October 2021

Internet Infrastructures for Large Scale Emulation with Efficient HW/SW Co-design

Aiden K. Gula
University of Massachusetts Amherst

Internet Infrastructures for Large Scale Emulation with Efficient HW/SW Co-design

Aiden K. Gula

INTERNET INFRASTRUCTURES FOR LARGE SCALE EMULATION WITH EFFICIENT HW/SW CO-DESIGN

A Thesis Presented

by

AIDEN GULA

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 ENGINNERING

September 2021

Electrical and Computer Engineering

INTERNET INFRASTRUCTURES FOR LARGE SCALE EMULATION WITH EFFICIENT HW/SW CO-DESIGN

A Thesis Presented

by

AIDEN GULA

Approved as to style and content by:

Russell Tessier, Chair

Fatima Anwar, Member

Paul Siqueira, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

DEDICATION

I want to dedicate this thesis work to my late father, who always stressed the importance of an education. Thank you for providing your guidance and wisdom throughout the years, which helped shape me into the person I am today.

ACKNOWLEDGMENTS

I want to express my gratitude to my advisor Professor Russell Tessier, for his support and guidance throughout my thesis work. I want to thank Professor Fatima Anwar and Professor Paul Siqueira for serving on my thesis committee and providing their expertise. Finally, I would like to thank former graduate students Xuzhi Zhang and Narendra Prabhu for their collaboration on a portion of this thesis work.

ABSTRACT

INTERNET INFRASTRUCTURES FOR LARGE SCALE EMULATION WITH EFFICIENT HW/SW CO-DESIGN

SEPTEMBER 2021

AIDEN GULA

B.Sc., UNIVERSITY OF MASSACHUSETTS DARTMOUTH

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

Directed by: Professor Russell Tessier

Connected systems are becoming more ingrained in our daily lives with the advent of cloud computing, the Internet of Things (IoT), and artificial intelligence. As technology progresses, we expect the number of networked systems to rise along with their complexity. As these systems become abstruse, it becomes paramount to understand their interactions and nuances. In particular, Mobile Ad hoc Networks (MANET) and swarm communication systems exhibit added complexity due to a multitude of environmental and physical conditions. Testing these types of systems is challenging and incurs high engineering and deployment costs. In this work, we propose a scalable MANET emulation framework using virtualized internet infrastructures that generalizes an assortment of application spaces with diverse attributes. We then quantify the architecture using various evaluation techniques to determine both feasibility and scalability. Finally, we developed a hardware offload engine for virtualized network systems that builds upon recent work in the field.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . **v**

ABSTRACT . . . . . **vi**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

CHAPTER

1. INTRODUCTION **1**

1.1 Docker Simulation Infrastructure 2
1.2 Network Offload Engine 2
1.3 Thesis Outline 3

2. EMULATION ARCHITECTURE **4**

2.1 Mobile Ad hoc Networks 4
2.2 Docker 5
2.3 Node Infrastructure 6
2.4 Containerized SDN Connectivity 9
2.5 Graphical User Interface 12

2.5.1 Main Control Panel 13
2.5.2 Control Processor Panel 14
2.5.3 Graphing Utility 15

2.6 Hardware Emulation Support 15

2.6.1 Node Hardware Emulation 16
2.6.2 System Hardware Emulation 16

3. QUANTIFYING SYSTEM PERFORMANCE 18

3.1 Simulation Performance 18

3.1.1 Single Stream Bandwidth 19
3.1.2 Traffic Model Bandwidth 22

3.2 Network Coverage 26
3.3 Network Resiliency 28

4. SILICON OPENFLOW SWITCH 30

4.1 OpenFlow Fundamentals 30
4.2 Switch Architecture 31
4.3 Header Parsing 33
4.4 Flow Table 34

4.4.1 FPGA Content Addressable Memory 36

4.5 Action Processor 37

5. PCIE SR-IOV VIRTUAL NETWORK ACCELERATOR 39

5.1 SR-IOV Virtualization Specification 40
5.2 High-Level Acceleration Solution 42
5.3 Linux Device Driver 44

5.3.1 Network Drivers and PCIe Support 44
5.3.2 Packet Representation in the Linux Kernel 45
5.3.3 Packet Lifecycle in the Linux Kernel 47
5.3.4 NAPI compliant drivers 49

5.4 PCIe Hardware Implementation 50

5.4.1 PCIe Specification Overview 50
5.4.2 Read Requests for High Bandwidth Applications 52
5.4.3 PCIe Endpoint Architecture 54

5.4.3.1 Receive Engine 56

5.4.4 System Memory and Programming Models 58

6. EVALUATION OF PCIE SR-IOV VIRTUAL NETWORK ACCELERATOR 61

6.1 Performance for Containerized Applications 61

6.1.1 Bandwidth Capabilities of Direct Memory Access 61
6.1.2 Performance Quantification of Network Communication 63

6.2 MANET Emulation HW/SW Performance 66
6.3 Edge Application Specific Acceleration 69
6.4 Limitations for Memory Bound Computation 73

7. CONCLUSION 76

7.1 Future Work 77

7.1.1 Scatter Gather I/O 77
7.1.2 FPGA Offload with OVS 77
7.1.3 Distribute Computation Across Hosts 78
7.1.4 Interconnected CPU and FPGA 80

BIBLIOGRAPHY 81

LIST OF TABLES

Table **Page**

3.1 Traffic Model Grid (Mbps) 25

3.2 Network Coverage 26

3.3 Network Resiliency 29

6.1 Repeated TCP/IP Connections (Kpps) 68

LIST OF FIGURES

Figure **Page**

2.1 Example MANET 5

2.2 Node Architecture 6

2.3 Master Coordinator Architecture 8

2.4 Routing Endpoint Architecture 9

2.5 Emulation Architecture 10

2.6 Visibility File 11

2.7 Topology in respect to time 12

2.8 Emulation Manager Main Control Panel 13

2.9 Emulation Manager Control Processor Panel 14

2.10 Emulation Manager Graph Utility 15

2.11 Hardware Based Environment 17

3.1 Inter-Node Bandwidth 19

3.2 Intra-Node Bandwidth 20

3.3 Full Message Bandwidth 21

3.4 Traffic Model Bandwidth 23

3.5 Connectivity Graph for Traffic Model Simulation 24

4.1 Open Flow Architecture 30

4.2 OpenFlow Pipeline 32

4.3 Ethernet Header Parser in the Interpreter 34
4.4 Flow Table 35
4.5 CAM FPGA Emulation 37
4.6 Action Processor 38
5.1 CPU Utilization 39
5.2 SR-IOV Architecture for Containerized Applications 41
5.3 PCIe SR-IOV Accelerator 43
5.4 Socket Buffer Kernel Structure 46
5.5 Linux Kernel Packet Lifecycle HW/SW 48
5.6 PCI Express Interconnect Network 51
5.7 Theoretical PCIe 3.0 Bandwidth 52
5.8 PCIe Read Request Comparison 53
5.9 PCI SRIOV Architecture 55
5.10 Receive DMA Engine 57
5.11 PCIe SR-IOV Address Space 59
6.1 DMA Throughput for OpenFlow 62
6.2 Performance Comparison between OVS and FPGA 64
6.3 Small Payload Bandwidth Comparison between Mobile Nodes 65
6.4 MANET Emulation FPGA vs OVS 66
6.5 Parallel FIR Filter Transposed 70
6.6 Edge Acceleration Speedup over UDP Protocol 72
6.7 PCIe Spec Comparison 74
7.1 Distributed Emulation with SR-IOV enabled FPGAs 79

CHAPTER 1

INTRODUCTION

In recent years, the number of connected computing devices has increased drastically. This growth has been predominantly driven by IoT and cellular technologies, including 4G, LTE, and 5G standards. The number of connected devices in 2030 is projected to be approximately 24 billion, with revenue exceeding 1.5 trillion dollars across the globe [32]. As the number of connected devices increases, so does the need for networked distributed systems' evaluation infrastructures. These infrastructures are particularly needed to support rapid prototyping and extensions in both algorithm development and routing protocols [8].

