Operating System Mechanisms for Remote Resource Utilization in ARM Microservers

John Velegrakis

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes Campus, Heraklion, GR-70013 Greece

Thesis Advisor: Prof. *Manolis Katevenis*

This work was performed in the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7th Framework Programme under the EuroServer (FP7-ICT-610456) project.

University of Crete
Computer Science Department

Operating System Mechanisms for Remote Resource Utilization in ARM Microservers

Thesis submitted by
John Velegrakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
John Velegrakis

Committee approvals: _____
Manolis GH Katevenis
Professor, Thesis Supervisor

_____
Angelos Bilas
Professor, Committee Member

_____
Dionisios N. Pnevmatikatos
Professor, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, March 2015

Abstract

Recent efforts towards performance and power optimization in large-scale Data Centers have brought the use of Microservers in the forefront. Compared to traditional architectures, Microservers consist of smaller, less power-hungry Compute Units (CUs) compared to traditional architectures. The key concept is that by integrating such smaller CUs in high numbers, the resulting many-core system can achieve high multi-threaded performance, while maintaining a low power profile. In such an environment, expensive resources must be shared among CUs, since it is costly and impractical to dedicate one to each CU. However, the management of these resources requires to implement sharing mechanisms in the Operating System (OS) and the software stack.

In this work, we investigated and implemented OS and user space software mechanisms that are necessary for the deployment of ARM-based CUs in such large-scale systems. We address the sharing of remote resources by fully exploiting the underlying hardware features, such as Remote Direct Memory Access (DMA) and Remote Load/Store. In particular, we have implemented software mechanisms to (1) enable access to remote memory and (2) allow usage of a shared virtualized 10 Gbps Network Interface Card (NIC) by several CUs simultaneously.

Remote memory access is implemented in three different ways: (1) As an extension of the local DRAM of the CUs, (2) As a remote Swap Device, and (3) as an I/O character device accessed directly from user space. We demonstrate that remote memory can be used effectively without performance penalty in a system running a full OS.

The sharing of a virtualized 10Gbps NIC is achieved by a kernel network driver, that we have implemented, which enables utilization of the customized hardware as a standard Ethernet Device. This allows legacy applications that use Berkeley Sockets to run unmodified. The network driver makes use of scatter-gather DMA for fast zero-copy packet transmission and reception and operates in Full-Duplex mode by using two independent DMA channels. Additionally, it supports Interrupt Coalescing and management of the MAC and PHY hardware blocks using the Management Data Input/Output (MDIO) interface.

In conclusion, this work shows that we can indeed utilize a system built upon ARM-based CUs that were not originally designed to operate in such an environment, by the sharing of remote and shared resources by the Linux OS and its user space environment. We expect this work to become even more relevant with upcoming 64-bit ARM-based platforms, targeting large-scale servers for Data Centers.

Περίληψη

Οι πρόσφατες προσπάθειες για βελτιστοποίηση της απόδοσης αλλά και της κατανάλωσης ισχύος των μεγάλης κλίμακας Δατα ἐντερς έφερε τους λεγόμενους Μικροσερερς στο προσκήνιο, οι οποίοι αποτελούνται από μικρότερης κατανάλωσης Υπολογιστικές Μονάδες (ΥΜ) συγκρινόμενοι με τις παραδοσιακές αρχιτεκτονικές. Η κεντρική ιδέα είναι ότι χρησιμοποιώντας μεγάλο αριθμό από τέτοιες ΥΜ μπορούμε να κατασκευάσουμε μία πολυπύρηνη μηχανή που θα έχει υψηλή απόδοση στις πολυνηματικές εφαρμογές και παράλληλα θα έχει χαμηλή ε νεργειακή κατανάλωση. Σε τέτοιες μηχανές, οι ακριβοί πόροι αναγκαστικά διαμοιράζονται μεταξύ των ΥΜ, καθώς δεν γίνεται να αποδοθούν πόροι για κάθε ΥΜ ξεχωριστά. Ωστόσο, η διαμοίραση των πόρων αυτών απαιτεί την υλοποίηση κατάλληλων μηχανισμών στο Λειτουργικό Σύστημα (ΛΣ) και γενικότερα στο επίπεδο του λογισμικού.

Σε αυτήν την εργασία, ερευνήσαμε και υλοποιήσαμε μηχανισμούς στο ΛΣ και στο επίπεδο διεργασιών, οι οποίοι είναι απαραίτητοι για την χρησιμοποίηση ΥΜ βασισμένων σε αρχιτεκτονικές ARM σε μεγάλης κλίμακας συστήματα. Υλοποιήσαμε την διαμοίραση πόρων δύο ειδών, χρησιμοποιώντας τους διαθέσιμους μηχανισμούς του υλικού, όπως η Remote Load/Store και Remote Load/Store. Συγκεκριμένα, υλοποιήσαμε μηχανισμούς για την απομακρυσμένη προσπέλαση μνήμης και την χρησιμοποίηση μιας κοινής και virtualized 10 Gbps διεπαφής δικτύου η οποία μπορεί να χρησιμοποιείται από πολλές ΨΜ ταυτόχρονα.

Η χρησιμοποίηση της απομακρυσμένη μνήμης υλοποιήθηκε με τους εξής ακόλουθους τρόπους: (1) Σαν επέκταση της τοπικής DRAM μίας ΥΜ, (2) Σαν απομακρυσμένη συσκευή για Swap, και (3) Σαν απομακρυσμένη συσκευή χαρακτήρων που χρησιμοποιείται απευθείας από τις διεργασίες. Δείχνουμε ότι η χρήση της απομακρυσμένης μνήμης σε ένα σύστημα με ΛΣ δεν επιφέρει μείωση της απόδοσης.

Η από κοινού χρήση της διεπαφής δικτύου 10 Gbps επιτυγχάνεται με ένα οδηγό στο ΛΣ, τον οποίο υλοποιήσαμε και ο οποίος επιτρέπει στο ΛΣ να βλέπει την διεπαφή αυτή σαν κλασσική διεπαφή Ethernet. Αυτό είναι απαραίτητο για να μπορούν να τρέξουν διεργασίες που χρησιμοποιούν τα Berkeley Sockets, χωρίς να χρειάζεται να τροποποιηθούν. Ο οδηγός χρησιμοποιεί Scatter/Gather DMA για γρήγορη και zero-copy μετάδοση και παραλαβή πακέτων και λειτουργεί σε Full-Duplex χρησιμοποιώντας δύο ξεχωριστά κανάλια της DMA μηχανής. Επιπλέον, υποστηρίζει μηχανισμούς Interrupt Coalescing και μπορεί να διαχειριστεί τα MAC και PHY μέρη του υλικού μέσω του πρωτοκόλλου Management Data Input/Output (MDIO).

Η εργασία αυτή δείχνει ότι πράγματι μπορούμε να φτιάξουμε ένα σύστημα βασισμένο σε ΥΜ αρχιτεκτονικής ARM, οι οποίες δεν είναι σχεδιασμένες να λειτουργούν σε ένα τέτοιο σύστημα. Πιστεύουμε ότι αυτή η εργασία θα γίνει ακόμα πιο σημαντική στο μέλλον, καθώς νέες 64-μπιτες πλατφόρμες ARM εμφανίζονται, οι οποίες στοχεύουν να χρησιμοποιηθούν σε μεγάλης κλίμακας Data Centers.

Acknowledgements

This work was performed in the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7th Framework Programme under the EuroServer (FP7-ICT-610456) project.

I would like to thank the Institute of Computer Science at FORTH for support my research, by providing access to equipment and financial support.

Furthermore, I would also like to thank the Computer Architecture and VLSI Laboratory hardware team and my advisor Prof. Manolis Katevainis. Special thanks to Dr. Manolis Marazakis and Mr. Giorgos Kalokairinos for their support in my work.

John Velegrakis, Staff Sergeant, Hellenic Army Reserve

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research in Computer Science and especially in the Data Center domain, has started to include energy consumption as a performance figure of merit. An important amount of the total budget accounts for the energy consumption of the data center that includes power consumption of the cooling infrastructure and the servers themselves (OPEX [1]) and also the construction and installation of those subsystems (CAPEX [2]) as well (Figures 1.1 and 1.2). So, it is essential to quantify and measure that cost and also find new methods and technologies for its reduction as well. As a result, computer scientists and processor architects started talking about performance of CPUs per Watt, and from now on every evaluation of a system includes such figures of merit that address energy consumption.

Therefore, growing amount of research effort in the Computer Architecture domain investigates new technologies that can reduce energy consumption costs, while preserving system performance compared to older or modern energy hungry systems. New processor architectures started to appear, demanding a large portion of the Data Center and High Performance Computing market, for example ARM processors, new RAM and Flash technologies. As shown in Figure 1.3 a rise of microserver units in the data center domain is expected, as more and more of those systems are produced and purchased. While these technologies were in the past mainly used in applications where energy consumption is a crucial factor, such as Embedded Systems, Cell Phones, Sensors, Controllers, etc, the latest breeds of these technologies start being used in large scale systems like Data Centers and HPC, without suffering from performance loss compared to traditional technologies (Intel x86, x64).

The key concept is that smaller and less-energy hungry processors can be assembled in large numbers in the same amount of space that traditional processors require and achieve the same performance. Each of those compute units is inferior in terms of performance compared to traditional processors. However, the aggregate

[1] Operating Expense: ongoing cost for running a product, business, or system.

[2] Capital Expenditures: are used by a company to acquire or upgrade physical assets such as equipment, property, or industrial buildings.

system performance of the system remains similar to the traditional systems, due to the larger number of cores. As a result, they must fully exploit multi-threaded and scalable workloads, as they cannot achieve sing-threaded performance. Thus, scalability is of great importance, when designing and implementing such a system, both in hardware and software levels. The hardware implementation must utilize a hierarchical organization of the whole system, partitioning groups of cores into same packages, server blades, etc and connecting them together with an appropriate hierarchical network. Similarly, the system software stack must fully utilize the underlying hardware mechanisms and allow common workloads to run on those systems transparently. Scalability is also of great importance in the system software implementation as well.

Due to the large number of cores in such a system - and the physical packaging - placement and management of resources (especially I/O) becomes a great concern and challenge for hardware and software designers, as resources cannot be dedicated to every compute node. It is very expensive and infeasible in terms of IC and PCB packaging to place such resources for each compute node. Instead, expensive and high-performance resources are distributed among groups of compute nodes. The correct management and secure sharing of these resources is of great significance and as a result efficient hardware and software mechanisms are required.

Except from sharing expensive I/O devices, even the main memory (DRAM) placement can become an issue when designing the physical packages of such systems and as a result, one may not be able to place enough DRAM memory for each compute node. In order not to suffer from performance loss, it is preferable to also share components such as the main memory as well, instead of using shared storage devices.

A big portion of the resource sharing problems burdens the system software, that means the Operating System and the User Space environment. It must simultaneously allow unmodified workloads to run in such systems and fully exploit and manage the underlying hardware features in a secure way. Unfortunately, the progress of software development and the software that exists in the wild is very poor for such systems, especially for those that consist of ARM processors, compared to that available for the traditional x86 systems. Currently, software development efforts for ARM architectures target the deployment of such processors in mobile and embedded devices. Systems software for ARM processors targeting large-scale Data Centers or HPC is still in its infancy.

A Discrete Prototype hardware platform that was constructed for the Euroserver Project and is explained in Chapter 3 was used for the development and evaluation of the software mechanisms that deal with remote memory usage and sharing of a virtualized 10 Gbps Network Interface Card (NIC). The prototype consists of platforms with ARM processors, several FPGAs and interconnection circuits and will be explained in detail in a following chapter.

1.1 Contributions

In this work, we designed and developed Operating System and User Space mechanisms, that are essential for the deployment of ARM processors in a large scale modern Data Center. These mechanisms address the use of local and remote resources in a many-core ARM system, which may not be in the same integrated chip and they even may not be connected by a network that allows coherency. As a result, these Mechanisms must work well in processors that belong to different Coherence Islands (CIs - not connected by a coherent network) or Compute Nodes (CNs).

Special research effort has been dedicated to the usage of remote RAM memory in a variety of ways and also for the secure use of a shared virtualized network interface by a group of Compute Nodes as well. For remote memory in particular, we implemented three mechanisms for its efficient usage, depicted in the list below as items (1), (2) and (3). The first (1) mechanism deals with the extension of the available physical memory that the Operating System sees, consisting of Local RAM and segments of other Compute Nodes' RAM. With the second (2) way, remote memory is used as a swap device, used by the Operating System only when the local RAM is full. These two methods, require the OS to do all the management of physical memory, leaving user space applications unaffected since they do not sense the presence of remote memory, so they do not need to modify their API calls. The last way (3), addresses the usage of remote memory by user space applications explicitly. The applications must be developed or modified for that purpose and can access remote memory either via Load/Store instructions, RDMA operations of I/O system calls.

Finally, I implemented a Linux kernel driver (4) that allows sharing of a high-speed network interface 10 Gbps optical NIC, that is virtualized and shared among the nodes. Sharing of a network interface is important for web services, where many Compute Nodes have the same entry/exit point to the Internet.

List of contributions:

1. Remote memory as main memory extension
2. Remote memory as swap device
3. Remote memory as an I/O character device
4. Linux network driver for a virtualized 10 Gbps Network Interface

The rest of this work is organized as follows: In Chapter 2 we explore related work that exists in the fields of Remote Memory and Network Interface sharing,

in Chapter 3 we describe the hardware testbed (Discrete Prototype) upon which the software mechanisms are implemented. In Chapter 4 we describe the usage of remote memory in a full system with Operating System and evaluate our implementation with some micro-benchmarks in Chapter 5. Next, in Chapter 6 we describe the sharing of the 10 Gbps NIC and the implementation of the network driver for the Linux kernel and we present an evaluation of network performance. Finally, we conclude in Chapter 7.

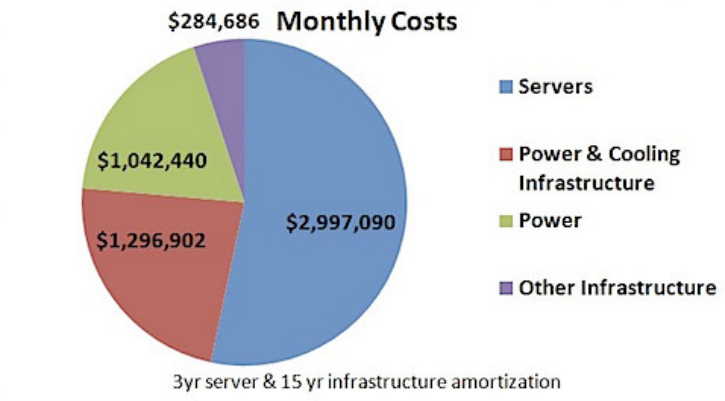

Figure 1.1: Data Center costs breakdown. *Source: James Hamilton, Amazon Web Services*

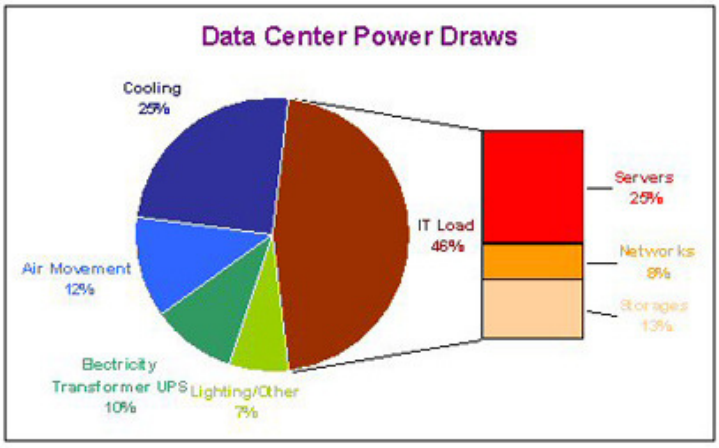

Figure 1.2: Data Center power draws. *Source: IMEX Research*

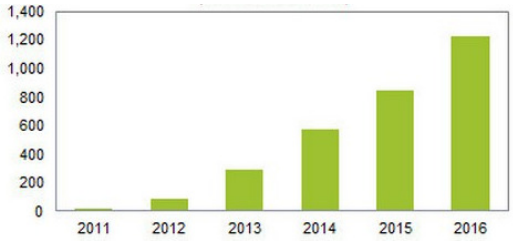

Figure 1.3: Worldwide microserver shipment forecast (Thousands of Units). *Source: IHS iSuppli Research, February 2013*

Chapter 2

Related Work

2.1 Remote Memory

As large-scale parallel and distributed systems evolve, efficient hardware and software mechanisms for using physical memory are investigated. Since more and more processors are deployed in the same Integrated Chip [26] [34], the placement of DRAM memory modules becomes an issue. It is simply unfeasible to dedicate a large amount of DRAM arrays to each core or processor. Furthermore, the ever increasing number of compute nodes in a large-scale distributed system or a NOW (Network of Workstations) makes the sharing of even the physical memory of the nodes unavoidable, especially for large-scale workloads. There are plenty of remote memory implementations that exists, but the great majority of those address the traditional x86 architectures. Additionally, the Operating Systems support for these architectures is huge compared to the new ARM processors. Our work, differs mainly on that remote memory utilization is implemented for the ARM architecture using custom hardware interconnection. There is poor systems software support for ARM architecture and is even poorer when addressing the use of such processors in the Microserver or Data-Center domain. That is a result of the past application target of the ARM microprocessors that are embedded and mobile devices, which consist of a limited number of cores and do not run the workloads that are found in data centers.

In [21] and [30] the authors implemented a remote memory swap device over traditional Ethernet network in a Network of Workstations. The swap device resides in the remote workstation's physical memory as a ramdisk device and is used when the local physical memory is getting full. The network that connects the nodes in this distributed system must be fast enough in order to benefit from the remote swap over the network, compared to the usage of the hard disk. Our remote swap mechanism does not use traditional Ethernet network, but it rather uses our custom interconnect that has a unified address space utilizing Physical Address Translation as described later. However, we also implemented a similar mechanism using the Network Block Device (nbd) [6] for remote swapping to compare its per-

formance with our approach. In [31] the authors use a similar approach for remote swapping, as the previous work described. They implement remote swap over traditional Ethernet or InfiniBand network in traditional 64 bit Intel architectures with 64 bit Operating System. Authors of [33] also implement on-demand remote swapping over an ATM network in a cluster that runs a Data Mining workload.

The FaRM system described in [19], uses unified physical address space for compute nodes in a cluster and utilizes RDMA to Read/Write to remote memory. However, they created a programming model (API) for applications to use, thus running commonly available applications without modifying their source code is unfeasible. In [29] the authors address the usage of remote memory in the Data Center domain, where physical memory blades are added to the racks and used as remote swap devices. The memory modules in those racks can be accessed by all compute nodes' OS and user space processes transparently, with the use of custom hardware subsystems. In the same domain of Data Centers, authors in [40] implement an optimized runtime system, specifically for Phoenix, that is a MapReduce runtime. The runtime automatically manages concurrency and locality in a cluster with shared memory.

In [23] the authors investigate and optimize remote memory accessing using the MPI3 [22] runtime system and utilizing RDMA operations, in a modern large-scale cluster.

There are many implementations that utilize RDMA either with special APIs or transparently, in order to increase performance of communication between workstations. Their target application is either remote memory, remote resource access or general intra-cluster communication. They are implemented above commonly used network interconnects, such as Ethernet, InfiniBand, etc, or they use custom hardware. All of these systems target traditional x86/x64 architectures with the corresponding commonly available operating systems Linux/Windows.

Virtual Interface Architecture (VIA) [16] is an abstract model of a user-level zero-copy network, utilizing RDMA, and is the basis for InfiniBand, iWARP and RoCE. RoCE or RDMA over Converged Ethernet [25] a network protocol that allows remote direct memory access (RDMA) over an Ethernet network and iWARP [5] is a computer networking protocol that implements RDMA for efficient data transfer over Internet Protocol networks. The Sockets Direct Protocol (SDP) [10] maintained by OpenFabrics Alliance, is a transport-agnostic protocol to support stream sockets over Remote Direct Memory Access (RDMA) network fabrics. Authors in [18] have implemented the SDP Protocol in the Microsoft Windows Operating System.

2.2 Shared Network Interface

Because the current trend in cloud computing is overcommitment of resources using multiple Virtual Machines, great amount of efforts has been spent investigating virtualization of resources, especially storage and sharing of those resources among

several Virtual Machines. Sharing of resources is also an important issue in data centers where large numbers of compute units must necessarily share expensive peripherals.

The most common virtualization and sharing of the same physical resource is SR-IOV [8], that is a protocol specification for both hardware and software and targets the PCI and PCI Express devices' virtualization. SR-IOV is device independent as long as the device connects to a PCI/PCIe bus and supports Virtual Functions. Network interface sharing can be achieved as specified in [28], where each Virtual Machine sees a dedicated network interface using the Intel VT-d subsystem. OpenStack [7] also support virtualization of Network Interfaces using SR-IOV as described in [11].

Virtio [35] is a popular software mechanism for virtualized I/O resources among multiple virtual machines and is current used by the KVM virtual machine hypervisor.

Authors in [20] and [15] investigate hardware mechanisms for enabling Network on Chip (NoC) interface sharing among multiple processor cores, that belong to the same coherence island (CPU), with special attention to fault tolerance issues. Network on Chip optimization for ARM platforms is done in [36], where the authors implement a Network Interface above the ARM AXI protocol and support different network topologies.

Chapter 3

Discrete Prototype

The implementation of a small-scale system by the Computer Architecture and VLSI Laboratory in ICS, FORTH for the Euroserver Project, is a proof of concept that we can, indeed, build a microserver consisting of ARM processors. In this Discrete Prototype, several mechanisms were implemented both in hardware and software, which are essential for a microserver. The term *Discrete Prototype* is derived from the fact that the system consists of parts that are available in the market today, without the need of custom Integrated Chips

To this moment, two generations of Discrete Prototypes have been constructed. Remote memory usage was implemented and tested in Generation 1 Prototype, while sharing of the virtualized network interface was implemented in Generation 2. In the near future, all hardware features and software mechanisms will be employed in the Generation 2 Prototype.

3.1 Discrete Prototype Generation 1

Figure 3.1 depicts a Generation 1 Discrete Prototype block diagram and in Figure 3.2 the actual system is shown. It consists of two Compute Nodes, connected together back-to-back with a custom interconnect. Zedboards assume the role of Compute Nodes in this prototype. Furthermore, each Zedboard is also a Coherence Island, since there is not any coherent interconnection between the boards. Thus, in our prototype the terms Compute Node and Coherence Island mean the same thing and can be used interchangeably. Avnet Zedboards [2] are populated with a Xilinx Zynq 7000 SoC that contains a dual core ARM Cortex A-9 processor and a ZC020 FPGA that runs at 100 MHz. The FPGA implements all the essential hardware mechanisms, such as the interconnection between the two Zedboards, physical address translation for remote access and some other features that help the software that runs in each Zedboard. ARM Cortex A-9 CPUs run at 667 MHz and have 512 MB of DRAM. They allow external communication with other hardware modules or co-processors by the use of master ports GP0 and GP1, ACP (Accelerator Coherence Port) and HP (High Performance) slave ports [13].

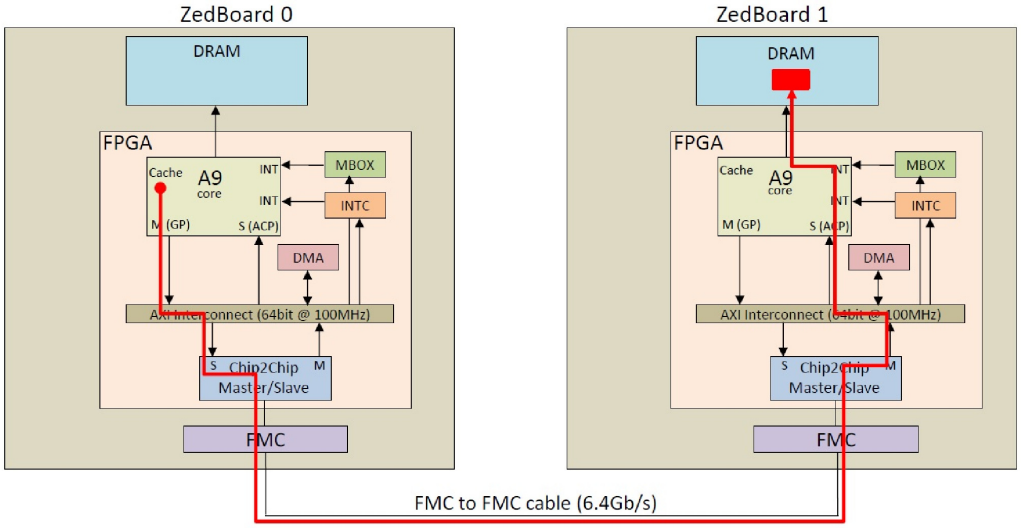

Figure 3.1: Generation 1 Prototype using two Zedboards

Additionally, the CPU chiplet contains an Interrupt Controller, the GIC (Global Interrupt Controller) [38] that handles all interrupt requests and disseminates them to the appropriate core.

