2015

Design and Implementation of a High Performance Network Processor with Dynamic Workload Management

Padmaja Duggisetty
University of Massachusetts Amherst

DESIGN AND IMPLEMENTATION OF A HIGH PERFORMANCE NETWORK PROCESSOR WITH DYNAMIC WORKLOAD MANAGEMENT

A Thesis Presented

by

PADMAJA DUGGISETTY

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2015

Department of Electrical and Computer Engineering

DESIGN AND IMPLEMENTATION OF A HIGH PERFORMANCE NETWORK PROCESSOR WITH DYNAMIC WORKLOAD MANAGEMENT

A Thesis Presented

by

PADMAJA DUGGISETTY

Approved as to style and content by:

Tilman Wolf, Chair

Russell Tessier, Member

Michael Zink, Member

C V Hollot, Department Head
Department of Electrical and Computer Engineering

DEDICATION

I dedicate this thesis to my Parents.Without their patience, understanding, support, and most of all love, the completion of this work would not have been possible.

ACKNOWLEDGEMENTS

I would like to thank Prof.Tilman Wolf for allowing me to work on this project and for his constant guidance and help. I am thankful to the University of Massachusetts, Amherst for allowing me to use the lab and other resources. I would like to express my gratitude to Prof.Russell Tessier and Prof.Michael Zink for their time and for accepting to be on my thesis committee. I am greatly thankful to Arman Pouraghily of Network systems lab for his support and guidance in this project. Finally, I extend my heartfelt thanks to my parents, family and friends who have been my constant source of support throughout my time here at the University.

ABSTRACT

DESIGN AND IMPLEMENTATION OF A HIGH PERFORMANCE NETWORK PROCESSOR WITH DYNAMIC WORKLOAD MANAGEMENT

SEPTEMBER 2015

PADMAJA DUGGISETTY

B.E., M S RAMAIAH INSTITUTE OF TECHNOLOGY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Tilman Wolf

Internet plays a crucial part in today's world. Be it personal communication, business transactions or social networking, internet is used everywhere and hence the speed of the communication infrastructure plays an important role. As the number of users increase the network usage increases i.e., the network data rates ramped up from a few Mb/s to Gb/s in less than a decade. Hence the network infrastructure needed a major upgrade to be able to support such high data rates. Technological advancements have enabled the communication links like optical fibres to support these high bandwidths, but the processing speed at the nodes remained constant. This created a need for specialised devices for packet processing in order to match the increasing line rates which led to emergence of network processors. Network processors were both programmable and flexible. To support the growing number of internet applications, a single core network processor has transformed into a multi/many core network processor with multiple cores on a single chip rather than just one core. This

improved the packet processing speeds and hence the performance of a network node. Multi-core network processors catered to the needs of a high bandwidth networks by exploiting the inherent packet-level parallelism in a network. But these processors still had intrinsic challenges like load balancing. In order to maximise throughput of these multi-core network processors, it is important to distribute the traffic evenly across all the cores.

This thesis describes a multi-core network processor with dynamic workload management. A multi-core network processor, which performs multiple applications is designed to act as a test bed for an effective workload management algorithm. An effective workload management algorithm is designed in order to distribute the workload evenly across all the available cores and hence maximise the performance of the network processor. Runtime statistics of all the cores were collected and updated at run time to aid in deciding the application to be performed on a core to to enable even distribution of workload among the cores. Hence, when an overloading of a core is detected, the applications to be performed on the cores are re-assigned. For testing purposes, we built a flexible and a reusable platform on NetFPGA 10G board which uses a FPGA-based approach to prototyping network devices. The performance of the designed workload management algorithm is tested by measuring the throughput of the system for varying workloads.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 An Introduction to Network Processors

With the rapid advancements in networking technology, the bottleneck no longer remains in the links which support high bandwidths but, rather at the nodes of the network where the packets are intercepted for processing. In the past, to overcome this performance bottleneck dedicated hardware, or ASICs were used to perform the processing at the network nodes instead of software running on general purpose processors. The dedicated hardware, by virtue of it being intended for only a limited set of operations, might be optimized to perform those operations and thereby attain higher performance. But a huge drawback was that the development cycle for ASICs was significant in terms of both time and cost. Since the time taken to deploy an ASIC is quite long say 18 month to 20 months, hence it could only be deployed after the solution was obsolete.Yet another drawback of ASICs is not being flexible and reprogrammable once customized to perform a certain operation. This is a huge drawback since the protocols deployed at the edges are constantly changing and ASIC deployed at the edge needs to be replaced each time a protocol is updated. It is also the case that it is desirable now for network nodes to perform an increasingly diverse set of operations. Finally, ASIC-based solutions also tend to be quite expensive. On the other hand though general purpose processors could be used to perform a variety of operations and can also be updated to go in hand with the advancements in the technology, the performance of these general purpose processors are less when compared to ASICs.

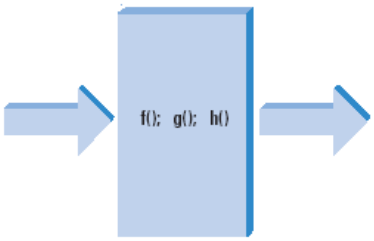

Figure 1.1. Single-Core Network Processor

The lack of a combined solution for flexibility and good performance was a motivation to design a dedicated hardware for packet processing at the nodes which was both flexible and had high performance. Network processors are processors tailored towards the computer network space, specifically packet processing. Network processors, which are high-performance and programmable solutions, offer an alternative to packet processing ASICs, which are high performance solutions but are not programmable, and also to general purpose processors, which are often lower performing devices, but are programmable. Initially, single core network processors were sufficient for the packet processing. A single-core network processor shown in Figure 1.1 must perform three functions f(), g(), h() on each packet. But as the internet transformed from a simple store and forward network to a more complex communication infrastructure. To meet the demands of such a complex system, a single core network processor was not sufficient since it was limited by the clock rates and memory access rates. All the efforts to improve the performance of a single core network processor were not enough to support the ever increasing bandwidth and complexity. This led to packet drops and delays in packet processing. To overcome this bottleneck at the nodes, the computer industry has started to migrate from single-core processors towards multi-core and many core processors. Multi-core or many core network processors as shown in Figure 1.2 typically contain multiple processor cores on a

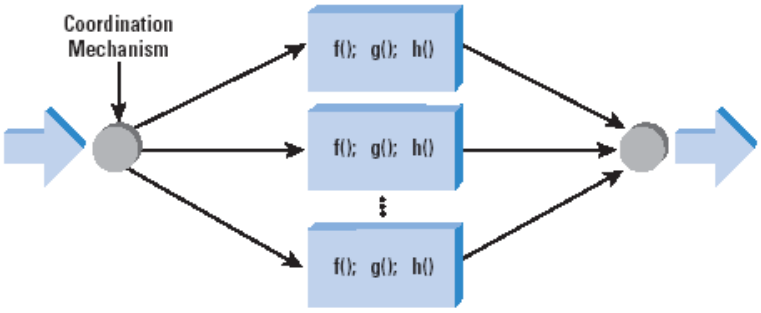

Figure 1.2. Multi-Core Network Processor

single chip, allowing multiple network packets to be processed concurrently. Figure 1.2 illustrates packet flow through an architecture that uses a parallel approach. A coordination mechanism on the ingress side chooses which packets are sent to which processor. Coordination hardware can use a simplistic round-robin approach in which a processor receives every Nth packet, or a sophisticated approach in which a processor receives a packet whenever the processor becomes idle. They exploit packet level parallelism in order to process multiple packets concurrently and thereby improve performance of the network processor. Hence multi-core network processors are a necessity in today's world of high packet volumes to keep up with the high bandwidth links.

1.2 Load balancing in Multi/Many-Core Network Processors

Multi-core network processors use multiple processing engines that process packets in a parallel of pipelined fashion. Most packets in network traffic do not exhibit inter dependencies and thus can be processed independently. This allows network processors to employ highly parallel architectures that are optimized for packet throughput. Though multi-core network processors are capable of improving packet processing rates by exploiting parallelism, they face some intrinsic challenges like load balancing. Load balancing is especially important in networks where it is difficult to predict

the amount of traffic at a given time at a node. In order to maximize throughput in such dynamic workloads, there is a need to distribute the workload evenly across all the cores. Otherwise, few cores might be overloaded when compared to others which might increase the packet processing times or may even lead to packet drops. Many algorithms have been previously designed to be able to distribute the workload evenly across the cores based on I-cache locality etc.,

1.3 Prototype Implementation on NetFPGA 10G

In our project, we build a platform i.e.,prototype of a high performance network processor with dynamic workload management. To build a flexible and a reusable platform we use NetFPGA 10G board which uses a FPGA-based approach to prototyping network devices. FPGA is an integrated circuit that can be reconfigured according to the required application by the designer. The configurable logic cell blocks are the basic logic units in an FPGA even though the features and number of logic cells used vary from device to device. Todays FPGAs are highly scalable, field debuggable, re-configurable, have a lower cost and are readily available in the market. They can be re-programmed using the powerful and versatile development tools commercially available today. It has a much shorter design cycle and requires much less engineering equipment costs compared with those for the design of Application Specific Integrated Circuits (ASIC). The FPGA has thus been widely used in both academy and industry to build test platforms to test design alternatives and validate theoretical research results. NetFPGA community has successfully implemented many testable platforms over the years which are targeted at academic researchers, industry users, students. For example, the community has successfully built many reference prototypes like reference router with line rates as high as 40 Gb/s, reference NIC, OSNT(open source network tester) etc., and hence is the motivation for us to choose NetFPGA 10G as our test bed. The specific contributions of our work are:

1. Design of a Multi-Core Network Processor prototype.

2. Development of an effective Dynamic Workload Management algorithm for the Multi-core Network Processor.

3. Prototype Implementation of the Multi-Core Network Processor on NetFPGA 10G. Competence of our system to adapt to the workload along with the the throughput performance, resource utilisation and processing capacity is evaluated.

CHAPTER 2

RELATED WORK

2.1 Trends in Network Processor Architecture

As the networking domain continues to grow in terms of bandwidth and the number of applications, it demands an equipment with very high throughput and also flexible to support newer applications and protocols. A network processor is such a software programmable device with architectural features and/or special circuitry for packet processing. Network processors share characteristics with many different implementation choices:

- network co-processors
- communication processors used for networking applications
- programmable state machines for routing
- reconfigurable fabrics (e.g. FPGAs)
- GPPs used for routing

Over the years, architecture of network processors has been continuously evolving to support a large number of applications and higher bandwidths. In today's era of dynamic workloads, interoperability between components present in a network processor affects the throughput and performance of a network processor and is highly researched on. Today's internet has grown from a traditional packet forwarding service to a network which supports various complex applications. This requires the processor at the nodes to be able to perform such complex operations and at

the same time maintain the throughput. The throughput is maintained by exploring the design space and coming up with a topology that best keeps up with the dynamic workload. Crowley et al. [7] explore various computer architectures and measure the performance of each of the architectures with varying work loads. The simulated workloads included packet forwarding routines, data encryption and authentication routines. Superscalar processors, fine grain multi-threaded processors, single chip multiprocessors and simultaneous multi-threaded processors were the different architectures which were explored in [7]. It was determined that simultaneous multi-threaded processor performs better than the rest of the architectures by exploiting both the thread level and instruction level parallelism. The chip multiprocessor and simultaneous multi-threaded processors achieve the greatest performance as a result of their ability to take the advantage of parallelism both within single threads and also across multiple threads. In [15] Wolf et al. analysed network workloads according to resource utilization to determine a performance model for various applications and topologies. They employed annotated directed acyclic graphs (ADAG) in order to capture runtime properties, such as number of instructions, memory reads and writes, and dependencies between instructions and clusters of instructions. The performance model takes the main system components into account and considers processing, memory access, inter-processor communication, and the effects of pipeline synchronization. The performance tradeoffs between different system topologies are presented and discussed and hence different system topologies are evaluated. This helps in understanding if the network processor is scalable and topology implemented could be modified for a better performance.

2.1.1 Network Processors in the Industry

Network processor based routers have been deployed in the industry by many major device vendors like Broadcom XLP832 [4], EZChip [14],Intel IXP [9], Cavium's

Octeon[5] or Cisco's Quantum Flow[6]. Commercial network processors first emerged in the late 1990's and were used in products as early as 2000. Though there were a lot of device vendors producing network processors around the year 2003, the number of vendors dwindled to less than 30 by the start of 2004. This was because many designs did not receive wide acceptance. The architecture of network processors has been continually evolving starting from a single core processor to multi-core and many-core processors. The number of processor cores in these chips range from as little as eight in the IXP2400 to over a hundred in the Cisco Silicon Packet Processor (SPP).

2.2 Load balancing algorithms

Multi-core network processors operating at a node avoid any bottlenecks caused due to high volumes of traffic flowing into a node by exploiting packet level parallelism and hence maintain the throughput. In [16] to overcome intrinsic challenges faced by multi-core network processors like load balancing, Wolf et al. attempt to address the issue of I-cache locality. When a core becomes idle, it searches for a packet of the same application as the previous one. They propose scheduling algorithms based on I-cache locality which clusters the processing engines which belong to a certain application. In [18], Wolf et al. consider a packet processing application as a graph where different tasks within application forms the nodes of the graph. It considers adjacency between nodes for task scheduling as the packet moves between different cores during processing. Using run-time profiling information about processing requirements and traffic characteristics, the system is able to adapt to dynamic changes in the workload and balance the utilization of all processing resources to maximize throughput. In [10] Iqbal et al. design and evaluate a scheduler for data plane packets in network processor. The packet scheduler adopts an efficient dynamic core allocation scheme for multiple services to improve throughput and to minimize out of order delivery

of packets by eliminating unnecessary flow migrations. Furthermore, the scheduler extends the hash based design for multi-service routers where the cores are dynamically allocated to services to improve I-Cache locality. Hence it reduces aggressive flow migration in multi-core network processors.

