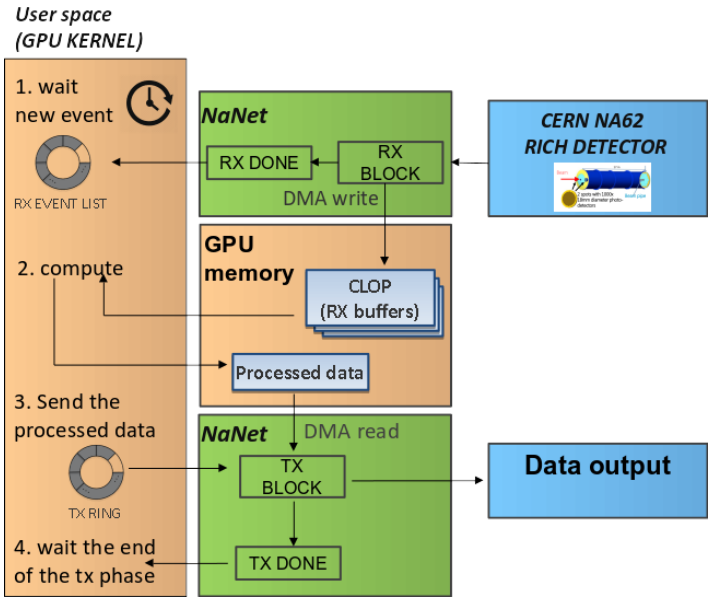


Figure 2.15: NaNet software overview using persistent CUDA kernel.

Preliminary results

The test-bed is composed by a SuperMicro server equipped with Intel Xeon E5-2630 CPUs, a NaNet NIC and an NVidia K20x GPU. We simulate the arrival of new events by sending packets from the Ethernet interface directly to the NaNet NIC, which writes the data into the memory of the GPU, then a dummy kernel is launched to simulate the rings search ($\sim$200μs) and, finally, data are sent from NaNet to another Ethernet board to mimic progressing further towards the NA62 FPGA trigger.

Using the idea presented in this chapter, *i.e.* a persistent CUDA kernel to directly drive the NIC (at the moment no further improvements are made, just a single thread is used), we reap benefits on two different sides:

- we eliminate the latency due to the user $\Leftrightarrow$ kernel space switch by accessing the board directly from the persistent CUDA kernel;
- we save the overhead of launching the CUDA kernel every time a new bunch of events arrives since the kernel is already running on the GPU.

The preliminary result in terms of total latency (adding up receive, compute and sending phases) are shown in the Fig. 2.16. The classical approach to control NaNet NIC, based on a kernel device driver, is compared with a persistent CUDA kernel in a synthetic test that reproduce the NA62 CERN experiment flow (gather data, process it and then send over an ethernet connection). Notice that the plot starts at 200μs to highlight the baseline of computation time. This has yielded encouraging results in terms of a latency reduction of about 10%. Besides these results, the most important achievement is having demonstrated the feasibility of using a CUDA kernel, usually confined to number crunching, to drive the NIC. This is of particular relevance in GPU-based real-time applications.

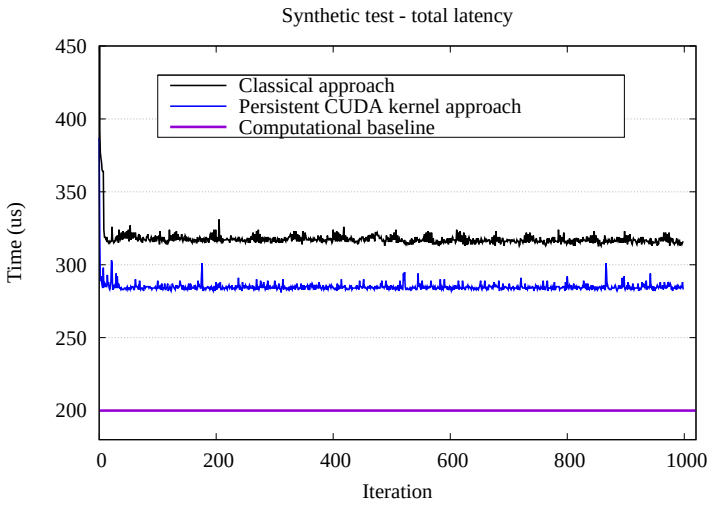


Figure 2.16: Latency test results for the persistent CUDA kernel approach.

Part III

ExaNeSt

3

ExaNeSt project

Contents

3.1 Introduction and related work

The ExaNeSt project [51], started on December 2015 and funded in EU H2020 research framework (call H2020-FETHPC-2014, n. 671553), is a European initiative aiming at developing the system-level interconnect, a fully-distributed Non-Volatile Memory (NVM) storage and the cooling infrastructure for an ARM-based Exascale-class supercomputer. The ExaNeSt Consortium combines industrial and academic research expertise in the areas of system cooling and packaging, storage, interconnects and the HPC applications that drive all of the above.

ExaNeSt will develop an in-node storage architecture, leveraging on low-power NVM devices. The distributed storage system will be accessed by a unified low-latency interconnect, enabling scalability of either storage and I/O bandwidth together with the compute capacity. The unified RDMA-enhanced network will be designed and validated using a testbed based

on FPGAs and passive copper and/or active optical channels, allowing the exploration of interconnection topologies, congestion-minimizing routing functions and support to system resiliency.

ExaNeSt also addresses packaging and liquid cooling – which are of strategic importance for the design of real systems – and aims at an optimal integration which will be dense, scalable and power efficient.

ARM-based servers are currently under evaluation as an alternative to the x86 and POWER-based servers which are prevalent in data-centers and supercomputers for both research and business [52, 53, 54]. This technological approach for a scalable and low-energy solution to computing is shared with other projects with the common goal to deliver a European HPC platform: (i) ExaNoDe [55] focuses on delivering low-power compute elements for HPC and (ii) ECOSCALE [56] focuses on integrating FPGAs and providing them as accelerators in HPC systems.

A set of relevant ambitious applications, including HPC codes for astrophysics [57, 58, 59, 60], spiking neural networks simulation [61], engineering [62, 63], climate science [64], materials science [65] and big data [66], will support the co-design of the ExaNeSt system to provide specifications during design phase and application benchmarks for the prototype platform.

A final prototype of the systems will be made available in 2018, which is the last year of the ExaNeSt project. In this chapter, after a general overview of project motivations, the interconnect technologies and the multi-tiered topologies are introduced to better understand the software stack we are working on. My contribution to the ExaNeSt project is mainly involved with the network architecture software interface of the so-called "KARMA" Test Framework, explained in Sec.3.3, where I worked on the NIC device driver to design and verify the hardware funcionalities. I am also collaborating in the hardware/software co-design of the final system prototype.