The FPGA of each Zedboard contains a block that implements the interconnection logic, connecting the two Compute Nodes that enables remote memory access. Furthermore several essential and useful peripherals such as DMA and Mailbox are implemented. The interconnection logic uses 4 ports of the ARM processor, which are available in the Zynq 7000 SoC chip and is compliant to the **AXI protocol (Advanced Extensible Interface)** [14]. Its also contains submodules for the interconnection of the two Zedboards with an FMC-to-MC cable that consists of 16 LVDS pairs. Figure 3.3 gives a detailed view of the connections of several FPGA blocks with the ARM processor inside each Zedboard.

The Chip2Chip FPGA block, which is seen in Figure 3.3, is the logic that allows us to pass AXI Requests through the FMC-to-FMC cable to the other Zedboard using LVDS (Low Voltage Differential Signal) signaling. We will not go into further details of this circuit in this work.

The Direct Memory Access engine we use for data transfers between the two memories is the Xilinx Central DMA or CDMA [37]. It can operate in a simple mode, transferring a large chunk of continuous data or in scatter-gather mode, which allows transfers of multiple chunks of data, residing in different parts of memory. Remote DMA or RDMA operations in our prototype are very efficient, since the DMA engine understands the AXI protocol and creates requests of interleaved AXI Read/Write bursts.

Mailbox is a hardware mechanism that allows transmission of small messages from one Zedboard to the other and the creation of a remote interrupt upon transmission completion.

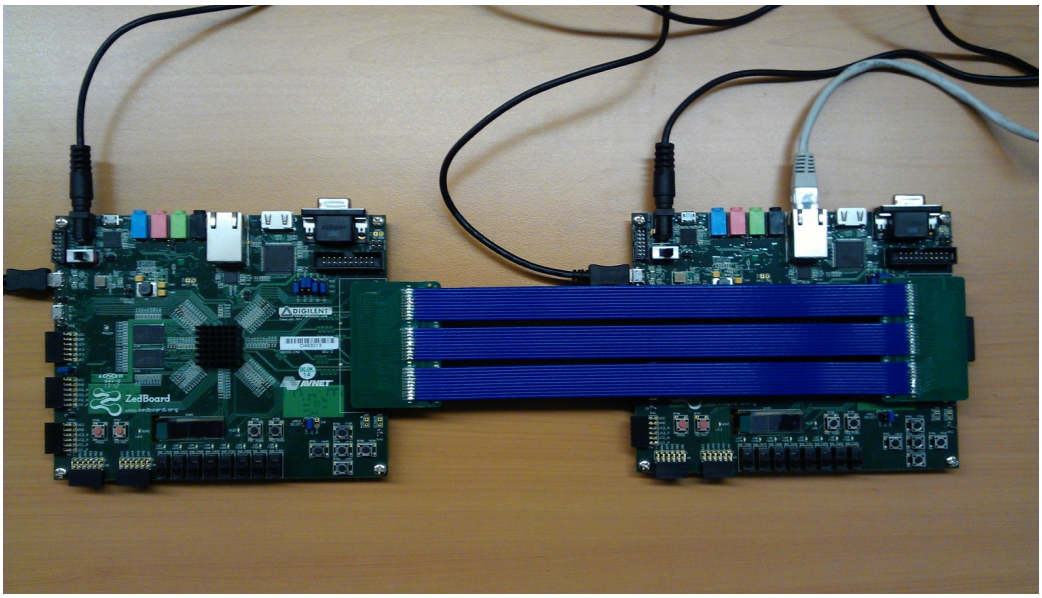

Figure 3.2: Physical View of the Gen.1 Prototype

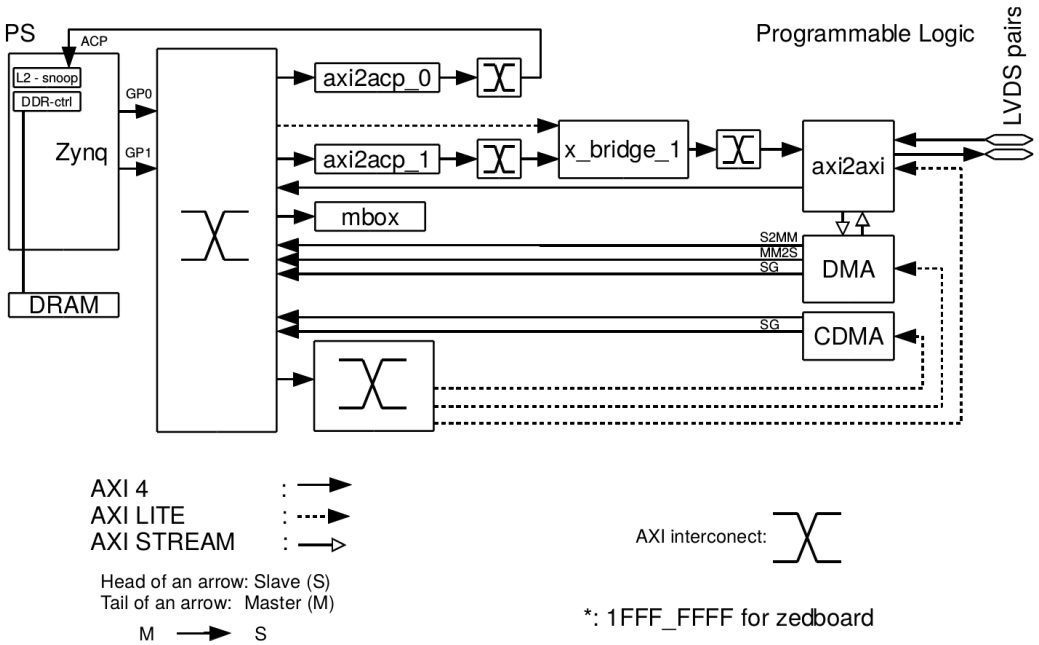

Figure 3.3: Detailed Hardware Blocks & Protocols Diagram of Gen.1 Prototype

A remote memory access flow and the hardware components that are involved are seen in Figure 3.1 and Figure 3.3 in more detail. GP0 and GP1 master ports of the ARM processor produce AXI requests when a the processor issues a Load/Store

to a remote physical address. ACP and HP slave ports, receive external AXI Requests that end up as Read/Writes in the physical memory. Figure 3.4 shows a typical connection of two modules (i.e. CPU and DSP processor) using the AXI protocol. Each AXI Operation consists of a Request from the master module and a Response from the slave module. For example, a Store instruction to a remote memory address (DRAM of the other Compute Node) that is issued by the ARM processor in our prototype, is passed through our interconnection as an AXI Read Request. The other node's ARM processor (in specific the Snoop Control Unit or the DRAM controller subsystem) responds back with an AXI Response with the corresponding data or error status.

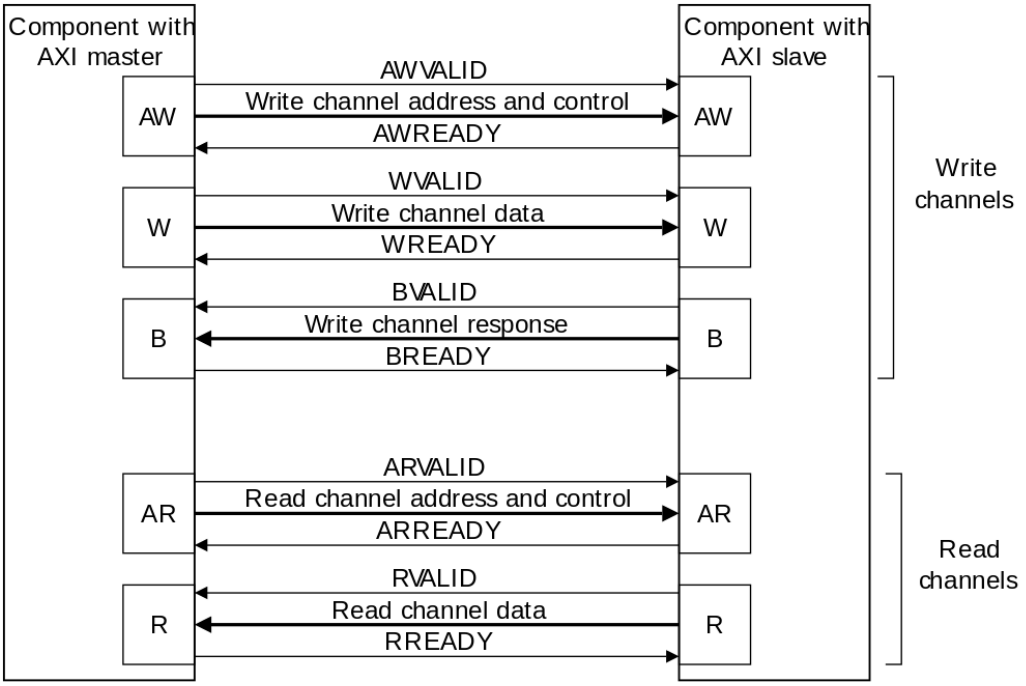

Figure 3.4: The AXI Protocol. *Source: ARM RealView ESL API v2.0 Developer's Guide*

ACP and HP slave ports of a remote Compute Node can both be used for accessing remote memory. When the ACP port is used, the AXI requests pass through the cache coherence system (Snoop Control Unit) of the ARM processor, while HP port allows AXI requests to reach the DRAM controller directly.

Hardware Block Details

In Figure 3.3 the various hardware blocks involved for the interconnection of the two Compute Nodes are shown. The flow of a remote Load/Store is the following: The initiating processor produces an AXI 3 32 bit request at GP0 master port. It must be converted to AXI 4 64 bit protocol via the AXI interconnect FPGA block.

The request then passes throigh some smaller interconnects and a bridge that is used for AXI Request ID conversion and reaches the *axi2axi* block that drives the 15 LVDS pairs of the FMC-to-FMC cable. Once the request passes that block it is outside of the Compute Node 0. When it reaches Compute Node 1 through the cable it passes the *axi2axi* and then undergoes an AXI protocol conversion again. It is converted from AXI 4 64 bit request (than was produces in the FPGA of Compute Node 0) to an AXI 3 64 bit request. because the ACP slave port of the processor works only with 64 bit AXI 3 requests.

The AXI response signals for that request go all the way back to Compute Node 0 than initiated the AXI request (signals *BVALID* for read and *WVALID* for write requests, as shown in Figure 3.4).

3.1.1 Physical Address Translation

The Physical Address Translation FPGA block, allows us to to map segments of DRAM memory of Compute Node 1 to Compute Node 0 physical address space and vice versa. Each mapped segment is accessible by the ARM processor as a valid physical memory range.

We can use either ACP or HP ports to access the memory of Compute Node 1 from Compute Node 0. This Physical Address Translation block is configurable at FPGA circuit synthesis time and any required memory mapping can be created. However there is an limitation in the ARM Cortex-A9, that prohibits mappings into physical addresses below 0x40000000. The ARM processor does not produce AXI requests in the GP slave ports, when the processor issues Load/Stores in physical addresses lower than 0x40000000. As a result, we can only create physical memory mappings in the range above that limit.

Figure 3.5 depicts the process of accessing the higher 256 MByte of DRAM memory of Compute Node 1 from Compute Node 0, using the Physical Address Translation through the ACP port. Compute Node 1's memory segment is mapped at physical address range 0x40000000 - 0x5FFFFFFF of Compute Node 0. When the processor of Compute Node 0 issues a Load/Store instruction at a physical address inside this mapped range, an AXI Request is generated by that processor's subsystem at comes out of the GP port GP port. Then the request passes through our Physical Address Translation block and converts it to a new AXI Request targeting the memory of Compute Node 1, with the appropriate new destination physical addresses that is valid in the physical address range inside Compute Node 1. In Figure 3.5, address 0x50000000 of Compute Node 0 is translated to address 0x10000000 of Compute Node 1. That translation is necessary, in order physical addresses to be valid and understandable by both nodes.

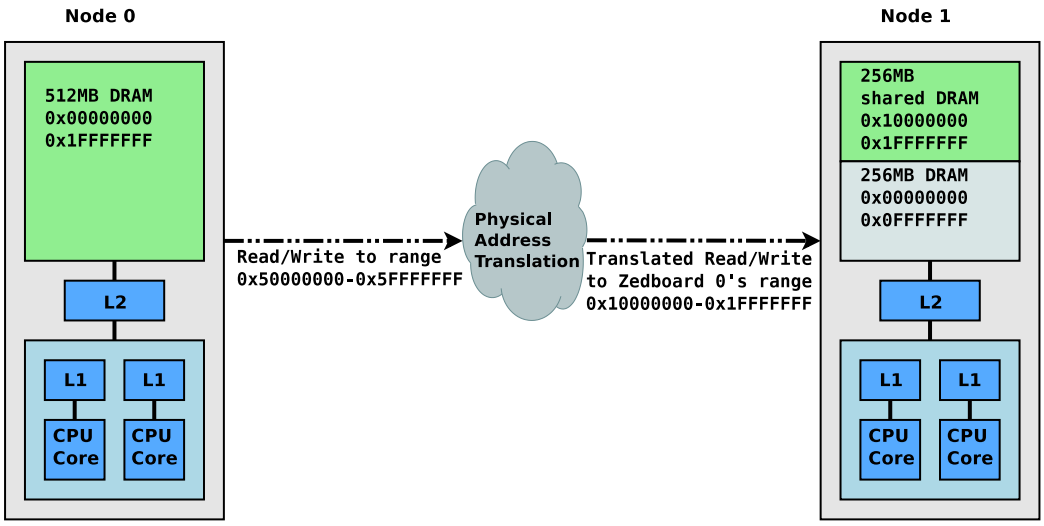

Figure 3.5: Physical address translation.

3.2 Discrete Prototype Generation 2

Discrete Prototype Generation 2, consists of 4 to 8 Compute Nodes, connected together with a custom interconnection network through a central main board. This time, the discrete Compute Nodes are the Microzed boards [1] connected into a HiTech Global main board, with a custom PCB designed by FORTH. Microzeds are populated with Xilinx Zynq-7000 SoC, which consists of an ARM Cortex-A9 dual core processor, 1 GByte of DRAM and a ZC7020 FPGA. HiTech Global main board contains a Xilinx Virtex-7 FPGA and a 10 Gbps Network Interface (or NIC) along with its optical transceiver. Figure 3.6 shows a hardware block diagram of the Gen. 2 prototype with 4 Compute Nodes (Microzeds) and Figure 3.7 shows a photograph of the actual prototype with 8 Microzeds connected.

The FPGA of the main board implements the 10 Gbps MAC Layer associated with the physical Network Interface. **The 10 Gbps NIC is shared and accessible by all Compute Nodes connected to the main board**.

The custom interconnection network consists of hardware blocks that reside in the FPGAs of each Microzeds and the main board, utilizing dedicated LDVS pairs for each Microzed. The multiplexing of traffic from/to different Microzeds is done in the main board. Load/Stores or DMA operations end up as AXI Read/Write Requests as described in Section 3.1. Furthermore, Physical Address Translation is implemented in the main board, to allow connected Compute Nodes to access peripherals that reside in the main board. Physical Address Translation is done the same way as in Section 3.1.

The Xilinx AXI DMA engine is used by each Microzed to transmit and receive Layer 2 Ethernet frames to/from the 10Gbps MAC in the main board. This DMA engine differs from the CDMA engine described in Section 3.1, because it does not

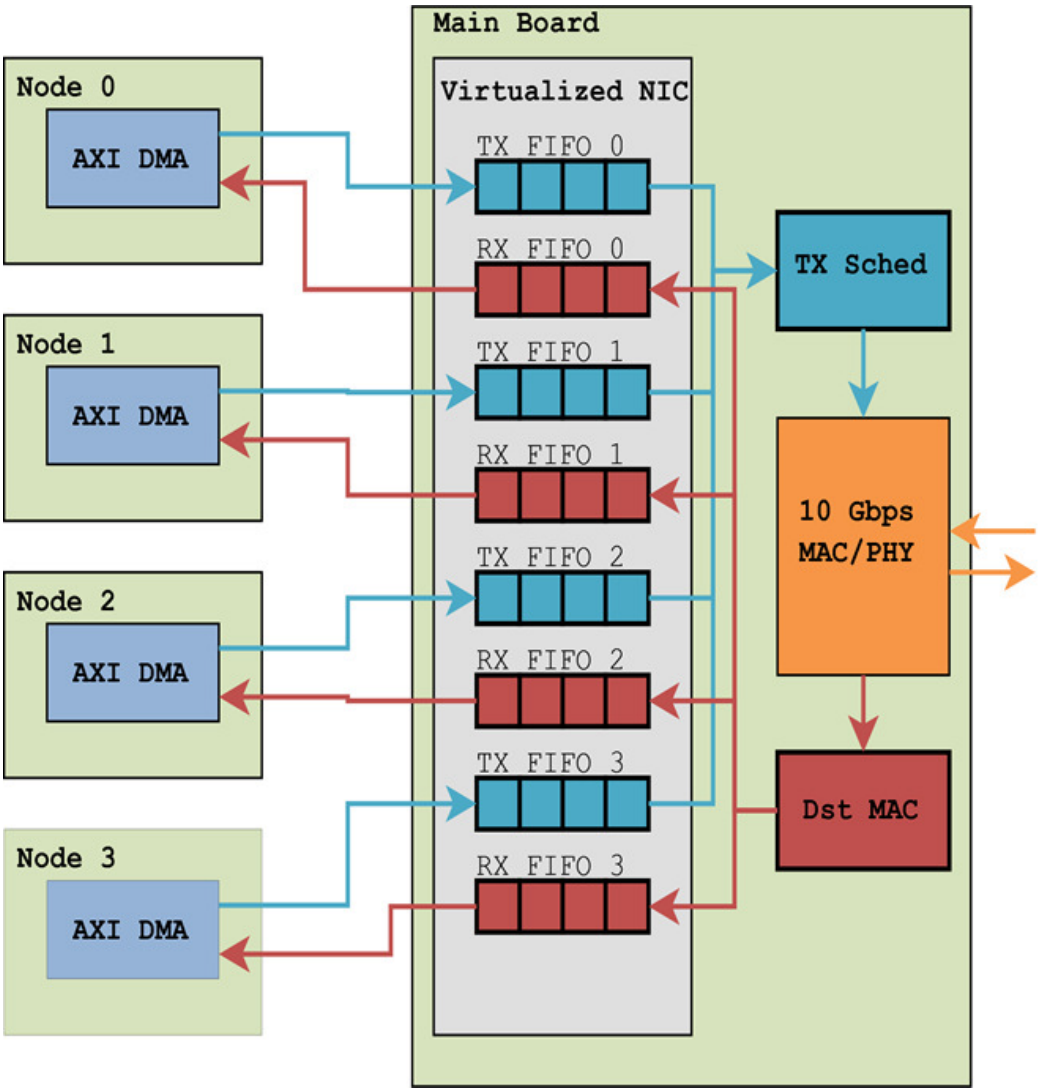

Figure 3.6: Gen.2 Prototype Hardware Block Diagram

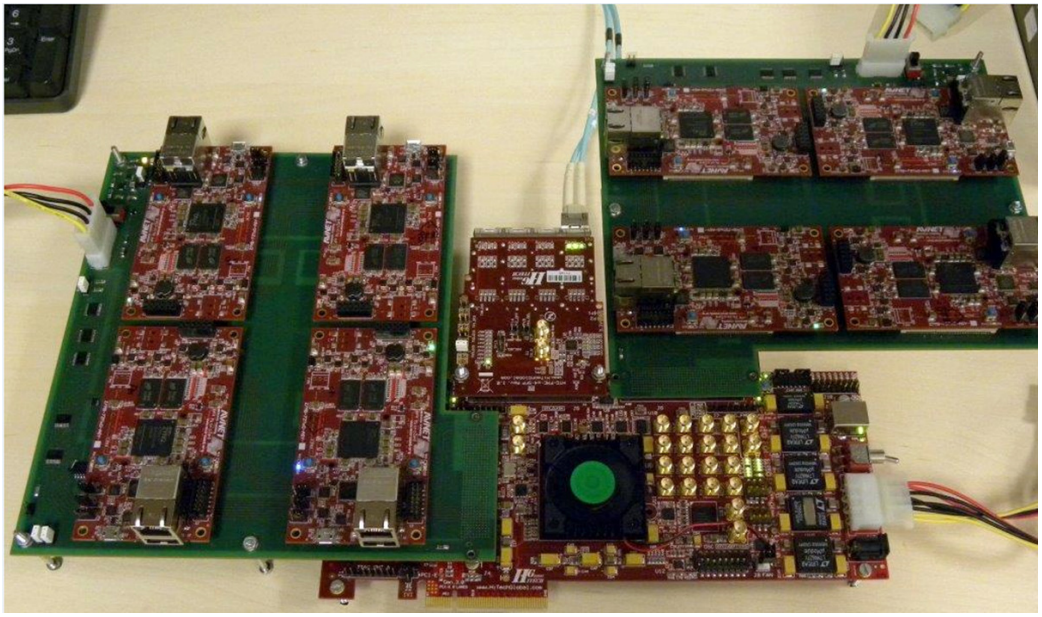

Figure 3.7: Physical View of the Gen.2 Prototype

transfer data from memory address to memory address. Xilinx AXI DMA does not accept destination memory address as an argument because the destination is not a memory controller, but a custom hardware module. Furthermore, it consists of two independent Data Movers - one for transmission and one for reception. The separation of TX/RX paths enables Full-Duplex data transfers when using the AXI DMA engine.

3.3 Hardware Virtualization

It is an essential prerequisite for the Network Interface to be virtualized in order the connected Compute Nodes to share it and transmit/receive packets properly. In our Discrete Prototype the NIC is virtualized by hardware assistance, in the following way: each Compute Node connected to the main board has a dedicated pair of TX/RX FIFOs that store Ethernet Frames that are transmitted/received from/to that node, as shown in Figure 3.6. A TX scheduler implemented in the central FPGA consumes the TX FIFOs in a **round-robin** fashion [1]. Finally, The TX scheduler passes the consumed frames to the 10 Gbps MAC. In a similar way, when frames arrive to the 10 Gbps MAC it pushes them into the appropriate RX FIFO according to the Destination MAC address field of the Ethernet header of each frame. Because of the round-robin scheduling algorithm, the nodes share the available bandwidth of the NIC evenly.

[1] Future work in the Euroserver project includes implementing a Quality-of-Service mechanism in the TX and RX paths

It is essential that nodes have different MAC addresses, that are known to the 10 Gbps MAC in order to differentiate network traffic between different nodes. Furthermore, different MAC addresses allow a remote end point to distinguish the Compute Nodes.

The configuration registers of the MAC are **not virtualized**[2]. This means that a change in the MAC settings by a Compute Node, will have a global effect in the system, affecting all connected Compute Nodes.

[2]In the same fashion the PCI/PCIe device virtualization in x86/x64 architectures works.

Chapter 4

Remote Memory

Remote memory usage is very important for microserver environments, where each Compute Node has limited local physical memory (RAM). Workloads that maintain a large set of their Working Set in memory, in order not to suffer from I/O latency, will experience issues in a system that has limited amount of physical memory. If the physical memory is not large enough to hold this Working Set, the I/O issue rises again, since the Working Set will be swapped into a storage device. As a result, remote memory usage to overcome this problem of limited local RAM is essential. Workloads like these are very common in Data Centers, like Data Analytics.

We would like to use remote memory segment from other Compute Nodes, when they do not need them, instead of using a slow storage device. Essential condition to gain benefit from remote memory usage is that the latency and throughput that applications will experience using this remote memory will be far better than using a storage device. In the following subsections, we describe how we utilized remote memory in a full system with Operating System and User Space environment.

4.1 Page Borrowing & Caching Policy

With the aforementioned we end up with the term *Remote Page Borrowing*, meaning that a Compute Node can borrow a memory page (or frame) from another Compute Node to expand its available physical memory. Since page borrowing occurs between **different Coherence Islands**, we must predefine its way of operation in order to avoid driving the system to inconsistency. Recall that different Coherence Islands are not connected with any network or bus that allows coherency between processors.