2.3 Prototype implementation of Network Processor

With the advent of programmable packet processors in the data path of routers, programmability of these processors is of prime importance in order to explore new arenas in network architectures. Hence, software development of programmable packet processors is a complex process since the programmer needs to make the best use of the resources. In [17] Wolf et al. have implemented a network processor design which separates programming from resource management. The prototype of a 4-core network processor has been implemented on a NetFPGA 1G platform. Each of the cores is a packet processing unit which is programmed for a certain application. This prototype has been simulated for two different types of applications i.e., IP forwarding and IPSec, one of which involves header processing and the other involves payload processing. The performance results claim that the architecture can be scaled when implemented on a platform with larger amount of resources than NetFPGA 1G board.

CHAPTER 3

DESIGN OF THE NETWORK PROCESSOR

3.1 General Framework

3.1.1 NetFPGA 10G Infrastructure

NetFPGA was a research project started in Stanford University in 2007 and was called NetFPGA 1G. This project allowed users to develop designs that are able to process packets at line-rate, a capability generally unafforded by software based approaches. NetFPGA focused on supporting developers that can share and build on each other's projects and IP building blocks. After 1G which had a Xilinx Virtex-II pro, the 10G board was designed with more resources and more upgrades. The NetFPGA-10G is an FPGA-based PCI Express board with 10-Gigabit SFP+ interface, a x8 gen1 PCIe adapter card incorporating Xilinxs Virtex-5 TX240TFPGA. The main features of this board are:

- Virtex-5 TX240T
- 10-Gigabit Ethernet networking ports
- 4 SFP+ connectors
- Quad Data Rate Static Random Access Memory (QDRII SRAM)
- Quad data rate (1.2 Giga transactions every second), synchronous with the logic
- Three parallel banks of 72 MBit QDRII+ memories
- Total capacity: 27 MBytes

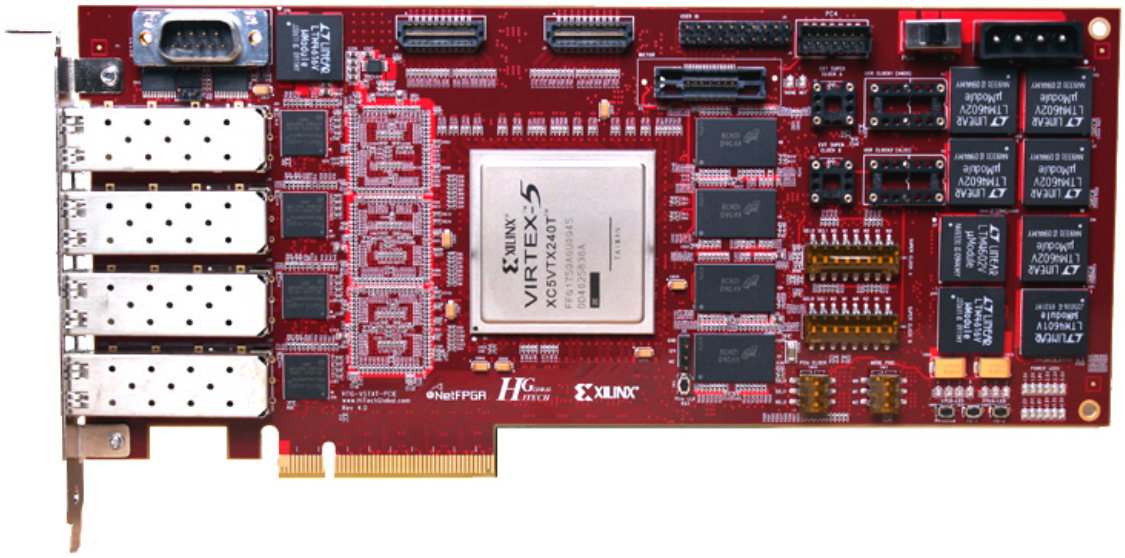

Figure 3.1. Front view of NetFPGA10G from [12]

- Cypress: CY7C1515KV18
- Reduced Latency Random Access Memory (RLDRAM II)
- Four x36 RLDRAMII on-board device
- First generation PCI Express interface, 2.5Gbps/lane
- Two FLASH devices
- Two SAMTEC QTH connectors
- (RS232) Connector
- User LEDs and Push Buttons

The FPGA top-level designs in the reference designs consist of a fair number of modules that are arranged in a somewhat flat hierarchy and are interconnected through standardized interfaces. In the reference designs, the hardware was divided into modules and these user-defined modules were implemented as IP cores. Each

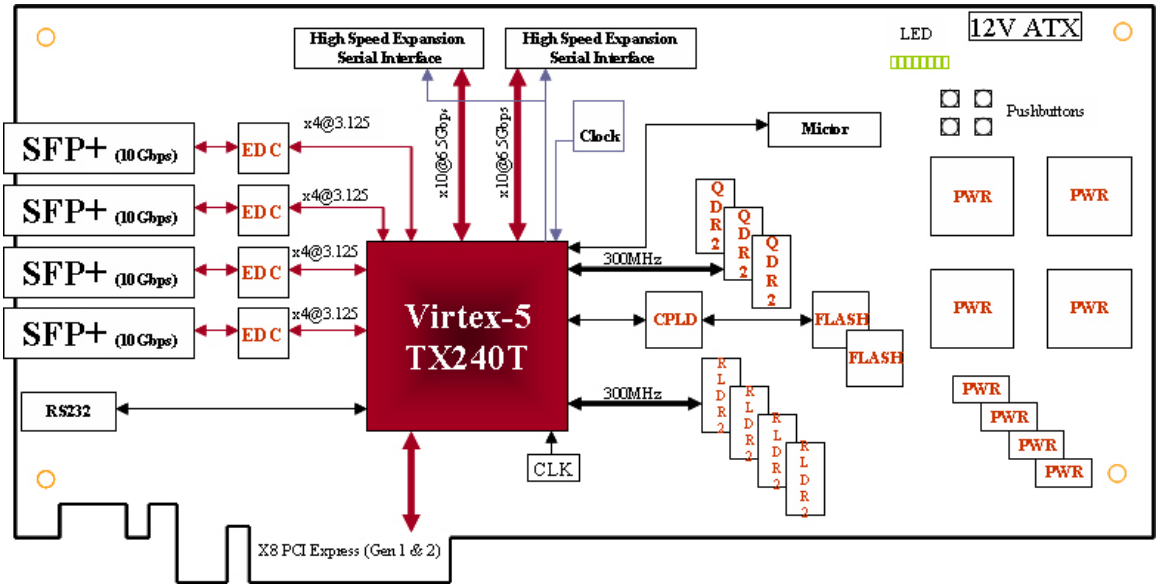

Figure 3.2. Block diagram depiction of various components in NetFPGA10G from [12]

reference design has a set of IP cores commonly called pcores and few of the most essential cores used in a reference design are described in the subsequent sections.

3.1.1.1 10G MAC

This pcore is a combination of Xilinx XAUI and 10G MAC IP cores, in addition to an AXI4-Stream adapter. Incoming XAUI signals from AEL2005 are firstly transformed into XGMII signals by Xilinx XAUI core. The XGMII signals are read in by Xilinx 10G MAC and finally transformed into AXI4-Stream. The TX side follows the exact same path but in the opposite direction.

3.1.1.2 Input Arbiter

The function of this block is to merge a number of input streams into one output stream. All input interface share the same bandwidth (and therefore width) as the output stream to ensure that maximum throughput can be achieved. The input port buffering is handled in the AXI Converter block.

3.1.1.3 Output Port Lookup

The function of this block is to set the destination port meta data field for all packets. The destination port is determined on the functionality of a reference design.

3.1.1.4 BRAM Output Queues

The function of this block is to dispatch packets from one input stream to a number of output streams whereby the destination port sub-band channel determines to which output the packets are routed. All input interfaces need to have the same bandwidth (and therefore width) as the output stream to ensure that maximum throughput can be achieved.

3.1.1.5 DMA Engine

This module serves as a DMA engine for the reference NIC design. It includes Xilinx' PCIe core and AXI4-LITE master module. To the other NetFPGA modules it exposes AXIS (master+slave) interfaces for sending/receiving packets, as well as a AXI4-LITE master interface through which all AXI registers can be accessed from the host (over PCIe). There is also included a set of AXI registers that can be connected via the AXI4-LITE slave bus to the same AXI interconnect for testing purposes. As there is only one AXIS set of interfaces, the module uses TUSER signal to multiplex between all four ports as defined in the NetFPGA standard IP interfaces documentation.

3.1.1.6 Standard IP interfaces

Use of interface standards is essential to the platform's goal of enabling rapid prototyping of networking applications. It increases IP interoperability, and therefore IP reuse. The NetFPGA-10G platform's standard interfaces themselves strictly adhere to a widely used interface standard released originally by ARM called AXI4, and later

refined by Xilinx. NetFPGA 10G currently specifies and supports the following set of interfaces:

- Control and Status Interface: The Xilinx standardized AXI4 lite interface which is derived from the AMBA4 bus specification is used as the control and status interface. AXI4 lite protocol is a simple read/write interface, single beat access, with 32bit address and 32bit data. It is basically implemented through 5 FIFO channels: read data, read address, write data, write address and write response. Components inside the FPGA design with AXI lite interface can be connected to the AXI4 lite interconnect in which all peripherals reside within a flat address space. Both PCIe and micorblaze act as masters on this interconnect and can write and read all peripherals. Addresses can be assigned manually, however EDK can automatically create an address map for all peripherals connected to this control/status network.

- Data Path Interface: The Xilinx AXI4 streaming interface (contiguous and aligned strobes) which is derived from the AMBA4 bus specification is used as data path interface.A data path interface is a group of five tightly coupled "channels". Each channel is a pre-configured Xilinx AXI4 Streaming Protocol v1.9 compliant stream. One of these channels is called the "data" channel. It is responsible for moving data from master to slave. All other channels included in a data path interface are for conveying metadata. These channels are referred to as "sub-band channels", and are "length", "source port", "destination port", and "error". Data channel and associated sub-band channels share the same clock and reset.

3.1.1.7 AXI4 Lite Protocol

The AXI4 Lite read and write transfers are as shown in the Figures 3.3 and 3.4.The AXI4-Lite interface is a subset of the AXI4 interface intended for communication with

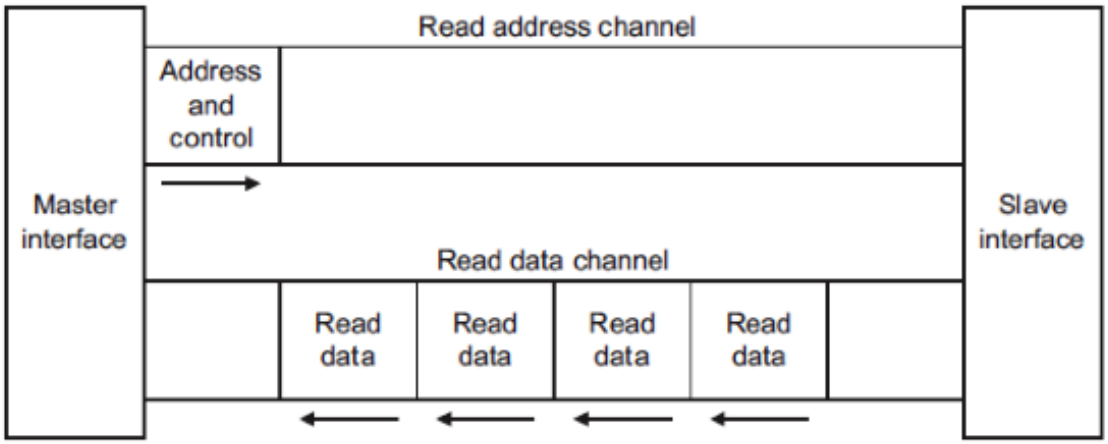

Figure 3.3. AXI4 Lite Read

control registers in components. The aim of AXI4-Lite is to allow simple component interfaces to be built that are smaller and also require less design and validation effort. Having a defined subset of the full AXI4 interface allows many different components to be built using the same subset and also allows a single common conversion component to be used to move between AXI4 and AXI4-Lite interfaces. The various signals used in AXI4 Lite are as shown in Figure 3.5.

3.1.1.8 AXI4 Stream Protocol

AXI4 stream protocol is designed for high speed data transfers. It supports unlimited burst mode.The data transfer always occurs between master and slave and not vice-versa. The AXI4 stream protocol does not have an address channel.The AXI4 streaming transfer is shown in Figure 3.6. Various signals used in AXI4 stream protocol are described in Figure 3.7.