3.2 Multi-tiered, scalable interconnects for unified data and storage traffic

The development of an interconnect technology suitable for Exascale-class supercomputers is one of the main goals of the project; we envision it as a hierarchical infrastructure of separate network layers interacting through a suitable set of communication protocols. Topologies in the lowest tiers are hardwired due to choices made in the prototype design phase.

The *Unit* of the system is the Xilinx Zynq UltraScale+ FPGA, integrating four 64-bit ARMv8 Cortex-A53 hard-cores running at 1.5 GHz. This device provides many features, the following being the most interesting:

- a very low latency AXI interface between the ARM subsystem and the programmable logic,
- cache-coherent accesses from the programmable logic and from the remote unit,
- a memory management unit (MMU) with two-stages translation and 40-bit physical addresses, allowing external devices to use virtual addresses, thus enabling user-level initiation of UNIMEM [67] communication.

The FPGA block diagram is shown in Fig. 3.1.

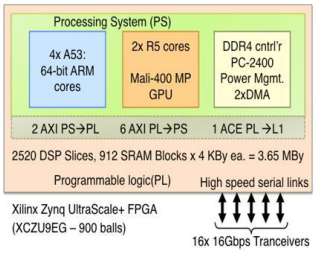


Figure 3.1: The *Unit* of the ExaNeSt system: the Zynq FPGA.

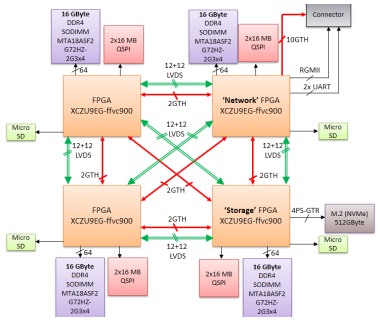


Figure 3.2: The *Node* is the Quad-FPGA Daughter-Board (QFDB).

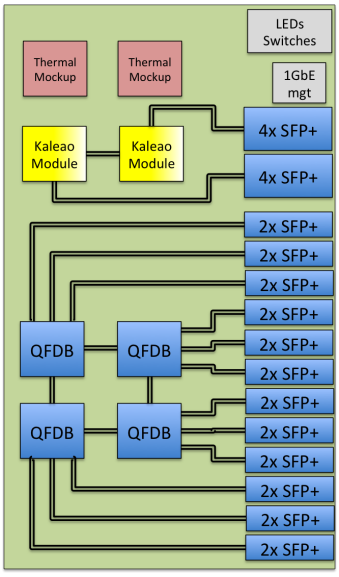


Figure 3.3: the *mezzanine*.

The *Node* (Fig. 3.2) is the Quad-FPGA Daughter-Board (QFDB), which contains four Zynq Ultrascale+ FPGAs, 64 GB of DRAM and SSD storage connected through the ExaNeSt Tier 0 network. The inter-FPGA communication bandwidth and latency affect the overall performance of the system. As a consequence, at QFDB level, ExaNeSt provides two different networks, one for low-latency exchanges based on LVDS channels via AXI protocol, the other for high-throughput transmissions through High Speed Serial links (HSS).

For inter-node communication, the QFDB provides a connector with ten bidirectional HSS links for a peak aggregated bandwidth of 20 GB/s. Four out of ten links connect neighbouring QFDBs hosted on the *Mezzanine/Blade* (Tier 1). The first Mezzanine prototype (Track-1), shown in Fig. 3.3, enables the mechanical housing of 4 QFDBs hardwired in a 2D cube topology with two HSS links (2×16 Gb/s) per edge and per direction. The remaining six HSS links, routed through SFP+ connectors, are mainly used to interconnect mezzanines within the same Chassis (Tier 2). Furthermore, they can also be exploited to modify the Intra-Mezzanine topology.

The Mezzanine sports additional slots to host thermal mockups to evaluate the liquid-cooled mechanics and optional off-the-shelf ARM-based computing modules, the Kaleao KMAX [68]. Nine such as mezzanines will fit within an 11U (approximate height) "half depth" chassis.

3.2.1 Topologies

ExaNeSt explores both *direct* blade-to-blade and *indirect* blade-switch-blade networks. The former type, with direct links (Inter-Mezzanine) between blades, is frequently called "switchless" and has been employed in many HPC installations. These interconnects distribute the switching and routing functions to units that are integrated close to computing elements. The latter will be tested connecting the blades to commercially available components, based on ASICs or FPGAs.

Each mezzanine provides 24 SFP+ connectors to communicate with other mezzanines within the same Chassis. So many independent channels allow for a high level of flexibility to experiment with several direct network topologies.

A first scenario is shown in Fig. 3.4, where 2D torus topology is chosen to interconnect the QFDBs of the 9 blades of a chassis. The solid and dotted lines are the intra-Mezzanine and inter-Mezzanine I/O interfaces respectively. Since local (within the mezzanine) and remote (neighbouring mezzanine) QFDBs are in the same network hierarchy, 2 HSS per direction for remote channels are used to balance the network capability. A 6×6 Torus topology is the resulting configuration where the longest path consists of 6 hops implementing a Dimension-Order Routing (DOR) algorithm (see Sec. 3.2.3).

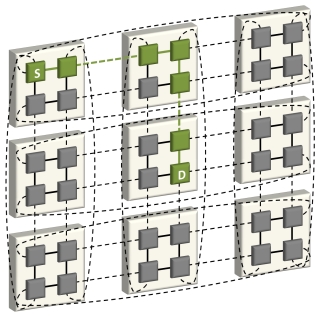


Figure 3.4: QFDBs within the chassis shape a 2D Torus topology (Tier 1/2).

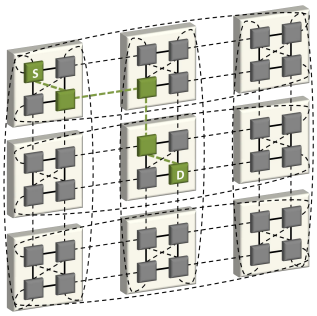


Figure 3.5: Performance boost due to the intra-Mezzanine (Tier 1) all-to-all topology.

An additional design option would use the "diagonal" links to interconnect the QFDBs in a mezzanine resulting in a all-to-all topology. With this simple modification – which also requires the implementation of a more complex routing algorithm – two hops are saved on average, as sketched in Fig. 3.5; our estimation for single hop latency is about 200 ns.

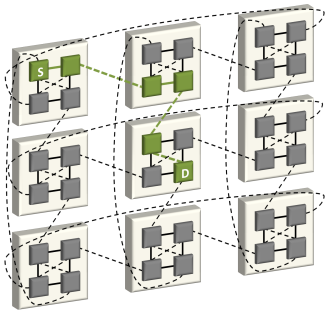


Figure 3.6: Dragonfly topology interconnecting Mezzanine Supernodes (Tier 2).