Network emulation and virtualization helps alleviate the need for expensive hardware testing and evaluation. Instead of utilizing multiple hardware devices, one can virtualize a compute host on a single Operating System (OS) and run directed tests to evaluate algorithms, routing protocols, and performance metrics [27]. Network virtualization flexibility helps assess potential future systems and determine system specifications, feature requirements, and costs before hardware acquisition and deployment. Obtaining this information early in the development process allows engineers to make informed design decisions for hardware and software sub-systems. This, in turn, decreases the need for expensive changes after the hardware has been acquired. Network emulation allows the infrastructure to behave the same way as it would on a physical device. Network emulation provides guarantees that the system software will run the same when deployed on distributed networked systems in the

field. This thesis discusses our network virtualization approach and emulation and describes how it integrates with specific applications and hardware-based acceleration.

1.1 Docker Simulation Infrastructure

The field of network virtualization is vast and complex with a myriad of approaches [38][40] each with its own pros and cons for specific application spaces. Network virtualization systems are even more challenging when considering MANETs (mobile area networks), in which the network topology changes as a function of time due to real-time position updates. The network connectivity relies on the nodes' relative position in 3D space, which is enforced by the communication medium range. In this work, we propose a network virtualization infrastructure that supports MANET topologies at a proper abstraction while closely emulating the functionality of distributed embedded testbeds. The virtualized system utilizes Docker [1] containers to simulate mobile communication nodes. Mobile node can communicate with each other using an external virtual switch called Open vSwitch (OVS) [3]. The external OVS instance allows dynamic traffic forwarding based on a set of rules to help support MANET topologies. The work in this thesis builds up this virtualization infrastructure to support a wide range of applications and system performance quantification tools while increasing usability to allow rapid prototyping and evaluation.

1.2 Network Offload Engine

Software-based network virtualization offers customization and flexibility at a low cost, but it suffers from performance degradation due to sequential packet processing. In software-based virtualized systems, user space programs, kernel space programs, hypervisors, switches, and daemons run on a Central Processing Unit (CPU). This implementation leads to large context switching overheads [26] for virtualized infrastructures since the number of processes often exceeds the number of CPU cores.

Researchers [37][39] have recently shown great interest in developing acceleration infrastructures that support end-to-end virtual networking stacks. For this thesis project, we propose an Field Programmable Gate Array (FPGA) based offload solution that is enabled by state-of-the-art virtualization technology standards such as Peripheral Component Interconnect Express (PCIe) Single Root Input/Output Virtualization (SR-IOV) [18]. The infrastructure supports the OpenFlow protocol for packet processing, matching, and forwarding.

1.3 Thesis Outline

This thesis work is divided into four major sections. The first section discusses our MANET emulation environment. We describe the architecture, capabilities, interfaces, and extensions that support various applications and use cases. The second section discusses the emulation performance metrics. We explain how system performance is quantified and discuss the implications of these results. The third section examines Software Defined Networking (SDN) solutions at the hardware level. This section describes hardware architectures at the Register Transfer Level (RTL) and a full software-defined networking switch on an FPGA is presented. The final section discusses how a network hardware data plane was integrated into our emulation infrastructure for hardware offloading. This chapter includes discussions of relevant PCIe specifications, kernel-level drivers, and hardware infrastructures for efficient HW/SW co-design.

CHAPTER 2

EMULATION ARCHITECTURE

2.1 Mobile Ad hoc Networks

Mobile Ad hoc Networks (MANET) are a type of decentralized network, where the topology of the network is determined autonomously based on each node's relative position. The network connectivity can experience drastic changes based on the environment, making modeling and simulation challenging. The nodes of a MANET are constrained by limited transmission ranges forcing each node within a network to act as a router to achieve multi-hop communication [23]. Network topologies like this are most prevalent in the IoT space that operates on mesh networking protocols such as ZWave or Zigbee [6].

An application space that is most prevalent in military operations is demonstrated in Figure 2.1. Each node within the network is an Unmanned Aerial Vehicle (UAV). The black arrows represent direct connections between nodes, while the red arrow demonstrates a virtual network substrate achieved by multi-hop routing. The ground station acts as a special purpose node that initiates traffic used for inter-node communication. It is important to note that the ground station can be generalized to represent any node within the system. Throughout the rest of the document, the ground station will be referred to as the master coordinator. The following sections will discuss how this type of scenario is emulated using modern technologies and frameworks.

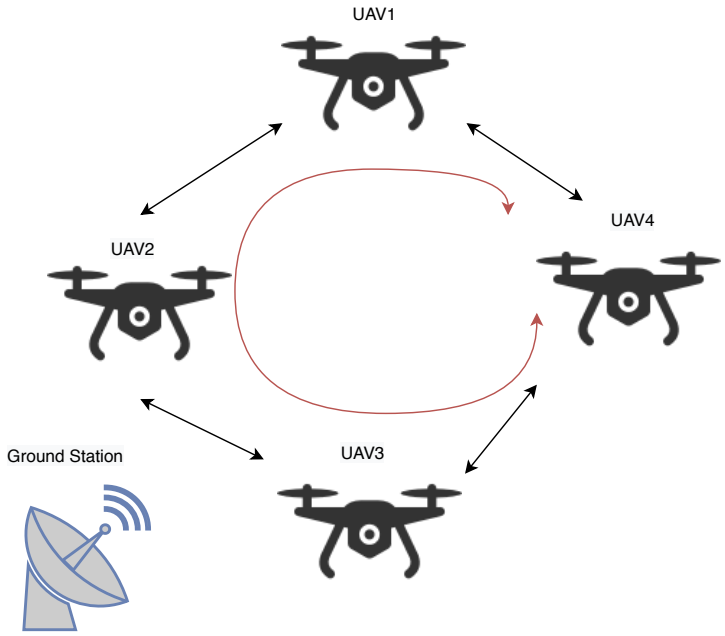


Figure 2.1: Example MANET

2.2 Docker

Docker is a framework that can be used in the development, shipping, and execution of applications. A Docker platform [1] provides isolation and security, allowing many containers to be virtualized on a single host. Docker operates similar to a Virtual Machine (VM), but runs directly in the host machine's kernel and avoids the need for a complete hypervisor. In this project, we leverage Docker to instantiate multiple containers where each container represents one node within the network. The Docker infrastructure uses an *image* to determine the functionality embedded within a container. In our case, each node is an Ubuntu operating system that contains its own network stack, device drivers, kernel, and most importantly, user space programs.