Two are the secure ways of implementing Page Borrowing:

1. Remote page can be cached only in the caches of the user Compute Node. (Figure 4.1).

2. Remote page can be cached in the owner Compute Node.(Figure 4.2).

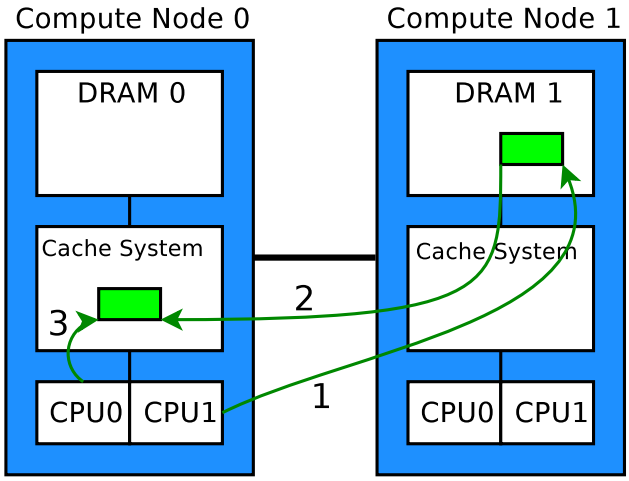

Figure 4.1: User Cached Page Borrowing

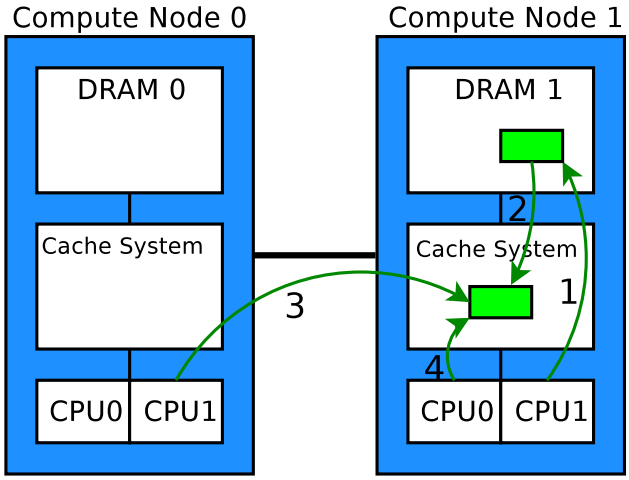

Figure 4.2: Owner Cached Page Borrowing

The advantages of the 1st way are emerged when a Compute Node needs to borrow a page from another Compute Node the time the second does not use it. In the scenario where we have only one page user Compute Node which uses almost the same data most of its lifetime, the 1st way increases latency and throughput, since every access to the borrowed page is done to its local cache. Only the first access will pass through the interconnection to the remote memory to access the remote page and bring it to the user's cache. Figure 4.1 depicts an operation, where Compute Node 0 accesses a remote page (Step 1) residing the DRAM of Compute Node 1 via the interconnection network, brings it to its cache system (Step 2) and finally the subsequent accesses are done to its local cache (Step 3).

The 2nd way addresses occasions where we want a page to be shared simul-

taneously by the two Compute Nodes. Then, it surely is more fair to keep that page into the cache of the owner. It is fair, when the two Compute Nodes, need that page for the same amount of time. The remote user will inevitably experience the latency of remote access through the interconnection network. Accesses from the remote user must not bring that shared page into the remote user's cache, so he must use that page as 'uncached'. In Figure 4.2, Compute Node accesses a shared page that resides in its local DRAM (Step 1) and brings it in its own cache. Then, Compute Node 0 accesses the shared page copy through the interconnection network to the remote cache system via the ARM ACP Port, without bringing the page to its cache (Step 3). Finally, Compute Node 0 accesses again the shared page from its local cache, because the cached copy is not dirty (Step 4).

There is another way of implementing sharing of a memory page that uses the ARM HP port. Accesses that pass through the HP port bypass the cache coherence system and talk directly to the DRAM controller. Although, both Compute Nodes can use the shared page as 'uncached' by accessing it directly from a DRAM, they will not benefit from the performance the cache systems can give. **In this work, we focus only on 1st way of Remote Page Borrowing.**

4.2 Remote Memory and the Operating System

In a modern microserver system, a full Operating System and user space environment is needed. Thus, it is essential to explore the ways we can utilize remote physical memory, without compromising security or require a huge software development effort by the user space application programmer. Remote physical memory utilization in an environment like this can be utilized in three different ways. They differ in the way the Operating System and the user space applications see and manage the available remote physical memory.

1. Use of remote physical memory as an extension of the available physical memory, seen by the Operating System.(Section 4.3
2. Use of remote physical memory as a swap device, managed by the Operating System.
3. Use of remote physical memory as an I/O character device. With the use of a kernel driver, the remote memory access and management is done by the user space applications or a user space runtime system.

Any software application (OS, user space program, or bare metal program) that runs in a processor can access memory in two ways. Either with Load/Store instructions or with DMA operations. I/O Read/Write system calls by a user-space application also end up as Load/Store instructions or DMA operations implemented by the low level kernel drive, which manages these system calls for a specific device.

In our system each Compute Node runs its own Operating System that is a Linux kernel 3.6.0 as provided by Digilent, Inc. We use the higher 256 MByte of

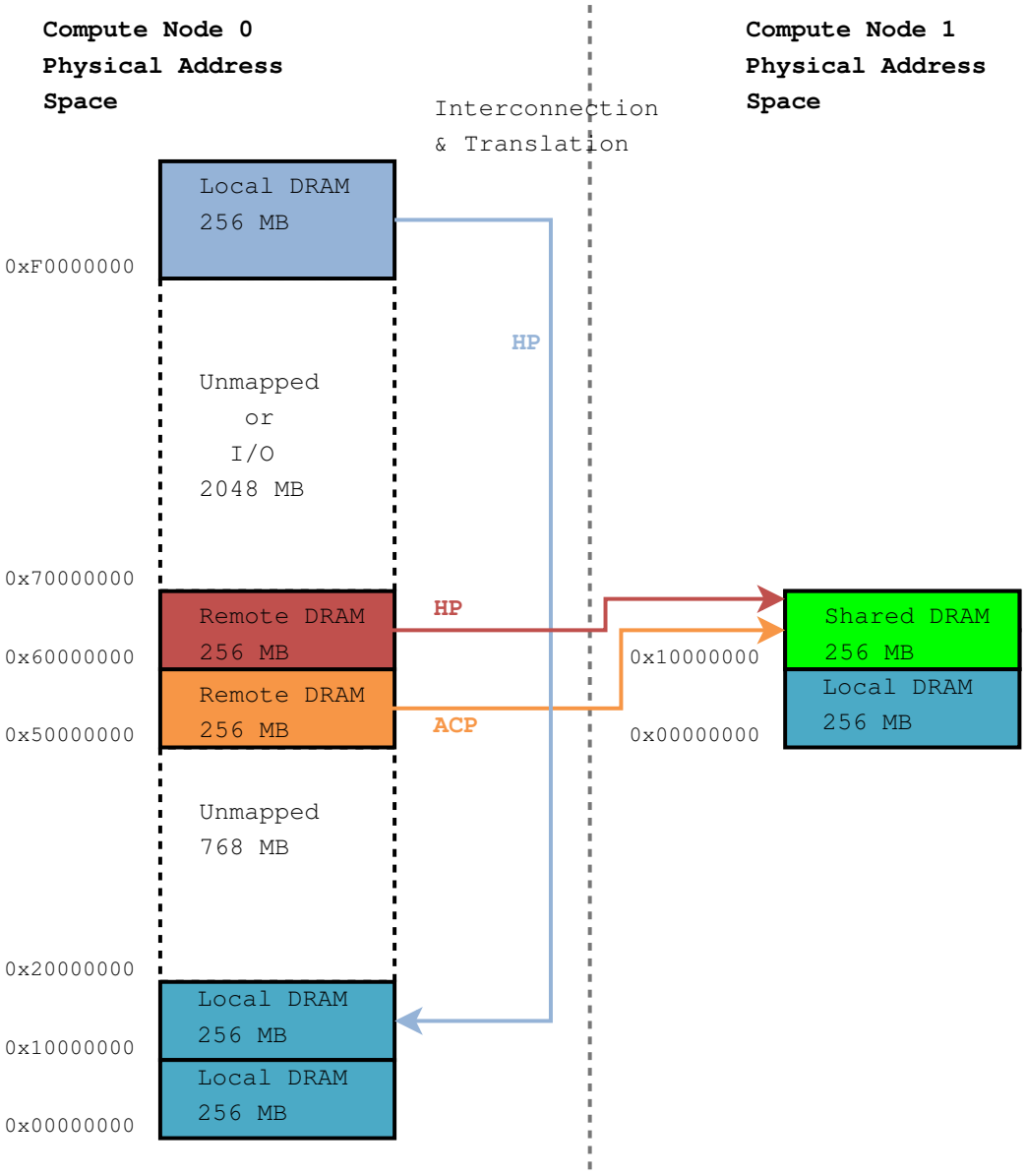

Figure 4.3: Memory Mappings in a Sparse Memory Model, as it is used in our system. Physical address space of Compute Node 0 at the left and Node 1's at the right. The arrows show the memory mapping and the port used (ACP or HP). HEX numbers at the left of each memory space show the start address of each segment.(*Note: Mappings for I/O peripherals is not shown.*)

Addresses	Translated Addresses	Destination Node	Destination Port
0x50000000 - 0x5FFFFFFF	0x00000000 - 0x1FFFFFFF	Node 1	ACP
0x60000000 - 0x6FFFFFFF	0x00000000 - 0x1FFFFFFF	Node 1	HP
0xF0000000 - 0xFFFFFFFF	0x10000000 - 0x1FFFFFFF	Node 0	ACP

Table 4.1: Physical Memory Mapping and Translation

Compute Node 1's DRAM as remote memory for Compute Node 0. The lower 256 MByte of Compute Node 1 are dedicated to its software only. Using the available ACP and HP ports we have several memory mappings available for use by the software as seen in Figure 4.3 and Table 4.1. Address ranges 0x50000000 - 0x5FFFFFFF and 0x60000000 - 0x6FFFFFFF target the same 256 MByte of Compute Node 1's DRAM, while address range 0xF0000000 - 0xFFFFFFF target the higher 256 MByte of Compute Node 0's own DRAM.

4.3 Remote Memory as Main Memory Extension

We can use a remote physical memory segment as an extension of the local physical memory of a Compute Node and include it in the set of physical memory that is available to the Operating System. This means that the kernel structures that describe the physical memory (table *memmap*), will also include the remote physical memory segment. This remote memory will also be fragmented to physical **page frames** as well, just like the 'normal' local physical memory. These physical page frames are available to store virtual pages belonging to user space applications and also the kernel itself as well.

Figure 4.4 depicts the correlation of virtual pages to physical frames with the use of page tables. In the left side the linear virtual address space of a user space application is shown. Available physical memory is shown in the right sight of the picture, along with its valid physical address ranges. Virtual to physical address translation, as we know, is achieved by the use of multilevel page tables, which it the case of ARM have two levels - *Page Directory* and *Page Table*. The address of the Page Directory resides in a register and changes each time a *context switch* is occurred. Using this register, *Page Walk* process can be done in order to find the corresponding physical address of a virtual one, when such a mapping does not exist in the *Translation Look-aside Buffer (TLB)*.

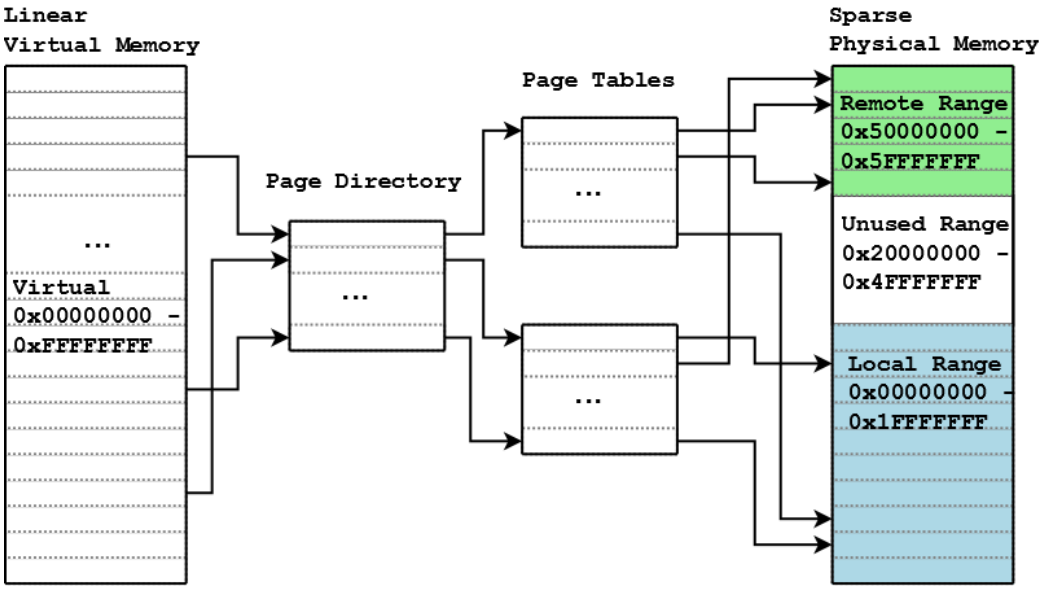

Figure 4.4: Remote Memory as Main Memory Extension

4.3.1 Defining Remote Memory

In order the Operating System to view the remote physical address range, we must describe it in the appropriate ***Device Tree*** segment. The Device Tree is a binary file, used by the Linux kernel in ARM architectures and describes the memory and I/O peripherals of the specific board. It is the equivalent of *BIOS* in x86 architectures. The Device Tree file is loaded by the *u-boot* (boot loader for ARM), then given as an argument to the Linux kernel. In our case, we had to add a segment in the device tree of Compute Node 0 that describes the physical address range that corresponds to Compute Node 1 physical memory (through the underlying hardware physical address translation).

```
memory@00000000 {
   reg = <0x0 0x20000000>;
   device_type = "memory";
};

memory@50000000 {
 reg = <0x50000000 0x10000000>;
 device_type = "memory";
};
```

Listing 4.1: Device Tree memory segments. Remote memory through ACP port.

Listing 4.1 shows the device tree segments that describe the available physical memory. The first segment describes 512 MByte of local DRAM that start at physical address 0x00000000 with length 0x20000000. The latter describes

remote physical memory that start at physical address 0x50000000 with length 0x10000000, resulting in 256 MByte of Compute Node 1's DRAM. Note that the address 0x50000000, makes use of the ACP port as described in Table 4.1.

We can instead map the same remote memory segment using the HP port, by adding the device memory shown in Listing 4.2.

```
   memory@00000000 {
      reg = <0x0 0x20000000>;
      device_type = "memory";
   };

   memory@60000000 {
    reg = <0x60000000 0x10000000>;
    device_type = "memory";
   };
```

Listing 4.2: Device Tree memory segments. Remote memory through HP port.

Using remote physical memory, borrowed from Compute Node 1, as an extension of physical memory for Compute Node 0, the Linux Operating System that we run at that node sees 768 MByte of total available physical memory. By running the *free* utility, we can confirm that. Listing 4.3 shows the output of the utility.

```
root@zedboard0:~# free
             total       used       free     shared    buffers     cached
Mem:        775168     129752     645416          0      10556      59600
-/+ buffers/cache:      59596     715572
Swap:            0          0          0
```

Listing 4.3: *free* command in Compute Node 0 to view available physical memory.

We observe that the available physical memory is larger 512 MByte (of local DRAM), almost 768 MByte. There are some MBytes reserved by the Operating System, to maintain *memory resident* chunks of its own code and data structures. Those, reserved MBytes are at the beginning of the physical memory, almost always starting at physical address 0x8000 for ARM architectures. Of course, this reserved memory segment is not available for user space applications and also it cannot be swapped to a storage device as well. We can also view the physical address ranges that are valid, we can read the special */proc/iomem* file. The output in our system is the following (Listing 4.4.)

```
root@zedboard0:~# cat /proc/iomem
00000000-1fffffff : System RAM
00008000-00429b8f : Kernel code
00450000-0049f66f : Kernel data
50000000-5fffffff : System RAM
e0001000-e0001ffe : xuartps
e0002000-e0002fff : /axi@0/ps7-usb@e0002000
e0002000-e0002fff : e0002000.ps7-usb
e000a000-e000afff : e000a000.ps7-gpio
e000d000-e000dfff : e000d000.ps7-qspi
e0100000-e0100fff : mmc0
f8000000-f8000fff : xslcr
f8003000-f8003fff : pl330
```

```
f8007000-f8007fff : xdevcfg
```

Listing 4.4: Valid physical memory ranges seen by Linux in Compute Node 0.

The output of the special file shows that physical address ranges 0x00000000 - 0x1FFFFFFF and 0x50000000 - 0x5FFFFFFF are labeled as *System RAM* and are available for applications. Physical memory segments that are reserved by the Operating System are shown as the ranges 0x00008000 - 0x00429B8F and 0x00450000 - 0x0049F66F. The rest of the ranges are I/O peripheral mappings.

By running the same utilities in Compute Node 1, we can see the following results (Listings 4.5 and 4.6):

```
root@zedboard1:~# free
             total       used       free     shared    buffers     cached
Mem:        254828      65760     189068          0       6948      34052
-/+ buffers/cache:      24760     230068
Swap:            0          0          0
```

Listing 4.5: *free* command in Compute Node 1 to view available physical memory.

```
root@zedboard1:~# cat /proc/iomem
00000000-0fffffff : System RAM
00008000-00480323 : Kernel code
004aa000-004fcfcf : Kernel data
e0001000-e0001ffe : xuartps
e0002000-e0002fff : /axi@0/ps7-usb@e0002000
e0002000-e0002fff : e0002000.ps7-usb
e000a000-e000afff : e000a000.ps7-gpio
e000d000-e000dfff : e000d000.ps7-qspi
e0100000-e0100fff : mmc0
f8000000-f8000fff : xslcr
f8003000-f8003fff : pl330
f8007000-f8007fff : xdevcfg
```

Listing 4.6: Valid physical memory ranges seen by Linux in Compute Node 1.

Compute Node 1 has only the lower 256 MByte of its DRAM available to its Operating System. That memory resides into address range 0x00000000 - 0x0FFFFFFF as Listing 4.6 shows.

4.3.2 Accessing Remote Memory from User Space

With the way of remote memory usage, detailed in this Section, user space processes do not have direct access to the remote memory. Ultimately, **they do not know that remote memory exists**, like they do not know about physical memory in general. They get only virtual addresses to virtual pages that the Operating System has set up and manages on their behalf. The main way, user space processes can dynamically request memory is the usage of the malloc() library function, which ends up calling system calls and requesting space from the Operating System. Only the Operating System, sets the page tables up for each user space process. As a result, the usage of a virtual page by a user space process can either be in local or remote physical memory, without the process knowing where it resides.

The main advantage in such an environment is that user space processes do not need to by modified or from the start developed for that purpose, using a special library and API. That way, we can run all processes we can run in a traditional environment, this time with additional remote physical memory.

To confirm that we indeed can get memory space larger than the 512 MByte of local DRAM for a user space process, we developed a program that uses the malloc() library function and requests large memory from the Operating System, sliced in several smaller chunks of various sizes. Since the Operating System, implements memory allocation for processes' request in a lazy way, called *lazy allocation*, we must ensure that the process indeed got the memory size it requested and given a positive response from the Operating System. To confirm that, we access all the requested pages. Running this utility we can almost allocate and access the whole physical memory that is available. We know that we accessed the physical memory, because we do not have any swap device enabled. Furthermore, as shown in Listing 4.7, running the *free* command during the operation of our utility we can see that a very large portion of the physical memory is in use and finally that we do not have any swap device enabled.

```
             total       used       free     shared    buffers     cached
Mem:        775168     764116      11052          0       4760      19860
-/+ buffers/cache:     739496      35672
Swap:            0          0          0
```

Listing 4.7: Available physical memory during large allocation test.

In specific, we observe that 764116 KByte from the total available 775168 KByte are used.

4.3.3 Sparse Memory Model

As mentioned before, the ARM processor (as populated with the Xilinx Zynq 7000 SoC) does not allow AXI Requests to be produced on the GP0 and GP1 ports for physical addresses lower than 0x40000000. As a result, we cannot map a remote physical memory segment into a Compute Node's physical address space lower than this limit. Thus, the set of physical addresses, in a Compute Node with remote memory, that is available, contains an unused and invalid physical address range - that is not mapped anywhere. That kind of physical memory model is called *Sparse Memory* .

In our Discrete Prototype, as shown in Figure 4.4 and 4.3, an unmapped physical address range exists at 0x20000000 to 0x4FFFFFFF, creating an unused segment of 768 MByte. The next usable address range is 0x50000000 to 0x5FFFFFFF which is mapped to the higher 256 MByte of DRAM of Compute Node 1, through ACP port. Using HP port we map again the higher 256 MByte of Compute Node 1, to range 0x60000000 - 0x6FFFFFFF of Compute Node 0. A large 2 GByte segment follows, that contains unmapped segment and mappings of the board's I/O peripherals. Finally, we mapped the higher 256 MByte of DRAM of Compute Node 0, so it can access its own DRAM segment through HP port, instead of the normal

way that occurs inside the CPU chip. That segment is mapped at 0xF0000000 to 0xFFFFFFFF.

Depending on what segment of Compute Node 1 we want to use from Node 0, an unused segment of different size is produced in the physical address space of Compute Node 0. In general, physical address space of Compute Node 0 and mappings depend on which segment of remote memory we want to use and the also configuration of the physical address translation block that resides in the FPGA as well.

4.4 Remote Memory as Swap Device

Another way of utilizing remote physical memory, is its usage as a remote swap device. In this case, **it is only used when local physical memory is full**. Using remote physical memory this way means that the remote range will not be split into physical frames available for hosting processes' pages. It is seen as an I/O device by the Operating System and is managed by the OS threads that do page swapping and the appropriate block drivers.

Access to the swap device is occurred only when a swapper kernel thread decides to store an unused physical memory frame, or when a **page fault** occurs and the Operating System must bring the stored physical frame back to memory. Figure 4.5 depicts the page fault procedure.

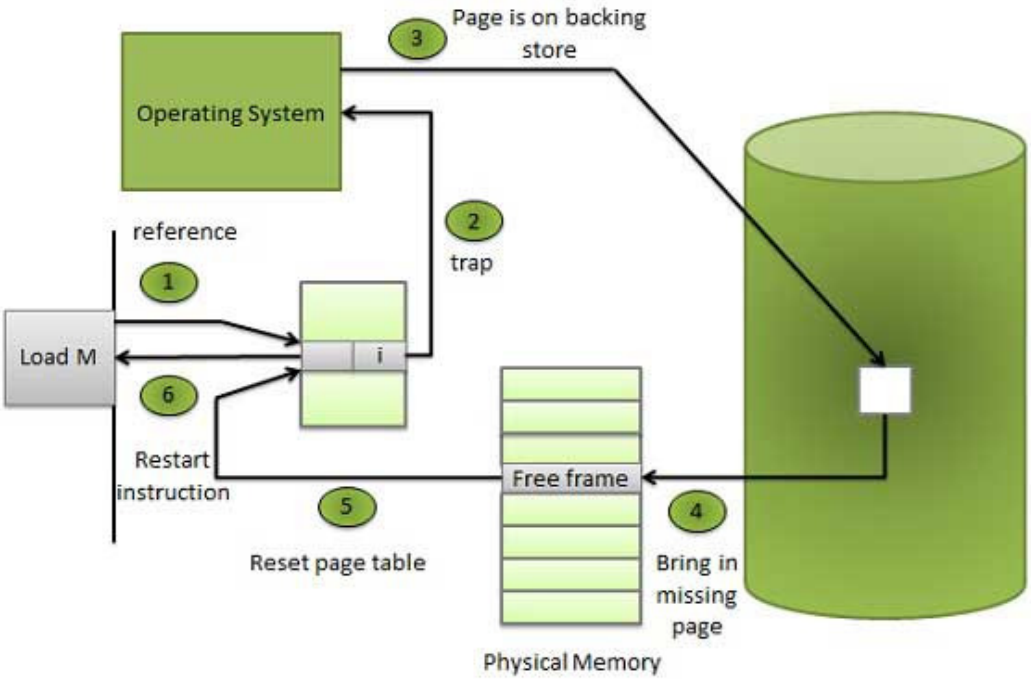

Figure 4.5: Page fault and physical frame recovery

In the figure, the processor issues a Load/Store instruction to a virtual address of a page that does not have a corresponding physical frame present in the physical memory. A page fault exception is raised, generating a trap. The Operating System, finds the corresponding physical frame from the swap device and brings it to a free physical memory frame. If there is not any free frame in the physical memory, the Operating System, will swap the least used physical frame.

4.4.1 Ramdisk Driver

In order to use physical memory of Compute Node 1 as a remote swap device in Compute Node 0, we developed a kernel driver that creates a *ramdisk* **for exactly that physical address range**. The ramdisk is represented by a block device entry in the */dev* directory of the Linux, namely */dev/krama*. When a read or write is occurred on this device, our driver is invoked and services that request. Note that the device can be used for many different application except for swapping. For example, one can use that ramdisk device and our driver to create an EXT4 file system. In our work, we use the driver and the device as a swap device.

The driver's main component is an array of *struct page* structures that describe the physical page frames the driver is responsible of. Reading and writing from/to those physical page frames is done by a function called *kram_make_request* that is invoked every time a read or write operation is done upon the */dev/krama* device entry. The function completes a read or write operation by reading data from the appropriate physical addresses and returning them back to the caller, or by writing the given data to the appropriate physical address.