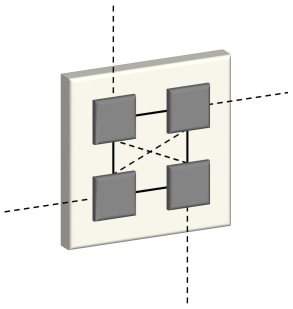


Figure 3.7: Each QFDB exploits only one SFP+ cable for inter-Mezzanine network.

A second scenario foresees a Dragonfly [69] network implementation as in Fig. 3.6. Each blade corresponds to a supernode (Fig. 3.7) connected to the neighbouring nodes with just one inter-Mezzanine channel.

A further latency reduction (3 hops for the longest path as depicted in Fig. 3.8) is gained by connecting each QFDB of a Mezzanine with their counterparts on neighbouring Mezzanines, shaping four 3×3 2D torus networks (Fig. 3.9). Moreover, counterparts QFDBs residing on Mezzanine in neighbouring chassis (Tier 3) can be arranged in a 3D torus; in this way we exploit two additional external inter-Mezzanine channels eliminating the diagonal links on the QFDB. Each set of QFDBs is a 3D torus interconnect $3 \times 3 \times C$ where C is the number of chassis.

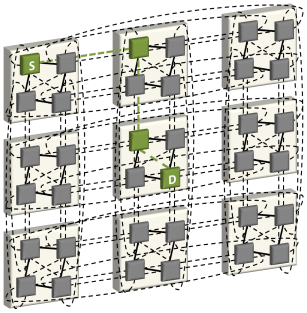


Figure 3.8: An alternative topology to the simple torus network.

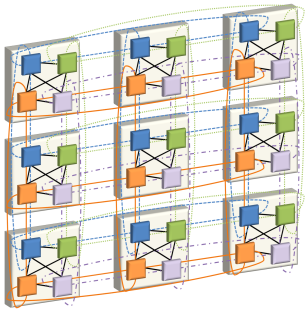


Figure 3.9: Four 2D torus networks interconnecting the mezzanines.

3.2.2 High-Throughput intra- and inter-Mezzanine communication

APElink [70] is the communication protocol for the management of data flow over the HSS links. It is based on a word-stuffing protocol, meaning that data transmission needs submission of a *magic* word every time a control frame is dispatched to distinguish it from data frames.

The word-stuffing APElink protocol includes two words – Magic and Start – into the data flow over the HSS links to establish the logical link between nodes. The transmission of the packet header is announced with this sequence. Since misrouted packets are disruptive for the network, the highly critical header integrity could be protected by an Error Detection Code (EDC) or Error Correction Code (ECC), depending on the Bit Error Rate (BER) that we experience in the network. To prevent the reception buffer from overflowing, the IP manages the flow between two neighbouring nodes by keeping track of the APElink words sent. Buffer availability is measured by *credit*; occupancy of the receiving buffer is contained in the *credit*. Outbound words consume it, causing transmission suspension as soon as a programmable credit threshold is reached – *i.e. credit* is exhausted – and resuming as soon as info about newly available space bounces back to the transmitter – *i.e. credit* is eventually restored. Furthermore, this information is mandatory for the Virtual Cut-Through (VCT) switching mechanism described in Sec. 3.2.3.

Information regarding the health of the node can be embedded in the *credits*, allowing a fault communication mechanism to avoid a single point of failure and guaranteeing a fast broadcast of critical status [71]. Embedding the diagnostic messages in the communication protocol limits the amount of additional overhead and avoids that this flow affects overall performance.

3.2.3 Routing

The APEnet IP [72] is the data flow handler over the HSS network (Tier 0/1/2), implementing low-latency and high-speed communications between the Mezzanines. The *Router* is the component in charge of determining the path the messages will follow to reach their destinations.

Current APEnet implementation adopts a deterministic Dimension-Order Routing (DOR or e-cube) policy: it consists in reducing to zero the offset between current and destination node coordinate along one dimension before considering the offset in the next dimension. The APEnet DOR router is able to handle more than one packet transaction at a time and specialized priority registers – writable at run-time – allow for limited but effective routing function customization.

The employed switching technique – *i.e.* when and how messages are transferred along the paths established by the routing algorithm – is Virtual Cut-Through (VCT): the router starts forwarding the packet as soon as the algorithm has picked a direction and the buffer used to store the packet has enough space.

A more sophisticated routing logic will of course be able to consume the coordinates in a more exotic way or recognize critical directions and then change appropriately the packet header to follow an alternative path to their the destination.

Finally, we will implement a set of effective collective communication functions, in order to relieve what typically acts as a bottleneck for the HPC systems. To enhance application performance at very large system scale, an enhanced design with hardware offloading of these functions is under development.

3.3 KARMA Test Framework

King ARM Architecture (KARMA) is a software-oriented test framework to validate the EXAnet Network IP (APEnet Crossbar Switch and APElink data transmission systems). The main idea behind its design is the use of the multicore ARM Cortex-A53 Processing System (PS) to emulate in software the functionalities of the Network Interface (NI), exploiting the AXI low latency communication capabilities between the PS and the Programming Logic (PL) that implements the systems under test. This approach turned out to be very effective, allowing for the test and validation of the EXAnet Network system since the earliest stages of its development, well before the actual integration with the NI. It also enabled the rapid prototyping of various architectural solutions for the interface between the NI and the Switch systems. Finally, using this framework we were able to characterize the performance of the

two systems in terms of latency.

On the hardware side, the intra-tile Switch FIFOs are directly connected to the ARM HPM AXI port through an in-house developed adapter IP, whose only purpose is the conversion between streaming and memory-mapped AXI protocols. This means ARM has to write every single word into header/data FIFOs, which is obviously suboptimal for bandwidth, but gives good latency results for small-sized packets, as shown later. Moreover, a set of configuration/status registers is accessible on the same AXI bus through the "Target Controller" IP, which allows configuration of the router (*e.g.* setting coordinates and lattice size) and probing FIFOs and link status. This no-DMA approach, summarized in Fig.3.10, was pursued to quickly provide a validation framework for the underlying network subsystem, without replicating efforts developing a Network Interface.

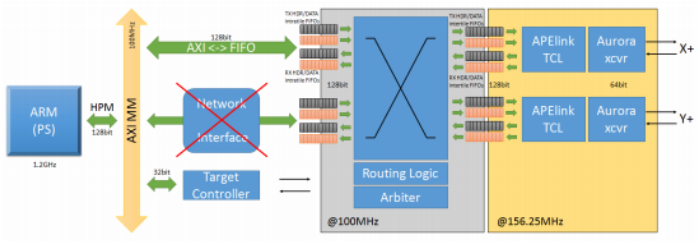


Figure 3.10: King ARM Architecture (KARMA) overview.