Furthermore, each container can support nested containers through Docker's standard Docker in Docker (DinD) [11] image. The DinD image allows any Docker infrastructure to be realized within an already existing container. This implies that each node can host complex network infrastructures, where each piece is its own

nested virtual host. The proposed architecture allows each node to act as a virtual distributed system. Each component within a node emulates its own compute environment with an isolated network stack and user space programs. The significance of this architecture will be further discussed in Section 2.6.

2.3 Node Infrastructure

In section 2.1, we introduced a hypothetical scenario where the nodes of the MANET network represent a single UAV. For the purposes of this discussion, a mobile node can be thought of as one of the UAVs in that scenario. A mobile node must make local decisions, initiate remote connections between nodes, and maintain global routing information. Figure 2.2 shows an architectural diagram of the components that make up a mobile node to achieve this functionality.

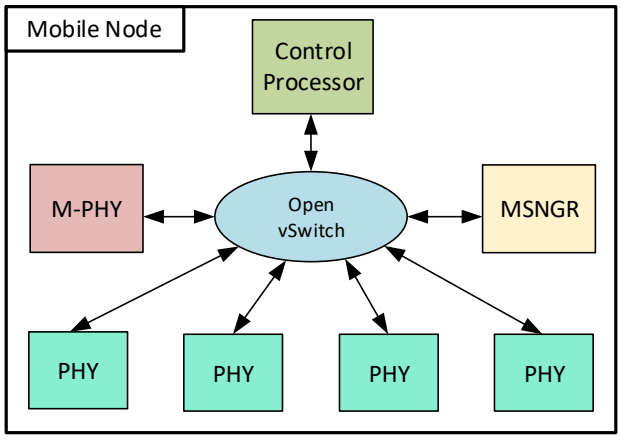


Figure 2.2: Node Architecture

A mobile node comprises three distinct components: the control processor, the messenger (MSNGR), and PHYs. The control processor is considered to be the brain of the node. This component maintains global information such as the state of all other components and all remote connections. The control processor manages all routing tables and distributes relevant information to all components within a

mobile node. This routing information is used to forward traffic from source to destination during intermediate hops. The messenger component is used to generate traffic models. In this context, the traffic model is a stream of data from source to destination of various sizes. The supported traffic types are image, command, and telemetry data, which all have a set payload size ranging from 100 to 65k bytes.

The final internal component is a PHY, which is derived from the physical layer of a communication stack. The component acts as a software abstraction that handles all connections and transmissions to remote mobile nodes. Each PHY component can initiate and manage one active connection at a time through a Transmission Control Protocol (TCP) socket. The TCP socket is necessary due to its connection-oriented nature; each PHY must be able to recognize that a connection has been made or broken to notify and update the control processor's internal routing tables. Therefore, the number of remote connections for each mobile node is limited by the number of PHYs. Figure 2.2 is capable of four remote mobile node connections due to the number of PHYs inside of the container. The M-PHY component is almost identical to the others but can only maintain remote connections with a master coordinator component, hence the *M* designator. The Open vSwitch in the container is utilized to route packets between components using Media Access Control (MAC) based forwarding. All internal components dynamically discover each other using a variation of the Address Resolution Protocol (ARP) protocol.

The master coordinator uses a control processor and a messenger but only utilizes M-PHYs. The M-PHY component in the master coordinator only makes remote connections with other M-PHY components. The control processor component has a unique identifier that restricts its connections to the correct component type. Internal identification allows other mobile nodes to identify a master coordinator node's messages. Figure 2.3 shows a high-level view of the master coordinator.

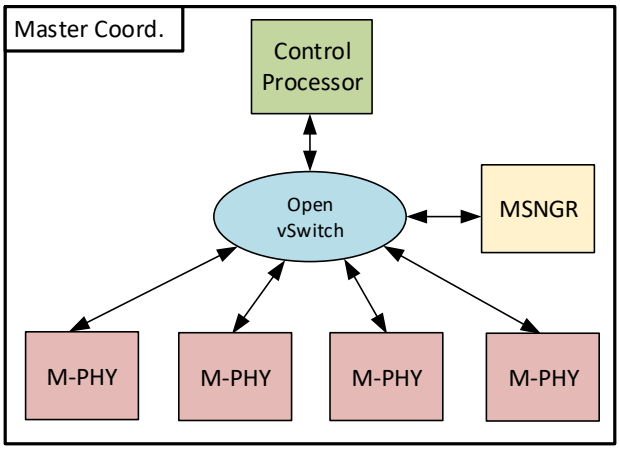


Figure 2.3: Master Coordinator Architecture

The final type of supported node infrastructure is a routing endpoint. This node type acts as an adapter that can expand connectivity between nodes. The routing endpoint's components are useful when connecting a master coordinator to other mobile nodes, expanding routing capabilities. The routing endpoint can be viewed as an antenna that extends communication when transmission distances are too far. For example, if a master coordinator cannot make a connection to a mobile node due to limited range, a routing endpoint can be used as an intermediate connections between source and destination. Section 3.2 will discuss how these components can be leveraged for network coverage. The routing endpoint does not require a MSNGR component because no messages are received or transmitted. Components in the node behave in the same manner as similar components in a master coordinator node. Figure 2.4 shows a high-level diagram of a routing endpoint with multiple M-PHY instances.

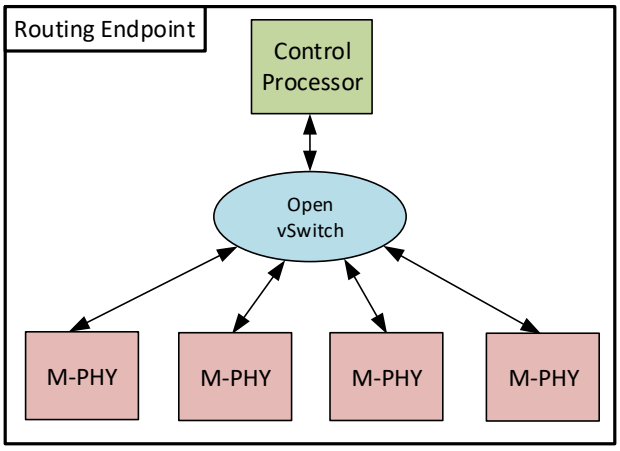


Figure 2.4: Routing Endpoint Architecture

2.4 Containerized SDN Connectivity

To emulate a MANET network, a network substrate that can dynamically change network topologies based on geo-spatial distances must exist. An external OVS (PHY Bridge) is utilized to connect PHYs within each node to implement network topologies. The OVS instance uses the OpenFlow protocol [28], which supports SDN. SDN decouples the control and data plane, allowing software to dictate how traffic is forwarded between ports. Chapter 4 provides protocol and implementation-specific details for SDN implementation. Figure 2.5 shows a high-level depiction of the infrastructure utilized to connect nodes represented as Docker containers.