Since we use physical pages that are not allocated by a kernel allocator, we must ensure that these pages are accessible by the kernel source code macros and helper functions. In order to achieve this, we must add those remote pages to the kernels physical memory tables, but reserve them for usage **only** by our swap driver. We modified the kernel source code and added a reservation segment during boot time. In the *arch/arm/mm/init.c* file we added the following code segment in the *arm_memblock_init()* function.

```
memblock_reserve(0x60000000, 0x10000000);
```

Listing 4.8: Reserve physical memory for exclusive usage.

Now, that this physical address range is reserved by the kernel for the driver, we use the code shown in Listing 4.9 to get the structures (*struct page*) that describe the corresponding physical page frames. Constant *PHYS_BASE_ADDR* is the starting address of the remote DRAM segment that we want to use as swap. In our default setup it is 0x60000000, that is the address of the higher 256 MByte of Compute Node 1's DRAM, using the HP port. Variable *pages* is the number of physical page frames the driver must use as swap space and is given as an argument to the driver upon the loading (*insmod*). Kernel macro *__phys_to_pfn()* gets the page frame number of a physical address by looking into the kernel's *mem_map* table and macro *pfn_valid()* checks if a given page frame number is a valid one. If

the remote physical memory segment was not reserved as mentioned before, this function call would have indicated the given page frame numbers as invalid. Finally, we make use of the *pfn_ to_ page()* macro that gets the corresponding *struct page* for a given page frame number. That way, the *memory* table of the driver is filled with valid physical page frame descriptions.

```
for(i=0; offset=0; i<pages; i++; offset+=PAGE_SIZE)
{
  pf_no = __phys_to_pfn(PHYS_BASE_ADDR+offset);
  if (!pfn_valid(pf_no))
  {
    return -ENOMEM;
  }
  memory[i] = pfn_to_page(pf_no);
  if (memory[i] == NULL)
  {
    return -ENOMEM;
  }
}
```

Listing 4.9: Get page frames for a physical address range.

Note that for using the remote memory segment as swap we must describe it in the device tree the same way as described in Section 4.3.1 and shown in Listing 4.2. This is essential, so we can reserve those physical page frames at kernel boot time.

When Linux boots at Compute Node 0, running the *free* utility, will show that the availably physical memory is 512 MBytes, since the remote 256 MBytes of DRAM have been reserved at boot time.

```
             total       used       free     shared    buffers     cached
Mem:        513024     125992     387032          0      10928      56984
-/+ buffers/cache:      58080     454944
Swap:            0          0          0
```

Listing 4.10: Available physical memory before swap driver loading.

We can load the driver by using the series of commands (that require root privileges) shown in Listing 4.11.

```
$> insmod ./kram_main.ko pages=256
$> mkswap /dev/krama
$> swapon /dev/krama
```

Listing 4.11: Load remote swap driver.

The first command loads the driver into the kernel. The argument *pages* instructs the driver to use 256 MBytes of the remote memory as swap device. The command *mkswap* creates a swap file system in the */dev/krama* device. Finally, the *swapon* command enables the usage of the */dev/krama* device as swap.

After enabling the swap, running the *free* utility again shows the available memory for the system (Listing 4.12).

```
             total       used       free     shared    buffers     cached
Mem:        513012     127040     385972          0      10864      57200
```

```
-/+ buffers/cache:      58976      454036
Swap:        262140          0      262140
```

Listing 4.12: Available memory when swap driver is loaded.

We observe that the physical memory is still 512 MBytes, but this time we have additional 256 MBytes of swap as shown in the *Swap* line Listing 4.12.

4.4.2 Driver with DMA

We also implemented the remote swap driver utilizing the DMA engine for Remote DMA data transfers instead of memory copying with Loaf/Store instructions. The function of the driver that services read/write requests to the */dev/krama* device initiates DMA transfers using the simple mode of the DMA engine. Simple DMA mode requires the data to be transfered to be serialized in memory. Swap driver utilizing DMA operations increases data transfer throughput as shown in the evaluation Sections 5.2.6 and 5.2.7.

4.5 User Space and Swap

User space processes only know about their virtual memory space and nothing else. Thus, they do not know about the existence of a swap device. Only the OS knows and manages it. When the physical memory (DRAM) is insufficient it starts storing pages from different processes into the swap (*swapping*).

Running the memory allocator application, as we shown in Section 4.3.2 we can confirm that the swap is indeed used by the OS, when a process requests memory larger than the available physical DRAM. Running the *free* utility at a moment during the allocator operation we can confirm that the swap is used. Listing 4.13 shows that almost all physical memory is used and furthermore that a large part of the swap device is used.

```
             total       used       free     shared    buffers     cached
Mem:        513012     501852      11160          0         68       2248
-/+ buffers/cache:     499536      13476
Swap:       262140     136564     125576
```

Listing 4.13: Using the swap device

4.6 Explicit Access of Remote Memory

It is possible for user space processes to access physical memory explicitly, using some underlying kernel features. Linux kernel provides a set of special device files in */dev* directory. The */dev/mem* special file in particular, gives access to physical address space (physical RAM and I/O peripherals) the Operating System can seen to user space processes. Using that special file a process can Read/Write from/to physical addresses using either I/O system calls read()/write(), just like

file operations or use the *mmap()* function to map a physical address range to its virtual memory space. In detail, calling mmap() upon */dev/mem*, ends up requesting from the Operating System to set up the process's page tables and add mapping of physical frames to virtual pages of that process. As detailed below, an example of mmap() call as used in our applications is shown.

```
memfd = open("/dev/mem", O_RDWR | O_SYNC);
mem = mmap(0, mapsize, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
                                              phys_addr & ~MAP_MASK);
```

Argument *mapsize* is the size in Bytes that we request to mmap and it always is a multiple of page size. Argument *PROT_READ | PROT_WRITE* defines the permissions for the mapped pages. *MAP_SHARED* argument tags the mapped pages as shareable among processes. We use it in our experiments to tag the mapped page as ***uncached***, in order to measure latency and throughput of the remote physical memory itself. The last argument is the file descriptor of */dev/mem*, which is page aligned using the *& MAP_MASK* bitwise operation. The mmap() function returns a pointer to the base address of the first virtual page that corresponds to the requested physical address range. If the mmap() is successful, the process can read or write any byte of the physical memory with Load/Store instructions. For example:

```
1. Load: a = mem[i];
2. Store: mem[i] = a;
```

4.7 Explicit Remote DMA Operations

Similar to explicitly physical memory access is the procedure of explicitly initiating DMA operations from user space processes. The DMA register space is memory mapped into some physical address range and therefore can be accessed using the same methods described in Section 4.6. A process can initiate a DMA transfer by defining the physical memory source and destination addresses, the size and some control flags. Those, are given as arguments to the DMA engine, by writing their values in the appropriate DMA registers. For example Listing 4.14 shows a DMA transfer initiated by a user space process that has used *mmap()* on */dev/mem* to obtain access to the DMA registers.

```
#define dma_write(base_addr, reg_offset, value) \
  *(unsigned int *)(base_addr + reg_offset) = ((unsigned int)data);

#define dma_read(base_addr, reg_offset) \
  *(unsigned int *)(base_addr + reg_offset);

  memfd = open("/dev/mem", O_RDWR | O_SYNC);
  dma_virt = mmap(0, mapsize, PROT_READ | PROT_WRITE, MAP_SHARED, memfd,
                                              dma_phys_addr & ~MAP_MASK);

  dma_write(dma_virt, CTRL_REG_OFFSET, INIT_VALUE);
  dma_write(dma_virt, SRC_REG_OFFSET, SRC_PHYS_ADDR);
  dma_write(dma_virt, DST_REG_OFFSET, DST_PHYS_ADDR);
```

```
dma_write(dma_virt, SIZE_REG_OFFSET, SIZE);

do{
  pollbit = dma_read(dma_virt, STATUS_REG_OFFSET);
  pollbit &= 0x00000007;
}while(pollbit != 0x02);
```

Listing 4.14: User space DMA transfer.

When the size register is written the DMA engine starts the transfer. The process waits for the DMA transfer to complete by polling the status register of the DMA engine. The macros *dma_write()* and *dma_read()* are used for convenience.

4.8 Remote Memory as I/O Device

The last way of "viewing" remote memory in a full system with Operating System and user space environment, as mentioned Section 4.2 is as an I/O character device. Using similar methods as in the mechanisms for explicit memory access from user space described in Section 4.6 we implemented a kernel driver that manages an character device. That character device is created in the */dev* directory of the Linux file system and represents the remote memory available in the system (*/dev/remotemem*). Our driver implements a set of functions that service read-/write requests to the */dev/remotemem* character device file. It also implements the *mmap()* function for usage of remote memory as described in Section 4.6.

That way a user space process can access the remote memory through the */dev/remotemem* device file either as a file using the read/write system calls or via direct Load/Store instructions using the *mmap()* function.

4.9 Operating System with Non-Uniform Memory Access (NUMA)

We described three ways of remote memory "viewing" and usage by the Operating System and the user space environment. The first one, described in Section 4.3 is the most convenient one, from the perspective of user space processes. It also is the better way, from a performance aspect, since there is not any I/O driver invoked when remote memory is accessed. However, using remote memory that way (as a main memory extension) can reduce performance, because the Operating System (and of course the user space environment) does not know which segments of its physical memory are local and which are remote. **Thus, management of user space code and data from the OS can result in data or code placement in remote memory, even though there is available local memory.** This is of course an idiot way of using remote memory. We want the Operating System to use remote physical memory only when the local physical memory is getting full, just like in Section 4.5 but without the swapping subsystem in between. In other words we want the Operating System to be aware of the underlying hardware topology.

This type of Operating System implementation is called **Non-Uniform Memory Access aware Operating System or NUMA aware**. Many NUMA systems employ hardware support and special NUMA Operating System implementations. x86 architectures provide hardware support and most Operating Systems for that architecture are NUMA aware. The main feature of a NUMA aware Operating System that we want in our Prototype is the use of **Distance Vectors**, that are latency metrics for different physical memory segments. That way, when the **Buddy Algorithm** of the Linux Operating System is invoked to doe physical page frame placement and replacement knows about the topology of the underlying memory system and uses the most efficient way to place/replace memory page frames. As a result, local physical memory is always used when it contains free page frames and remote memory is only used when the local memory is getting full.

The Linux implementation for NUMA platforms, partitions both CPU cores and physical memory segments into different groups or **NUMA nodes**. Figure 4.6 shows an example of NUMA platform, where for each 2 CPU cores there is a dedicated DRAM segment. Of course all course can access all available DRAM segments, but accessing segments that are not "local" to them will suffer from greater latency.

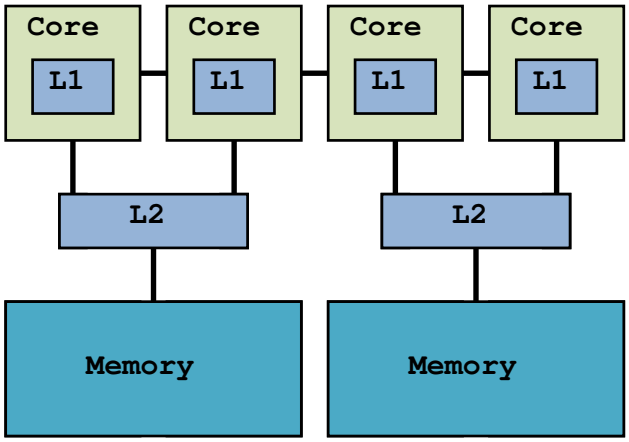

Figure 4.6: A simple NUMA platform.

The Linux operating system for such platforms has different physical memory description structures than the default flat memory model. It organizes memory segments into node structures, that describe a numa node. A node structure contains array of physical memory frame structs. Consequently, when the kernel needs to allocate a physical page frame it always considers memory frames that belong to the numa node the request came from firstly. For more details on how physical memory is managed in a NUMA aware Linux kernel refer to [4].

4.10 Deadlock Scenario

In this work we have described using remote memory in several ways. However, all those ways require Compute Node 0 to use remote memory from the other and Compute Node 1 to use only its remaining DRAM memory. We cannot use remote memory by both Compute Nodes at the same time, using the ACP port. During remote memory implementations, our team discovered that a deadlock is in the processor occurred when trying to use remote memory from both Compute Nodes. The hardware team of the laboratory traced that problem and found out that it occurs when two AXI Write Requests with different destinations arrive in the processor Snoop Control Unit, through the ACP port. Because using remote memory in both Compute Nodes requires the issue of AXI Write requests by both nodes' processors the deadlock is unavoidable. As a result, we cannot use remote memory from both Compute Nodes simultaneously.

To create the deadlock situation, we implemented a bare-metal application that creates AXI Write Requests to specified remote memory addresses using Store instructions. One instance of the application runs on Compute Node 0's processor and the other one at Compute Node 1's processor. Compute Node 0 application runs a loop Store instructions to remote memory (that is the memory of Compute Node 1). When application at Compute Node 1 starts the same procedure, the first Store instruction that is issued creates the deadlock in the whole system. The same happens, the other way, when Compute Node 1 starts first and Compute Node 0 later.

Chapter 5

Remote Memory Evaluation

We implemented some micro-benchmarks both in bare-metal and Linux process forms, to measure latency and throughput achieved when using remote memory. We implemented both forms (bare-metal and Linux process) to confirm that running a full system with Operating System and User Space environment does not negatively impact the performance seen by processes.

All experiments were done in the Discrete Prototype described in Section 3.1. The two Compute Nodes (Zedboards) are connected with a high speed FMC-to-FMC cable that consists of 15 LVDS pairs and the interconnection logic is implemented in the FPGAs of each board. The FPGA runs at 100 MHz and the digital logic allows transfers of 64 bits each clock cycle. As a result, the maximum throughput that the interconnection circuit can achieve is 6.4 Gbps. Keep in mind that this data rate does not take into account the processor to FPGA interconnection (ACP/HP and GP ports) latencies.

5.1 Bare Metal Evaluation

5.1.1 Latency

Running a bare metal application at Compute Node 0, we can access remote memory of Compute Node 1, either with Store/Load instructions or with DMA operations and we can measure latency and throughput.

It is important to measure the latency of a simple Read or Write operation for one word (4 Byte). With a basic setup we can measure the Round Trip Time for such an operation. That word must **not be cached anywhere**, in order to measure the latency of the interconnection network itself.

A single Load or Store instruction to a remote memory region produces an AXI Read or Write request and its corresponding response from the remote processor. Thus, a single Load or Store to an uncached remote segment suffers the Round Trip latency of the interconnection network.

We created an application that does a large number of Stores/Loads to the same memory address and then calculate the time it takes for a pair of Store/Load

instructions to complete by dividing the total time to the number of iterations. In our Discrete Prototype we measured

1450 nsec for a Store/Load pair to complete

Thus the **AXI Request Round Trip Time is about 725 nsec**. The pair of the instructions produce 2 AXI Requests: one Write AXI Request and one Read AXI Request. Although the latency seems quite large, it is measured in the Discrete Prototype in which the interconnection logic is implemented in the FPGAs that run with a slow clock of 100 MHz.

Many of the FPGA clock cycles are spent to the *axi2axi* blocks that handle the LDVS pairs of the FMC-to-FMC cable as described in Section 3.1. When 8 LVDS pairs are utilized this subcircuit consumes almost 39 to 40 clock cycles. The rest 33-34 clock cycles are spent in the essential AXI protocol conversion blocks and the processor's memory-to-ACP port and memory-to-GP port subsystems.

Logic Analyzer

The measurement is in accordance with the results seen using a Logic Analyzer (Xilinx ChipScope). With a Logic Analyzer we can see an AXI Read or Write Request and their corresponding responses, determining the FPGA clock cycles it took to complete. (FPGA clock runs at 100 MHz). The Logic Analyzer logic probes (ChipScore FPGA blocks) are put inside the FPGA logic in the Zedboard that is the initiator of the experiment (Compute Node 0). In particular they are inserted at the point the GP ports of ARM processor connects to the interconnection logic. Each time an AXI Request is produced to the GP port by the processor the probe captures it. The logic probes connect to the Logic Analyzer software of a PC via the JTAG port of the boards.

We can measure the FPGA clock cycles needed for an AXI Read Request to complete by measuring the time difference between the signals ARVALID (Read Address Valid) and RVALID (Read Valid). In a similar fashion, we check the signals AWVALID (Write Address Valid) and BVALID (Write Valid).

Figures 5.1 and 5.2 show screen shots of the Logic Analyzer software for the AXI Read and Write Requests. The vertical cursors mark the activation of the aforementioned AXI signals. The AXI Read Request takes 73 clock cycles to complete, while the AXI Write Request takes 65. As a result a pair of a Load and Store instructions needs 138 clock cycles to complete. The 145 clock cycles we obtained from software experiments contain the additional overhead of the GP and remote ACP ports latency that cannot be measured with the Logic Analyzer.

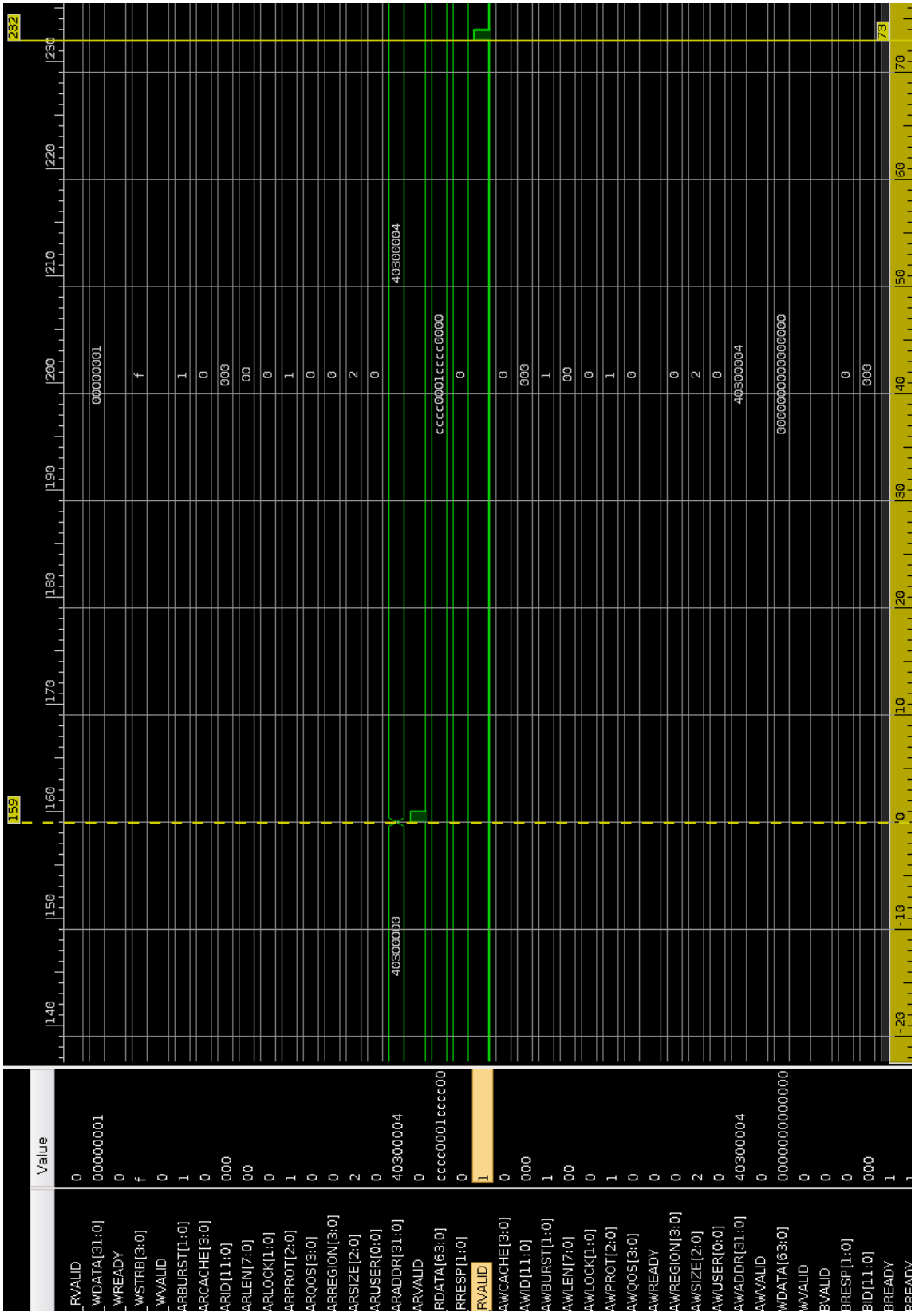

Figure 5.1: AXI Read Request latency in Logic Analyzer

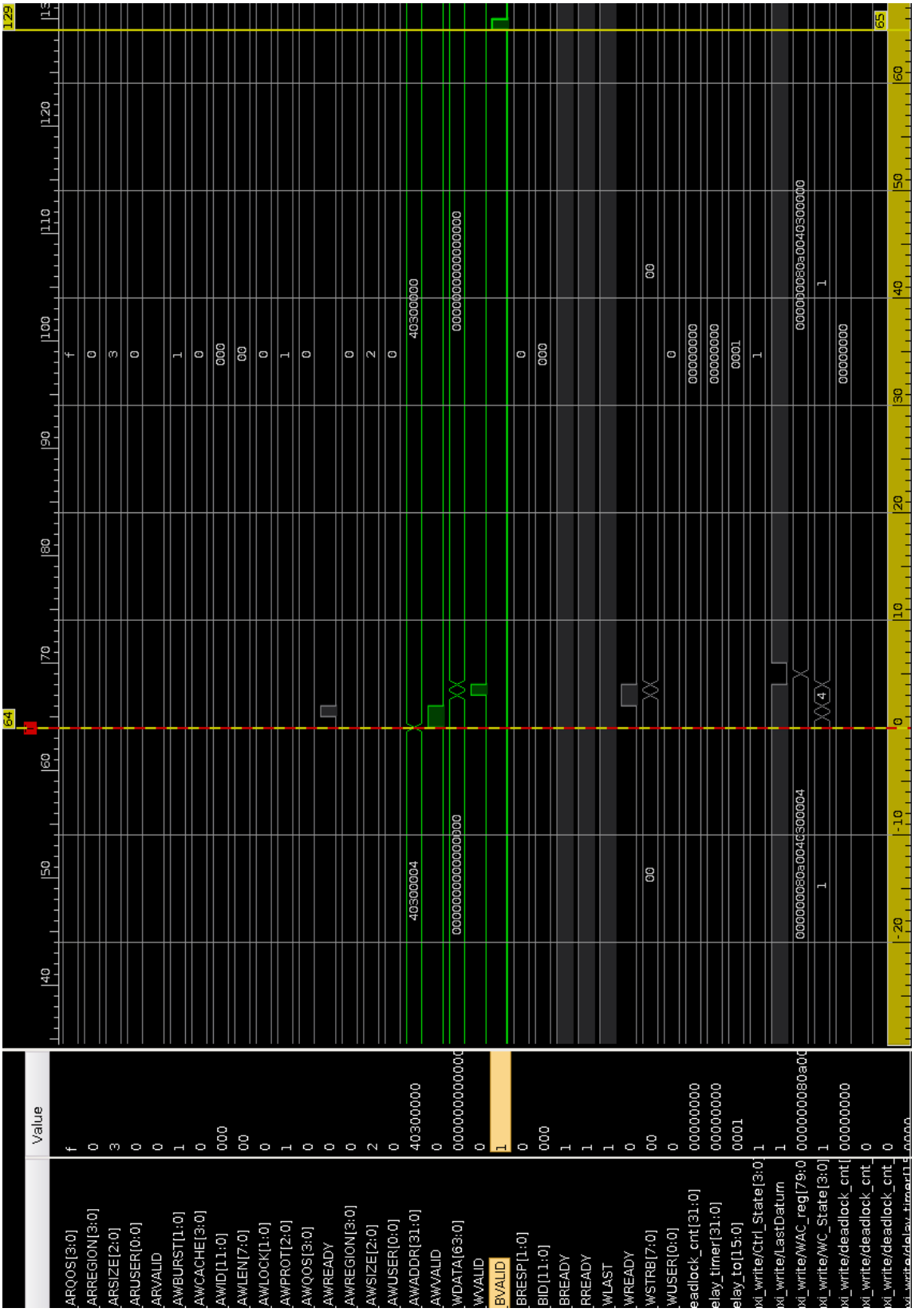

Figure 5.2: AXI Write Request latency in Logic Analyzer

5.1.2 Data Transfer Throughput

We also measured the throughput achieved for data transfers from local to remote memory and vice versa, with a bare-metal application that uses a loop of Load/Store instructions to copy 100 MBytes of data (the application does many iterations of that copying to amortize the measurements). We used the memcpy() function that is included in the bare metal libraries that come with the Xilinx SDK suite. Using either ACP or HP ports we made some experiments and the results shown in Table 5.1.

	Direction	Destination Port	MB/s	Gbps
1	Local Read - Remote Write	ACP	110	0.88
2	Local Read - Remote Write	HP	112	0.90
3	Remote Read - Local Write	ACP	45	0.36
4	Remote Read - Local Write	HP	40	0.32

Table 5.1: Data transfer throughput (for bare-metal application).