3.3.1 KARMA Software User Interface

On the software side, we first created a Linux user-space test program capable of stimulating the devices under test by writing commands and data to the corresponding hardware FIFOs (see Fig.3.10). This was achieved by exploiting the `/dev/mem`device to map the AXI bus in virtual memory and allowing an application to directly access TX/RX FIFOs and their occupancy counters. In this setup there was no support for interrupts, system-wide locking and virtual-to-physical address translation when handling the receive part of a RDMA operation. We then developed a Linux kernel-space device driver supporting a `proc`file-system entry for outputting device status and debug information, together with the output of the internal configuration/status registers.

The module also parses the device tree to find the IRQ number associated to the APEnet Crossbar Switch and then assigns a callback function (`irq_handler`) to handle the interrupt request generated by the arrival of new data.

Interrupts are handled both in the "top half" by a kernel module callback function, and in the "bottom half" by using a work-queue: to this end, `_karma_wq_schedule`wraps the

call to the standard `queue_delayed_work`kernel space function to wait the arrival of the new data, *i.e.* when the interrupt rises, the driver starts polling for the new data:

```
void _karma_wait_data(struct delayed_work *work)
karma_dev_t *xdev= container_of(work, karma_dev_t, delayed_work);
    int reg;
    volatile int timeout_j = jiffies + msecs_to_jiffies(TOUT_ms);

    //loop while the new data arrives or timeout
    while((reg=karma_reg_read(PAYLOAD_COUNT_REGISTER))==0){
        if (!time_is_before_eq_jiffies(timeout_j)){
        ERROR("TIMEOUT!");
        goto exit_timeout;
        }
    }

    //new data arrived, handle it
    karma_recv(xdev);

    exit_timeout:
        karma_enable_irq(xdev); //enable the interrupts
```

Listing 3.1: Pseudo-code implementation of `_karma_wait_data`

`karma_recv`is in charge of asynchronously copying the data just landed into the pre-registered user buffer (see below); if the buffer is not registered, it will use a bounce buffer instead:

```
static int karma_recv(karma_dev_t *xdev){
    header_t header;
    footer_t footer;
    int num_payloads,i=0;
    karma_buffer_t *buf=NULL;
    payload_t *payload;

    karma_header_read(header); //consumes header

    //this is the size of the arriving packet
    num_payloads = header.s.packet_size;
```

```
    //naive implementation of payload words read
    for(i=0;i<num_payloads;i++){
        karma_payload_read(payload[i]);
    }
    karma_footer_read(footer); //consume the footer
    karma_set_buffer_ready(buf); //ready for the user
}
```

Listing 3.2: Pseudo-code implementation of `karma_recv`

Functions `karma_footer_read`, `karma_header_read`and `karma_payload_read`reads the data from the memory mapped FIFOs.

The userspace application then needs to register/unregister data buffers and the kernel space module needs to keep track of the registered buffers to copy the new data on arrival. This is done by using the linked lists data structures built it in the kernel. When a new packet arrives (or when the user needs to delete a buffer) we need to search in the list of the buffers. Once the buffer is found in the list of preregistered buffers, it can be used to store the new data. The userspace application asynchronously polls for the arrival of new data. This is translated into a kernel "request for new data" call. The device driver searches the buffers list and checks the "ready" bit for that particular buffer requested by the user (set by the kernel driver). If it is set, the new buffer is ready to be consumed by the user application.

Fig. 3.11 shows the normal execution of a generic send/receive test execution using the kernel space module. The kernel-space test is as follows: in the sending phase, the kernel-space module copies data from the source buffer to a kernel bounce buffer, then prepares header and footer and writes them onto the corresponding FIFOs. The receiving phase is just the opposite: arriving data are copied into a kernel bounce buffer (waiting for the user-space process to request them) while header and footer are "consumed".

In the kernel-space test, the performance is heavily affected by the user-/kernel-space traversal, influencing the total time required for the send/receive. This also affects the fluctuations, as shown in Fig. 3.13, where latency decreases moving from 16 to 32 payload Bytes.

Since `clock_gettime`calls are not accurate enough at this time scale, a large number of packets is sent ($\sim$100000), averaging the time to get an estimate of the latency for a single send call.

Because of the aforementioned suboptimal bounce-buffer mechanism and the notoriously slow interrupt handling by GNU/Linux, a user space ping-pong test is provided (see Fig.3.12) exploiting `/dev/mem`to directly access the memory-mapped hardware to assess the system

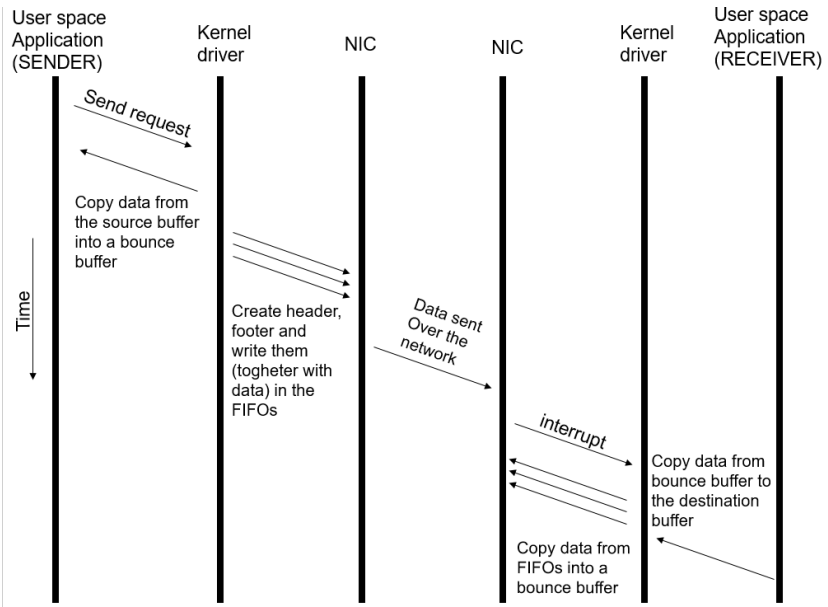


Figure 3.11: Generic send/receive test execution using the KARMA kernel driver.