The Simulation Controller (SC) is a component within the emulation infrastructure that helps manage connections through the external OVS. The switch allows a set of programmable rules to be defined to control how data is handled at each input port. In this architecture, the SC programs flow rules into the switch that manages which nodes can communicate with each other. The SC component can also communicate with all control processors through the control bridge, a switch that provides IP based forwarding. The SC has a global view of all mobile nodes, master coordinators, and routing endpoints and helps facilitate network topology changes

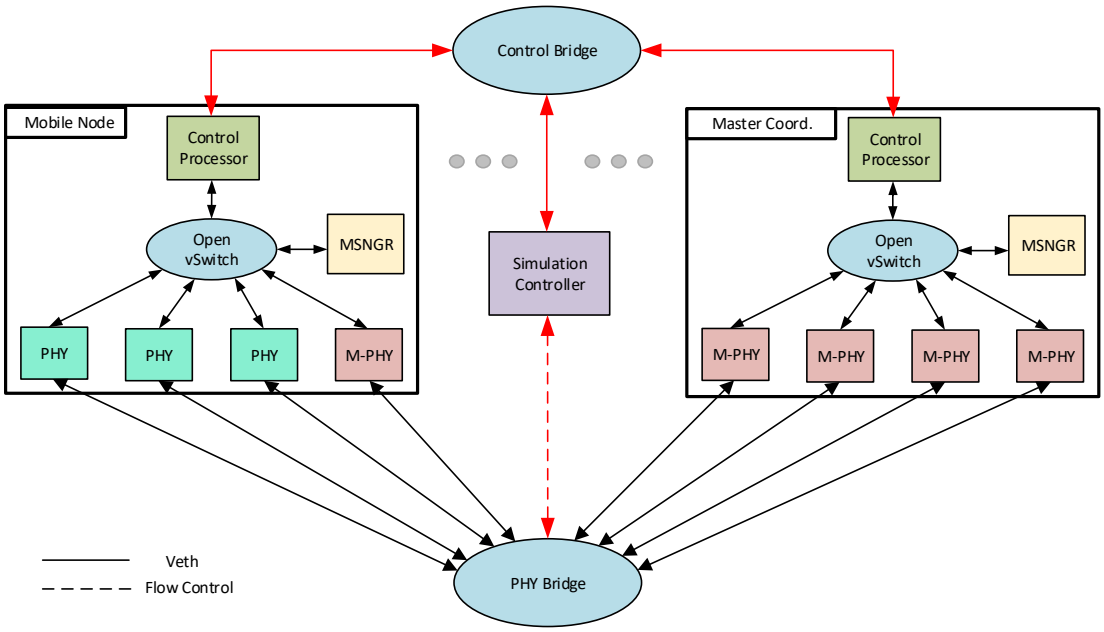


Figure 2.5: Emulation Architecture

based on visibility data. The visibility data in a JavaScript Object Notation (JSON) file defines which mobile nodes can be connected and the X, Y, Z distances between the nodes. Figure 2.6 shows the structure of visibility data that defines the network topology during a simulation. The communication medium range can be adjusted to help model Radio Frequency (RF) protocols.

```
{
  "PosZ": -19000454.467008,
  "PosX": -36862067.71915131,
  "nodeName": "GPS BIIF-4  (PRN 27)",
  "nodeID": "39166",
  "PosY": -39842795.50091413,
  "visibilityGraph": [
    {
      "PosZ": -13630816.71157064,
      "LoS": true,
      "PosX": -11472273.568443207,
      "nodeName": "GPS BIIF-3  (PRN 24)",
      "nodeID": "38833",
      "PosY": 54751839.352035485
    },
    {
      "PosZ": -37361965.893144995,
      "LoS": true,
      "PosX": 23082173.55403828,
      "nodeName": "GPS BIIF-2  (PRN 01)",
      "nodeID": "37753",
      "PosY": 36647151.35005301
    },
    {
      "nodeZ": -32985736.750322387,
      "LoS": true,
      "PosX": 27115357.742739305,
      "nodeName": "GPS BIIF-1  (PRN 25)",
      "nodeID": "36585",
      "PosY": -38078628.518545
    }
  ],
  "Time": "2019-07-25T15:15:00.000"
}
```


Figure 2.6: Visibility File

The visibility file includes a node's position, identifier, and name. The visibility graph object key provides a list of all visible nodes indicated by the Boolean line of sight (LoS) key. Furthermore, the distance between the current node and the visible node is available. The connection is only made if the distance is less than a certain threshold dictated by the communication medium being modeled. The same structure is repeated for every valid time step over the simulation time frame. A simulation time step is the unit of time where the current network topology is guaranteed to be correct. After a simulation time step is over, new position data is evaluated. The network is modified if the distance between nodes is no longer supported based on the range of the communication medium. The opposite is also true; new connections can be made if the range between nodes is shortened to a supported distance. The SC manages this visibility data and reevaluates the network topology at each new time step. If links need to be broken or added, each respective node's control processor is

notified so all internal routing tables can be updated appropriately. Figure 2.7 shows an example simulation, where the node connectivity changes based on visibility time steps. The red lines indicate links that have been broken, while the green lines indicate newly established connections.

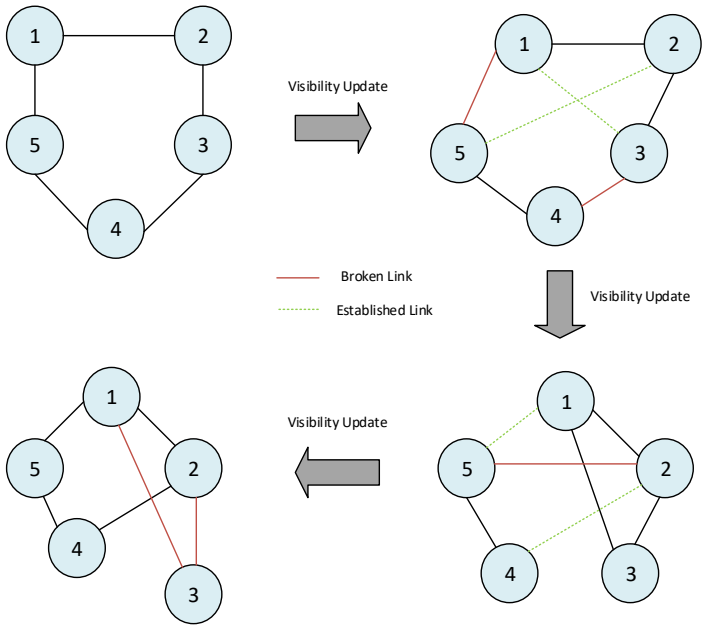


Figure 2.7: Topology in respect to time

The example shows a simple four-node environment with four different time step evaluations of 10 minutes each. The time step can vary depending on the application space. For example, a UAV may update positions multiple times per minute, but GPS satellites may update much more slowly. The topology changes show that the network is self-organizing, allowing nodes to be disconnected and connected based on visibility data based on relative positions.

2.5 Graphical User Interface

The emulation environment can be controlled through a command-line interface using a set of scripting tools. The environment can be built, deleted, and run from

the scripting tools but modifications require an understanding of the architecture. To utilize all features of the environment, a general knowledge of Docker and OVS commands is needed. Furthermore, the data produced and displayed to the user can be difficult to interpret and visualize, making emulator operation less intuitive and interactive. To provide a cohesive user experience with proper technical abstraction, a Graphical User Interface (GUI) was developed using the Qt framework [35] to operate, manage, and interact with the emulation environment. The GUI allows the environment to be easily controlled, providing responsive feedback enabling rapid prototyping, evaluation, and data visualization. Figure 2.8 shows the main control panel and supported tabs.