3.1.1.9 AXI4 Stream Width Converter

This block converts the width of AXI stream (only integer ratio is supported in this version). For example, when used in 10G reference NIC, it converts 64bit datapath of 10G MAC to 256bit NetFPGA datapath. Optionally, it can set up the default

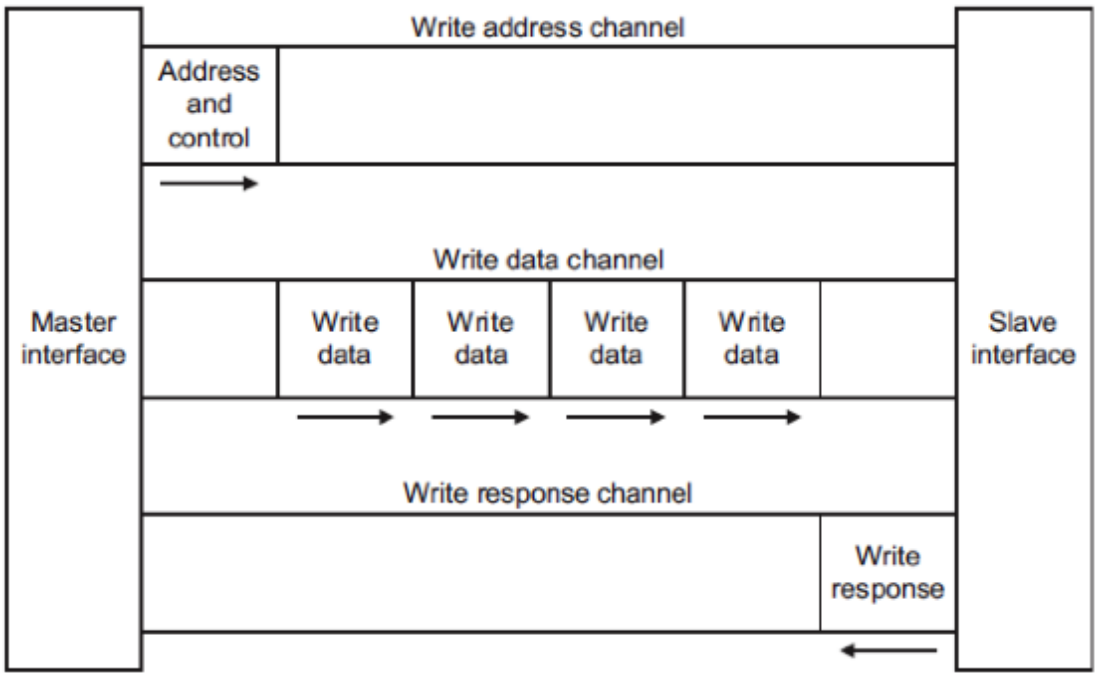

Figure 3.4. AXI4 Lite Write

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
–	AWADDR	WDATA	BRESP	ARADDR	RDATA
–	AWPROT	WSTRB	–	ARPROT	RRESP

Figure 3.5. AXI4 Lite Signal List

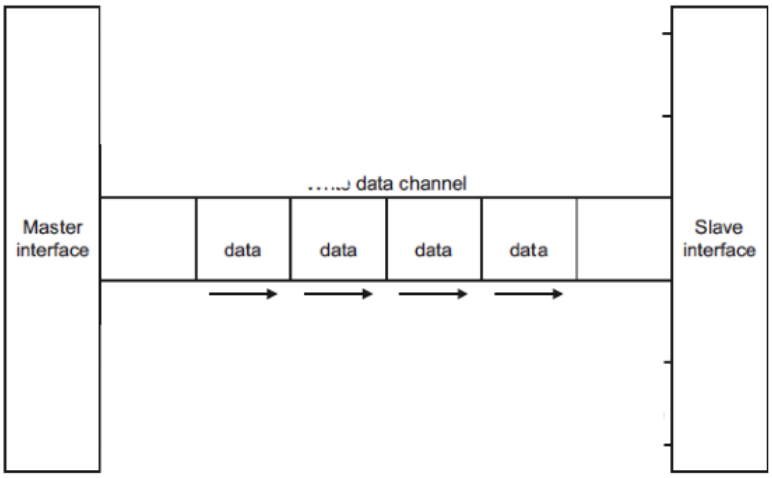

Figure 3.6. AXI4 Streaming Transfer

Signal	Source	Description
ACLK	Clock	Sample on the rising edge
ARESETn	Reset	Active Low
TVALID	M	A transfer takes place when both TVALID and TREADY are asserted
TREADY	S	A transfer takes place when both TVALID and TREADY are asserted
TDATA	M	Primary payload
TSTRB	M	Mark Position Byte
TKEEP	M	Mark NULL Byte (=Byte not enabled)
TLAST	M	Boundary of a packet
TID	M	Indicates different streams of data. Usually used by routing infrastructures
TDEST	M	Provides routing info. Usually used by routing infrastructures
TUSER	M	sideband info transmitted alongside the data stream

Figure 3.7. AXI4 Stream Signal List

metadata, e.g. destination port, source port etc. in the TUSER metadata field. It also calculates the length of packets and put it in TUSER field.

3.1.1.10 Microblaze

The Microblaze soft-core processor is a 32-bit Reduced Instruction Set Computer (RISC) architecture optimized for embedded applications. Microblaze can be user configured like pipeline depth, cache size, embedded peripherals, and bus-interfaces. The Microblaze core block diagram is shown in Figure 3.8. The fixed feature set of the processor includes:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

In addition to these fixed features, the Microblaze processor is parameterized to allow selective enabling of additional functionalities like floating point arithmetic, multiplication and division.There are a number of choices for Microblaze soft-core processor interconnections. It follows the Harvard architecture with separate paths for data and instruction accesses. Processor Local Bus (PLB) is a fully synchronous bus that provides connection to both on-chip and off-chip peripherals and memory. The Local Memory Bus (LMB) provides single-cycle access to on-chip dual-port Block RAM. The Xilinx Cache Link (XCL) interface is intended for use with specialized external memory controllers. Memory located outside the cache area is accessed through PLB or LMB. The debug interface is used with the Microblaze Debug Module (MDM) and is controlled through JTAG port by the Xilinx Microprocessor Debugger (XMD). MicroBlaze can be configured with up to 16 Fast Simplex Link (FSL) or

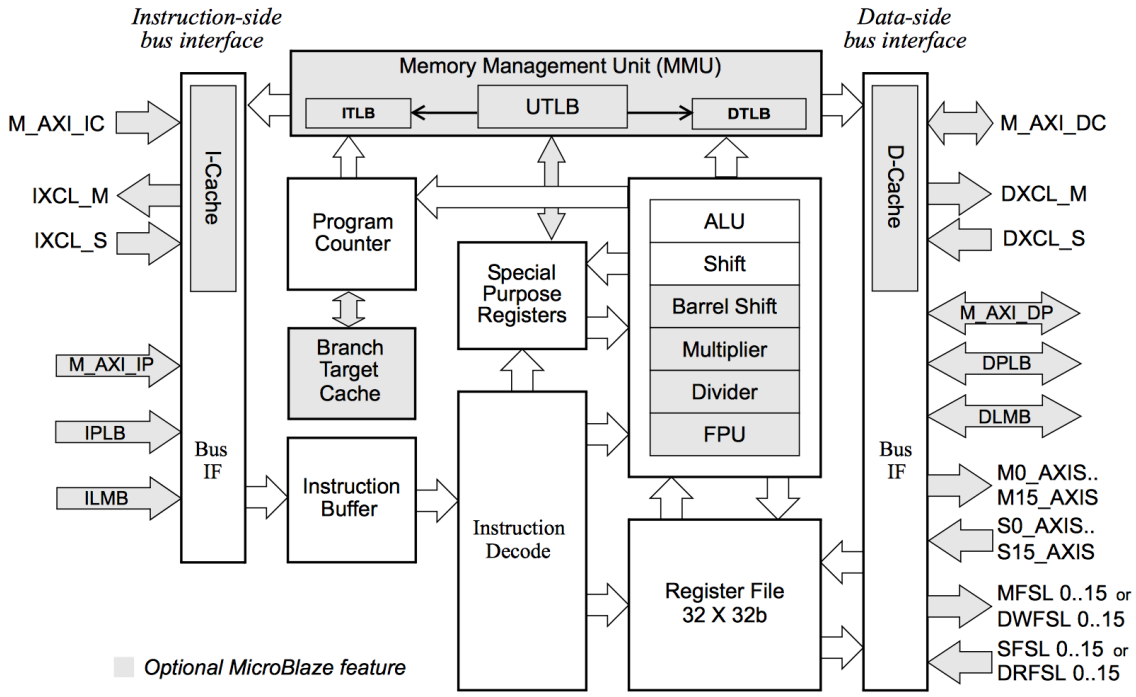

Figure 3.8. Microblaze Core Block Diagram from [1]

AXI4-Stream interfaces, each consisting of one input and one output port. The channels are dedicated uni-directional point- to-point data streaming interfaces.

Currently, NetFPGA-10G projects use the MicroBlaze soft processor to program the four on board AEL2005 PHY chips at start-up. The MicroBlaze writes a sequence of registers through the MDIO pcore after the FPGA gets programmed.

Our project is based on the reference NIC project built on NetFPGA 10G by the NetFPGA community. In the subsequent section we describe this project.

3.1.1.11 NetFPGA 10G Reference NIC

This is a reference NIC project [3] using the DMA engine. It includes many of the standard NetFPGA modules (microblaze, AXI interconnect, nf10 interface), but in addition it uses the DMA module to connect to the host over PCIe. The DMA module is connected to the rest of the NetFPGA system through AXIS (master+slave) for streaming packets and AXI4-LITE over which user code on the host can read/write

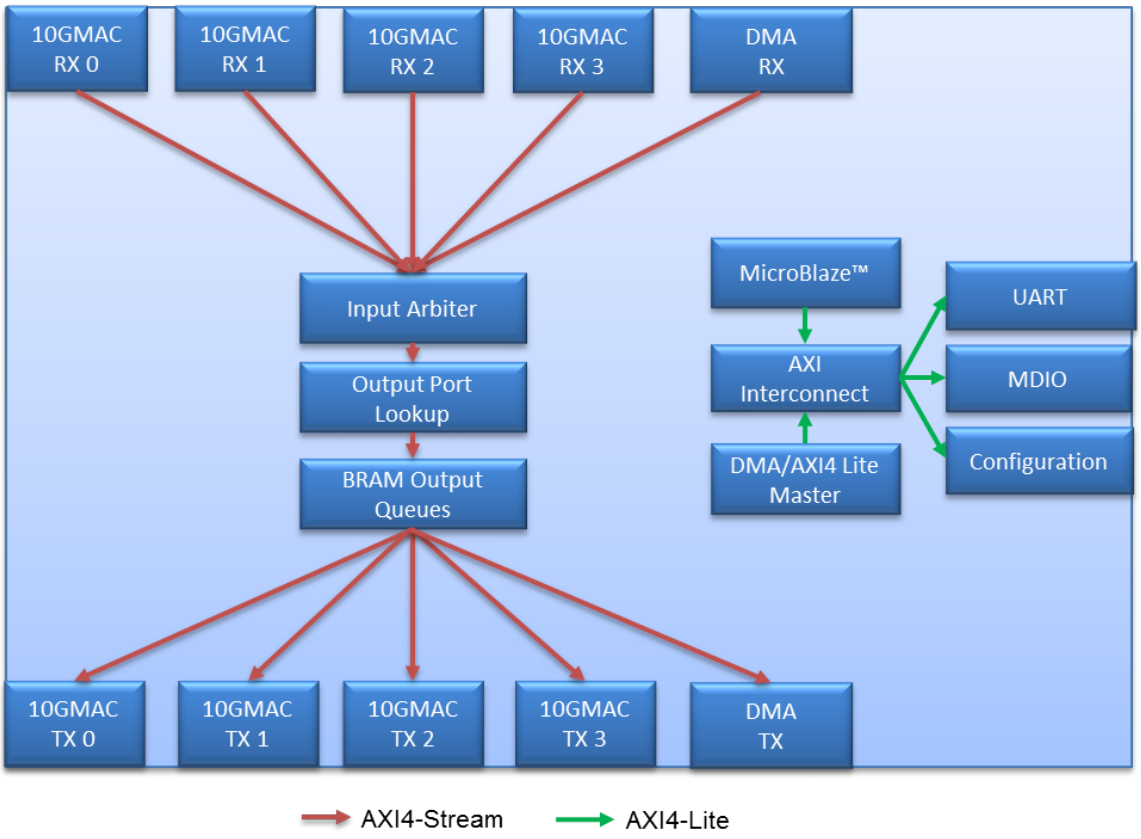

Figure 3.9. Datapath of Reference NIC from [3]

any AXI register in the system. This design is a pipeline where each stage is a separate module. A diagram of the pipeline is shown in the Figure 3.9.

Packets first enter the device through the 10G interface Rx modules, which connect next to the input arbiter module. The arbiter can operate in 1G or 10G mode; this is setup through selecting the data width accordingly i.e., 64-bit for 1G and 256-bit for 10G. The input arbiter has five input interfaces: four from the 10G interface modules and one from a DMA module. Each input to the arbiter connects to an input queue, which is in fact a small fall-through FIFO. The simple arbiter rotates between all the input queues in a round robin manner, each time selecting a non-empty queue and writing one full packet from it to the next stage in the data-path, which is the output port lookup module. The output port lookup module is responsible for deciding which

port a packet goes out of. After that decision is made, the packet is then handed to the output queues module. The lookup module implements a very basic lookup scheme, sending all packets from 10G ports to the CPU and vice versa, based on the source port indicated in the packet's header.

3.2 System Architecture

In this section we describe the system architecture of the network processor and the prototype implementation of the network processor is discussed in Chapter 5. The network processor consists of a processing grid and the movement of packets between the processors are taken care by a control system.

3.2.1 Resource Management

The network processor implemented in the current prototype simplifies the usual design of a commercial network processor in which the resources are managed in software. In this architecture we take care of the resource management using hardware modules. This eases the software development. Using special purpose hardware it is possible move packets between the processor cores, switch processing context between different applications, feedback to ensure multiple application processing on a single packet and allocate processing resources to applications based on the workload.

3.2.2 Software Development

Software development becomes a notch easier, since the resource management is performed in the hardware. Since special purpose hardware takes care of moving packets between processor cores and switch processing context between different applications, the software development is simplified. By allocating fixed memory locations to processing instructions, data structures and the current packet, the program can easily access the memory with static references.