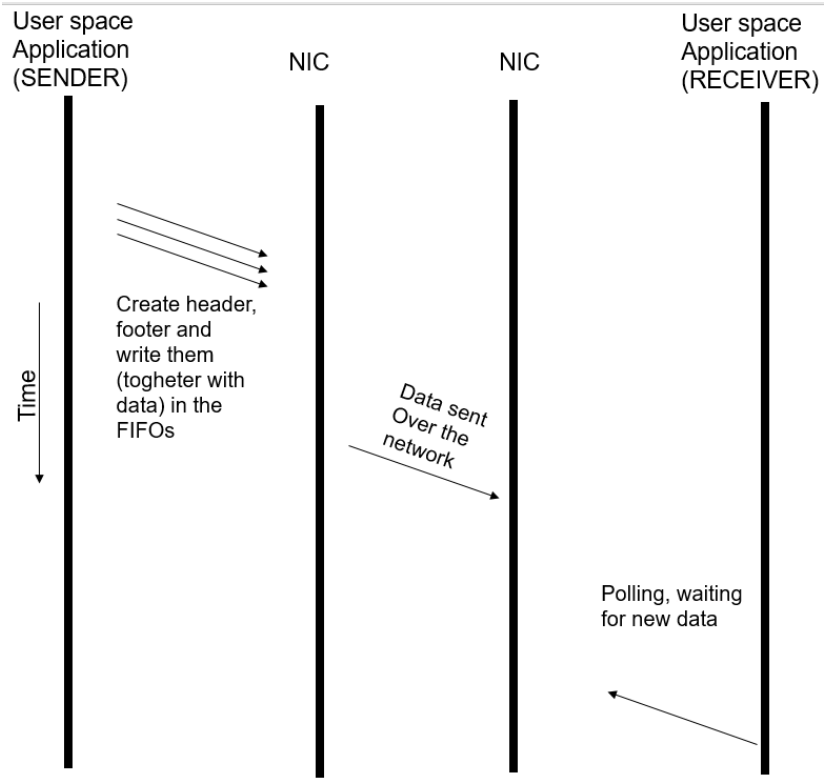


Figure 3.12: Generic send/receive test execution using the KARMA user space software.

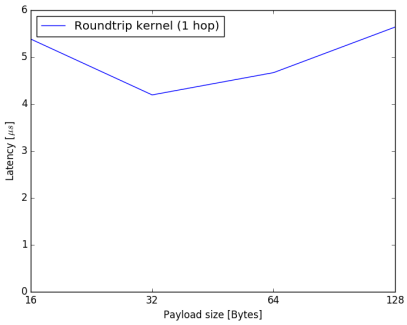


Figure 3.13: Round-trip latencies between two boards using KARMA kernel driver: the performance is heavily affected by the user-/kernel-space traversal.

latency. In Fig. 3.14, the results are shown for both direct connection between sender and receiver, and through an intermediate hop. Times spent by ARM for intra-tile port reading ($\sim 0.4\mu$s, about 20 clock cycles per word) and writing ($< 0.1\mu$s, 4 clock cycles per word) are independent from the number of hops. Differences in time measurements in the two cases provide an estimate of the contribution of single hop traversal to total latency (0.46 μs).

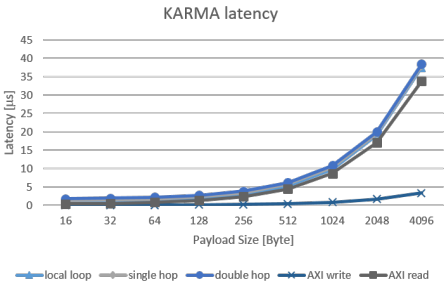


Figure 3.14: KARMA test framework latency.

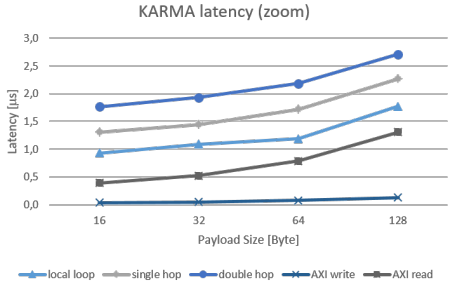


Figure 3.15: Zoom of KARMA test framework latency.

Current KARMA does not implement any DMA-access to the intra-tile ports, so that ARM must issue a write for every single word into header/data FIFOs, which is obviously suboptimal for bandwidth but appropriate for gauging the latency of small-sized packets.

3.4 ExaNeSt project status

The ExaNeSt project started at 1Q16 and foresees two consecutive, 18 months-long time slots. The first phase, ended in July 2017, focused on the design of general system architecture and the realisation of the main hardware building blocks. During the second phase, the project consortium will integrate, test and evaluate the delivered system prototype through

application benchmarks. The current status of ExaNeSt hardware blocks is:

- The QFDB has been designed and it is under production; first release is expected for 4Q17.
- The Mezzanine board, hosting up to 4 QFDBs and/or thermal mockups for power dissipation analysis will be released few weeks after the QFDB.
- The design of QFDB FPGA firmware has been started on a Xilinx FPGA development kit. We procured a number of Trenz(R) Zynq UltraScale+ systems (TE0808-03ES2-S) to implement an emulator of final ExaNeSt hardware.
- Small-size clusters based on Trenz FPGAs, interconnected via 10 Gbps custom links, have been installed: (i) to test and evaluate performance of the ARM V8 multi-core programmable systems and HSS transceivers and (ii) to deliver a first release of the ExaNeSt network, based on INFN APEnet router and Forth Unimen-based network interface.

On the basis of the project budget and expected cost of the components at 4Q18, the final prototype will be made of an assembly of a dual liquid cooled chassis hosting $\sim$ 50 QFDBs (*i.e.* 200 FPGAs) distributed on 9 Mezzanines per chassis and interconnected via the brand new custom ExaNeSt network architecture based on multiple 10 Gbps channels.

We are clearly in a crucial phase of the project and the results are just preliminary. Such an ambitious project could not be concluded in three years and its natural follow-up has been foreseen in some projects that have already started or are about to.

Concluding remarks

The employ of GPUs and FPGAs in the field of HPC is a relatively new topic of great interest in the scientific community.

Systems based on reprogrammable components as FPGAs allow a reduction in costs and in development time by providing solutions that can be easily repurposed for different applications, not only the one for which they were originally designed.

In this dissertation I addressed some of the most challenging issues of using accelerators in FPGA-based networks for different application scenarios.

In the first part of this thesis, an analysis of the critical issues of HPC-dedicated interconnects is presented; APEnet+, a 3D Torus direct interconnect based on an FPGA NIC, is therein described in detail.