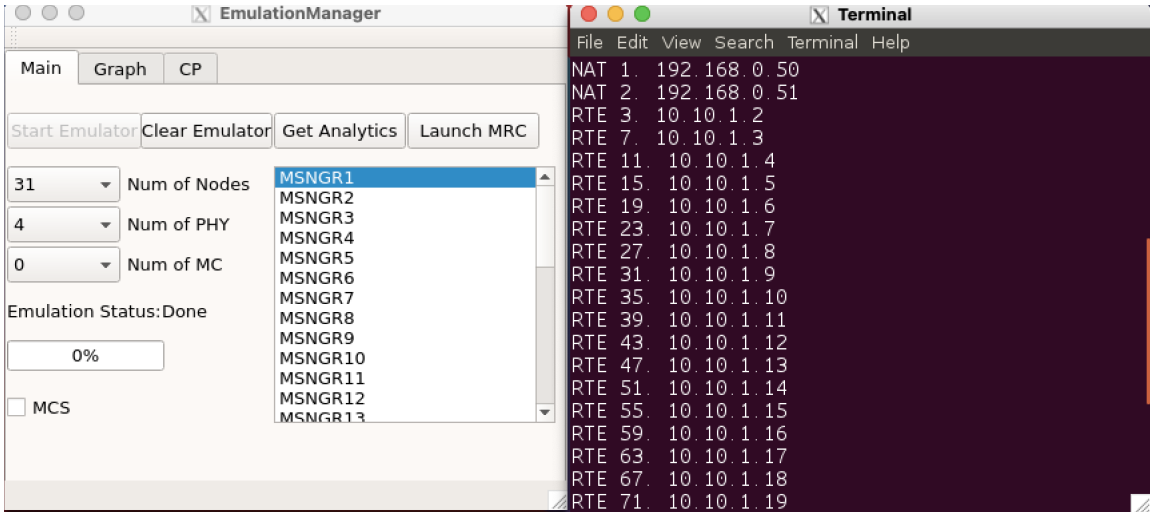


Figure 2.8: Emulation Manager Main Control Panel

2.5.1 Main Control Panel

The main control panel allows a user to build a Docker-based emulation environment with nodes, PHYs, and master coordinators. The GUI uses low-level operating system primitives to manage Docker container instantiation processes, OVS instances, and virtual networking. The GUI uses User Datagram Protocol (UDP) sockets to communicate with all control processors to manage and monitor internal statistics.

The control panel also allows MSNGR components to be dynamically spawned inside a node, enabling the user to generate traffic between mobile nodes of varying payload sizes. The user can also view each node's internal routing tables to gauge the routing algorithms' connectivity. The control panel mainly provides a means to abstract the Docker specific tasks needed to manage and run a simulation. A progress bar is utilized to provide user feedback during the environment construction since this can be quite time consuming for large scale emulation.

2.5.2 Control Processor Panel

The control processor panel allows the GUI to communicate with all mobile nodes to check internal statistics and dispatch commands to the environment. The feature enables a user to query a mobile node to check PHY statistics and external mobile node connections. This allows a user to verify external connectivity and check which nodes have direct connections. The panel also enables traffic models to be generated from source to destination, making the feature useful for evaluating single-hop communication performance. Figure 2.9 shows how the panel is used to view connectivity and the form used to generate arbitrary traffic model streams.

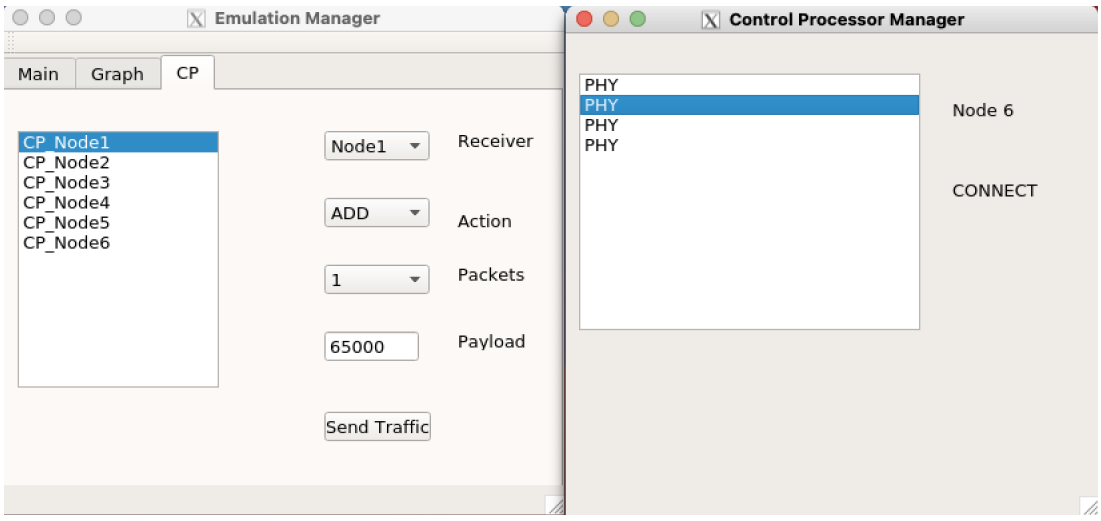


Figure 2.9: Emulation Manager Control Processor Panel

2.5.3 Graphing Utility

The graphing utility allows real-time data visualization, which provides system performance feedback to the user. The display statistics show the latency of inter-node communication links using small payload sizes and the average bits per second (bps) flowing through each communication link. This feature allows a user to visualize data flow through the MANET emulation environment and provide real-time performance metrics when deploying and evaluating traffic models. Figure 2.10 shows the two data displays. These data displays can be toggled by pressing the space bar when the graphing utility is in focus. The graphing utility dynamically scales based on the number of nodes within the environment. For example, a six node environment will have 24 active links, while a 32 node environment will have 128.