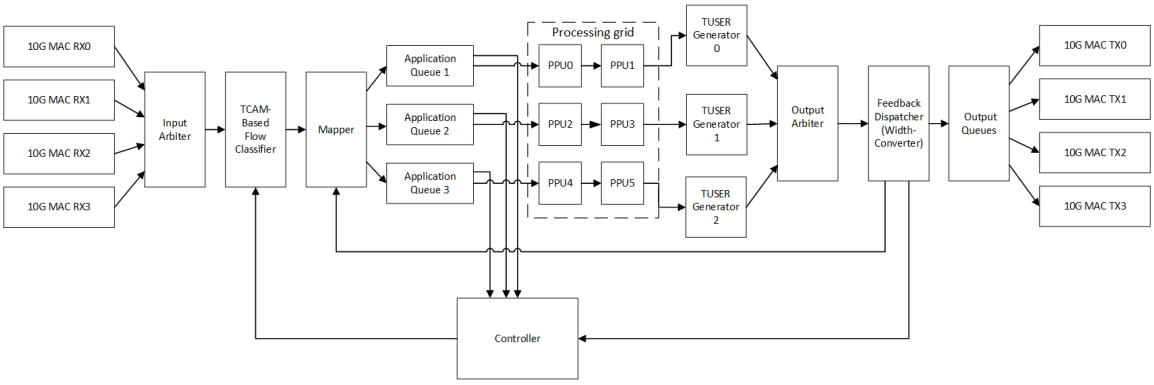

Figure 3.10. System Architecture of Multi-Core Network Processor on NetFPGA 10G

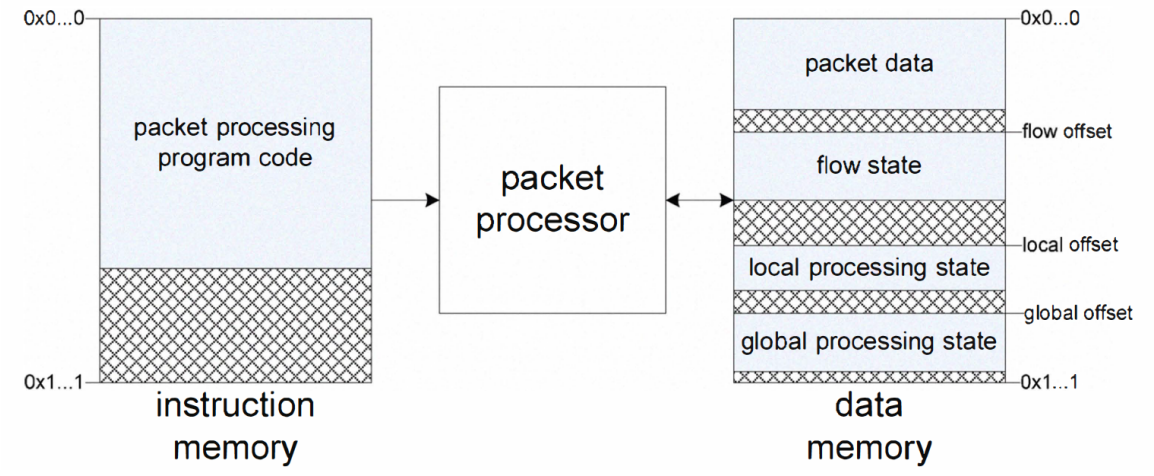

Figure 3.11. Processing context in network processor design in our prototype from [18]

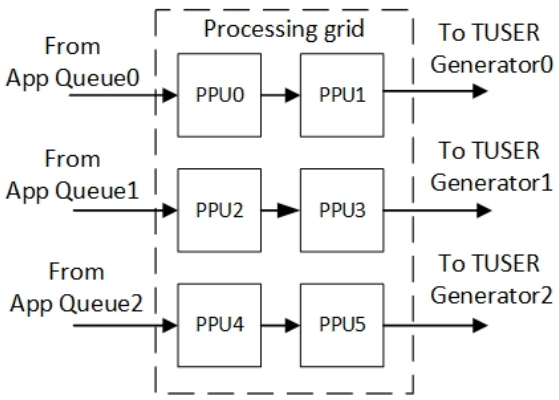

Figure 3.12. Processing Grid

3.2.3 Processing Grid

The Processing grid in our system consists of packet processor units connected from left to right as shown in the Figure 3.12. This architecture enables us to scale the system to multiple processors. A packet entering this grid is classified and the processing steps are determined before hand. The packet also carries information that determines its path among the processors and the operations to be performed on it by different processors. The processing context switching is taken care by the controller using the control packets.

3.3 Summary

This chapter introduces several concepts that are essential for understanding the prototype system implemented in the current project. It also describes the system architecture of the prototype.

CHAPTER 4

WORKLOAD MANAGEMENT

Over the years, network processors have transformed from a single core processor into a multi-core processor to meet the performance requirements of ever growing internet users and enormous amount of internet applications. To maximise the throughput of a multi-core network processor in a dynamic workload environment, it is essential to distribute the workload evenly across the cores. This helps in avoiding overloading of one particular core while the others are less-loaded. In order to achieve this, it is important to have a good load balancing algorithm implemented on a multi/many-core network processor which can not only perform in a normal or a static workload but also be able to prove its performance in a dynamic workload. The algorithm should hence be able to keep a track of the workload and be able make changes in the resource allocation in a multi-core network processor at run time. Though there are many load balancing algorithms already designed for such scenarios in network processors, it is always important to customise an algorithm and implement it on a custom platform and hence maximise its throughput. In this chapter, the algorithm used to balance the workload among multiple cores in the prototype system is described. The workload balancing algorithm designed for the prototype is based on the task mapping and duplication algorithms described in [18].

The high-level system architecture of the prototype was explained in Chapter 3. In the prototype, each of the processor core is pre-loaded with a certain set of tasks to be able to switch the processing context at run-time. The main idea of the algorithm is to maintain a resource allocation table and update it dynamically with the changing

traffic. The controller module would then consult the resource allocation table at pre-defined intervals to re-allocate resources if it detects any over-loading of a core. After the re-allocation of processing resources, the controller module updates a task identifier table. The resource allocation table consists of the information regarding the task running on a processor core and the ID associated with it, the expected running time of a task on a core, utilisation of a core and workload on a core. The resource allocation table and the task identifier table are shown in Figure 4.1. During a cold boot of the prototype system, the table is preloaded from the values obtained from offline profiling. We have a wide range of applications ranging from a simple forwarding to complex packet processing applications like IPSec. Each application has a run time different from each other which depends on the number of operations to be performed on a packet. For example, a simple forwarding application does not involve deep packet inspection where as certain security applications like IPSec perform deep packet content inspection where service time is completely dependent on the size of packet. For a given set of applications, we use offline profiling tools to measure the time taken by an application to complete the processing on a packet. This data is stored in the resource allocation table. In a dynamic workload, it is difficult to predict the behaviour of the incoming traffic and hence it is difficult to assign an application without knowing the utilisation of a core. Hence initially we set the utilisation of all the cores constant at 1 which indicates full utilisation of the core. As the traffic flows in, the utilisation of each core varies. The utilisation is then updated in the resource allocation table.

4.1 Task Mapping and Duplication

We allocate the tasks to a processor core depending on the workload. When the workload of a particular task is higher than the rest of the tasks, more number of processor cores are allocated for that task. Hence the number of processor cores

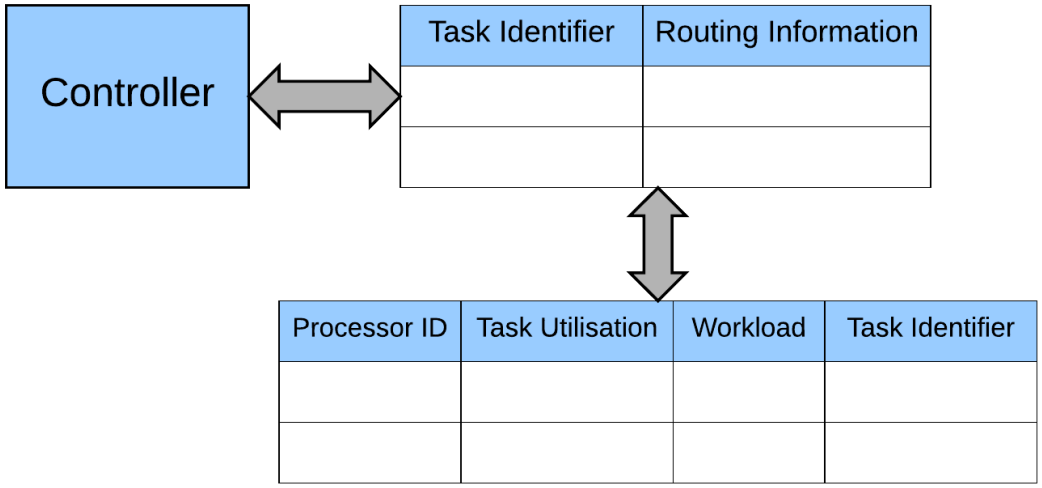

Figure 4.1. Resource Allocation Table

allocated is directly proportional to the workload. Increasing the number of cores decreases the utilisation of the cores and results in a lower workload for each of the cores running that particular task. This means that there are more resources to process packets waiting for a certain task to be performed on them and hence this reduces packet wait times. If a processor core gets over-loaded, it is indicated by an increase in the workload. Hence the task running on a particular processor core is duplicated i.e., a core which is has a lower workload is chosen and the context is switched to that particular task. Each task has a taskID and a fixed offset in the instruction memory of a processing core. By task duplication, the workload reduces since the task utilisation reduces.

CHAPTER 5

PROTOTYPE IMPLEMENTATION

In this chapter, we present the detailed system architecture of each of the modules present in our prototype implemented on NetFPGA 10G.A block diagram depiction of our prototype implementation is as shown in Figure 3.10. The subsequent sections describe the various user-defined IP cores present in our prototype.

5.1 Input Arbiter

The function of input arbiter is to multiplex a number of input streams into one output stream. In our system input arbiter merges the data from the four 10G MAC interfaces in a round robin fashion. The 10G MAC interface is a 64-bit wide. A stream width converter converts the 64-bit data into a 256-bit data out. In our system the data width of the input arbiter at both the input and output of the module is 256-bit. The input arbiter has an input buffer at each of the four inputs. A buffer is implemented as a small fall through FIFO. The output of this module is fed into a flow classifier module.

5.2 Flow Classifier

Flow classifier performs packet classification to be able to determine the services to be performed on the packet. This module extracts the 5-tuple of each Ethernet frame i.e., Source IP, Destination IP, Source port, Destination Port and Protocol information associated with each frame. The extracted tuple is then used as an index to perform a TCAM look up on a flow table which has the information regarding

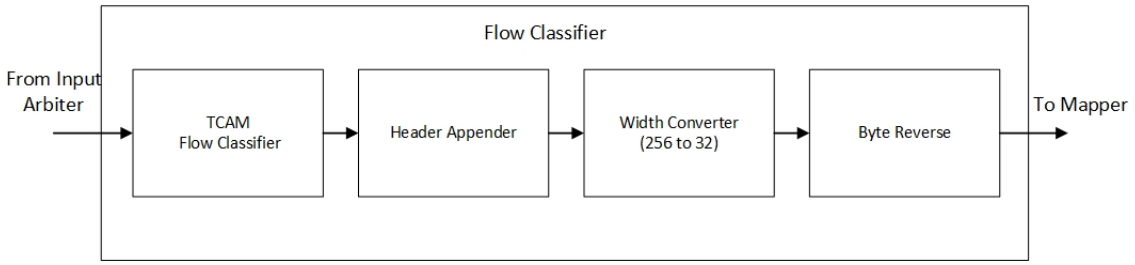

Figure 5.1. Block Diagram of a Flow Classifier

the application to be performed on the packet. Each packet is then appended with a 32-bit control information header that contains information for each processor that is traversed in the grid. The block diagram of the flow classifier is shown in Figure 5.1. This module also consists of a width converter and a byte reverse module.

5.2.1 Control Information Header

The structure of the 32-bit Control Information Header is as shown in the Figure 5.2. Given below is the description of the bits present in the header.

- Bits 0-10 represent the size of the packet. These are filled in by the flow classifier.
- Bits 11-18 represent the destination interface of the packet. These are filled in by the flow classifier.
- Bits 19-26 represent the task information i.e., each application is represented by 2 bits. The maximum number of applications supported in this prototype are 4. These are filled by the flow classifier.
- Bits 27-30 represent the routing information. This information indicates if a particular processor should process that packet or forward it to the next processor in the gird. These bits are filled in by the controller module.

Row [1:0]	Route [2:0]	Seq [7:5]	Seq [4:0]	DST [7:5]	DST [4:0]	PktSize [10:8]	PktSize [7:0]

Figure 5.2. Packet Structure Of Control Information Header

- Bits 31-32 represent the row of the processing grid. This information indicates the row of the processing grid to be taken by the packet. These bits are filled in by the controller module.

5.2.2 Width Converter

This converter down sizes the AXIS stream width from 256-bit datapath to a 32-bit data path. This is done to enable smooth functioning of the datapath interface to processor cores since the AXIS bus interfaces of the processor cores are only 32-bit wide. The output of this module is connected to a byte reverse module.

5.2.3 Byte Reverse

A byte reverse module is used to reverse the order of the bytes since the processor core functions in big endian format and the network packets in the hardware flow in to the modules in little endian format.

5.3 Mapper

The output of the flow classifier is fed to the Mapper. A packet with the control information header appended is received by the Mapper. Mapper maps each of the packet to its respective queue according to the processing to be performed on it. There are as many queues as applications. The queue in which the packet needs to be queued, is determined using a queue mapping table which consists of a queue ID and an application ID. The application ID is used as the index to obtain the queue ID.