Column *Direction* shows the data transfer direction and *Destination Port* mentions the ARM slave port used for that data transfer. The last two columns shows the result for the data transfer in MBytes/sec and Gbps. Experiments 1 and 2, read from local physical memory and write to the remote memory - the first using the ACP and the latter the HP port. The last two experiments read from the remote physical memory and write to the local memory - experiment (3) through ACP and (4) through HP port.

Note that data transfers that copy data word-by-word or byte-by-byte, do not require the AXI Requests for each word or byte to be completed. Instead, the hardware produces multiple interleaved AXI Requests, since it is allowed by the AXI protocol.

5.1.3 DMA Throughput

Besides data transfers using Load/Store instructions, we can utilize DMA operations to the mapped remote physical memory segment (RDMA operations). The RDMA is a *Data Mover*, which transfers the content of a physical memory segment to another. Four kinds of data transfers can be performed with RDMA:

1. From local to remote
2. From remote to local
3. From local to local
4. From remote to remote

There is an FPGA block that implements the Xilinx CDMA in each Zedboard (Compute Node). It resides outside of the CPU cores, thus it has to create 2 AXI requests for the transfer of a single word (or Byte). It produces an AXI Read Request for reading the data from the source physical address and an AXI Write Request to write those data to the destination physical address. The DMA engine utilizes several techniques to improve performance, such as multiple interleaved AXI Requests, Read/Write bursts, etc.

Using a bare metal application that handles the DMA engine and executes 1 million iterations of 8 MByte DMA transfers (8 MBytes is the largest data size the CDMA can handle), we measured the throughput achieved for data transfers in various directions and different source and destination port utilizations (ACP or HP).

Results are shown in Table 5.2. Columns *Source* and *Source Port* show the port used for reading data, while columns *Destination* and *Destination Port* mentioned the port used for writing data. The last two columns show the measured throughput in MByte/sec and Gbps for each data transfer.

	Source	Source Port	Destination	Destination Port	MB/s	Gbps
1	Local	ACP	Remote	ACP	310	2.48
2	Local	ACP	Remote	HP	305	2.44
3	Local	HP	Remote	ACP	610	4.48
4	Local	HP	Remote	HP	507	4.06
5	Remote	ACP	Local	ACP	264	2.11
6	Local	ACP	Local	ACP	263	2.10
7	Local	HP	Local	HP	716	5.73

Table 5.2: DMA data transfer throughput (for bare-metal application).

We observe that when using the HP Port for reading, the throughput almost doubles. It is reasonable, since use of the HP port bypasses the cache coherency system of the ARM processor.

5.2 Linux Microbenchmarks

Section 4.6 describes a method by which a Linux user space process can explicitly access physical memory. We used that method to implement microbenchmarks as user space processes to measure remote memory latency and throughput for data transfers. Furthermore, we used the method described in Section 4.7 to measure throughput for RDMA data transfers, from the Linux user space.

5.2.1 Latency

We measured the latency for a sing read/write of one word (4 Bytes) which belongs to a remote memory page and the throughput achieved for data transfers from the local physical memory to the remote and vice versa. The latency for a Load/Store pair of one word of the remote memory is
1450 nsec or **725 nsec per read or write operation**.
The results are the same to the measurements acquired by the bare metal application. As a result, we observe that we can use remote memory without penalty in a full Linux environment.

5.2.2 Data Transfer Throughput

Furthermore, we measured throughput achieved for data transfers in various directions. From local to remote physical memory, remote to local and local to local. The experiment is identical to that in Section 5.1. The results are shown in Table 5.3.

	Direction	Destination Port	MB/s	Gbps
1	Local Read - Remote Write	ACP	110	0.88
2	Local Read - Remote Write	HP	112	0.89
3	Remote Read - Local Write	ACP	45	0.36
4	Remote Read - Local Write	HP	40	0.32

Table 5.3: Data transfer throughput using Load/Store instructions.

As we can see in Figure 5.3, throughput achieved with Load/Store instructions is relatively small compared to Local to Local memory data transfers inside the processor. However, the throughput achieved is still far better than data transfers from/to I/O storage devices.

In Figure 5.3 we see than transferring data from remote to local memory gives a maximum throughput of 45 MByte/sec. (*If the ARM processor had no read buffers so that no outstanding AXI read requests could exist, then the maximum throughput that could be achieved would be 5.2 MByte/sec.*) Writing data from local to remote memory gives a higher throughput of 110 MByte/sec.

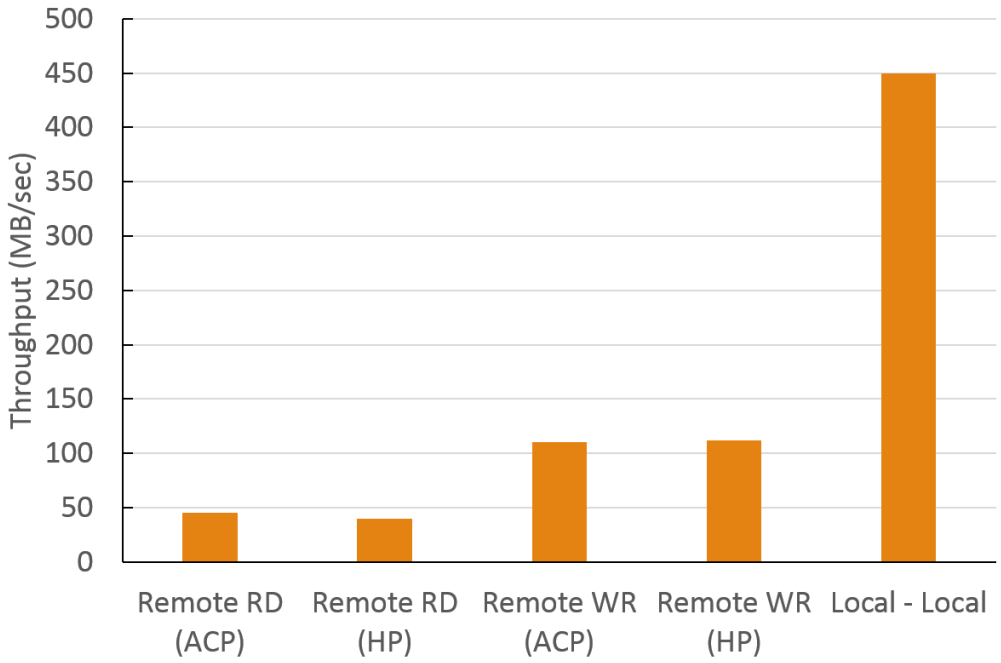

Figure 5.3: Data transfer throughput using Load/Store instructions.

5.2.3 Local Memory Throughput

For completeness of the benchmarks we include an evaluation of the local DRAM and create a machine profile for the Compute Nodes. Since Zedboards contain a DDR3 DRAM memory module clocked at 553 MHz utilizing a 32 bit *rambus*, the maximum theoretical data rate the memory can handle is

(Clock Frequency) x (Bus Width) = 533 MHz x 32 bit = 15.89 Gbps or 2033 MB/sec.

This is the maximum raw data rate the DRAM can handle. Of course, the Read or Write bandwidth differs when the Load/Stores refer to sequential or random memory addresses. The data transfer throughput, achieved by an application when copying a buffer to another one, is even smaller because it consists of pairs of Load/Store instructions per word. Table 5.4 shows throughputs measures when using different buffer copying methods. Methods (1) and (2) use the default library functions *memcpy()* and *bcopy()* and achieve high throughputs of 7.6 and 5.5 Gbps respectively. Methods (3), (4), (5) and (6) use for loops that copy 4 Byte words using Load/Store instructions. Method (3) copies one word per loop iteration, while methods (4), (5) and (6) implement unrolled loops that copy 4, 8 and 32 words per loop iteration.

We observe that performance of remote memory in our Discrete Prototype is comparable to the performance of local DRAM.

	Methods	MB/sec	Gbps
1	memcpy()	950	7.6
2	bcopy()	687.5	5.5
3	loop()	250	2
4	Unrolled loop() x4	650	5.2
5	Unrolled loop() x8	762.5	6.1
6	Unrolled loop() x32	937.5	7.5

Table 5.4: Data transfer throughput using Load/Store instructions (for Local DRAM)

5.2.4 DMA Throughput

We made the same experiments as in Subsection 5.1.3 plus some more measurements to include all possible data transfer directions and port configurations in our prototype, when using DMA operations. We also made those measurements to confirm that a system with a full Linux environment does not have a negative impact on DMA performance.

The Xilinx CDMA engine that is used in these experiments is set up by a user space process using the method described in Section 4.7. Although this method should be disabled in a final product, it is very useful for debugging and evaluation purposes in the development process.

The process executes 1 million iterations of 8 MByte DMA transfers, just as the bare metal application described in Section 5.1.3.

The results of all possible DMA configurations are shown in Table 5.5.

The first four lines address data transfers from local to remote node, with all port configurations (reading from local DRAM and writing to remote DRAM). Lines 5 to 8 address remote to local data transfers (Reading from remote DRAM and writing to local DRAM). Lines 9 to 12 the transfer throughput when the DMA is used on local DRAM only and finally, the last four lines (13 to 16) show throughput for DMA transfers from remote DRAM to remote DRAM again (also called *3rd party DMA*). In Figure 5.4 the results seen in 5.5 are visualized in a graph plot, where the horizontal axis describes the data transfer direction and port configuration and the vertical axis shows the DMA throughput achieved. We observe that DMA transfers, especially for large data sizes, are far more beneficial than simple Load/Store operations. As a result, it is better to use DMA operations for large data sizes when possible.

We confirm that having a full environment with Operating System and user space applications does not have a negative impact on performance of DMA. We also, observe that in general, when reading through the ACP port (regardless if the source is the local or remote DRAM) the performance is decreased compared to transfers that read through the HP port.

The Discrete Prototype and the CDMA engine gives the software engineer and

	Source	Source Port	Destination	Destination Port	MB/s	Gbps
1	Local	ACP	Remote	ACP	310	2.48
2	Local	ACP	Remote	HP	305	2.44
3	Local	HP	Remote	ACP	610	4.48
4	Local	HP	Remote	HP	507	4.06
5	Remote	ACP	Local	ACP	264	2.11
6	Remote	ACP	Local	HP	280	2.24
7	Remote	HP	Local	ACP	280	2.24
8	Remote	HP	Local	HP	701	5.60
9	Local	ACP	Local	ACP	250	2.00
10	Local	ACP	Local	HP	241	1.92
11	Local	HP	Local	ACP	715	5.72
12	Local	HP	Local	HP	716	5.73
13	Remote	ACP	Remote	ACP	213	1.70
14	Remote	ACP	Remote	HP	230	1.84
15	Remote	HP	Remote	ACP	605	4.84
16	Remote	HP	Remote	HP	501	4.00

Table 5.5: DMA data transfer throughput

the systems designer many available options to choose from. Depending on the memory allocation policy (which node uses what part of DRAM) and the type of memory coherency needed, one can use the appropriate configuration to maximize

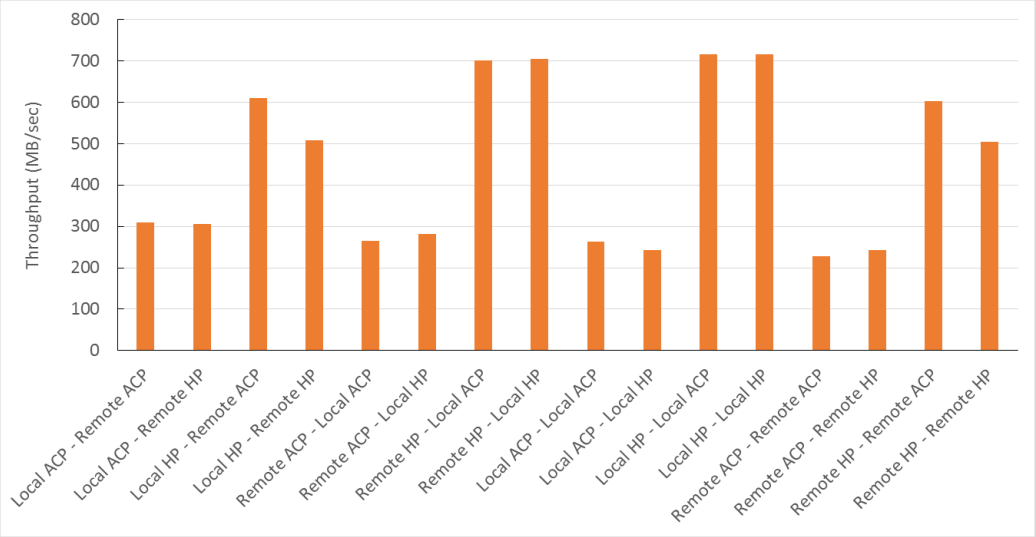

Figure 5.4: DMA data transfer throughput

performance.

5.2.5 DMA Throughput vs. Data Size

In this section we examine the impact that different sizes have upon throughput of DMA transfers.

In Figure 5.5 a graph of DMA transfer throughput against different transfer sizes is plotted. The DMA source and destination correspond to Line 3 of Table 5.5, where the source is Local DRAM through the HP port and the destination is the remote DRAM through the ACP port. As seen in the plot, there is little benefit to be gained when using DMA operations for small transfer sizes. The performance is decreased due to DMA setup time and the fact that one cannot fully utilize the DMA features, such as large bursts, etc. As a result, the throughput of small data DMA transfers is comparable to throughput achieved with Load/Store data transfers for the same size, so one can use Load/Stores instead. The critical transfer size is 512 Bytes, which is the first transfer size that DMA operations start to gain greater throughput than Load/Stores. As seen in Figure 5.5, DMA data transfer throughput for 512 Bytes is 122 MByte/sec. The Load/Store upper throughput limit when writing to remote memory via HP or ACP ports is 110 MByte/sec as seen in 5.3. So, for smaller data sizes one can use Load/Store instructions instead of DMA operations.

In Figure 5.6 a DMA transfer of the opposite direction, is shown. The DMA engine reads data from the remote DRAM through the ACP and writes them to the local DRAM through the HP port. As mentioned in Table 5.5 of Section 5.2.4, the upper limit for DMA throughput using this direction and port configuration is 280 MByte/sec, that is reached only for transfers utilizing large data sizes. For small data sizes, 4 up to 128 Bytes, data transfers using Load/Store instructions will give better results in terms of throughput, because as seen in Figure 5.3 Load/Store data transfers in this configuration can achieve 45 MByte/sec throughput.

In Figure 5.6 a DMA transfer of the opposite direction, is shown. The DMA engine reads data from the remote DRAM through the ACP and writes them to the local DRAM through the HP port. As mentioned in Table 5.5 of Section 5.2.4, the upper limit for DMA throughput using this direction and port configuration is 280 MByte/sec, that is reached only for transfers utilizing large data sizes. For small data sizes, 4 up to 128 Bytes, data transfers using Load/Store instructions will give better results in terms of throughput, because as seen in Figure 5.3 Load/Store data transfers in this configuration can achieve 45 MByte/sec throughput.

5.2.6 Remote Swap I/O Throughput

Viewing remote memory as a remote swap device, as described in Section 4.5 requires the I/O driver stack of the Linux kernel to intervene for each *read()* or *write()* from/to that device. First of all, we measured the I/O throughput the swap device can achieve using the standard *dd* utility. We ran *dd* for different block sizes

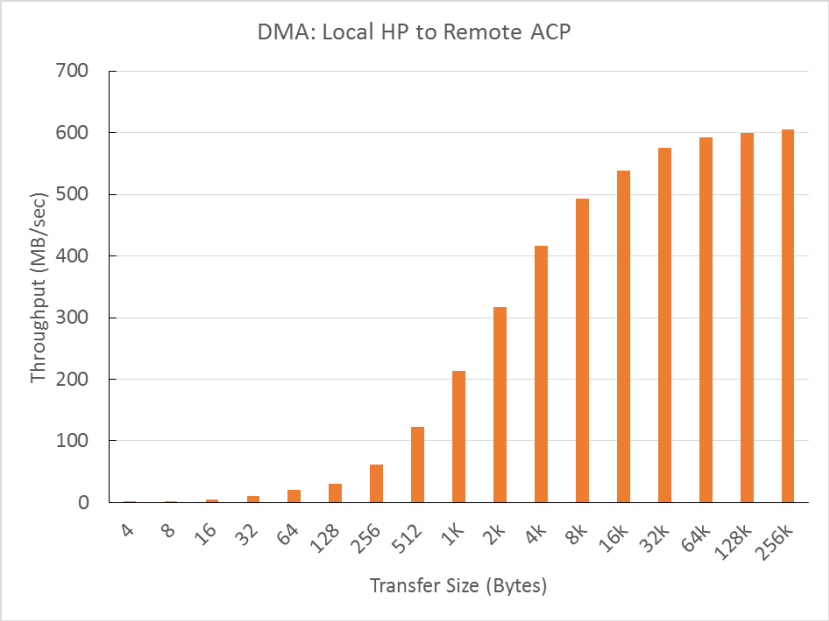

Figure 5.5: DMA data transfer throughput vs. data size (Local to Remote)

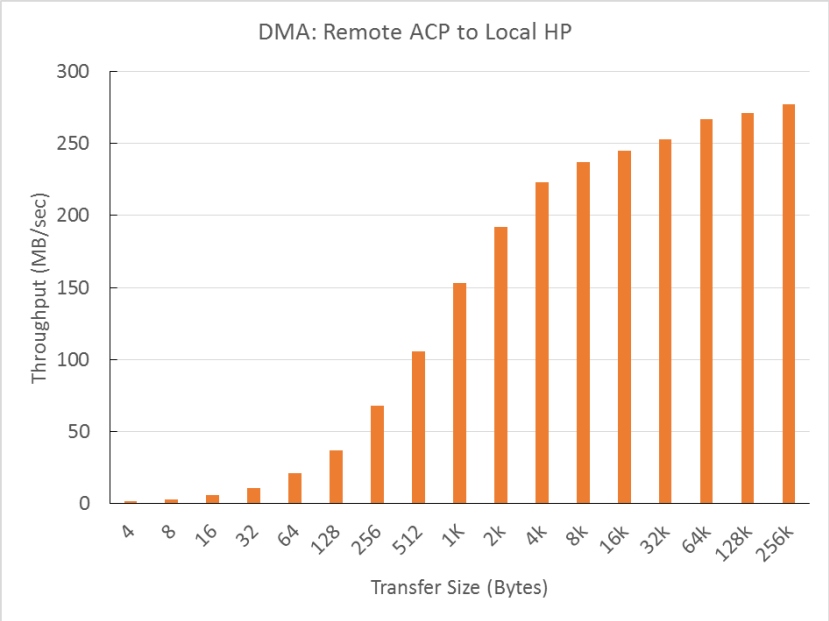

Figure 5.6: DMA data transfer throughput vs data size (Remote to Local)

and the results are shown in Figures 5.7 and 5.8. Figure 5.7 shows that the I/O throughput reaches the limit of Local to Remote memory data transfer throughput, 110 MByte/sec or 0.88 Gbps, as measured by the bare-metal application in Section

5.1.2 Table 5.1. Different block sizes have a negligible impact on I/O throughput. In a similar fashion, as depicted in Figure 5.8, the I/O Read throughput achieved, also reaches the data transfer throughput limit that is 45 MByte/sec or 0.36 Gbps. Block size does not affect the I/O throughput. We observe that the intervention of the I/O software stack does not reduce data transfer throughput compared to pure Load/Store data transfers, since it reaches the throughput limits seen by the bare-metal micro-benchmark described in Section 5.1.2.

5.2.7 Remote Swap I/O Throughput with DMA

We measured I/O throughput achieved using the remote swap device, when it is implemented utilizing DMA transfers instead of Load/Store data copying. We used the same utility, *dd* as in the previous Section. The Write and Read data transfer throughputs with DMA operations reach the DMA throughput limit, seen by bare-metal DMA operations, described in Section 5.1.3.

5.2.8 Swap Devices Comparison

As shown in Figures 5.7 and 5.8, we compared different devices used for swapping. In those Figures, the Gray line represents throughput achieved when using remote swap over Ethernet with the *Network Block Device (nbd)*. The orange line shows throughput for a swap partition that resides in the SD card that contains the Linux kernel and the Ubuntu 12.04 root file system.

We implemented remote swap over Ethernet using the *nbd* in the following way. Using the native 1 Gbps Ethernet interface of the Zedboard platform, we connected the node with a PC with a Linux OS *back-to-back*, using a *cross* Ethernet cable. The swap partition on the PC resides in main memory (*ramdisk*), in order not to suffer from hard disk throughput limitations. Using the *nbd* driver we created a block device in the Compute Node's Linux. The client driver is invoked when read()/write() occur at that device and sends those requests over the Ethernet to the PC, where the server side of the *nbd* accesses the swap ramdisk and services the requests. Results shown in Figures 5.7 and 5.8, show that our implementation achieves double data transfer throughput for Write requests for all block sizes. For Read throughput, block size has a huge impact on the *nbd* throughput, resulting in a better performance than our implementation for large block sizes. However, at the most important block size, that is 4K - the size of a page, our implementation achieves 3 times greater throughput. In general, TCP throughput achieved using the native 1 Gbps Ethernet interface is limited to 450 Mbps Half-Duplex, as we measured in similar experiments using the *iperf* utility.

Using a swap partition in the SD card, we measured Write data transfer throughput at 50 Mbps and Read throughput from 50 to 100 Mbps depending on the block size. We observe, that the swap I/O throughput is limited by the SD card device I/O throughput, that is much smaller than throughput achieved in our implementation.

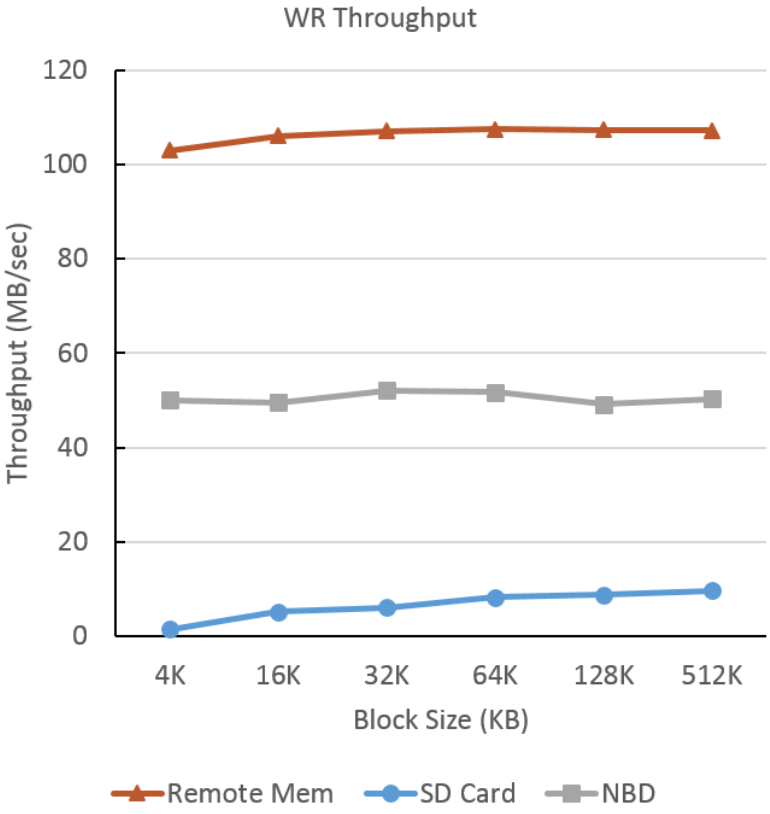

Figure 5.7: Swap write I/O throughput

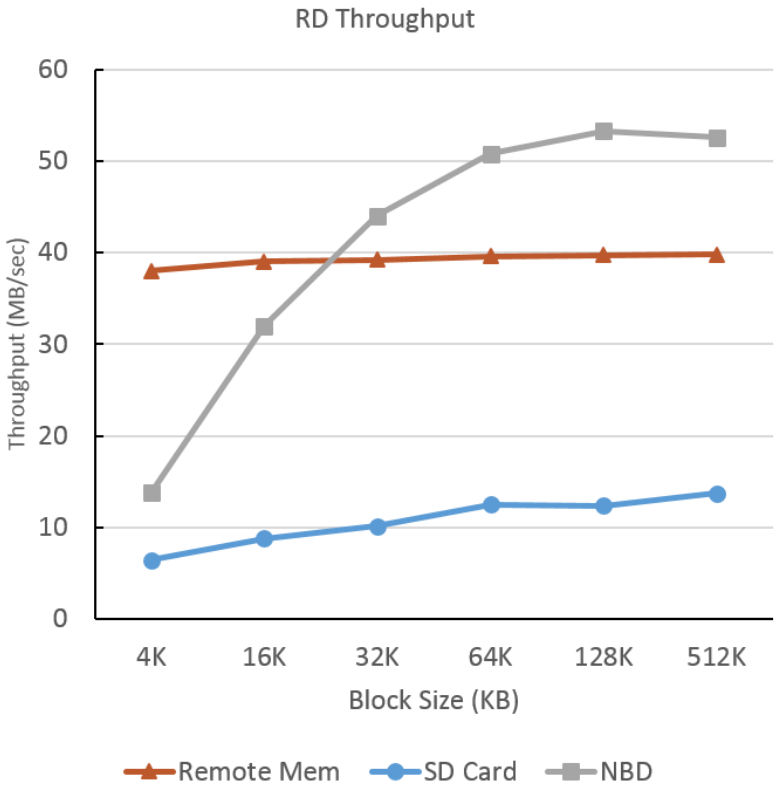

Figure 5.8: Swap read I/O throughput

Chapter 6

Shared Network Interface

In a microserver environment where many Compute Nodes are assembled to build a large system, it is unfeasible to dedicate I/O resources to each node, firstly because they are expensive and second because these resources cannot fit in the same chiplet. Thus, it is essential for expensive resources to be shared among Compute Nodes. The same happens with fast network interfaces that allow communication with the external world. (It is a common case for data centers to have a common entry/exit point to/from the Internet. All nodes inside the data center share that point, which also acts as a network router and many times as a load balancer.)