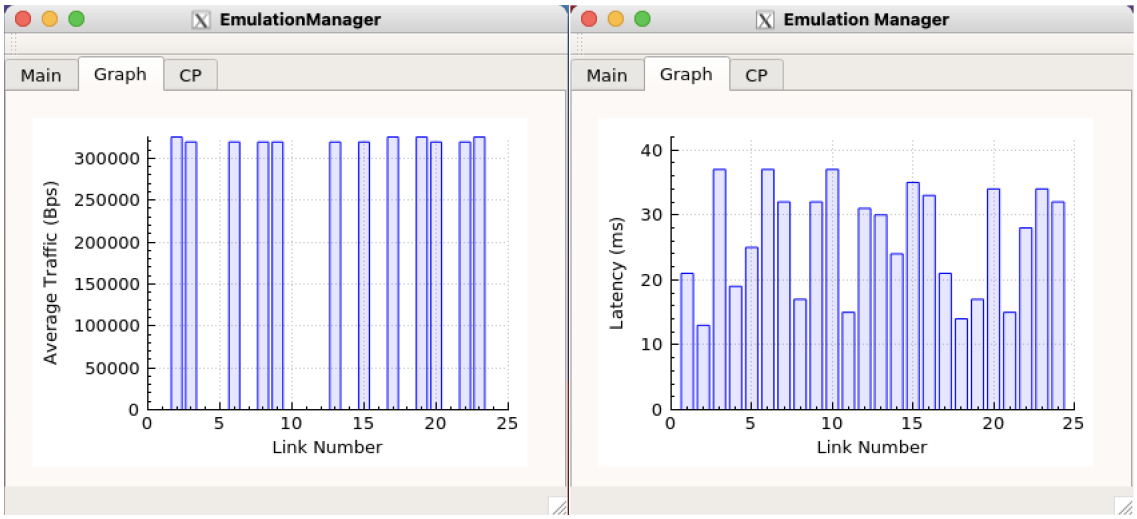


Figure 2.10: Emulation Manager Graph Utility

2.6 Hardware Emulation Support

To support hardware emulation, the environment must function the same way as a distributed embedded testbed. Leveraging nested Docker containers provides a flexible deployment environment while maintaining a network infrastructure that can be realized on hardware devices.

2.6.1 Node Hardware Emulation

A mobile node is comprised of many components that help represent a MANET capable node, such as a MSNGR, Control Processor, and multiple PHYs. Each of these components is represented using a user space program that operates using socket-based communications. The Docker infrastructure allows the program to be run in a secure and isolated operating system with a separate network stack. The virtual Ethernet interface then communicates with all other components using Linux network primitives. The virtualized environment operates in the same way as an operating system on a physical computing device. This allows each component to run transparently on a Raspberry Pi with the Ubuntu operating system. When a component tries to initiate a connection, a low-level driver will translate packets out of the physical Network Interface Card (NIC). This abstraction allows each component to behave similarly to an embedded device. This approach allows a mobile node to be emulated in hardware using multiple Raspberry Pi's. A physical switch can be utilized to connect the components. Figure 2.11 shows how a MANET node can be represented as a cluster of Raspberry Pis.

2.6.2 System Hardware Emulation

Each hardware-based mobile node can be evaluated in a MANET simulation system. The emulated PHY using a Raspberry PI can be connected to an OpenFlow switch using its onboard NIC. Thus, any arbitrary sized network can be emulated in hardware as long as there are enough available computing and networking devices. The software SC can be instantiated on a Raspberry Pi and connected directly to the OpenFlow switch to program flow rules into the device.

Furthermore, the environment can operate in a para-virtualized manner where it only partially represents the underlying hardware. In this scenario, each Raspberry PI acts as one mobile node where the node's internals are abstracted using Docker and

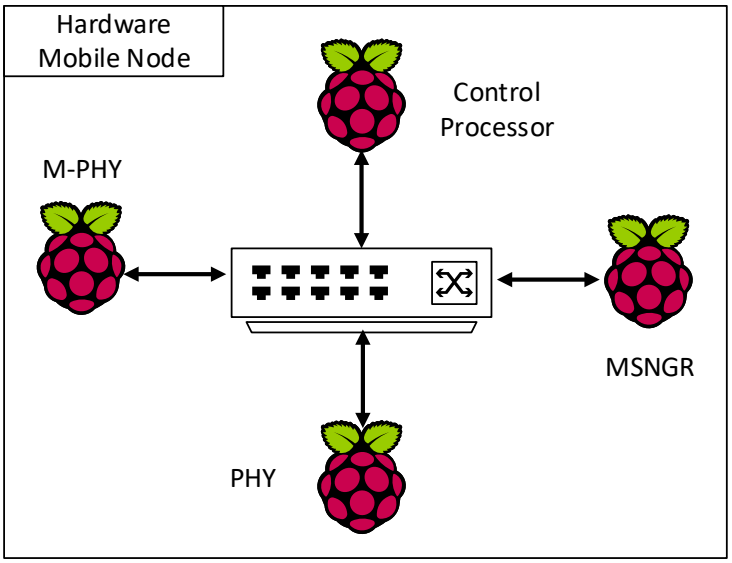


Figure 2.11: Hardware Based Environment

OVS. The external connections between mobile nodes would still be hardware-based with an OpenFlow capable switch controlled by a software entity.

The deciding factor in deployment is based on cost and hardware acquisition. A hardware-based system would require $(2 + PHY) * Nodes$ embedded devices, while a para-virtualized environment would only require an embedded device count equivalent to the number of nodes being emulated. Of course, this type of hardware-based simulation is not entirely scalable because the number of necessary devices becomes high when emulating anything more than a few nodes. Another limiting factor is the number of ports available on an OpenFlow capable switch. For example, a 32 Node environment with 4 PHY devices would require a 128 port OpenFlow switch, which is not available commercially.

CHAPTER 3

QUANTIFYING SYSTEM PERFORMANCE

The emulation infrastructure helps model, prototype, and explore networked distributed systems. The infrastructure emulates a distributed system's interactions and communication by virtualizing a CPU. This compromise leads to a disparity in system performance between the emulation infrastructure and an implemented distributed system. In a distributed system, all system processes can be run concurrently across many CPUs. There is no need for context switching, synchronization methodologies, hypervisors, or resource overhead for virtualized interfaces. This chapter will discuss how system performance is quantified for the emulation environment. The evaluation is split into three broad categories: simulation performance, network reachability, and resiliency. The server used for performance evaluation is a Dell PowerEdge T640 with 192 GB of memory and a 20-core Intel Xeon processor with 40 virtual threads. The server operating system is Ubuntu 16.04 LTS [36] with kernel version 4.4.0-138.

3.1 Simulation Performance

In the emulation environment shown in Figure 2.5, there are two distinct types of network links, inter-node and intra-node communication links. These communication links operate on two distinct transport level protocols, TCP and UDP, respectively. This section evaluates the emulated bandwidth for both types of communication links during a simulation. This analysis highlights which communication link acts is a bottleneck for message passing. Two simulation environments are evaluated, one environment with six mobile nodes and another with 32 mobile nodes. Each

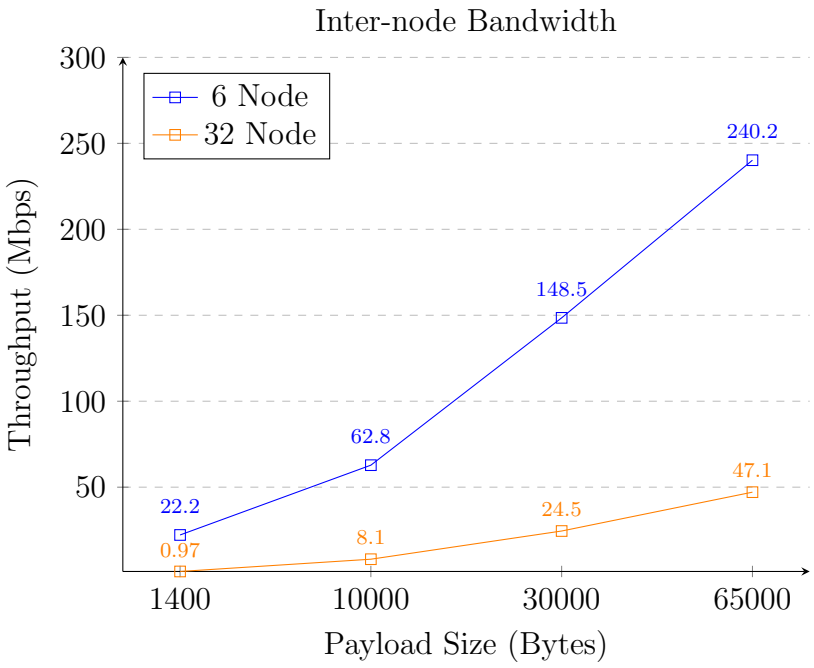


Figure 3.1: Inter-Node Bandwidth

simulation environment runs both a simple point-to-point communication network and a traffic model with multiple streams transmitting simultaneously.