My contribution to APEnet+ initially concerned the design of a stable Linux Device Driver for the PCI Express Gen3 FPGA-based NIC (APEnet+ V5), whose results showed encouraging improvements with respect to the previous Gen2 card version (APEnet+ V4). I then focused on the optimization of the device driver in order to improve the network performance experienced by the user application. This activity has deepened my knowledge of the system stack software architecture and of the relationship between user space and kernel space programming, allowing me to devise and implement specific solutions to enhance the overall network performance. The developed software, leveraging on ad-hoc features introduced in the FPGA NIC design, was able to efficiently exploit the PCI Express Gen3 capabilities with a $\sim$70% gain in bandwidth, passing from 2.8 Gb/s of APEnet+ V4 to 4.2 Gb/s of the V5 version. Nevertheless, I experienced an increase in latency for small packets (with sizes ranging from 32 B to 2 kB): $\sim$5 us of APEnet+ V5 compared to the $\sim$4 us of the previous version. I believe this could be due to the 3^{rd}-party IP core used or to the PCIe Gen3 protocol itself.

The project is still under development and can be optimized with new functions and technologies. For example, NVIDIA announced the release, in future CUDA versions, of the "GPUDirect Async" feature; this should allow the optimization of inter-node GPUDirect communications by providing a safe mechanism to issue commands from the GPU directly to the NIC, meaning RDMA communications directly triggered by the GPU itself. We are eager to see the documentation of these new APIs in order to put them to use in improving the network efficiency.

In the second part of this dissertation, the NaNet project was introduced.

The NaNet-10 board and the related software stack are currently under test in a prototype system deployed at the NA62 [17] experiment at CERN, where they receive raw data from

one of the detectors through a 10GbE channel and DMA-copy them into the GPU memory via the PCIe bus. This data transport mechanism allows for a low and predictable communication latency and enables real-time processing of physics data in the GPU, boosting the efficiency of the experiment data selection system (the so-called trigger). Besides the data transport, NaNet-10 implements a processing stage on the data stream (*i.e.* data decompression and re-alignment) in order to ease the data processing on the GPU. The resulting system can be viewed a heterogeneous stream processing pipeline, where NaNet-10 is also in charge of sending the processed data to its consumers. The measured total latency of this heterogeneous processing pipeline and its stability endorse the NaNet framework as a real-time data stream processing platform.

In this context, my work has focused on the best way to implement the sending phase of this "GPU-processed" data in order to attain the lowest latency, which, in this peculiar latency-bound environment, is by far the single, most important performance index.

A novel approach based on a "persistent" CUDA kernel was presented in the Sec.2.4.1, where handling of the NIC is demanded to the GPU without the intervention of the host CPU. I tested this new approach in a synthetic environment to reproduce the NA62 CERN experiment flow — gather data, process it and then send it over an Ethernet connection. This has yielded encouraging results in terms of a latency reduction of about 10%. Besides these results, the most important achievement is having demonstrated the feasibility of using a CUDA kernel, usually confined to number crunching, to drive the NIC. This is of particular relevance in real-time applications.

In the last part of the thesis I described the ExaNeSt project, a European initiative aiming at developing the system-level interconnect, a fully-distributed NVM storage and the cooling infrastructure for an ARM-based Exascale-class supercomputer. This is a relatively different scenario with respect to the projects presented in the first two sections since the computing node architecture relies on the AXI system bus rather than PCIe. My contribution in the ExaNeSt project was mainly on the architectural design of the prototype node, especially for what regards the network subsystem and its software stack. To this end, I am collaborating on the hardware/software co-design and on the NIC device driver to test and verify the functionalities of the hardware under development. I started by developing a kernel-based device driver and a set of user-space test functions to stimulate the network hardware. This setup highlighted the heavy effect that user-/kernel-space traversals bear on the communication latency, leading to the development of a user-space only test framework, the so-called "KARMA" Test Framework (introduced in Sec. 3.3), that roughly halved the measured ping-pong latencies for a 16 Bytes packet in zero-load conditions. In the future I will continue working at INFN (Istituto Nazionale di Fisica Nucleare) APE Lab, where all

my research activity was carried out during my PhD. There I had the invaluable opportunity to directly interact with the hardware designers, in this way being involved in the hardware/-software co-design while working actively on the design and optimization of the device driver and software interface of the different projects I am participating in. This allowed me to study and implement architectural specifications that can provide innovative features, ensuring maximum efficiency (high bandwidth and low latency) for intra- and inter-node communications.

As a concluding remark, this work has made its contribution in FPGA-based networks together with experimental results that show a great potential for further research activities.

Acknowledgment

This work was carried out within the ExaNeSt project, funded by the European Union Horizon 2020 research and innovation programme under grant agreement No 671553.

Bibliography

[1] G. E. Moore, "Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]," vol. 20, pp. 36 – 37, 10 2006.

[2] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc, "Design of ion-implanted mosfets with very small physical dimensions," *IEEE J. Solid-State Circuits*, p. 256, 1974.

[3] Accessed: 2017-02-02. [Online]. Available: http://www.top500.org

[4] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, Jan 2015.

[5] *The OpenPOWER Foundation*, accessed: 2017-10-24. [Online]. Available: http://openpowerfoundation.org/

[6] *The CCIX Consortium*, accessed: 2017-10-24. [Online]. Available: http://www.ccixconsortium.com

[7] O. Lawlor, "Message passing for gpgpu clusters: Cudampi," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 01 2009, pp. 1–8.

[8] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending mpi to accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD '11. New York, NY, USA: ACM, 2011, pp. 19–23. [Online]. Available: http://doi.acm.org/10.1145/2377978.2377981

[9] A. Venkatesh, K. Hamidouche, S. Potluri, D. Rossetti, C.-H. Chu, and D. Panda, "Mpi-gds: High performance mpi designs with gpudirect-async for cpu-gpu control flow decoupling," 08 2017, pp. 151–160.

[10] *NVIDIA GPUDirect technology*, accessed: 2017-02-02. [Online]. Available: https://developer.nvidia.com/gpudirect

[11] M. Albanese, P. Bacilieri, S. Cabasino, N. Cabibbo, F. Costantini, G. Fiorentini, F. Flore, A. Fonti, A. Fucci, M. Lombardo, S. Galeotti, P. Giacomelli, P. Marchesini, E. Marinari, F. Marzano, A. Miotto, P. Paolucci, G. Parisi, D. Pascoli, D. Passuello, S. Petrarca, F. Rapuano, E. Remiddi, R. Rusack, G. Salina, and R. Tripiccione, "The

ape computer: An array processor optimized for lattice gauge theory simulations," *Computer Physics Communications*, vol. 45, no. 1, pp. 345 – 353, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/001046558790172X

[12] C. Battista, S. Cabasino, F. Marzano, P. S. Paolucci, J. Pech, F. Rapuano, R. Sarno, G. M. Todesco, M. Torelli, W. Tross, P. Vicini, N. Cabibbo, E. Marinari, G. Parisi, G. Salina, F. D. Prete, A. Lai, M. P. Lombardo, R. Tripiccione, and A. Fucci, "The ape-100 computer: (i) the architecture," *International Journal of High Speed Computing*, vol. 05, no. 04, pp. 637–656, 1993. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129053393000268