In our system we have a shared 10 Gbps NIC that follows the Ethernet protocol and is used as a common entry/exit point for all compute nodes connected to the main board. As mentioned in Section 3.3 it is virtualized by the hardware, providing dedicated TX/RX FIFOs for each Compute Node. Writing and reading to/from the shared NIC is done by the use of AXI DMA [39] that uses our custom interconnect.

6.1 Network Driver

We implemented a Linux kernel network driver [17] that allows the Operating System and the user space processes to view the 10Gbps NIC as a standard Ethernet device. That way, processes have an entry/exit point to the world outside the Discrete Prototype, without any need for modification or use of a special API. Using Berkeley Sockets [3], processes can use the TCP/IP protocol and also all protocols supported by Layer 2 Ethernet to communicate with the outside world.

Figure 6.1 shows the flow chart of the send/receive operations initiated by user space processes that use the Berkeley Socket API and the software and hardware components involved. A socket send() library call ends up as a system call trap, which gives control to the Operating System's network stack. (In Linux and Windows the whole TCP/IP Layer stack is implemented entirely in kernel space.) Data given by the process, are processed by each TCP/IP layer and finally, the kernel calls the drivers transmit function *hard_start_xmit()*. Finally, the driver that

manages the underlying hardware initiates a DMA transfer to the network device. The receive path is the other way around.

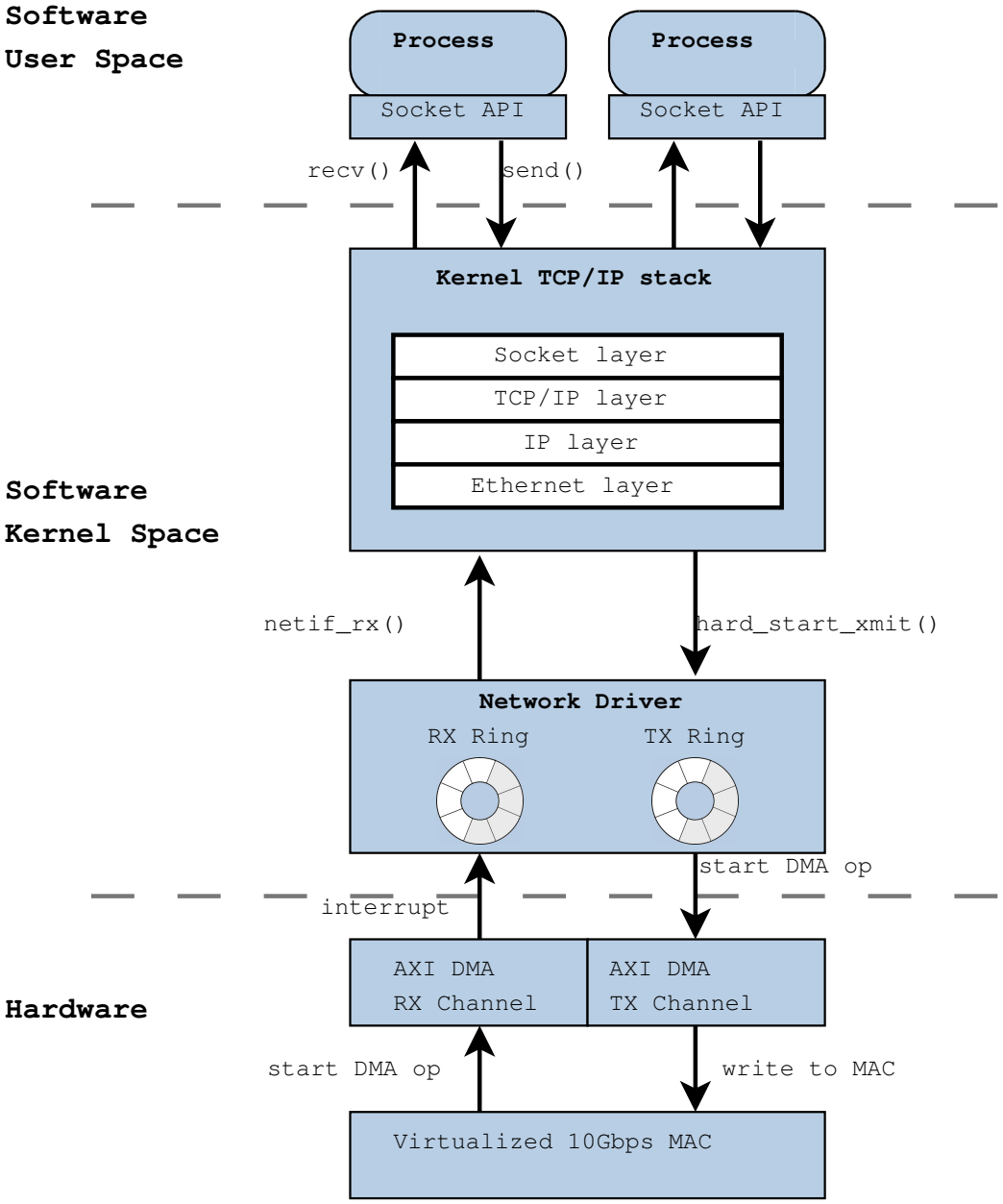

Figure 6.1: Network traffic send/recv flow chart.

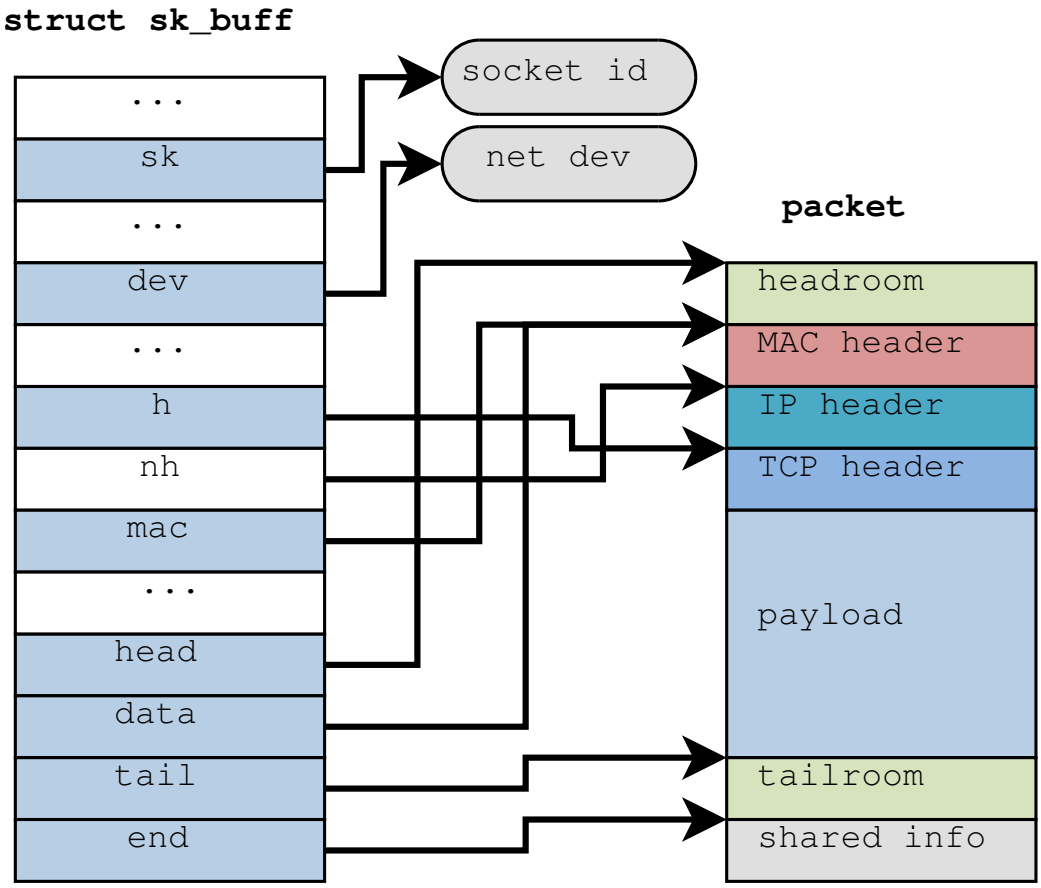

Figure 6.2: sk_buff structure.

Packet processing in the kernel is done with the use of the *sk_buff* structure, that is depicted in Figure 6.2. The *sk_buff* contains pointers to various segments of the packet for fast access.

As each layer takes control of the packet processing inside the kernel stack, it uses the corresponding pointer from the sk_buff structure to add/modify segments of the packet. The common case is every layer to add its corresponding header and some checksum. Payload is the segment of the packet that contains actual user space application data.

6.1.1 Simple Operation

In the simplest mode of operation, a user space process calls the send() system call stub with a pointer to a data buffer as an argument. When the kernel is invoked to handle the system call, the *inet* network layer of the kernel will copy the data buffer from the user space into a kernel space memory region and will also create an *sk_buff* structure to decribe that data. Then, the TCP layer will process the data

and add its own TCP header and checksum in the packet. The IP layer will add its header and header checksum in a similar way and finaly the Ethernet layer will add its header to the packet. At this moment, the packet data must be marshallized, in order to have each layer's header in the right order. As a result, the packet must be copied again before it is passed to the network driver for processing and hardware submission.

When the network driver takes control of the packet it submits it to the DMA engine to pass it to the hardware MAC block. When the packet transmission is completed the DMA engine raises an interrupt and the driver's interrupt handler is invoked to clear and reset its internal structures.

The receive path works in a similar fashion. The DMA engine writes the incoming data to a predefined memory region and raises an interrupt. The network driver's interrupt handler is then invoked, checks the packet and copies it to a new kernel memory region that is given to the kernel network stack.

6.1.2 Scatter-Gather & Ring Descriptors

With the simple operation described in Section 6.1.2, the packet needs to be processed before given to the driver's hard_start_xmit() function. As a result, the packet is copied twice until it reaches the hardware and this can negatively affect performance. Modern Operating Systems and network drivers operate in **scatter-gather** mode, which allows a packet to be split in several fragments that reside in different pages in memory. If the network driver supports scatter-gather operation, the kernel can give the fragmented packet directly to the driver without marshaling its content first, resulting in **zero-copy operation**. The common case in scatter-gather operation is the kernel network stack to have the headers of different layers in different memory pages as the packet travels through the layer stack and the layer headers are created. The sk_buff structure allows fragmentation of packets, because it includes pointers to the fragments of a packet. In Figure 6.3 an sk_buff structure describing a fragmented packet is shown.

Hardware and driver support is needed to enable scatter-gather operation of the network stack. The hardware must be able to get those fragments from the physical memory automatically, when given the appropriate arguments, marshal the fragments and transmit them using the physical device (PHY). Further more the driver must be implemented in a way that supports scatter-gather operation, in order to accept fragmented packets and set up the hardware appropriately.

In our implementation, we used the AXI DMA hardware block to transmit and receive data to/from the 10 Gbps MAC block. The AXI DMA resides in the FPGA of the Compute Node, whereas the 10 Gbps MAC resides in the FPGA of the central board. Refer to Section 3.2 for a description of Discrete Prototype Generation 2. The DMA engine is configured to operate in scatter-gather mode. It can accept a linked-list of descriptors, with a head and tail pointer. When given as arguments in the DMA, the head and tail descriptors physical addresses and the linked-list has been properly set up, then the DMA will start fetching all packet

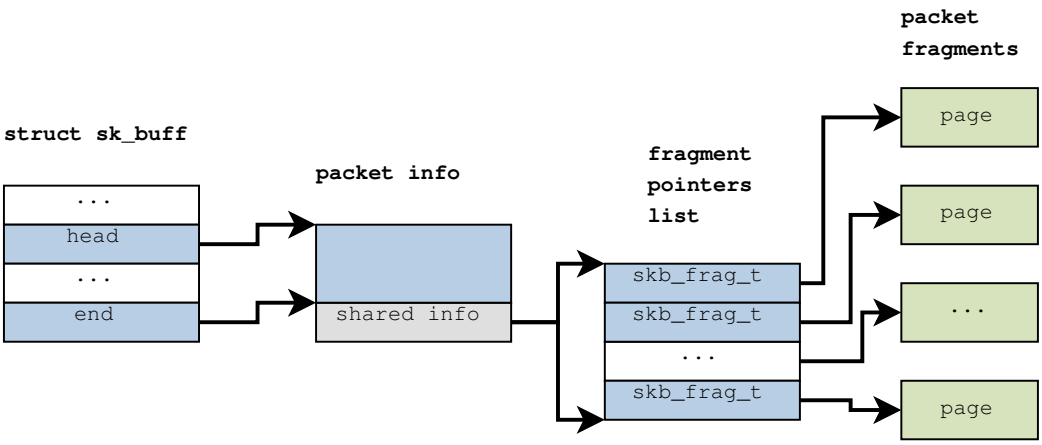

Figure 6.3: sk_buff structure describing fragmented packet.

fragments by reading each descriptor in the linked-list. A descriptor, compatible to the AXI DMA engine, as used in our design is shown in Figure 6.4. Each such descriptor is 16 words = 64 Bytes large and contains several pointers and control registers about packet fragments. As seen in the Picture, word *next desc phys addr* is the **physical address** of the next descriptor and it is set up fr each descriptor start up of the network stack by the driver to create a linked-list of descriptors. Word *frag phys addr* is the **physical address** of the packet fragment. Word *ctrl* is a DMA control register for that fragment and it usually contains the size of the fragment and some additional tags. Word *status* is written by the DMA engine and contains the error code upon transfer completion or abortion. Word *app4* is a DMA user field, not used by the hardware in our design. It is used by the driver to keep the **virtual address** of the *sk_buff* that describes a series of fragments that comprise a packet. The rest of the words are not used by the software.

Note that the physical addresses stored in the fragment descriptors must be DMA capable as described in the Linux kernel manual. The DMA engine must be able to read/write from/to those addresses. Whenever a driver needs such a physical address, it requests from the kernel by the use of the *dma_map_single()* function.

The chain of descriptors can be implemented in many ways. We implemented our driver with two circular buffers of descriptors, called the **descriptors rings**. The transmit path has 64 such descriptors in its ring and the receive path has 128. Each descriptor in the ring is a structure the same as described in Figure 6.4. Furthermore, we maintain a head and tail pointer for each ring, to keep track of used and free descriptors. This implementation allows **simultaneous access by the hardware and the software** to the descriptor rings and their corresponding fragments in physical memory, without any issue. Simultaneous access can further be described as simultaneous access of the external DMA engine and the CPU.

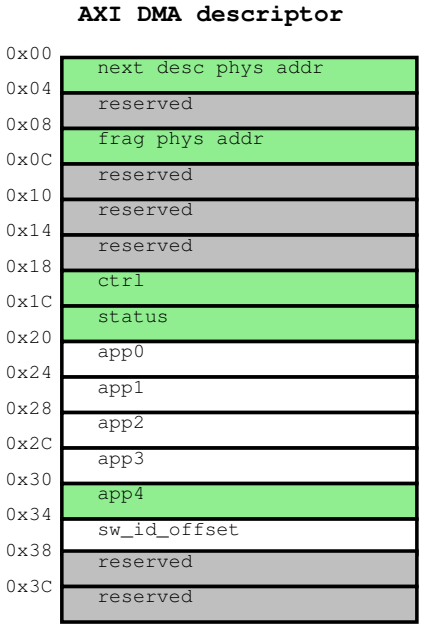

Figure 6.4: AXI DMA descriptor.

6.1.3 Set Up

When the driver module is loaded by the kernel it sets up the descriptors for the TX/RX path and configures the DMA engine. For the TX path, 64 descriptors of 64 Byte each are allocated next to each other, by the use of the *dma_ alloc_ coherent()* function, that returns both the physical and the virtual address of the allocated space. Their field, *next desc phys addr* is set up in such a way to create a circular array (or buffer), with the last descriptor pointing back to the first the array. The RX descriptors ring is set up in a similar fashion. For the RX path we also need to allocate additional space, in order the DMA to automatically put the incoming frames. For each descriptor in the RX ring we also allocate a buffer with size equal to the network MTU, using the *netdev_ alloc_ skb_ ip_ align()* function, which allocates and prepare that space for placement of an skb. The virtual address of the allocated skbs are given to the field *sw_ id_ offset* of their corresponding descriptors. Finally, a DMA capable physical address is requested for each allocated skb, using the *dma_ map_ single()* function. That physical address is stored in the *frag phys addr* of the appropriate descriptor.

Finally, the DMA engine is initialized, given the TX/RX head pointers. The RX DMA channel starts running and is ready to accept frames. The TX DMA channel is waiting to transmit when its register for the tail pointer is written.

6.1.4 Transmitting a Frame

When a whole or fragmented frame is ready to be sent by the network stack, the kernel calls the driver's *hard_ start_ xmit()* function with the corresponding *sk_ buff* pointer as an argument. This function must find of how many fragments the packet is consisted and find equal number of free descriptors in the TX ring. If there are available descriptors, it must set their fields to point to the corresponding fragments of the frame. Finally, it marks the first and last fragment of the frame as START_OF_FRAME and END_OF_FRAME accordingly. If the frame consists of only one fragment, its corresponding descriptor is marked with both START_OF_FRAME and END_OF_FRAME. It updates the tail pointer accordingly and writes it to the DMA TX TAIL register. Upon write, the DMA starts fetching those descriptors and their corresponding for transmission.

When transmission of a frame is completed by the DMA engine, it will raise an interrupt to the CPU. The driver's TX interrupt handler will run, check for DMA errors for each transmitted descriptors' *status* field and finally release all used descriptors, by updating the head pointer. If an error has occurred, it will schedule a task that will later reset the DMA engine and reinitialize the TX descriptors ring. Flow chart of a transmit operation from the user space process to the hardware and interrupt generation is shown in Figure 6.5.

In Figure 6.6 the descriptors ring with the pointers is depicted. *head* points to the first descriptor of the fragment subset that is process by the DMA, while tails points to the last. *curr* is DMA register that keeps track of the current descriptor processed by the DMA and is updated automatically. The *head* pointer is updated by the driver's transmit interrupt handler and *tail* is updated by the *hard_ start_ xmit()* function. The following subsets of descriptors can exists in the ring the same time:

1. head - curr: descriptors for fragments that have been transmitted by the DMA, but not yet released by the driver.
2. curr - tail: descriptors for fragments that have been submitted to the DMA, but not yet transmitted.
3. tail - head: free descriptors available for *hard_ start_ xmit()*

Function *hard_ start_ xmit()* takes the following steps when called by the kernel:

1. Finds how many fragments the frame (described by *sk_ buff*) consists of.
2. Checks if there is an equal number of free descriptors in the TX ring.
3. Reads the *tail* pointer and starts setting up descriptors for each frame fragment. In the *frag phys addr* field it sets the physical address of each fragment obtained by *dma_ map_ single()*. Stores the size of the fragment in the *ctrl* field and the virtual address of the *sk_ buff* in the *app4* field, to remember the sk_buff each fragment belongs to. For each descriptor completed

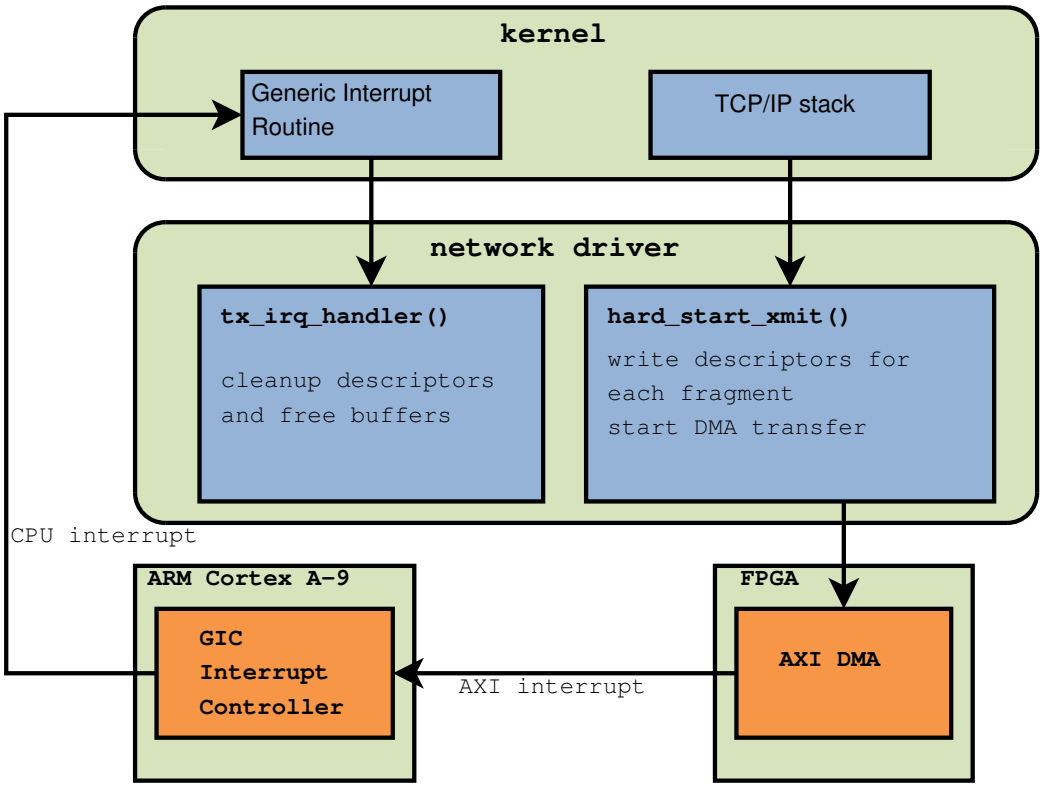

Figure 6.5: TX flow chart.

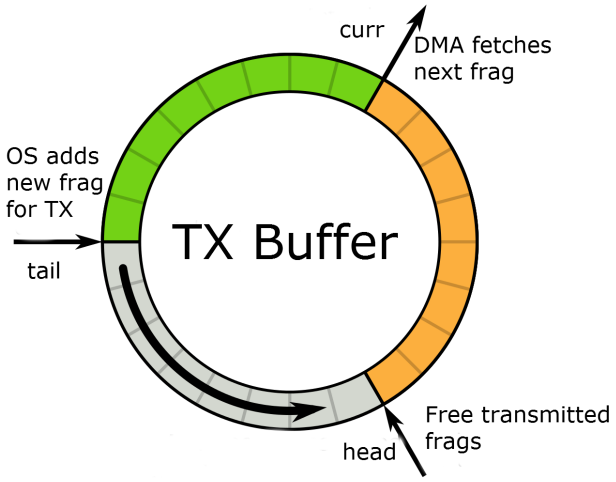

Figure 6.6: TX descriptors ring.

it increases the *tail* pointer, always in a circular fashion. (*tail %= NUMBER_OF_DESCRIPTORS*).

4. Marks the first descriptor of the frame as START_OF_FRAME and the last one as END_OF_FRAME.

5. Starts the DMA transfer by writing the physical address of the new tail descriptor to the appropriate DMA register. DMA starts operation automatically upon write on this register.

6.1.5 Receiving a Frame

Once the RX descriptors ring, the appropriate buffer space for new *sk_buff* frames has been allocated and set up, and the DMA RX channel has been configured, the driver is ready to accept incoming frames. **Driver is always accepting incoming frames regardless of blocked user space processes waiting to receive packets.**

The reception of network data involves two independent paths, in order an incoming packet to find its way up to the user space process. For a process to receive data from network it must notify the kernel network stack that it is waiting incoming data, by calling the *recv()* system call. The reception of data is a **synchronous** operation from the user space perspective, meaning that the call of *recv()* is a **blocking** one. In Figure 6.7 the events that occur in data reception and the hardware and software components involved is shown, both from the process' and the driver's perspective.