3.1.1 Single Stream Bandwidth

Single stream bandwidth is defined as the total number of bits per second transmitted from source to destination, when only one active node is transmitting data. The same tests are evaluated for both a TCP link used to communicate with components inside a mobile node and a UDP link used to communicate between nodes.

Figure 3.1 shows the performance metrics for inter-node communications using the TCP protocol. In each environment, there is only one active link being used for data transmission. The initial results highlight the discrepancy of performance between six- and 32-node systems. For example, in a distributed hardware-based system, processes running on node C will not affect the bandwidth of data transfer between nodes A and B. In our emulation environment, every process on the system competes

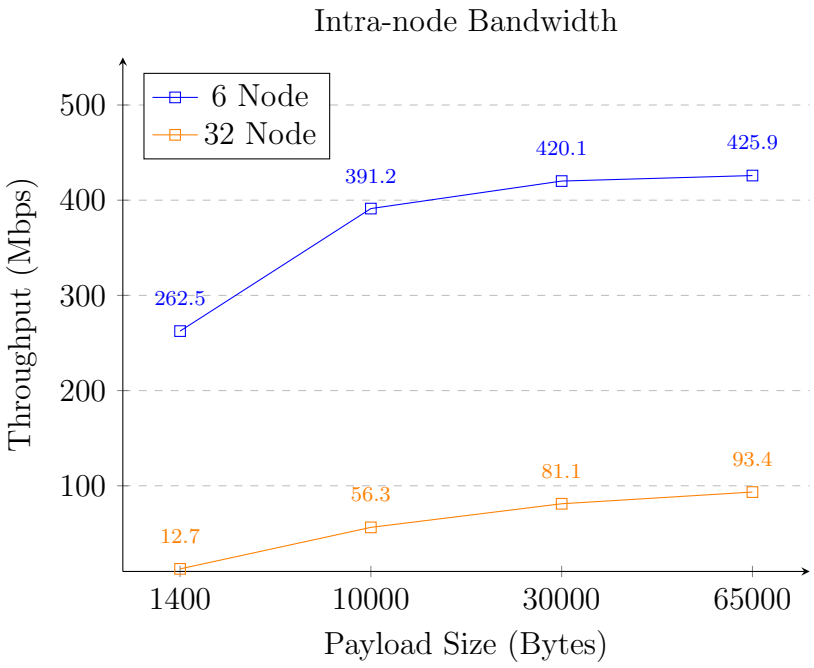


Figure 3.2: Intra-Node Bandwidth

for cycle time on one CPU, which degrades performance. Larger payload sizes yield better bandwidth utilization of the links. This result is attributed to TCP's windowing and the associated overhead enforced by collision detection, retransmission, and sequencing. For small payloads, this transmission overhead becomes a significant percentage of the payload size, reducing link utilization.

Figure 3.2 shows the intra-node communication link bandwidth. This data shows that UDP based payloads have a higher overall bandwidth between the two emulation environments. The UDP protocol can achieve higher bandwidth due to its simplicity but does not guarantee delivery, collision detection, or in-order retrieval. The protocol header of UDP is more simplistic and has less overhead and impact on overall throughput. The results show that the TCP link acts as a major bottleneck for message passing capabilities, especially for smaller payload sizes. The difference in bandwidth is more than 10x when evaluating payloads of 1400 bytes.

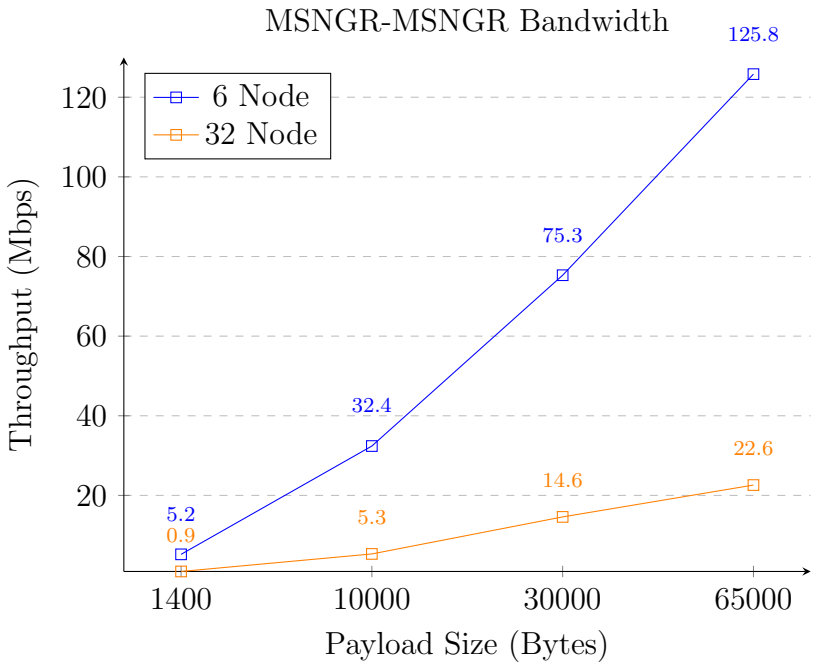


Figure 3.3: Full Message Bandwidth

The previous tests showed inter-node and intra-node communication link performance for point-to-point connections. However, they do not measure performance across entire communications paths. In Figure 2.5, there are a total of three communication links and two virtual switches between the MSNGR components of two mobile nodes. The combination of these links and switches form a full source-destination communication path. Since MSNGR components generate and receive traffic in each mobile node, the path bandwidth determines the message passing capability of the system. Figure 3.3 shows the message passing bandwidth between messenger nodes in both six- and 32-node environments with varying payload sizes.

Higher bandwidth can be seen at larger payload sizes, influenced by the TCP bandwidth for inter-node communication links. Limited inter-node bandwidth is the dominant effect since TCP packets have considerable overhead and limit total performance. In TCP protocol schemes, lower level kernel drivers have multiple quality assurance checks to help enforce arrival guarantees, which UDP does not provide at

the transport level. Additionally, the external switch has many connections in a 32-node environment, which drastically slows down packet processing, classification, and forwarding. Overall, the results show that the total bandwidth is much closer to the inter-node communication link bandwidth, demonstrating that these links limit system performance. The limiting factor for system performance is the overhead of the TCP protocol and virtual OVS switch used to connect mobile nodes. The overhead is even more significant with larger sized environments as the external switch scales with the number of ports.