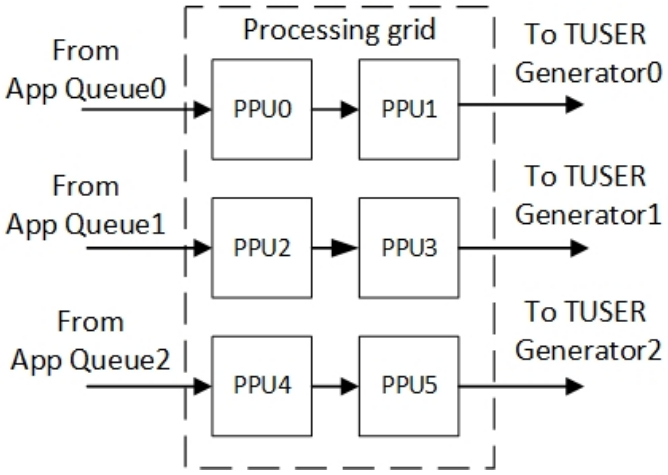

Figure 5.3. High Level Architecture of a Processing Grid

5.4 Application Queues

Application Queues are a bunch of FIFO's with each FIFO carrying the packets waiting to be processed for a certain application. In our prototype, we support three different applications and hence 3 queues. Mapper decides the queue to be taken by a packet depending on the application. Packets from each of the application queue then flow into the respective row of the processing grid indicated in the control information header.

5.5 Processing Grid

The high level architecture of the processing grid implemented in our prototype is as shown in the Figure 5.3. The packet processing units are connected in rows from left to right through AXIS streaming interfaces. This architecture minimises the bus interfaces required to connect all the cores but still be flexible.

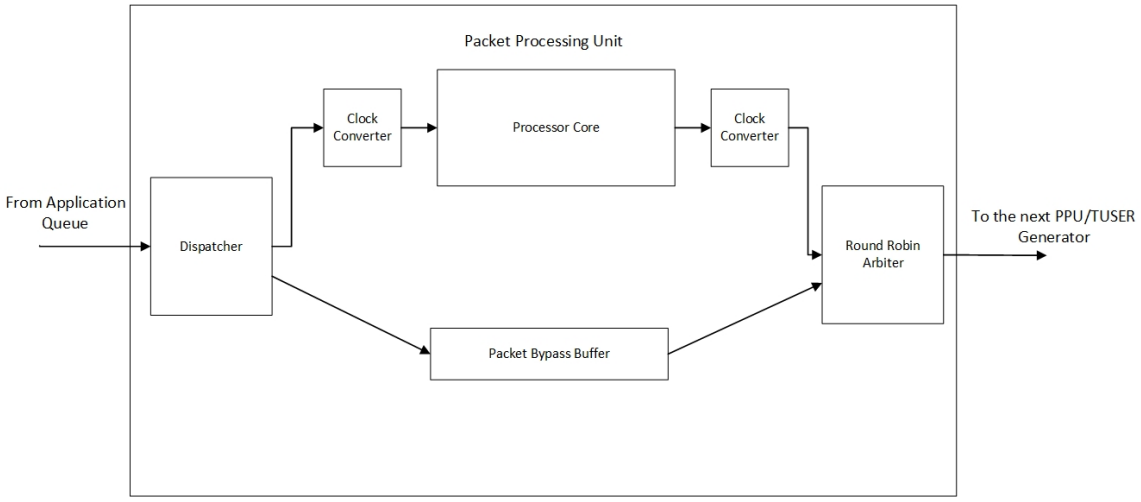

Figure 5.4. Packet Processing Unit

5.5.1 Packet Processing Unit

The packet processing unit is shown in the Figure 5.4. This unit consists of a processor core, dispatcher, clock converter and a round robin arbiter. Each of these modules are explained in the subsequent sections.

5.5.1.1 Dispatcher

A dispatcher module decides if the packet is to be processed by that particular processor core or a different core. If the packet needs to be processed by that particular core it forwards the packet to a clock converter module, else it forwards the packet to a packet bypass buffer.

5.5.1.2 Clock Converter

Clock converter is a combination of a dual clock FIFO and a synchronizer. The IP cores and the processor cores function at two different frequencies. This created the need for a clock converter to synchronize the data between the two different clock frequencies i.e., processor cores functioning at 100 Mhz and IP cores functioning at 160 Mhz. This dual clock FIFO is designed as a way for two circuits operating in different clock frequencies to communicate with each other. There is a read side and

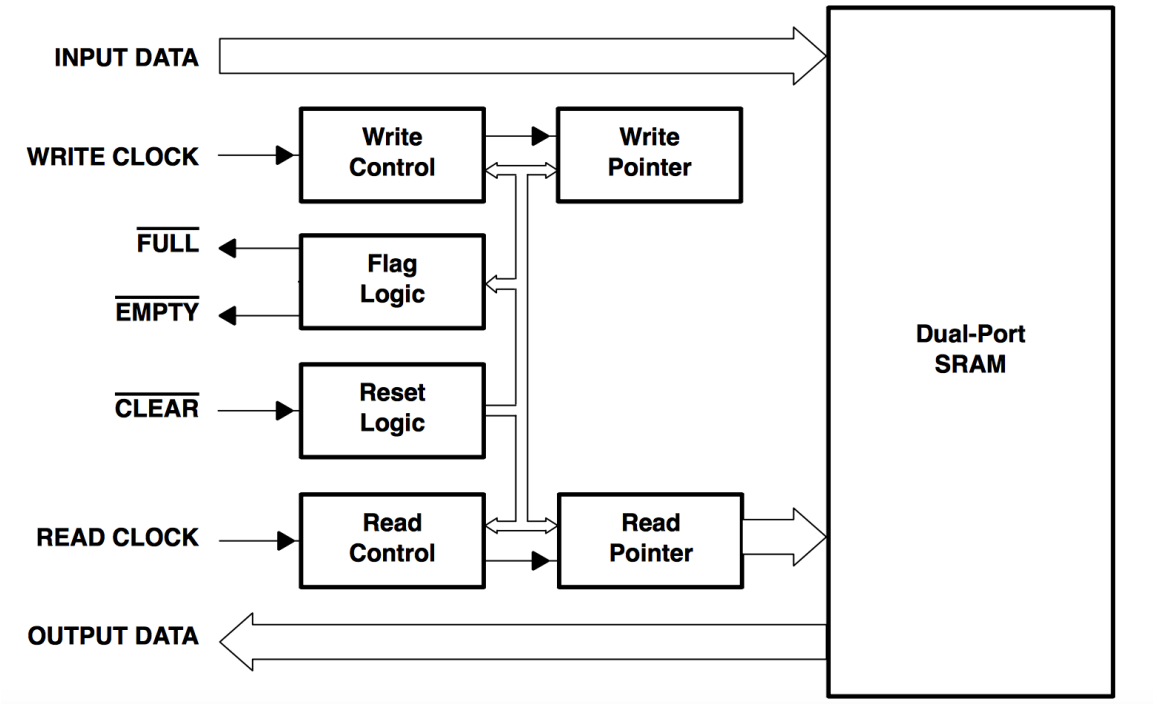

Figure 5.5. Dual Clock FIFO with Static Memory

write side where data is stored into the internal memory of the FIFO using the write side clock and then read from the internal memory using the read side clock. This module is meant to be flexible, allowing to easily change the data width as well as the size of the internal memory. A dual port SRAM is used for storage of the data as it comes in.

The block diagram of the dual clock FIFO with a static memory is as shown in the Figure 5.5. Read addresses are generated by the read pointer and write addresses by the write pointer. The write-control and read-control blocks control operation during write and read access. The full and empty status signals are generated by separate flag logic. The output of this module is fed to a processor core.

5.5.1.3 Processor Core

The processor cores in the processing grid are implemented using Microblaze which is a 32-bit soft-core processor provided by Xilinx. The streaming interfaces on Mi-

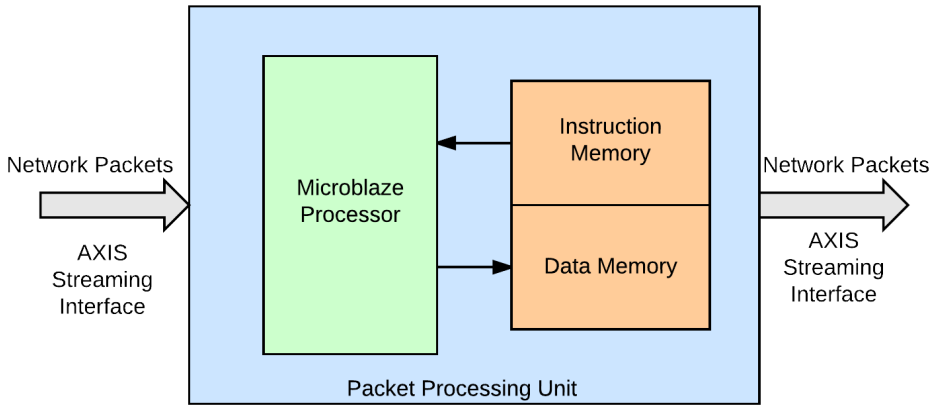

Figure 5.6. Processor Core

croblaze are 32-bit wide and each microblaze has upto 16 streaming interfaces. The 16 AXI streaming interfaces enable us to scale the system to build popular multi-core topologies, such as the tree, mesh, or torus structure. The get instruction in the MicroBlaze ISA is used to transfer information from a port to a general purpose register. The put instruction is used to transfer data in the opposite direction. Since the AXI interconnect is a memory mapped interconnect, it makes the communication of microblaze with the other hardware modules easier. Microblaze can also access the on-chip memory via OPB (on-chip peripheral bus) in a single cycle which makes the operation of accessing the instruction and data memory faster and easier. In the instruction memory, the code for a certain application is placed at a well-defined and a fixed location. IPv4 forwarding(two modes) and IPSec are the three different applications programmed on each of the core. At any given time only one application runs on a processor core. The controller decides at run time which application needs to run on a processor core. A processor core is as shown in Fig 5.6. The processed packets with the utilisation statistics appended as a trailer are then fed into a clock converter module.

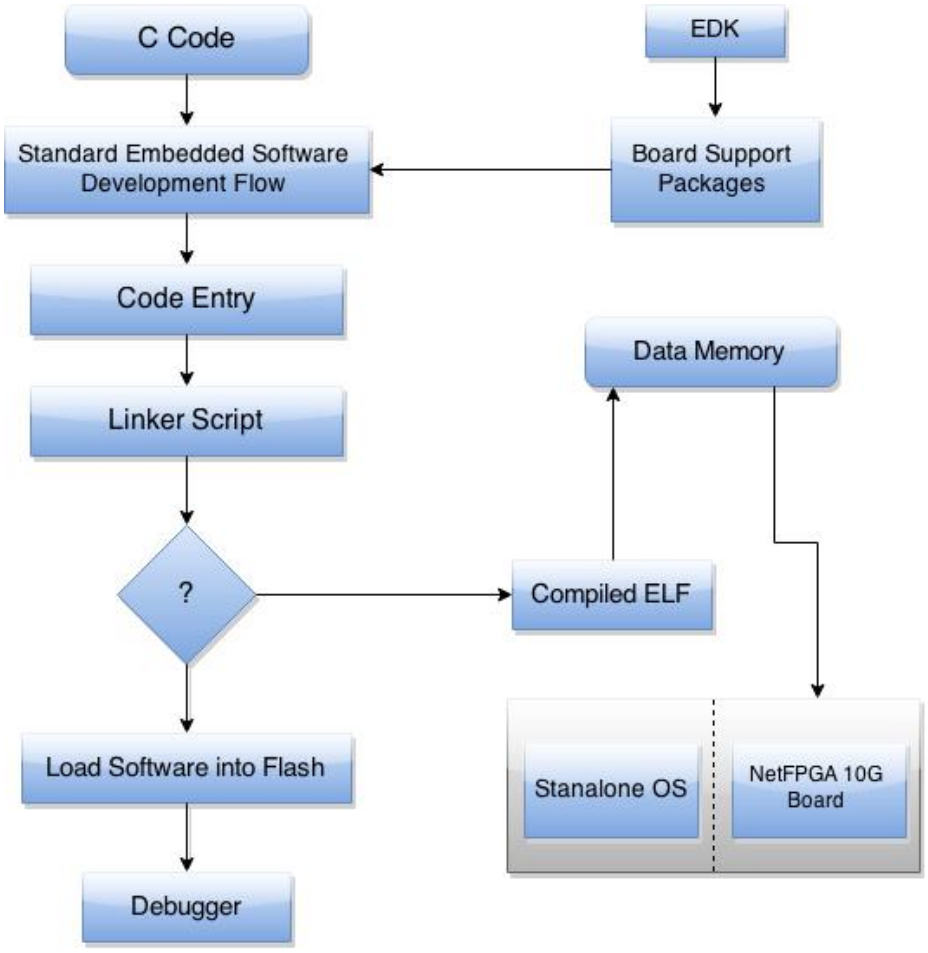

Figure 5.7. Software Design Flow

5.5.1.4 Software Development

Xilinx SDK was used to develop the applications to run on the processor cores. Writing software to control the MicroBlaze processor is done in C/C++ language. The EDK tools have built in C/C++ compilers to generate the necessary machine code for the MicroBlaze processor. The software design flow is as shown in the Fig 5.7.

As the packets come in through the streaming interfaces, the control header of the packet is inspected to check if the particular packet is to be processed on that processor. If the packet needs to be processed it is stored in the data memory for

further processing else it moves further in the grid. After the processing, the packet moves further in the grid.