The 10Gbps MAC hardware block uses the DMA engine to write an incoming frame to the predefined memory segments. When the DMA has finished writing the frame into memory it raises an interrupt that ends up to the execution of the driver's receive interrupt handler. The descriptors in the RX ring are updated by the DMA automatically when writing the frame into memory. The interrupt handles checks the appropriate status DMA register for errors and gives the received frame to the upper network stack in an *sk_buff* format, using the *netif_rx()* function. It releases the corresponding descriptors and allocates new *sk_buff* space for each descriptor, the same way as in set up process (Section 6.1.3). Note that the DMA engine writes an incoming frame to a single buffer space, using only one descriptor for each frame. If errors have occurred during frame reception, the driver will schedule a task to reset the DMA engine and reinitialize the RX descriptors ring in the near future.

After the frame has been given to the kernel, the network stack does some processing of the frame and checks if there are any processes waiting for packets with that destination information. If there are such processes, it **copies** the received packet payload to the process's user space buffer and wakes up that process that is blocked in the *recv()* system call.

There are not any frame copies occurred between the driver and the kernel.

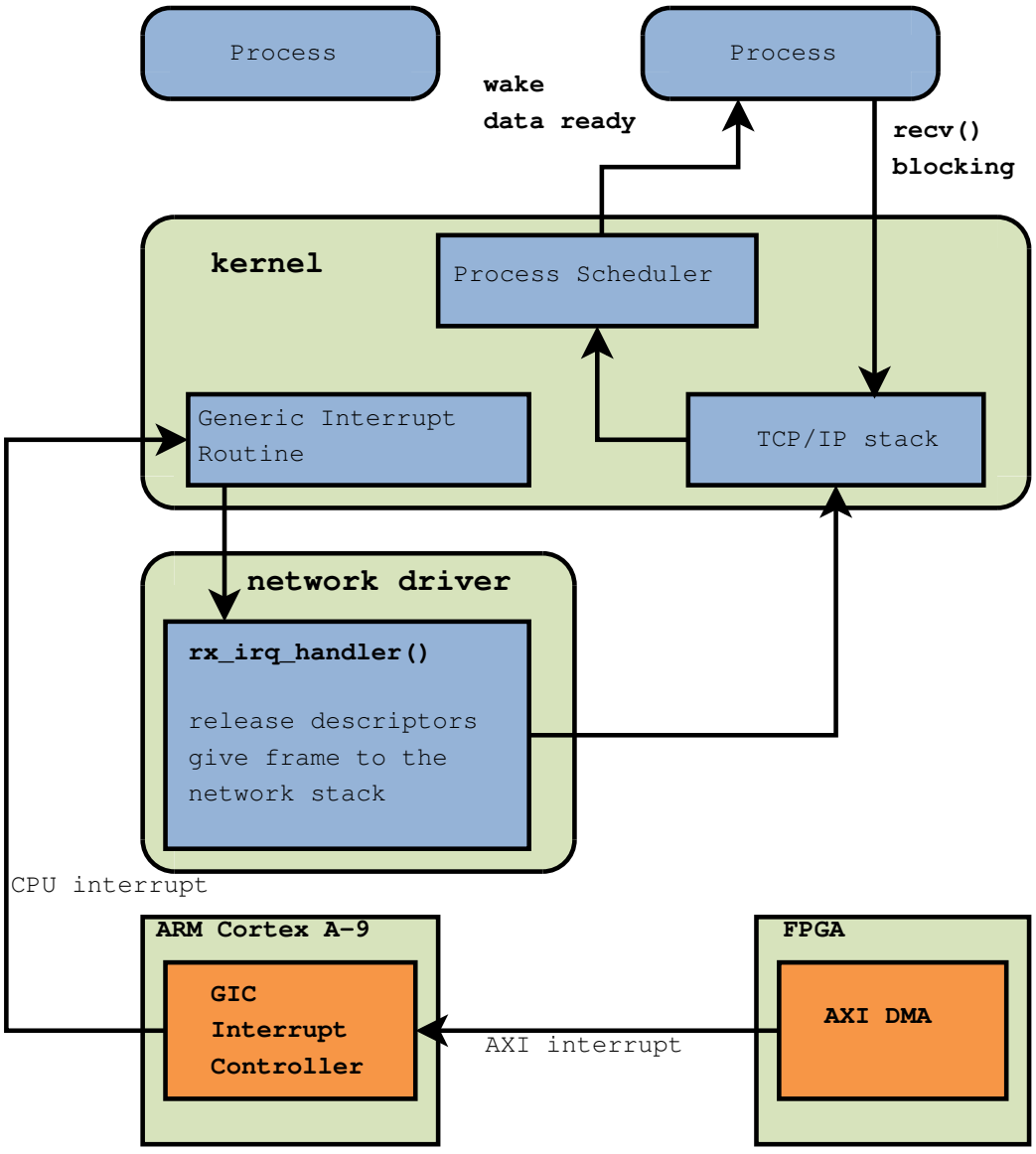

Figure 6.7: RX flow chart.

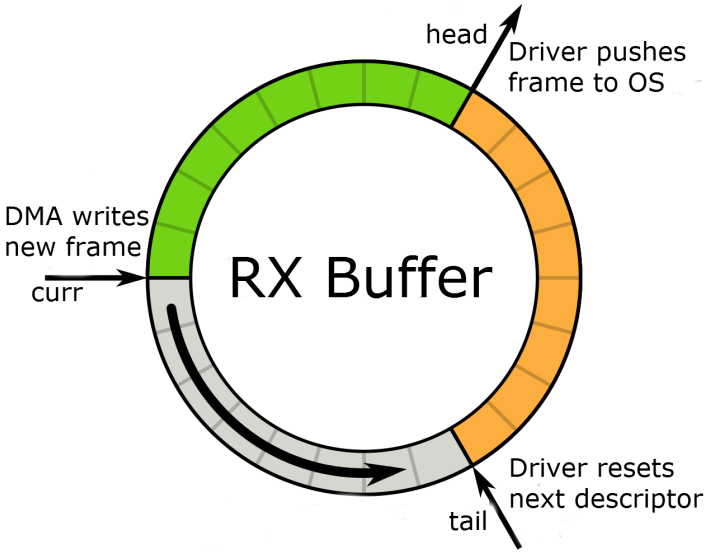

Figure 6.8: RX descriptors ring.

Along with the descriptors ring, the driver maintains a head and a tail pointer (Figure 6.8.) The first points to first descriptor written by the DMA engine, but not yet processed by the receive interrupt handler. The later points to the last free descriptor that can be used by the DMA engine. The DMA also maintains a pointer that keeps track of the current descriptor the DMA is writing. Head and tail pointers are updated by the driver's receive interrupt handles, when it is invoked and their new values are given to the DMA.

Three descriptor subsets can exist in the ring during operation:

1. head - curr: descriptors recently written by the DMA, but not processed yet by the driver's interrupt handler.

2. curr - tail: remaining free descriptors, submitted to the DMA for writing.

3. tail - head: descriptors written by the DMA that are being processed by the interrupt handler.

In our implementation each descriptor represents a whole Ethernet frame, since the DMA writes the received Ethernet frame non-fragmented in one memory buffer, as mentioned before. When the interrupt handler is invoked it does the following operations:

1. Checks for errors reported by the DMA engine in the status register. If an error has occurred, it schedules a task that will reset the DMA and reinitialize the RX ring in the near future.

2. Reads the tail and head pointers and then reads all descriptors in that range.

3. For each descriptor, it reads its status field for errors.
4. Marks the frame as CHECKSUM_UNNECESSARY, to tell the kernel not to reevaluate the checksum.
5. Pushes the frame to the kernel network stack, by calling the *netif_rx()* function.
6. Allocates a new buffer and sets it up as an *sk_buff* space, using the *netdev_alloc_skb_ip_align()* function.
7. Increases the tail pointer.
8. Submits the physical address of tail descriptor to the DMA

6.1.6 Checksum Offloading

In the Linux Operating System it is **required** to have TCP and IP header checksum hardware support, in order to enable scatter-gather operation. The process of calculating those checksum in the hardware is called *Checksum Offloading*. This policy of the Linux Operating System, ensures that the scatter-gather operation will not be slowed down by the calculation of TCP and IP header checksums in the network stack. At this moment, FORTH is developing a TCP and IP header checksum hardware block that will reside in the FPGA of the main board. We were not able to use that block in this work. As a result, we had to find a work-around, in order to enable scatter-gather operation and also have the kernel calculate the TCP and IP header checksums as well. This was achieved by enabling two flags, that are conflicting according to the Linux network driver implementation guidelines. The first flag enables the scatter-gather operation (tells the kernel that the driver can handle fragmented frames) and the latter enables checksum calculation in the kernel's network stack. There is not any issue, in the Linux 3.12.0 kernel version by enabling both those flags, but we are not sure if this applies for other kernel versions. We will use the hardware checksum block in the near future. Segmentation offloading can further improve performance, because it allows the kernel network stack to bypass many steps of the TCP layer that do checks and segment the packet if necessary. TCP packet segmentation, is a costly process, because the kernel must correctly split the packet into segments with appropriate headers, sequence numbers, etc.

6.1.7 Interrupt Coalesce

Since interrupt handling by the Operating System is an expensive procedure, that can increase CPU utilization when incoming/outgoing frame rate increases. When the CPU utilization is high enough, packet drops occur and throughput is limited. Modern network drivers and devices, support different interrupt rates. For example, a network device may raise interrupts upon the arrival of every 10 frames, as

opposed to interrupt per frame arrival. This leads, to reduced interrupt service time per number of frames and most importantly increased throughput, because of the reduced CPU utilization for the same frame rate. This method is called *Interrupt Coalesce*, because it groups events of many frame arrivals into a single interrupt ([32], [27] and [41]). Of course, this comes at a cost of response latency for both TX/RX paths. To mitigate this effect, hardware blocks employ a timer, which raises an interrupt when it expires.

Our network driver and the underlying hardware blocks support interrupt coalescing. The transmit and receive functions and interrupt handlers are implemented in such a way, that can operate in all interrupt coalesce settings, from one interrupt per frame to one interrupt per many frames. The AXI DMA engine we use, can be configured to a required interrupt coalesce. The timer can also be configured.

6.1.8 *ethtool* Interface

Our network driver implements a set of functions and exports them to the user space for usage by the *ethtool* utility. Supporting those features is a standard in modern network drivers, since one can access and configure many settings of the network device's MAC and PHY blocks and the network driver. Our implementation involves the access of the 10 Gbps MAC and PHY hardware blocks. The custom underlying hardware interconnect with the Physical Address Translation block, enables the software of a Compute Node to access the 10 Gbps MAC layer that resides in the main board's FPGA, since it is mapped in to the node's physical memory address space.

We implemented the following functions of the *ethtool* interface (Table 6.1:

	Function	Description
1	get_drvinfo()	Reports driver and device information
2	get_link()	Reports whether the physical link is up
3	get_ring_param()	Shows TX and RX descriptor rings Sizes
4	get_regs_len()	Gets buffer length required for *get_ regs()*
5	get_regs()	Gets device registers
6	get_coalesce()	Gets interrupt coalescing parameters.

Table 6.1: Implemented functions of the *ethtool* interface

6.1.9 MAC Configuration

The network driver can access all registers of the MAC block, since they are mapped on the physical address space of the Compute Nodes. Those registers keep the MAC configuration settings and also network statistics as well. MAC Configuration registers can be either writable or read only. Since the MAC register space is **not virtualized**, modifying a writable configuration register will affect all Compute

Nodes that shared the network device. The configuration registers of the network device are shown in table 6.2.

	Register	Description
1	RCW0	RX configuration word 0
2	RCW1	RX configuration word 1
3	TCW	TX configuration word
4	FCC	Flow control configuration word
5	RSC	Reconciliation sublayer configuration word
6	RXMTU	RX MTU configuration word
7	TXMTU	TX MTU configuration word
8	VR	Device version register (Read only)
9	CR	Capability register (Read only)

Table 6.2: 10Gbps MAC Configuration Registers

Except the aforementioned configuration registers, the MAC also contains a large set of detailed statistics registers that keep track of all transmitted Ethernet frames, errors, etc. It also keeps statistics about different frame sizes.

procfs

To expose the MAC configuration and statistics registers to the user space environment, we used the Linux *procfs* file system. For each MAC register the network driver creates a procfs entry that can be either writable or non- writable. When a user space process reads or write from/to that procfs entry (just like an ordinary file), the appropriate method of our network driver is called that either reads the requested register from the MAC block and returns its value, or writes the register with the data given by the process. All the procfs directories and files for the MAC registers are created upon driver loading.

6.1.10 Management Data Input/Output (MDIO) & XGMII

Management Data Input/Output (MDIO) is is a serial bus interface and protocol (cite) that is used to transfer management information between the MAC and the PHY hardware blocks in modern network devices. With our hardware, a software on Compute Node, can access the registers of the network device that contain the information of the MDIO. Our driver can access those registers and expose them to the user space, in the *procfs* file system (See subsection 6.1.9). One can view information or configure the PHY hardware block by reading or writing the appropriate bits of the according MDIO registers.

XGMII

10 Gigabit Media Independent Interface (XGMII) is a standard defined in IEEE 802.3 ([24] for connecting full duplex 10 Gigabit Ethernet (10GbE) physical blocks to each other and to other electronic devices on a printed circuit board. The XGMII is exposed to the Compute Nodes in our Discrete Prototype, through the MDIO interface, discussed in Section 6.1.10.

The network driver implements two methods for reading and writing to two MDIO registers that result in reading or writing raw byte data from/to the PHY block through the XGMII. Writing raw data to the PHY, results in actual physical transmission over the 10Gbps optical cable. Those read/write methods are exposed to the user space, through the *procfs* system and are useful for debugging of the hardware prototype.

6.1.11 Network Statistics

During its operation, the network driver collects statistics about the IP and Ethernet traffic that passes through it. It also counts the dropped packets and the transmission reception errors occurred. Update of those counters is done in the transmit and receive functions and interrupt handlers.Those statistics can be obtained by the standard Linux utility for IP networking: *ifconfig*.

6.1.12 Device Tree

To load the network driver and map the DMA and MAC register space into physical memory the Operating System needs to know about them at boot time. In ARM architectures the appropriate Device Tree file must be used, that replaces the function of BIOS in traditional x86 architectures. In our system, two device tree entries must be added, in order the Linux kernel to load our driver and map the hardware register spaces. The device tree segments are shown in Listing 6.1

```
axi-dma@82000000 {
  eusrv-connected = <0x6>;
  compatible = "xlnx,axi-dma-6.03.a";
  interrupt-parent = <0x2>;
  interrupts = <0x0 0x1e 0x4 0x0 0x1d 0x4>;
  reg = <0x82000000 0x10000>;
  xlnx,family = "zynq";
  xlnx,generic = <0x0>;
  linux,phandle = <0x7>;
  phandle = <0x7>;
};

eusrv-eth@41000000 {
  eusrv-connected = <0x7>;
  compatible = "eusrv,eusrv-ethernet-1.00.a";
  device_type = "network";
  interrupt-parent = <0x2>;
  local-mac-address = [00 00 1a 12 34 56];
  reg = <0x41000000 0x10000>;
  linux,phandle = <0x6>;
  phandle = <0x6>;
```

```
};
```

Listing 6.1: Device Tree memory segments for the Network Interface

The first entry named *axi-dma@82000000* is the segment that describes the AXI DMA block that resides in the FPGA of each compute node. Its member field *reg* instructs our network driver to map the register space of the DMA into physical memory region 0x82000000 - 0x82010000. The second entry, named *eusrv-eth@41000000*, describes the MAC block that resides in the main board's FPGA. Its field *compatible*, instructs the Linux kernel **to load our network driver** (the driver has a same tag in its source code). Field *local-mac-address*, tells the network driver to create a standard Ethernet interface in the Linux kernel with MAC address 00:00:1a:12:34:56. The field *reg*, instructs the driver to map the MAC register space into physical memory region 0x41000000 - 0x41010000. Lastly, the field *interrupt-parent*, defines that the interrupt controller responsible for interrupts generated from the device is the *Generic Interrupt Controller (GIC)* of the Xilinx Zynq-7000 SoC.

The two device tree entries are **linked** together by the fields *phandle*. This results, by use of both devices by the network driver.

6.1.13 Zero Copy Anecdotes

As mentioned previously, scatter-gather operation allows socket buffers not to be copied from the kernel to the driver. With the use of descriptors rings, hardware and software can process the data simultaneous without problems. Although, this operation is mentioned *zero copy* in the existing literature, it is not an absolute zero-copy, since the whole procedure still contains a buffer copy between user and kernel space. It is essential to copy the buffer that is given as an argument to a socket send() from the process context to the kernel memory space. When the system call trap is done, the kernel uses the function *copy_from_user()* to replicate the buffer given to a *send()* system call as an argument. One may wonder why is that procedure necessary? Why does the kernel not simply pin the corresponding user pages into memory in order to achieve absolute zero copy? Well, the necessity of the replications comes from the uncertainty of the buffers lifetime. The kernel cannot know beforehand if the buffer used in a *send()* call will be not modified long enough, for the network transaction to be completed. A user space process code might do the following:

```
send(sockfd, buff, size, flags);
memcpy(input_buff, buff, new_size);
```

In Listing 6.1.13 the process uses the buffer *buff* as a network data buffer given as an argument to the *send()* system call. When the process is unblocked from the *send()* call it uses again the same buffer to do something else. However, the TCP/IP protocol stack of the kernel may need to dome some retransmissions of data in the near future. No one can predict if that retransmission is going to

happen or not. **If it is essential to retransmit some data, what happens if the buffer used before has not been copied and was modified as in the example (Listing 6.1.13)?**.

The aforementioned example shows why replication of network data buffers from user space to kernel space is essential. However, there is a way to achieve **absolute zero copy** that is suitable for optimized and **carefully written** applications, such as an Apache Web Server. The use of the *sendfile()* system call [9], forces the kernel network stack not to duplicate the user space buffer. Instead it pins the corresponding user space pages into memory with the function *get_user_pages()*. Thus, the buffer data travel through the network stack to the network device driver without a copy. This is the real zero copy operation. Note that the Linux kernel must be configured to allow real zero copy when using the *sendfile()* system call. Otherwise, this system call behaves just like *send()*.

In the same fashion, the Windows Operating Systems supports absolute zero copy with the use of *TransmitFile()* function [12].

6.2 Evaluation

The Generation 2 Discrete Prototype described in Section 3.2 and shown in Figure 3.6 has a theoretical throughput limit of 10Gbps. The interconnection circuit of the 4 Compute Nodes to the main board and the custom PCB board can achieve data rates higher than 10Gbps. The 10 Gbps NIC available bandwidth is the upper limit the whole system can achieve.

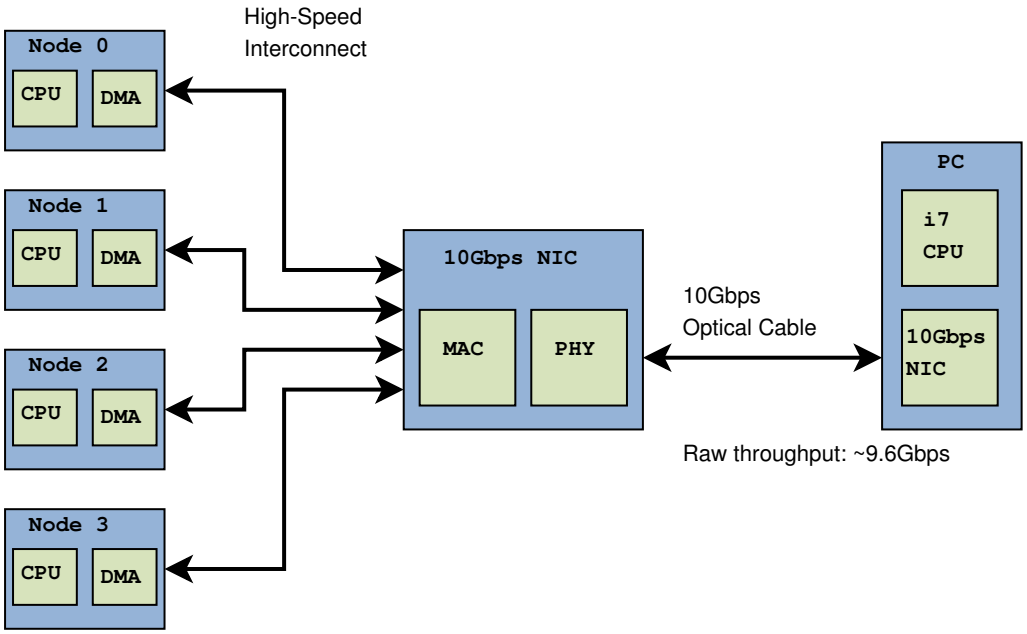

Figure 6.9: Shared NIC evaluation setup.

The evaluation setup is shown in Figure 6.9 and comprises of our Discrete Prototype connected to a PC with a 10Gbps capable optical cable. The PC consists of a modern Intel i7 CPU with 8 GByte of DDR3 RAM and has a PCIe network card that has an optical transceiver and is capable of 10Gbps. The Operating System in the PC is Xubuntu Linux 14.04, with the official drivers of the network card installed.

6.2.1 Bare-Metal Throughput

To saturate the throughput limit of the 10Gbps NIC, we implemented a bare metal application that runs in each Compute Node and uses the AXI DMA engines in the nodes in scatter-gather configuration. This mode, allows the AXI DMA to read /write ring buffers without stopping its operation. When used for transmission it reads whatever data it finds in the ring and transmits them. Although, there is no applicable usage of this type of DMA operation, it is useful to debug hardware components by achieving the maximum throughput the DMA engine can reach.

By running this experiment, we confirmed that we can get very close to the theoretical throughput limit, by achieving **9.8 Gbps** total throughput when all Compute Nodes generate traffic. The throughput was equally shared between the Compute Nodes, because of the round-robin TX scheduler of the MAC hardware block. Each node achieves **2.2 Gbps** throughput.

When only one node is active, the maximum throughput it can achieve is **3.2 Gbps**. This time, the upper limit is the CPU. It cannot generate more packets per unit of time. **Traffic from a single node cannot saturate the 10 Gbps link.**

6.2.2 TCP Throughput

The most important evaluation is the throughput user space applications can achieve using the TCP protocol, since it is the most common for network communication. We measured the TCP throughput achieved by a user space process in a full system with Operating System and user space environment in each node. We used the *iperf* utility that is a standard tool used for measuring TCP and UDP throughput in networks.

First of all we measured the maximum transmission throughput a node can achieve. An *iperf server* is set up in the PC listening for incoming TCP connections. Then at a node we set up an *iperf client* that connects to the server and transmits process payload using TCP sockets (STREAM). The maximum usable TCP throughput achieved by each node is **960 Gbps**, for large packet sizes that are close to the Ethernet MTU size. During transmission, the CPU utilization was always at 150% (as shown by the *top* utility, meaning that both CPU cores were utilized.

In a similar way, we measured the maximum receive throughput a node can achieve by setting up an *iperf server* in a node and an *iperf client* in the PC. The result was **880 Mbps** for packet sizes close to the Ethernet MTU. CPU utilization was high (150%) in the receive process too.

Using the 4 nodes together, each one transmitting TCP packets, the aggregate throughput seen at the 10 Gbps NIC is the sum of the individual throughputs achieved by the nodes. As a result, the aggregate throughput is 4 x 880 Mbps = 3.52 Gbps. The sum of the individual throughputs is also confirmed when the 4 nodes receive packets.

In Figure 6.10 the aggregated TCP throughput is shown by the red line. The horizontal axis shows the number of nodes generating network traffic and the vertical axis is the aggregated throughput achieved in Gbps for that number of nodes. We see, that when only one node is participating in network traffic, the aggregated throughput is 880 Mbps that is the maximum TCP throughput a single node can obtain. When two nodes generate traffic the aggregated throughput is 2 x 880 Mbps = 1.76 Gbps and when three nodes generate traffic the aggregated throughput is 3 x 880 Mbps = 2.64 Gbps. Finally, when four nodes generate network traffic the aggregated throughput is 4 x 880 Mbps = 3.52 Gbps. We can see that each time a node is added in the experiment it adds its maximum TCP throughput to the

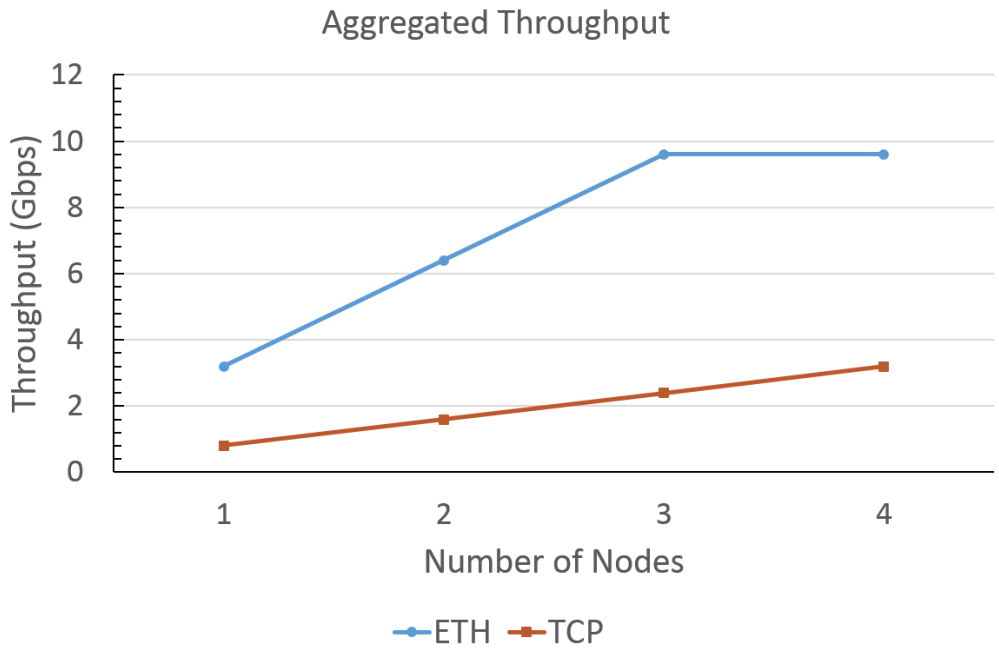

Figure 6.10: Throughput for TCP and Raw Eth

aggregated throughput. Aggregated TCP throughput with 4 nodes is far from link saturation (9.6 Gbps), due to high CPU utilization as described in the following Section.