[13] F. Aglietti, A. Bartoloni, C. Battista, S. Cabasino, M. Cosimi, A. Michelotti, A. Monello, E. Panizzi, P. Paolucci, W. Rinaldi, D. Rossetti, H. Simma, M. Torelli, P. Vicini, N. Cabibbo, W. Errico, S. Giovannetti, F. Laico, G. MagazzÃź, and R. Tripiccione, "The teraflop supercomputer apemille: architecture, software and project status report," *Computer Physics Communications*, vol. 110, no. 1, pp. 216 – 219, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S001046559700180X

[14] F. Belletti, S. F. Schifano, R. Tripiccione, F. Bodin, P. Boucaud, J. Micheli, O. Pene, N. Cabibbo, S. de Luca, A. Lonardo, D. Rossetti, P. Vicini, M. Lukyanov, L. Morin, N. Paschedag, H. Simma, V. Morenas, D. Pleiter, and F. Rapuano, "Computing for lqcd: apenext," *Computing in Science Engineering*, vol. 8, no. 1, pp. 18–29, Jan 2006.

[15] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Hardware and Software Design of FPGA-based PCIe Gen3 interface for APEnet+ network interconnect system," *Journal of Physics: Conference Series*, vol. 664, no. 9, p. 092017, 2015. [Online]. Available: http://iopscience.iop.org/article/10.1088/1742-6596/664/9/092017

[16] R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, F. L. Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, L. Pontisso, D. Rossetti, F. Simula, M. Sozzi, L. Tosoratto, and P. Vicini, "A multi-port 10GbE PCIe NIC featuring UDP offload and GPUDirect capabilities." *Journal of Physics: Conference Series*, vol. 664, no. 9, p. 092002, 2015. [Online]. Available: http://stacks.iop.org/1742-6596/664/i=9/a=092002

[17] The NA62 Collaboration, "2013 NA62 status report to the CERN SPSC," CERN, Geneva, Tech. Rep. CERN-SPSC-2013-009. SPSC-SR-115, Mar 2013. [Online]. Available: http://na62.web.cern.ch/NA62/Home/Home.html

[18] *The ExaNeSt project*, accessed: 2017-10-24. [Online]. Available: http://http://exanes.eu/

[19] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "GPU Peer-to-Peer Techniques Applied to a Cluster Interconnect," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 806–815.

[20] H. Fröning and L. Oden, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *IEEE International Conference on Cluster Computing 2013*, Indianapolis, IN, USA, 2013. [Online]. Available: http://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/2013cluster-ggas_public.pdf

[21] S. Neuwirth, D. Frey, and U. Bruening, "Communication models for distributed intel xeon phi coprocessors," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2015, pp. 499–506.

[22] S. Neuwirth, D. Frey, M. Nuessle, and U. Bruening, "Scalable communication architecture for network-attached accelerators," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 627–638.

[23] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 36–43.

[24] H. Fröning, M. Nüssle, H. Litz, and U. Brüning, "A case for FPGA based accelerated communication," in *Proceedings of the 2010 Ninth International Conference on Networks*, ser. ICN '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 28–33. [Online]. Available: http://dx.doi.org/10.1109/ICN.2010.13

[25] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale

datacenter services," *Commun. ACM*, vol. 59, no. 11, pp. 114–122, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2996868

[26] C. Álvarez, E. Ayguadé, J. Bosch, J. Bueno, A. Cherkashin, A. Filgueras, D. Jiménez-González, X. Martorell, N. Navarro, M. Vidal, D. Theodoropoulos, D. N. Pnevmatikatos, D. Catani, D. Oro, C. Fernández, C. Segura, J. Rodríguez, J. Hernando, C. Scordino, P. Gai, P. Passera, A. Pomella, N. Bettin, A. Rizzo, and R. Giorgi, "The axiom software layers," *Microprocess. Microsyst.*, vol. 47, no. PB, pp. 262–277, Nov. 2016. [Online]. Available: https://doi.org/10.1016/j.micpro.2016.07.002

[27] R. Giorgi, "Axiom: A 64-bit reconfigurable hardware/software platform for scalable embedded computing," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, June 2017, pp. 1–1.

[28] D. Mayhew and V. Krishnan, "Pci express and advanced switching: evolutionary path to building next generation interconnects," in *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, Aug 2003, pp. 21–29.

[29] Z. Pang, M. Xie, J. Zhang, Y. Zheng, G. Wang, D. Dong, and G. Suo, "The th express high performance interconnect networks," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 357–366, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11704-014-3500-9

[30] W. An, X. Jin, X. Du, and S. Guo, "A flexible fpga-to-fpga communication system," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, Jan 2016, pp. 586–591.

[31] A. T. Markettos, P. J. Fox, S. W. Moore, and A. W. Moore, "Interconnect for commodity FPGA clusters: Standardized or customized?" in *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, 2014, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/FPL.2014.6927472

[32] J. Anderson, A. Borga, H. Boterenbrood, H. Chen, K. Chen, G. Drake, M. Donszelmann, D. F. is, B. Gorini, D. Guest, F. Lanni, G. L. Miotto, L. Levinson, J. Narevicius, A. Roich, S. Ryu, F. S. euder, J. Schumacher, W. Vandelli, J. Vermeulen, W. Wu, and J. Zhang, "Felix: The new approach for interfacing to front-end electronics for the atlas experiment," in *2016 IEEE-NPSS Real Time Conference (RT)*, June 2016, pp. 1–2.

[33] T. Kuhara, T. Kaneda, T. Hanawa, Y. Kodama, T. Boku, and H. Amano, "A preliminarily evaluation of peach3: A switching hub for tightly coupled accelerators," in *Comput-*

ing and Networking (CANDAR), 2014 Second International Symposium on, Dec 2014, pp. 377–381.

[34] Y. Thoma, A. Dassatti, and D. Molla, "Fpga2: An open source framework for fpga-gpu pcie communication," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–6.

[35] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An efficient and flexible host-fpga pcie communication library," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–6.

[36] "Altera Nios Processor Reference Handbook." [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[37] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "Gpunet: Networking abstractions for gpu programs," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 201–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685065

[38] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "APEnet+: a 3D Torus network optimized for GPU-based HPC systems," *Journal of Physics: Conference Series*, vol. 396, no. 4, p. 042059, 2012. [Online]. Available: http://stacks.iop.org/1742-6596/396/i=4/a=042059

[39] R. Ammendola *et al.*, "Virtual–to–Physical address translation for an FPGA–based interconnect with host and GPU remote DMA capabilities," in *Field-Programmable Technology (FPT), 2013 International Conference*, Dec 2013, pp. 58–65.