5.5.1.5 Packet Bypass Buffer

A packet bypass buffer is used to store packets which are meant to be processed by a different processor core rather than the present core. The output of this module is fed into an arbiter module.

5.5.1.6 Round Robin Arbiter

An arbiter is used to multiplex the two inputs i.e., the processed packets from the processor core or the packets from the bypass buffer. The output of this module is then connected to the next packet processing unit or to a TUSER generator if it is the last packet processing unit in the grid.

5.6 TUSER Generator

TUSER generator appends the TUSER block to each processed packet. TUSER is a metadata field which is required for proper routing of the packets through the output queues. This module also performs an endian conversion i.e., from big to little endian format since the packets are received from the processor cores in the big endian format.

5.7 Controller

The controller module is responsible for mapping the incoming packets to the respective row of the processing grid using a task identification table. The respective row is indicated in the control information header and these bits are modified by the controller as and when the resources are re-allocated. The task identification table is updated at regular intervals by referring to the resource allocation table. Resource allocation table contains information regarding the application running on

a particular processor core and other information such as the task utilisation, expected service time, workload on a processor core. This ensures that the processing cores work independent of each other. The task identification and resource allocation tables are stored in a BRAM which are continually accessed and updated. This flow routing mechanism enables the flexibility of the processing gird. In our prototype, overloading of a core is detected by monitoring the size of the application queues. When the size of an application queue goes beyond a threshold, it indicates overloading. A processor core monitor the thresholds of the application queues to detect any overloading of an application, which means that there is need for more processing resources for that particular application. If an application overloading i.e., core overloading is detected it re-assigns the applications to be performed on the cores by referring to the resource allocation table.

5.8 Output Arbiter

The function of the output arbiter is to merge a number of input streams from the processing grid into one output stream. The packets after processing get queued into a small fallthrough FIFO present at the input of the output arbiter module. The output arbiter is modelled as a state machine with two states i.e., an idle state and a transmit state. The output arbiter waits for a complete packet to be read from an input buffer and once the complete packet is received it dispatches the packet to the feedback dispatcher.

5.9 Feedback Dispatcher

In our prototype, each time a packet passes through the processing grid it is processed only by one of the processor cores i.e., only one application. If a packet needs multiple application processing i.e., processing by more than one processor, feedback dispatcher decides if the packet needs to be sent back to the mapper module

to enable further processing of the packet. After the packet has been completely processed they are forwarded to the output queues. The datapath width is then converted from a 32-bit to 256-bit datapath using a width converter.

5.10 Output Queues

The function of this block is to dispatch packets from one input stream to a number of output streams whereby the destination port sub-band channel information stored in TUSER determines to which output the packets are to be routed. To maximise throughput, all input interfaces need to have the same data width as the output stream.

5.11 Experimental Approach

In this section, we describe the experimental setup and various tests performed on the prototype implementation.

5.11.1 Development of Hardware/Software test bed

In order to develop the embedded system design on the NetFPGA 10G, we use a tool kit provided by Xilinx i.e., Embedded Development Kit (EDK) suite. In our project, we use EDK design suite 13.4 version [8]. Using this suite of tools we can incorporate a wide range of Hard and soft-IP cores, such as microprocessors, interconnects, memories, and an assortment of peripherals. The main advantage of EDK suite is that in a single environment we can perform design, simulation, synthesis and compilation. In EDK embedded system design, we have two separate steps, hardware and the software design which interact each other. Firstly, we develop and design the hardware part where a custom circuitry i.e., IP cores are developed using a hardware description language like verilog or VHDL. Xilinx ISE design tools are used to develop IP cores required for our project. In order to test the IP cores,

Xilinx ISE supports three types of simulations i.e., Behavioural, structural and timing accurate. All the generated and tested IP cores are incorporated into a single design using XPS. XPS can be run in both gui and batch mode. We target our design for NetFPGA 10G. Once we have completed the embedded design, it is translated into an implementation suitable to the board. Here we have two phases in the implementation process: (a) Synthesis phase and (b) Compilation phase. Under the synthesis phase, a Xilinx synthesis tool translates the hardware description language into a gate level description. EDK provides Xilinx Synthesis Technology (XST) which performs the following, which happen in the synthesis phase: (1) System-on-Chip elaboration: It translates the HDL into a computer readable format. (2) Soft-IP core synthesis: Converts soft-IP core into a netlist which is a logical circuit description. (3) Physical Mapping: Maps the low-level netlist into a physical description circuit. (4) Netlist placement and routing: Physical Net lists are placed into the FPGA and channel is established between the different components for communication. (5) Bit-stream generation: Physical design is converted into a bit-level description i.e., bit-stream files which can be downloaded into the board for execution. In the compilation phase, the software program design is compiled from the C source and converted into the binary format as used by the microprocessor. After the software has been compiled into a binary executable file, the hardware and software are combined into a single bit-stream. This bit-stream is then used to initialize the SRAM with the hardware design, and the chip memories with the software design. Thus, when the system-on-chip is boot-strapped, any microprocessor present in the system executes the software associated with it.

5.11.2 Simulation tests

Packet stimuli and verification scripts are built upon scapy, by which stimuli may be constructed from scratch, using scapy packet primitives, or packets read in

from a pcap trace captured elsewhere.The workflow is illustrated in the following diagram in Figure 5.8. An end-user supplied script, run.py, writes out two sets of AXI Stream files: the stimuli, and the expected results. The project's MHS file is automatically translated into the subsystem simulation by means of the script "nf10-sim-autosubst.py". The simulation is run, and the resulting log files are automatically reconciled with the expected output by "nf10-sim-reconcile-axi-logs.py". Instead of specifying times for packet and register operations, the new infrastructure uses a barrier statement for synchronization. The barrier blocks until all expected packets arrive, or it times out, causing the test to fail. This ensures that register operations occur at the correct time relative to the packet operations.

Packets are constructed manually from scapy primitives. For example to make an ip packet the primitive with the following format make-IP-pkt (srcMAC, dstMAC, EtherType, srcIP, dstIP, TTL) is used. For the sake of performance, rather than simulate the complete design, including the 10G MACs, PCI Express end-point, and the respective testbench peers required to drive them the tool nf10-sim-autosubst.py automatically replaces the pcores that represent these interfaces to the external world with instances of nf10-axis-sim-stim and nf10-axis-sim-record, respectively one for each of the AXI4 Stream master and slave ports of those pcores. The script nf10-sim-reconcile-axi-logs.py is a generic tool which, for every AXI trace -expected.axi, attempts to load the corresponding AXI log (-log.axi) using axitools.axis-dump(). It then does a bit-wise comparison between actual and expected packets, and reports expected and actual packet counts, along with a simple pass/fail result. Simulation tests provided by the NetFPGA community are run on our prototype system to check the functionality and the results are observed.

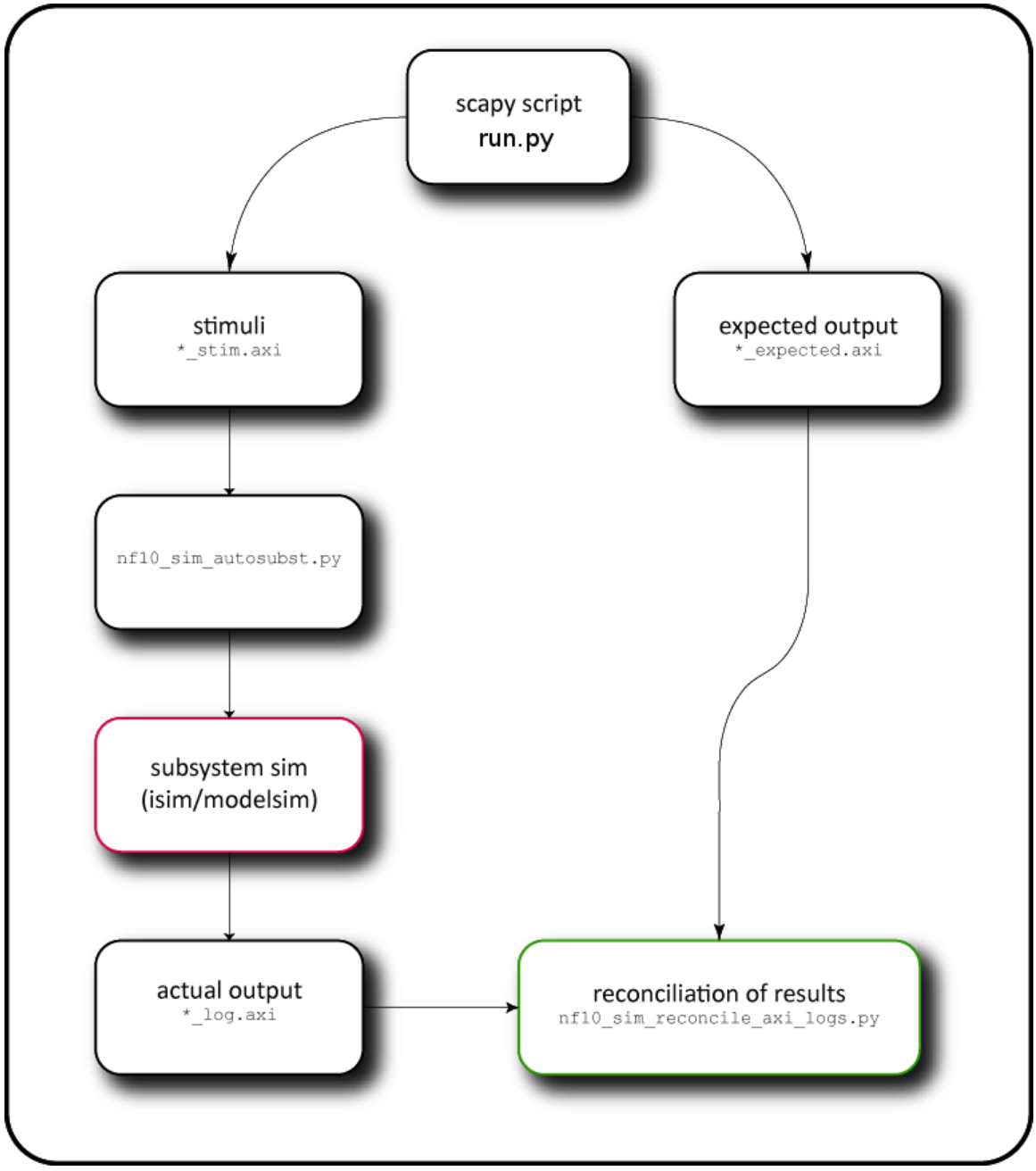

Figure 5.8. AXI4-Stream Simulation Workflow From [12]

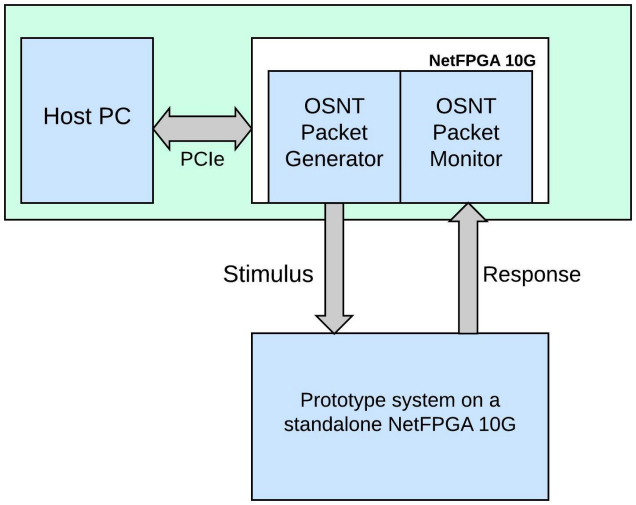

Figure 5.9. Test Topology

5.11.3 Hardware Tests

The test topology that will be used to verify the performance of our monitoring system in hardware is shown in Figure 5.9.

The setup is as described in the hardware test section [2] and the tests provided by NetFPGA community are run on the prototype system to check the functionality and performance of the single core network processor which is then extended to a multi-core system.Initially, we plan to test a single-core network processor prototype using Microblaze as soft-core. To generate packets and to capture packets forwarded from the prototype system, we plan to use open source network tester(OSNT) running on a NetFPGA 10G. OSNT [13] is an open-source hardware traffic generator and capture system built on NetFPGA 10G. The OSNT traffic generator generates packets according to pre-loaded PCAP traces. It allows for customizing the size, the number of iterations, and the throughput rate for the test packet. The OSNT traffic generator and monitor code is downloaded to one NetFPGA 10G board connected to a Host PC via PCIe interface. The GUI on a host PC aids in monitoring the traffic which

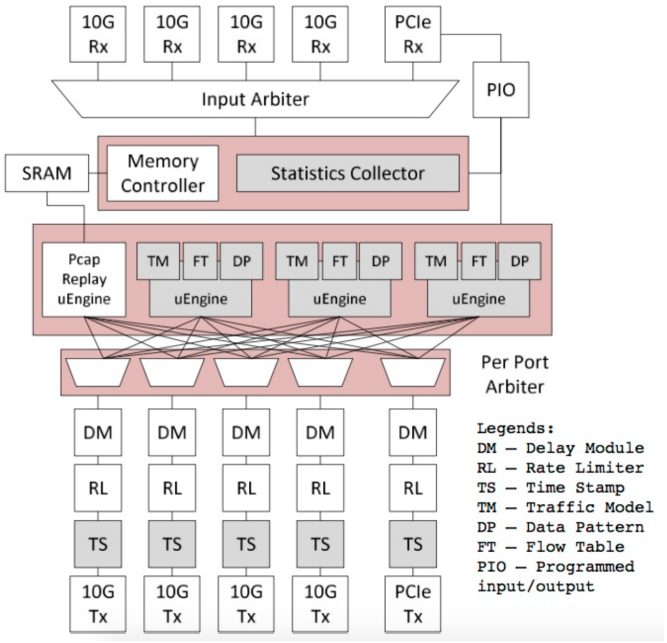

Figure 5.10. OSNT Traffic Generator From [13]

is received at the ports. The multi-core network processor system, is downloaded to another standalone NetFPGA 10G board.