CPU Utilization

The CPU utilization is the main cause that user space process cannot experience throughput close to the data rate seen by bare-metal applications that transmit/receive raw Ethernet packets. CPU utilization is very high due to the TCP and IP header checksums calculations in the kernel TCP/IP stack. In Section 6.1.6 we mentioned that we *fooled* the kernel to allow scatter-gather operation and at the same time calculate the checksums for us, because we do not have a TCP checksum offload engine yet.

We extracted the TCP checksum calculation function from the Linux kernel sources and ran that code in a user space application to measure the CPU clock cycles (or time) it takes to calculate one TCP checksum. The result is that for Ethernet frames that are the size of the MTU, the TCP checksum calculation accounts for more than 9000 CPU cycles (about 13 msec) per packet.

Concurrent TCP Flows

The high CPU utilization puts a limit to the throughput achieved when using TCP packets, so running the experiment with multiple concurrent TCP flows does not increase performance as it should. When we run 2 iperf client threads in a node (as the number of cores) that transmitted packets to the PC the throughput achieved was only 20 Mbps higher than that achieved with single-threaded iperf client, resulting in CPU utilization of 200% (both cores fully utilized).

The same situation occurs when measuring reception throughput with 2 iperf client threads running in the PC. The reception throughput seen in the node was only 20 Mbps higher compared to the single-threaded experiment.

Interrupt Coalescing

Using interrupt coalescing that is supported by our driver implementation, as mentioned in Section 6.1.7, also does not improve performance, as it should, because of the already high CPU utilization. We ran similar transmission and reception experiments with different coalescing settings and the throughput achieved is almost the same as when interrupt coalescing is not set.

Full-Duplex Operation

When TCP transmission and TCP reception flows are concurrent, the total throughput (both TX and RX) achieved in the node is 880 Gbps (same as the TCP transmission throughput). The throughput is almost equally split between the transmit and receive flows. Because of the high CPU utilization Full-Duplex operation does not result in the sum of the transmission and reception throughputs.

6.2.3 Raw Ethernet Throughput

To confirm that TCP throughput is limited by the high CPU utilization, cause by checksum calculation, we ran raw Ethernet frame transmission and reception benchmarks. We implemented a kernel module that is loaded in the nodes and the PC. The module uses the Layer 2 packet transmission function *dev_ queue_ xmit()* that is a pointer to our driver's *hard_ start_ xmit()* function. In order to send raw Ethernet frames, we had to create the packets in the form of *sk_ buff* that is described in Section 6.1. The module runs a loop transmitting a random Ethernet frame.

The first experiment measures the transmission throughput achieved at the nodes, so the raw frames module is loaded at the nodes. The PC runs the *ethstats* utility that measures the incoming data and packet rate at a specific network interface. The result was that the node achieved 3.0 Gbps of transmission throughput, very close to the throughput seen by a bare-metal application.

In a similar way, we measured reception throughput at the nodes. This time, the raw frames module is loaded in the PC and the *ethstats* utility is run in the

node. The reception throughput is 2.8 Gbps.

The aforementioned experiments, confirm that the high CPU utilization because of TCP and IP header checksum calculation is the dominant factor for throughput limitation when using the TCP protocol.

When using the 4 nodes together, we can saturate the 10 Gbps link, achieving almost 9.8 Gbps of raw Ethernet throughput as seen in Figure 6.10 by the blue line. When only one node generates raw Ethernet traffic it achives 3.2 Gbps of throughput. When two nodes generate traffic, the aggregated throughput seen is 2 x 3.2 Gbps = 6.4 Gbps. When three nodes are connected the aggregated throughout reaches the link saturation limit that is 3 x 3.2 Gbps = 9.6 Gbps. When four nodes generate traffic the aggregated throughput is still the link saturation but the throughput per node is decreased, so each node gets the same portion of the available data rate, due to the round-robin transmit scheduler in the FPGA.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work we showed that we can indeed deploy ARM architectures in larger-scale designs, but implementing essential mechanisms in the software level. The existing software support for ARM architectures, especially for many-core non-SMP designs is very poor, so essential steps were taken towards running Operating System efficiently in such customized hardware designs.

Resource allocation between different nodes, becomes a very important issue in large-scale designs and this work is focused on exactly that problem. We implemented Operating System and user space mechanisms for accessing and sharing those resources and demonstrated how they are used in an efficient way.

In particular, we implemented three different mechanisms for utilizing remote physical memory and we showed the pros and cons of each mechanisms addressing both efficient hardware mechanisms exploitation and software transparency. Furthermore, we implemented a network device driver enabling the Operating System and the processes that run on a node to access a shared and virtualized 10 Gbps NIC. Processes can send/receive TCP, UDP or Raw Ethernet traffic through the shared NIC using the standard Berkeley Sockets API, thus they do not have to be modified.

This work is one of the first steps implemented for incorporating ARM architectures in large-scale designs. We believe that this work will become more relevant with the upcoming 64-bit ARM architectures which target large-scale servers in the Data Center domain.

7.2 Future Work

Ongoing work, in the Euroserver Project, includes incorporating all mechanisms implemented on Discrete Prototype Generation 1 to Discrete Prototype Generation 2. Using remote memory requires modifying the boot sequence of the Linux kernel, to allow the Operating System to export or import shared physical pages.

For remote memory usage mechanisms in particular, we will implement the remote swap driver utilizing scatter-gather DMA. This will further increase data transfer throughput.

Using the remote physical memory as an I/O character device gives us many options for managing remote memory in the user space. We are exploring, user space allocators that will manage that remote memory in behalf of the processes. In specific, we are investigating a malloc() libc call interception mechanism that will allow processes to run unmodified, while memory allocation management will reside in a user space runtime environment.

As mentioned in previous chapters, viewing remote physical memory as an extension to the local DRAM raises some memory management issues, because the Operating System running on the platforms is not NUMA-aware and does not include memory distance vectors to add weights to the page replacement algorithm. Thus, we investigated modifying the available ARM Cortex A9 Linux kernel to support NUMA architectures and memory distance vectors. This work is still in progress, until a new NUMA-aware kernel becomes available with the upcoming 64-bit ARM processors.

Bibliography

[1] Avnet microzed. http://zedboard.org/product/microzed.

[2] Avnet zedboard. http://zedboard.org/product/zedboard.

[3] Berkeley sockets. http://en.wikipedia.org/wiki/Berkeley_sockets.

[4] Chapter 2: Describing physical memory. https://www.kernel.org/doc/gorman/html/understand/understand005.html.

[5] iwarp. http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf.

[6] Network block device (tcp version). nbd.sourceforge.net.

[7] Openstack. https://www.openstack.org/.

[8] Pci-sig single root i/o virtualization (sr-iov). https://www.pcisig.com/home/.

[9] sendfile(): Linux man page. http://linux.die.net/man/2/sendfile.

[10] Sockets direct protocol. http://en.wikipedia.org/wiki/Sockets_Direct_Protocol.

[11] Sr-iov passthrough for networking. https://wiki.openstack.org/wiki/SR-IOV-Passthrough-For-Networking.

[12] Transmitfile function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms740565%28v=vs.85%29.aspx.

[13] ARM. *Cortex-A9 MPCore Technical Reference Manual.*

[14] ARM. *AMBA AXI and ACE Protocol Specification.* 2011.

[15] Attia, B., Chouchene, W., Zitouni, A., and Tourki, R. Network interface sharing for socs based noc. IEEE.

[16] Compaq/Intel/Microsoft. *Virtual Interface Architecture Specification, Version 1.0.* 1997.

[17] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[18] D., G., D., T., and S., G. Architecture and implementation of sockets direct p rotocol in windows. IEEE.

[19] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[20] Ferrante, A., Medardoni, S., and Bertozzi, D. Network interface sharing techniques for area optimized noc architectures. IEEE.

[21] Flouris, M., and Markatos, E. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster Computing 2*, 4 (1999), 281–293.

[22] Forum, M. *MPI: A Message-Passing Interface Standard, Version 3.0.* 2012.

[23] Gerstenberger, R., Besta, M., and Hoefler, T. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)* (Nov. 2013), ACM, pp. 53:1–53:12.

[24] IEEE. *Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and Physical Layer specifications.* IEEE, 2008.

[25] InfiniBand. *RoCEv2: RDMA over Converged Ethernet, Version 2.* 2014.

[26] Intel. Intel xeon phi product familty. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[27] Intel. *Interrupt Moderation Using Intel GbE Controllers.*

[28] Intel. *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology.* 2011.

[29] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 267–278.

[30] Markatos, E. P., and Dramitinos, G. Implementation of a reliable remote memory pager. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996* (1996), pp. 177–190.

[31] Midorikawa, Kurokawa, Himeno, and Sato. Dlm: A distributed large memory system using remote memory swapping over cluster nodes. In *2008 IEEE International Conference on Cluster Computing* (Tsukuba, 2008), IEEE.

[32] Mogul, J., Western, D., Mogul, J. C., and Ramakrishnan, K. K. Eliminating receive livelock in an interrupt-driven kernel. vol. 15, pp. 217–252.

[33] Oguchi, M., and Kitsuregawa, M. Using available remote memory dynamically for parallel data mining application on atm-connected pc cluster. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International* (Cancun, 2000), IEEE.

[34] Oracle. *SPARC T4 Processor*. Oracle, 2011.

[35] Russell, R. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 95–103.

[36] Tung, N., Duy-Hieu, B., Hai-Phong, P., Trong-Trinh, D., and Xuan-Tu, T. High-performance adaption of arm processors into network-on-chip architectures. IEEE.

[37] Xilinx. *Xilinx AXI Central DMA Controller*.

[38] Xilinx. *Zynq-7000 All Programmable SoC: Technical Reference Manual, UG585 (v1.10) February 23, 2015*.

[39] Xilinx. *LogiCORE IP AXI DMA v7.1: Product Guide*. 2014.

[40] Yoo, R., Romano, A., and Kozyrakis, C. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *2009. IISWC 2009. IEEE International Symposium on Workload Characterization* (Austin, TX, 2009), IEEE.

[41] Zec, M., Mikuc, M., and Žagar, M. Estimating the impact of interrupt coalescing delays on steady state tcp throughput. In *in Proceedings of 10-th SoftCOM* (2002).

Appendices

Appendix A

Remote Memory Device Tree

The device tree for used in Compute Node 0 in Discrete Prototype Generation 1 is shown in Listing A.1. The higher 256 Mbytes of the DRAM of Compute Node 1 are used as remote memory for Compute Node 0, through the ACP port of the ARM processor.

```
/ {
        model = "Xillinux for Zedboard";
        interrupt-parent = <0x1>;
        compatible = "xlnx,zynq-zed";
        #size-cells = <0x1>;
        #address-cells = <0x1>;

        axi@0 {
                ranges;
                compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
                #size-cells = <0x1>;
                #address-cells = <0x1>;

                zed_oled {
                        spi-sdin-gpio = <0x3 0x3c 0x0>;
                        spi-sclk-gpio = <0x3 0x3b 0x0>;
                        spi-speed-hz = <0x3d0900>;
                        spi-bus-num = <0x2>;
                        dc-gpio = <0x3 0x3a 0x0>;
                        res-gpio = <0x3 0x39 0x0>;
                        vdd-gpio = <0x3 0x38 0x0>;
                        vbat-gpio = <0x3 0x37 0x0>;
                        compatible = "dglnt,pmodoled-gpio";
                };

                leds {
                        compatible = "gpio-leds";

                        mmc_led {
                                linux,default-trigger = "mmc0";
                                gpios = <0x3 0x7 0x0>;
                                label = "mmc_led";
                        };
                };

                swdt@f8005000 {
                        clock-frequency = <0x69f6bc7>;
                        reg = <0xf8005000 0x100>;
```

```
                compatible = "xlnx,ps7-wdt-1.00.a";
                device_type = "watchdog";
        };

        xadc@f8007100 {
                reg = <0xf8007100 0x20>;
                interrupts = <0x0 0x7 0x0>;
                compatible = "xlnx,ps7-xadc-1.00.a";
        };

        ps7-usb@e0002000 {
                reg = <0xe0002000 0x1000>;
                phy_type = "ulpi";
                interrupts = <0x0 0x15 0x0>;
                interrupt-parent = <0x1>;
                dr_mode = "host";
                compatible = "xlnx,ps7-usb-1.00.a";
        };

        serial@e0001000 {
                xlnx,uart-clk-freq-hz = <0x2faf080>;
                xlnx,has-modem = <0x0>;
                reg = <0xe0001000 0x1000>;
                interrupts = <0x0 0x32 0x0>;
                interrupt-parent = <0x1>;
                current-speed = <115200>;
                device_type = "serial";
                compatible = "xlnx,ps7-uart-1.00.a", "xlnx,xuartps";
                clock = <0x2faf080>;
        };

        ps7-ttc@f8001000 {
                reg = <0xf8001000 0x1000>;
                interrupts = <0x0 0xa 0x0 0x0 0xb 0x0 0x0 0xc 0x0>;
                interrupt-parent = <0x1>;
                compatible = "xlnx,ps7-ttc-1.00.a";
                clock-frequency-timer2 = <0x69f6bc8>;
                clock-frequency-timer1 = <0x69f6bc8>;
                clock-frequency-timer0 = <0x69f6bc8>;
        };

        ps7-sdio@e0100000 {
                xlnx,sdio-clk-freq-hz = <0x2faf080>;
                xlnx,has-wp = <0x1>;
                xlnx,has-power = <0x0>;
                xlnx,has-cd = <0x1>;
                reg = <0xe0100000 0x1000>;
                interrupts = <0x0 0x18 0x0>;
                interrupt-parent = <0x1>;
                compatible = "xlnx,ps7-sdio-1.00.a",
                             "xlnx,ps7-sdhci-1.00.a",
                             "generic-sdhci";
                clock-frequency = <0x2faf080>;
        };

        ps7-scuwdt@f8f00620 {
                reg = <0xf8f00620 0xe0>;
                interrupt-parent = <0x1>;
                compatible = "xlnx,ps7-scuwdt-1.00.a";
                clock-frequency = <0x13de4360>;
        };
```

```
ps7-scutimer@f8f00600 {
        reg = <0xf8f00600 0x20>;
        interrupt-parent = <0x1>;
        compatible = "xlnx,ps7-scutimer-1.00.a";
        clock-frequency = <0x13de4360>;
};

ps7-qspi-linear@fc000000 {
        xlnx,qspi-clk-freq-hz = <0xe4e1c0>;
        reg = <0xfc000000 0x1000000>;
        interrupt-parent = <0x1>;
        compatible = "xlnx,ps7-qspi-linear-1.00.a";
};

ps7-qspi@e000d000 {
        xlnx,qspi-mode = <0x0>;
        xlnx,qspi-clk-freq-hz = <0xbebc200>;
        xlnx,fb-clk = <0x1>;
        speed-hz = <0xbebc200>;
        reg = <0xe000d000 0x1000>;
        num-chip-select = <0x1>;
        is-dual = <0x0>;
        interrupts = <0x0 0x13 0x0>;
        interrupt-parent = <0x1>;
        compatible = "xlnx,ps7-qspi-1.00.a";
        bus-num = <0x0>;
};

ps7-iop-bus-config@e0200000 {
        reg = <0xe0200000 0x1000>;
        interrupt-parent = <0x1>;
        compatible = "xlnx,ps7-iop-bus-config-1.00.a";
};

ps7-gpio@e000a000 {
        phandle = <0x3>;
        linux,phandle = <0x3>;
        xlnx,mio-gpio-mask = <0xfe81>;
        xlnx,emio-gpio-width = <0x38>;
        reg = <0xe000a000 0x1000>;
        interrupts = <0x0 0x14 0x0>;
        interrupt-parent = <0x1>;
        #gpio-cells = <0x2>;
        compatible = "xlnx,ps7-gpio-1.00.a";
};

eth@e000b000 {
        xlnx,slcr-div1-10Mbps = <0x32>;
        xlnx,slcr-div1-100Mbps = <0x5>;
        xlnx,slcr-div1-1000Mbps = <0x1>;
        xlnx,slcr-div0-10Mbps = <0x8>;
        xlnx,slcr-div0-100Mbps = <0x8>;
        xlnx,slcr-div0-1000Mbps = <0x8>;
        xlnx,ptp-enet-clock = <0x69f6bc7>;
        #size-cells = <0x0>;
        #address-cells = <0x1>;
        phy-handle = <0x2>;
        interrupt-parent = <0x1>;
        interrupts = <0x0 0x16 0x0>;
        reg = <0xe000b000 0x1000>;
        compatible = "xlnx,ps7-ethernet-1.00.a";
```

```
                mdio {
                        #size-cells = <0x0>;
                        #address-cells = <0x1>;

                        phy@0 {
                                phandle = <0x2>;
                                linux,phandle = <0x2>;
                                marvell,reg-init = <0x3 0x10 0xff00 0x1e
                                                    0x3 0x11 0xfff0 0xa>;
                                reg = <0x0>;
                                device_type = "ethernet-phy";
                                compatible = "marvell,88e1510";
                        };
                };
        };

        ps7-dev-cfg@f8007000 {
                reg = <0xf8007000 0x1000>;
                interrupts = <0x0 0x8 0x0>;
                interrupt-parent = <0x1>;
                compatible = "xlnx,ps7-dev-cfg-1.00.a";
        };

        ps7-ddrc@f8006000 {
                xlnx,has-ecc = <0x0>;
                reg = <0xf8006000 0x1000>;
                interrupt-parent = <0x1>;
                compatible = "xlnx,ps7-ddrc-1.00.a";
        };

        pl310-controller@f8f02000 {
                reg = <0xf8f02000 0x1000>;
                interrupts = <0x0 0x22 0x4>;
                compatible = "arm,pl310-cache";
                cache-unified;
                cache-level = <0x2>;
                arm,tag-latency = <0x2 0x2 0x2>;
                arm,data-latency = <0x3 0x2 0x2>;
        };

        interrupt-controller@f8f01000 {
                phandle = <0x1>;
                linux,phandle = <0x1>;
                reg = <0xf8f01000 0x1000 0xf8f00100 0x100>;
                interrupt-controller;
                compatible = "arm,cortex-a9-gic";
                #interrupt-cells = <0x3>;
        };
};

memory@00000000 {
        reg = <0x0 0x20000000>;
        device_type = "memory";
};
memory@50000000 {
        reg = <0x50000000 0x10000000>;
        device_type = "memory";
};

cpus {
        #size-cells = <0x0>;
        #cdpus = <0x2>;
```

```
                #address-cells = <0x1>;

                ps7_cortexa9_1@1 {
                        xlnx,cpu-clk-freq-hz = <0x27bc86c0>;
                        xlnx,cpu-1x-clk-freq-hz = <0x69f6bcb>;
                        timebase-frequency = <0x13de4360>;
                        reg = <0x1>;
                        model = "ps7_cortexa9,1.00.a";
                        i-cache-size = <0x8000>;
                        i-cache-line-size = <0x20>;
                        device_type = "cpu";
                        d-cache-size = <0x8000>;
                        d-cache-line-size = <0x20>;
                        compatible = "xlnx,ps7-cortexa9-1.00.a";
                        clock-frequency = <0x27bc86c0>;
                };

                ps7_cortexa9_0@0 {
                        xlnx,cpu-clk-freq-hz = <0x27bc86c0>;
                        xlnx,cpu-1x-clk-freq-hz = <0x69f6bcb>;
                        timebase-frequency = <0x13de4360>;
                        reg = <0x0>;
                        model = "ps7_cortexa9,1.00.a";
                        i-cache-size = <0x8000>;
                        i-cache-line-size = <0x20>;
                        device_type = "cpu";
                        d-cache-size = <0x8000>;
                        d-cache-line-size = <0x20>;
                        compatible = "xlnx,ps7-cortexa9-1.00.a";
                        clock-frequency = <0x27bc86c0>;
                };
        };

        chosen {
                bootargs = "consoleblank=0 console=ttyPS0 root=/dev/mmcblk0p2
                rw rootwait earlyprintk bootmem_debug=1 memblock_debug=1";
                linux,stdout-path = "/axi@0/serial@e0001000";
        };
};
```

Listing A.1: Device tree for remote memory.

Appendix B

Bootable Media

The platforms that we use in our Discrete Prototype are the Avnet Zedboard for Generation 1 and Avnet Microzed ZC020 for Generation 2, as described in Section 3. Both of these platforms, require a bootable media, configured appropriatly, to boot an Operating System and the user space environment. The primary bootable media used is an SD card (SD card for Zedboard, MicroSD card for Microzed) formatted with 2 partiotions in it.

1. FAT32 partition that contains all files required to boot.
2. EXT4 partition that contains the root file system (optional).

The FAT32 boot partition is essential and contains all the files that are needed for a full system (or a bare-metal) boot. Figure B.1 shows the files that must be present in the partition.

File *fsbl.elf* is the **First Stage Boot Loader** for the ARM processor required to setup the processor and run a bare-metal application or an Operating System boot loader. It is created with the Xilinx SDK suite and is a compilation of the following

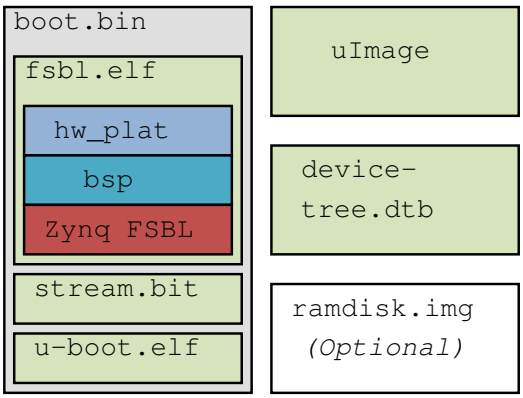

Figure B.1: Boot Parition of the SD card.

libraries: *hw_plat* that is a hardware platform description and is generated by importing the hardware description file and FPGA configuration bitstream from the Xilinx Vivado toolchain. The *bsp* or *Board Support Package* is the library that contains the API required for the specific platform, and is created by the Xilinx SDK. Finally, the *Zynq FSBL* is the application that setups the ARM processor and prepares it for a bare-metal application or an OS boot loader. Both three software parts, are linked to a single executable in ELF format, called *fsbl.elf*.

File *stream.bit* is the FPGA configuration (called bitstream) and is created by the Xilinx Vivado toolchain.

File *u-boot.elf* is the **Second Stage Boot Loader**. It is the standard Linux boot loader for ARM architectures.

All the aforementioned files are packaged together into a single archive, called *boot.bin*, using the *bootgen* utility that comes with the Xilinx Vivado suite. U-boot can be compiled from the sources in a Linux machine using the ARM cross compiler.

File *uImage* is the Linux kernel image in an uncompressed form. It is loaded by the u-boot into physical memory and executed. It can be compiled from the sources in a Linux environment using an ARM cross compiler and the appropriate compile settings.

devicetree.dtb is the Device Tree in a compiled binary form. It is loaded into physical memory by u-boot and is used by the kernel to load the appropriate drivers.

Optional *ramdisk.img* file is a root file system image of a Linux distribution. In our setupo, we do not use a ramdisk image. When used, it is loaded into physical memory by the u-boot.

If a permanent root file system is needed, the SD card must be formatted to contain an additional partition in the EXT4 (or any Linux supported filesystem) format. In our setup, the rest of the SD card (7.8 GByte) is formatted as an EXT4 partition and contains the Linaro Ubuntu 12.04 filesystem.

Selecting the permanent root file system rather than the ramdisk image, the Linux kernel boot arguments must contain the parameter shown in Listing B.1 that specifies that the root file system is the second partition of the SD card.

```
root=/dev/mmcblk0p2
```

Listing B.1: Permanent rootfs.