[40] J. T. Anderson *et al.*, "FELIX: a High-Throughput Network Approach for Interfacing to Front End Electronics for ATLAS Upgrades," CERN, Geneva, Tech. Rep. ATL-DAQ-PROC-2015-014, May 2015. [Online]. Available: http://cds.cern.ch/record/2016626

[41] P. Durante, N. Neufeld, R. Schwemmer, G. Balbi, and U. Marconi, "100 Gbps PCI-Express readout for the LHCb upgrade," *Journal of Instrumentation*, vol. 10, no. 04, p. C04018, 2015. [Online]. Available: http://stacks.iop.org/1748-0221/10/i=04/a=C04018

[42] J. Mitra *et al.*, "Common Readout Unit (CRU) - a new Readout Architecture for ALICE Experiment," *Journal of Instrumentation*, 2015, in preparation. [Online]. Available: https://indico.cern.ch/event/357738/session/11/contribution/209/attachments/1160497/1677069/CRU_TWEPP_15.pdf

[43] V. Halyo, A. Hunt, P. Jindal, P. LeGresley, and P. Lujan, "GPU enhancement of the trigger to extend physics reach at the LHC," *Journal of Instrumentation*, vol. 8, no. 10, p. P10005, 2013. [Online]. Available: http://stacks.iop.org/1748-0221/8/i=10/a=P10005

[44] D. Emeliyanov and J. Howard, "Gpu-based tracking algorithms for the atlas high-level trigger," *Journal of Physics: Conference Series*, vol. 396, no. 1, p. 012018, 2012. [Online]. Available: http://stacks.iop.org/1742-6596/396/i=1/a=012018

[45] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Virtual–to–Physical address translation for an FPGA–based interconnect with host and GPU remote DMA capabilities," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 58–65.

[46] R. Ammendola, A. Biagioni, O. Frezza, G. Lamanna, A. Lonardo, F. Lo Cicero, P. S. Paolucci, F. Pantaleo, D. Rossetti, F. Simula, M. Sozzi, L. Tosoratto, and P. Vicini, "NaNet: a flexible and configurable low-latency NIC for real-time trigger systems based on GPUs," *Journal of Instrumentation*, vol. 9, no. 02, p. C02023, 2014. [Online]. Available: http://stacks.iop.org/1748-0221/9/i=02/a=C02023

[47] F. Daoud, A. Watad, and M. Silberstein, "Gpurdma: Gpu-side library for high performance networking from gpu kernels," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '16. New York, NY, USA: ACM, 2016, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/2931088.2931091

[48] "Cuda apis," accessed: 2017-10-31. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[49] L. Oden, H. Froning, and F.-J. Pfreundt, "Infiniband-verbs on gpu: A case study of controlling an infiniband network device from the gpu," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 976–983.

[50] "Gpudirect async," accessed: 20/Oct/2017. [Online]. Available: https://github.com/gpudirect/libgdsync

[51] M. Katevenis *et al.*, "The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug 2016, pp. 60–67.

[52] R. V. Aroca and L. M. G. GonÃğalves, "Towards green data centers: A comparison of x86 and {ARM} architectures power efficiency," *Journal of Parallel and Distributed Computing*, vol. 72, no. 12, pp. 1770 – 1780, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512002122

[53] R. P. Luijten and A. Doering, "The dome embedded 64 bit microserver demonstrator," in *Proceedings of 2013 International Conference on IC Design Technology (ICICDT)*, May 2013, pp. 203–206.

[54] N. Rajovic *et al.*, "Supercomputing with commodity cpus: Are mobile socs ready for hpc?" in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[55] "ExaNoDe," accessed: 2017-02-02. [Online]. Available: http://exanode.eu/

[56] I. Mavroidis *et al.*, "Ecoscale: Reconfigurable computing and runtime system for future exascale systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 696–701.

[57] R. Capuzzo-Dolcetta, M. Spera, and D. Punzo, "A fully parallel, high precision, n-body code running on hybrid computing platforms," *Journal of Computational Physics*, vol. 236, pp. 580 – 593, 2013. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0021999112006900

[58] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, p. 1105, 2005. [Online]. Available: http://dx.doi.org/10.1111/j.1365-2966.2005.09655.x

[59] P. Monaco, T. Theuns, and G. Taffoni, "The pinocchio algorithm: pinpointing orbit-crossing collapsed hierarchical objects in a linear density field," *Monthly Notices of the Royal Astronomical Society*, vol. 331, no. 3, p. 587, 2002. [Online]. Available: http://dx.doi.org/10.1046/j.1365-8711.2002.05162.x

[60] T. Theuns *et al.*, "Swift: Task-based hydrodynamics and gravity for cosmological simulations," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*, ser. EASC '15, 2015, pp. 98–102.

[61] P. S. Paolucci *et al.*, "Dynamic many-process applications on many-tile embedded systems and HPC clusters: The EURETILE programming environment and execution platforms," *Journal of Systems Architecture*, vol. 69, pp. 29–53, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762115001423

[62] M. Januszewski and M. Kostur, "Sailfish: A flexible multi-gpu implementation of the lattice boltzmann method," *Computer Physics Communications*, vol. 185, no. 9, pp. 2350 – 2368, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465514001520

[63] "OpenFOAM," accessed: 2017-02-02. [Online]. Available: http://openfoam.org/

[64] "RegCM," accessed: 2017-02-02. [Online]. Available: http://www.ictp.it/research/esp/models/regcm4.aspx

[65] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[66] "MonetDB," accessed: 2017-10-24. [Online]. Available: https://www.monetdb.org/

[67] M. Marazakis *et al.*, "Euroserver: Share-anything scale-out micro-server design," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 678–683.

[68] "KALEAO," accessed: 2017-10-24. [Online]. Available: https://www.kaleao.com/Products/kmax

[69] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 77–88.

[70] R. Ammendola *et al.*, "APEnet+ 34 Gbps Data Transmission System and Custom Transmission Logic," *Journal of Instrumentation*, vol. 8, no. 12, p. C12022, 2013. [Online]. Available: http://stacks.iop.org/1748-0221/8/i=12/a=C12022

[71] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "A Hierarchical Watchdog Mechanism for Systemic Fault Awareness on Distributed Systems," *Future Generation Computer Systems*, vol. 53, pp. 90 – 99, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X14002751

[72] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, A. Salamon, G. Salina, F. Simula, L. Tosoratto, and P. Vicini, "APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters," *Journal of Physics: Conference Series*, vol. 331, no. 5, p. 052029, 2011.