5.11.3.1 OSNT Traffic Generator

OSNT follows the same modular hierarchy as any reference design on NetFPGA 10G. The various modules used in OSNT traffic generator as shown in Figure 5.10 are explained in this section. The system actually consists of four functional units:

- The arbiter selects packets and forwards them at their departure time.
- A pcap replay engine is used to replay the traffic captured in the form of PCAP files. These PCAP files are stored in the on-board SRAMs, by the host.The maximum trace dimension allowed strictly depends on the SRAM resources available on board. The replay engine reads the packets from these PCAP files and based on their destination port sends them out to the next module in the pipeline through the respective egress AXIS channel.

- A delay module is used to produce inter-packet delay. This block provides up to five instances of inter-packet-delays (e.g., four incase of NetFPGA-10G). This module adds necessary delay between packets either (1) calculated by the traffic model or (2) set by the host software as a fixed value.

- Rate limiter module limits the rate at which the packets can be sent through the egress interface(s).

- Finally, the packet is passed to the (10GbE) MAC which transmits it onto the wire.

5.11.3.2 OSNT Traffic Monitor

The architecture of a OSNT traffic monitor is as shown in Figure 5.11. The OSNT traffic monitor provides four functions:

- packet capture at full line-rate

- packet filtering permitting selection of traffic-of-interest

- high precision, accurate, packet timestamping

- statistics gathering

A module positioned immediately after the Physical interfaces and before the receive queues timestamps incoming packets as they are received by hardware. It also provides a functionality to monitor traffic of interest by a flow classification approach for example a 5-tuple filter implemented in the Core Monitoring module. As for the software side, it provides a python-based GUI that allows the user to interact with the HW components (e.g., enable cut/hash, set filtering rules, check statistics). A C-based application that comes with it records the received traffic in both PCAP or PCAPNG format. This allows offline use of common libpcap-based tools (e.g., TCPDump, Wireshark.)

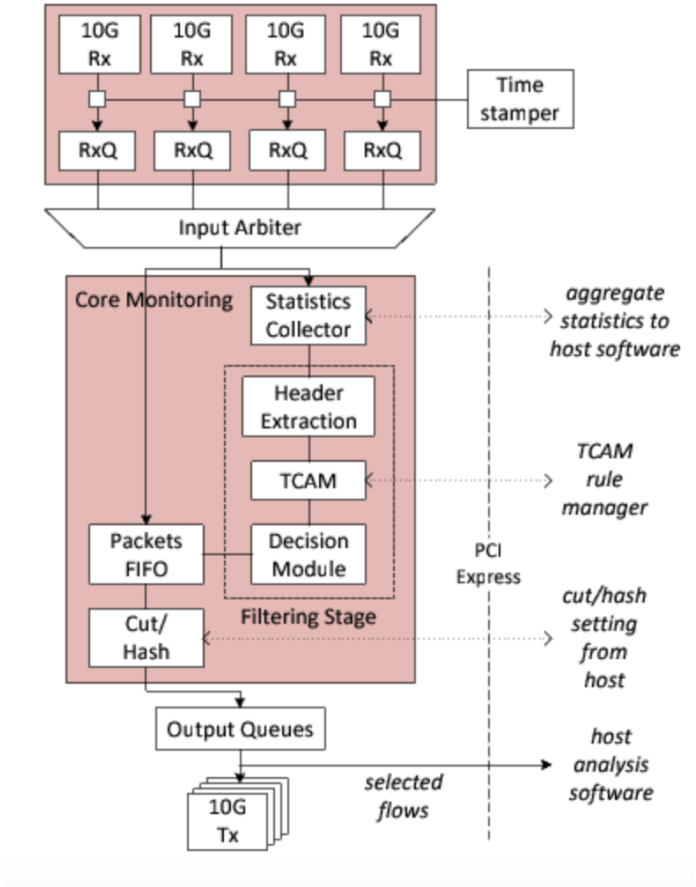

Figure 5.11. OSNT Traffic Monitor [13]

5.12 Evaluation Metrics

The prototype system is tested in simulation using a ModelSim-Xilinx simulator [11], and in hardware using the HW tests provided by NetFPGA community[2]. After the prototype is tested for it functionality, various tests to evaluate certain metrics are performed. The evaluation metrics are as listed below:

- Data Path Throughput: The throughput of the prototype system is measured by executing a simple IP forwarding on the processor cores. The forwarding is expected to performed at a full line rate.

- Processing Capacity: To evaluate the processing capacity, we compile processing-intensive applications like IPsec onto the processor core with a proposed load balancing algorithm for different packet sizes.

- Resource Utilization: We scale the system to include multiple cores and check resource utilisation as the prototype system is scaled to include multiple processing units.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter we discuss the results of the various tests performed on the multi-core network processor architecture.

6.1 Simulation Results

6.1.1 Control Header Insertion

When a packet is forwarded by the flow classifier, it inserts a 32-bit control information header at the start of each packet. The Figure 6.1 shows the header appended by the flow classifier at the start of a packet. It can be observed that the various fields like packet size, destination interface, applications to be performed, routing information are updated.

6.1.2 Width Conversion

A 256-bit datapath is converted to a 32-bit using the width converter. The Figure 6.2 shows the simulation of this operation on a network packet of 256-bit being divided into 32-bit chunks.

6.1.3 Clock Converter

A clock converter is used to synchronize between the different clock frequencies between the processor cores and rest of the hardware modules and to prevent any data loss due to the difference in clock frequencies. The Figure 6.3 shows the simulation of this operation on a network packet. It can be observed that the packets are transferred to the output after a 2-cycle latency but without any data loss.

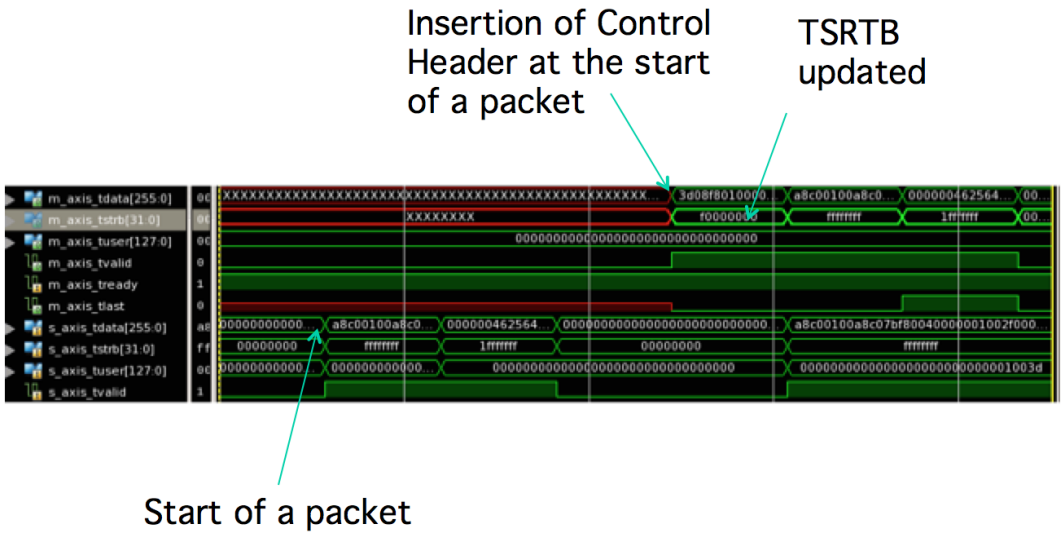

Figure 6.1. Simulation of Flow Classifier showing the Appended Header

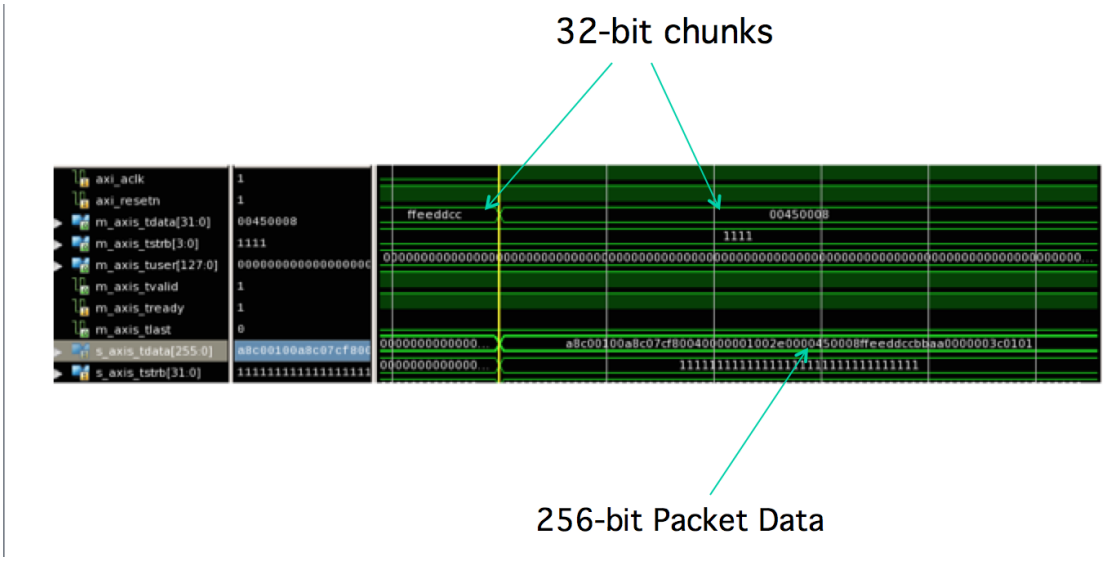

Figure 6.2. Simulation of Width Converter

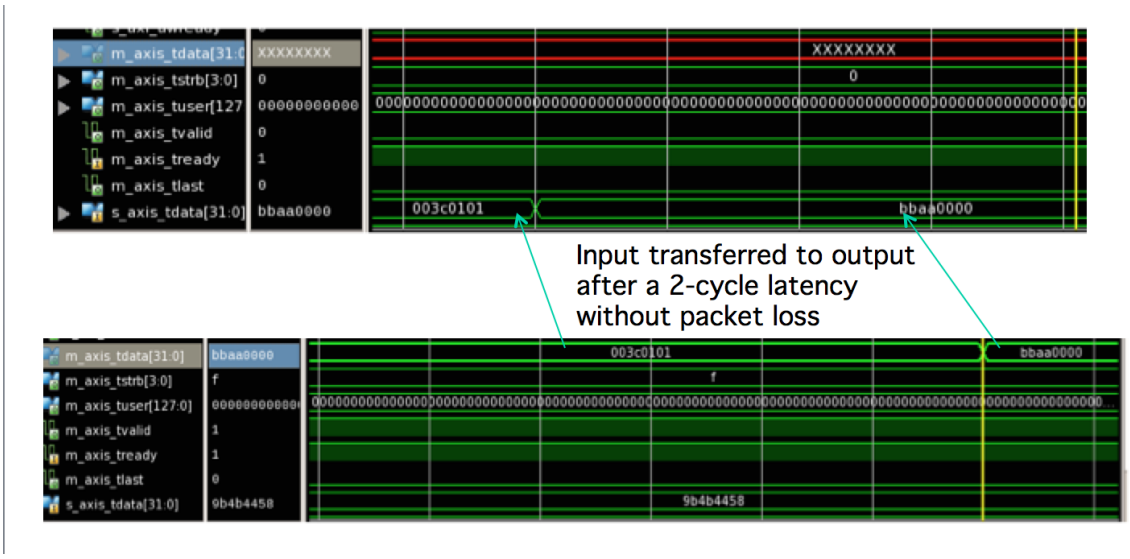

Figure 6.3. Simulation of Clock Converter

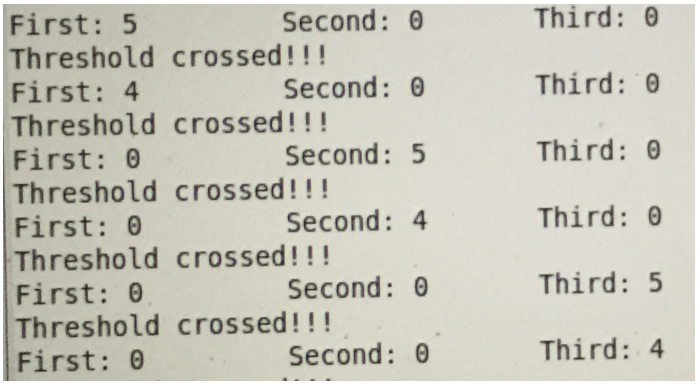

Figure 6.4. Simulation of Controller

6.1.4 Controller

Controller module monitors the threshold of the application queues to detect any overloading. When it detects any overloading, it displays a warning along with the current lengths of the each of the queues and assigns a new processor core. The Figure 6.4 shows the simulation of this operation on a network packet. In this simuation, the threshold of the queues are set at 4. When any of the queue is overloaded, a warning is issued along with the sizes of all the three queues. This helps us determine which of the application needs more processing resources.

```
while(1){
       for(i = 1; i <= 16; i++)
       {
       getfslx(got,0,FSL_DEFAULT);
       packet[i]=got;
       }
      int ip = (packet[5]>>16) & 0XFFFF;
      int header = (packet[5]>>8) & 0XFF;
      quotient = packet[7];
      quotient_lo = quotient & 0XFFFF;
      quotient_hi = (quotient>>16) & 0XFFFF;
      ttl_hi = (quotient_lo>>8) & 0XFF;
      ttl_lo = (quotient_lo) & 0xFF;

      if(ip==0x800 && header==0x45 && ttl_hi>1)
      {
       ttl_hi = ttl_hi-1;
       quotient_lo = (ttl_hi<<8) | ttl_lo;
       quotient = (quotient_hi<<16) | quotient_lo;
       packet[7] = quotient;
```

Figure 6.5. IPv4 Application

6.2 Throughput Performance

Using a standard IPV4 packet forwarding application in the processor core, the throughput performance of the multi-core system with 3 cores and 6 cores was tested. Network packets of 60-byte and 1500-byte packet sizes were generated from the pcap packet generator, and sent through the 10Gbps MAC ports of the NetFPGA 10G board via OSNT Generator. The forwarded packets were received back at the OSNT Monitor and the prototype systems transmit-receive statistics were measured. The packet forwarding application works by comparing destination IP address in each packet header with IP address values stored in processor memory to select an output port. A reduction in packet size increases the per packet processing operation, and thus reduces the overall throughput performance. A part of IPv4 application compiled in the processor core is as shown in the Figure 6.5. The resulting throughput performance with 60-byte and 1500-byte input packet sizes is illustrated in Figures 6.6 and 6.7. The throughput performance improves for a 1500-byte packet size since the per packet processing reduces.

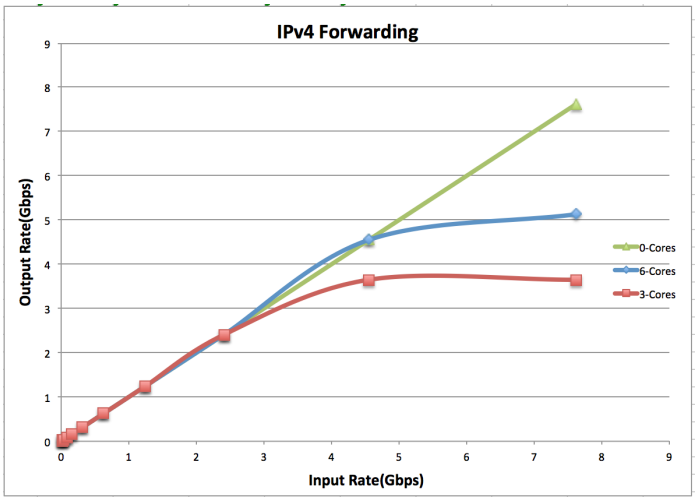

Figure 6.6. Throughput Performance of a Multi-Core Network Processor performing IPV4 with a 60-byte Packet Length

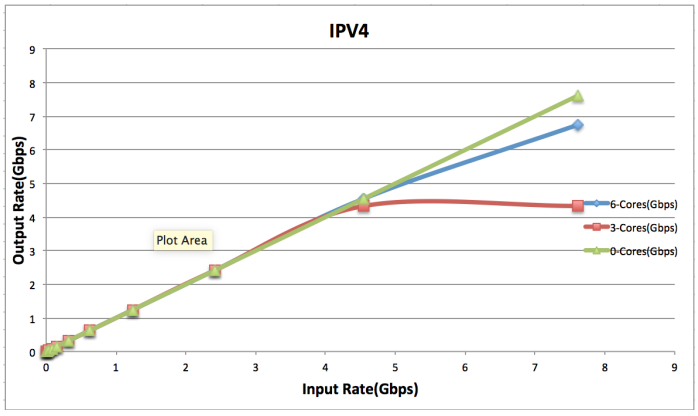

Figure 6.7. Throughput Performance of a Multi-Core Network Processor performing IPV4 with a 1500-byte Packet Length

```
if(ip==0x800 && header==0x45 && ttl_hi>1)
{
 ttl_hi = ttl_hi-1;
 quotient_lo = (ttl_hi<<8) | ttl_lo;
 quotient = (quotient_hi<<16) | quotient_lo;
 packet[7] = quotient;



 unsigned char plain_text[32]={0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                               0x00, 0x00};
  unsigned char encrypted_text[32];
  unsigned char key_ring[3][8]={
          {0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x010, 0x01},
          {0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01},
          {0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01}
  };
  tdes_init(key_ring);
  tdes_encrypt(18, plain_text, encrypted_text);
}
```

Figure 6.8. IPSec Application

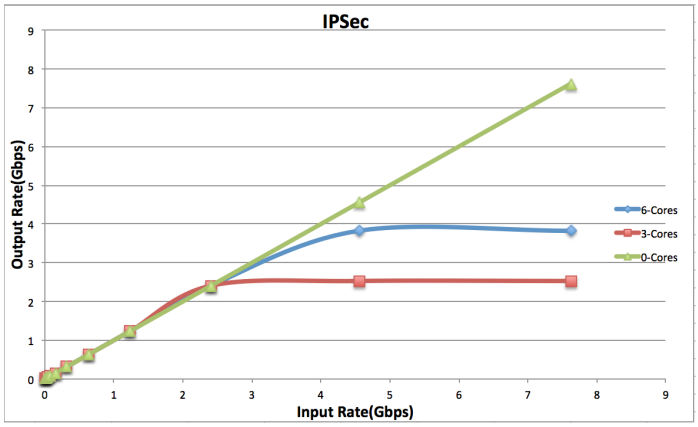

Figure 6.9. Throughput Performance of a Multi-Core Network Processor performing IPSec with a 60-byte Packet Length

6.3 Processing Capacity

A part of IPSec application as shown in the figure 6.8 is compiled on the processor core to measure the processing capacity of the multi-core network processor. The resulting throughput performance of the system with 60-byte and 1500-byte input packet sizes is illustrated in Figures 6.9 and 6.10. It can be observed from figures 6.8 and 6.9 that the throughput performance reduces for a 1500-byte compared to a 60-byte packet size since the per packet processing increases as the packet size increases.

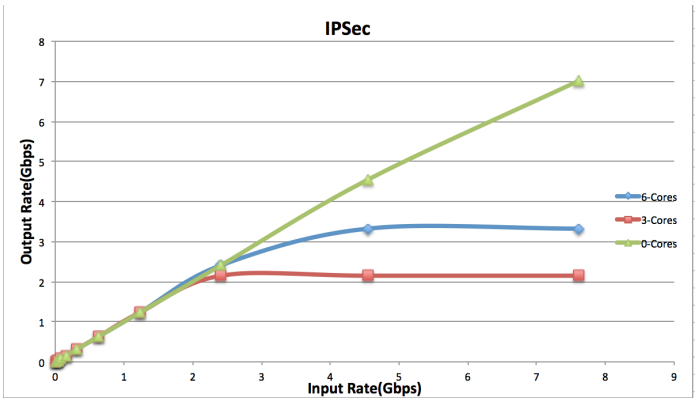

Figure 6.10. Throughput Performance of a Multi-Core Network Processor performing IPSec with a 1500-byte Packet Length

6.4 Throughput Performance During Overload

Throughput of our system during overload of a particular core is measured by using a packet trace which emulates overloading of one of the application queues. In this experiment, a 4-core network processor prototype was used. A packet trace which overloads the first application queue is used for testing the system. Initially only 3 processor cores are assigned for packet processing. Once an overloading is detected a 4th processor is assigned by the controller to improve the throughput. The improvement in throughput of the experimental prototype over a 4-core prototype without the workload balance is illustrated in Figure 6.11. When a queue is overloaded i.e., when the size of the queue exceeds the set threshold, the controller displays the lengths of the overloaded queues. In figure 6.12, variation in the length of the overloaded queue over time is illustrated. The threshold of each of the queues was set at 100. And the time at which the length of the first queue exceeded the threshold was marked as the start time. The length of the queue varying over time until it falls below the threshold is measured. It can be observed from the figure 6.12 that the length of the queue reduces as soon as the overloading is detected which indicates the assignment of more resources to the application in run-time.

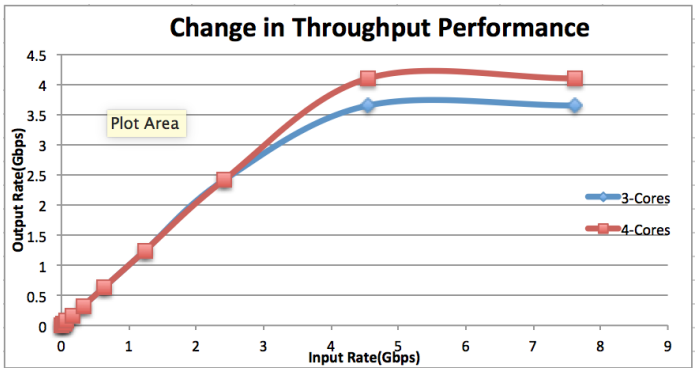

Figure 6.11. Throughput Performance of a Multi-Core Network Processor during Overloading

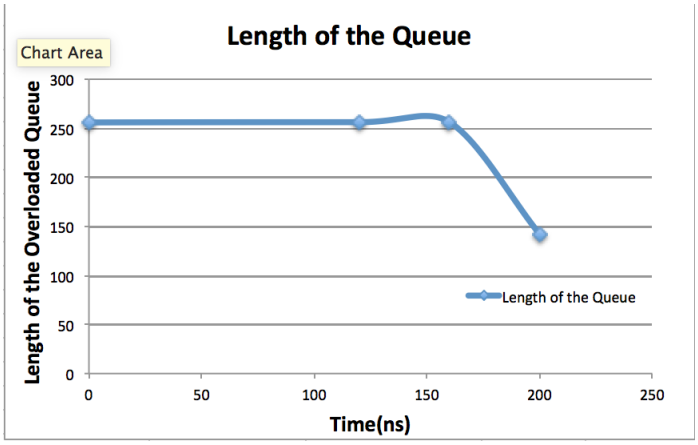

Figure 6.12. Queue length variation during overloading

6.5 Resource Utilisation

This section explains the different resource utilization details of our proposed network processor system. The synthesis results were provided by Xilinx Synthesis Tool(XST). The lookup table (LUT), flip flop (FF), and memory resources required for the single network processor core, multi-core network porcessor core, and other interface circuitry for the processor (e.g. buffers, input arbiter, queuing control) are shown in Table 6.1.

Table 6.1. Resource Utilisation

Resources	**Single-Core**	**Multi-Core (3-Core)**	**Multi-Core (6-Core)**	**Available**
LUT's	38762	46363	53000	149760
Flip Flops	59460	68137	76732	149760
Memory	2,273	2665	2942	39,360

CHAPTER 7

CONCLUSION

This thesis has described a Multi-Core network processor with a dynamic workload balance. As the workload changes dynamically, the processing resources are re-assigned to balance the workload across the processing cores. Our effective workload balancing algorithm monitors the utilisation of the processing cores at run time. When the queue carrying the packets for a certain application overflows i.e., exceeds beyond the set threshold, the utilisation of the cores is checked to identify the under-utilised core and the processing resources are re-allocated accordingly. The multi-core network processor with 6 processor cores is implemented on a NetFPGA 10G platform. A possible extension to this work would be to scale the multi-core network processor to more than 6 cores and implement the dynamic workload balancing algorithm designed for the prototype in this thesis.

BIBLIOGRAPHY

[1] Microblaze processor reference guide. www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf, 2014.

[2] Netfpga 10g hardware tests. https://github.com/NetFPGA/NetFPGA-public/wiki/HW-Tests, 2014.

[3] Netfpga 10g reference nic. https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-Reference-NIC, 2014.

[4] Broadcom. Broadcom xlp832 multicore processor. http://www.broadcom.com/products/Processors/Enterprise/XLP800-Series.

[5] Cavium. Octeon multi-core processor family. http://www.cavium.com/OCTEON_MIPS64.html.

[6] Cisco. The cisco quantumflow processor. , 2008.

[7] Crowley, Patrick, Fluczynski, Marc E., Baer, Jean-Loup, and Bershad, Brian N. Characterizing processor architectures for programmable network interfaces. In *Proceedings of the 14th International Conference on Supercomputing* (New York, NY, USA, 2000), ICS '00, ACM, pp. 54–65.

[8] EDK. Xilinx edk 13.4 tutorial. www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/est_rm.pdf.

[9] Intel. Intel second generation network processor. http://www.intel.com/design/network/products/npfamily/, 2005.

[10] Iqbal, M.F., Holt, J., Ryoo, Jee Ho, John, L.K., and De Veciance, G. Flow migration on multicore network processors: Load balancing while minimizing packet reordering. In *Parallel Processing (ICPP), 2013 42nd International Conference on* (Oct 2013), pp. 150–159.

[11] ISE. Xilinx ise 13.4 tutorial. www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/ise_tutorial_ug695.pdf, 2015.

[12] NetFPGA10G. Netfpga 10g. http://netfpga.org/2014/#/.

[13] OSNT. Osnt. http://osnt.org, 2015.

[14] Tilera. Ez chip. http://www.tilera.com/files/drim__NP-5_Product_Brief_short_Jan2015_7599.pdf.

[15] Weng, Ning, and Wolf, Tilman. Pipelining vs. multiprocessors - choosing the right network processor system topology. In *in Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with The 31st Annual International Symposium on Computer Architecture (ISCA 2004* (2004).

[16] Wolf, Tilman, and Franklin, Mark A. Locality-aware predictive scheduling of network processors. In *In Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS* (2001), pp. 152–159.

[17] Wu, Qiang, Chasaki, D., and Wolf, T. Implementation of a simplified network processor. In *High Performance Switching and Routing (HPSR), 2010 International Conference on* (June 2010), pp. 7–13.

[18] Wu, Qiang, and Wolf, Tilman. On runtime management in multi-core packet processing systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2008), ANCS '08, ACM, pp. 69–78.