3.1.2 Traffic Model Bandwidth

MANET communication is typically not limited to one message stream within an entire network. Therefore, the tests in the previous subsection do not evaluate message passing bandwidth under typical conditions. The tests described in this subsection quantify message passing bandwidth in the presence of multiple data streams. Our traffic model allows the emulation environment to evaluate realistic MANET payloads with command, image, and telemetry data. In the previous subsection, it was shown that maximum link utilization is achieved with larger payload sizes. The tests in this section use payloads at the limit of TCP transmission, roughly 65k bytes. Traffic stream types that support these payload sizes include image data, a common payload type in MANET based networks including those consisting of UAVs. Emulation environments with six and 32 mobile nodes are evaluated in the traffic model. Figure 3.4 shows traffic model performance for five image data streams for both six and 32 node environments. The performance data highlights the bandwidth of each stream.

Figure 3.5 shows the connectivity for six- and 32-node systems. The red nodes indicate those reached through intermediate hops, and the gray indicate those that are a source or destination. The red edges represent links with active traffic based on

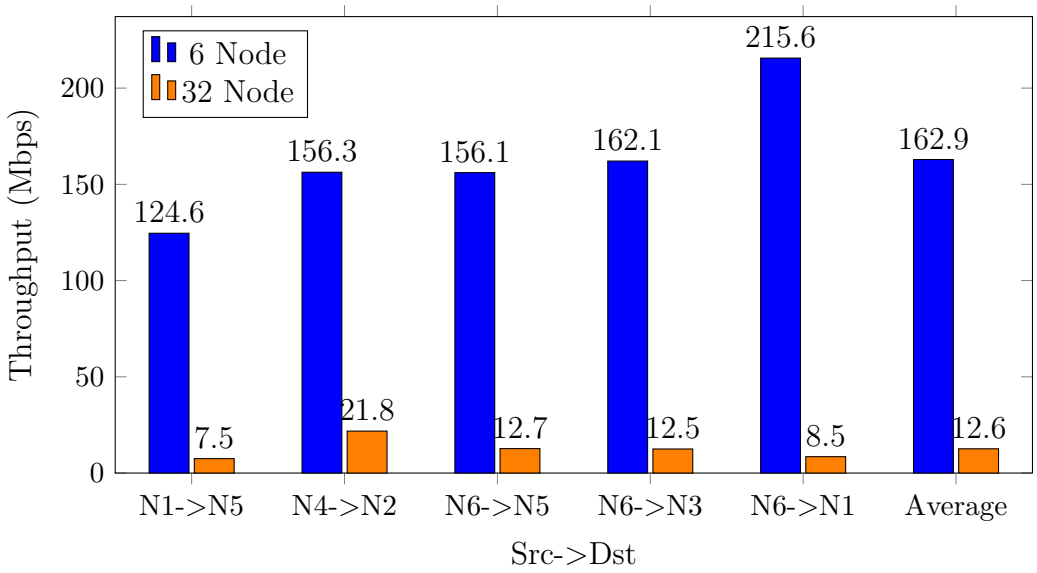


Figure 3.4: Traffic Model Bandwidth

the five image data streams. All of the six-node streams are transmitted in a single hop, while the 32-node environment is transmitted across 4, 1, 2, 2, and 3 hops, respectively, for the results in Figure 3.4. The number of hops is defined as the number of intermediate links needed for a packet to reach its destination from its respective source. From this data, we can see that each image stream has a different performance profile. In this environment, all five traffic streams run concurrently. The number of processes that compete for CPU cycles is greater than the number of cores and virtual threads. Performance deviation is attributed to the overhead caused by context switching and the importance of each process in the Linux scheduling algorithm. All user-space processes in the emulation infrastructure are given the highest priority, effectively making them real-time. The variance between test runs is high due to the number of high priority processes running concurrently, including kernel-space processes. If higher priority is given to specific mobile nodes, a performance shift is seen, favoring processes with higher priority.

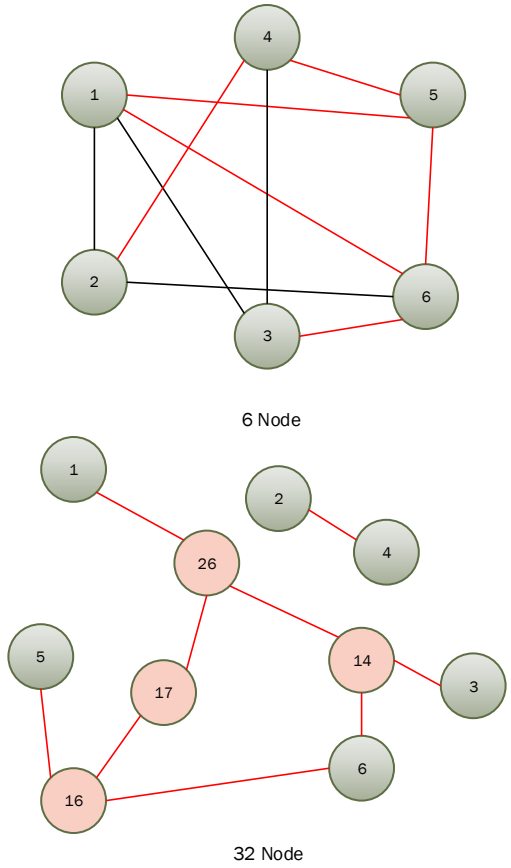


Figure 3.5: Connectivity Graph for Traffic Model Simulation

The variation is even more significant in the 32-node environment due to multi-hop routes. In the six-node environment, most data streams reach their destination in a single hop. For example, in an environment with four PHYs, Node 1 communicates directly with all but one node. In a 32-node environment, Node 1 can only directly reach four distinct nodes, 13% of the entire network. Therefore, most communication is made through multi-hop routes, degrading system performance, and increasing the variance between test runs. Our routing algorithms are optimized to choose the shortest path from a traffic stream's source to a destination.

The 32-node environment has 28x more possible communication paths than a six-node environment. To test system performance in the larger environment, the number

of active traffic models was increased to span a more complete subset of mobile nodes. The traffic models were intentionally overlapped to demonstrate how performance is impacted when nodes receive or transmit streams concurrently.

Table 3.1: Traffic Model Grid (Mbps)

	N15	N16	N17	N18	N21	N22	N23	N24	N25	N26
N1	x	x	11.1(2)	x	12.3(3)	x	x	x	x	x
N2	x	x	x	6.2(4)	8.5(5)	x	x	x	x	x
N3	x	x	x	x	x	x	x	7.2(4)	9.4(2)	12.5(2)
N4	14.1(2)	9.5(3)	x	x	x	x	x	x	x	x
N5	x	x	x	x	x	x	x	x	8.8(3)	x
N6	x	x	x	x	x	x	x	7.6(4)	x	x
N7	x	x	x	x	20.5(1)	12.6(2)	x	x	8.3(3)	x
N8	x	x	x	5.5(5)	x	x	x	x	8.7(3)	12.6(2)
N9	x	x	x	x	x	x	x	8.1(3)	x	x
N10	x	x	x	x	x	13.1(2)	22.5(1)	x	x	x

Table 3.1 shows performance metrics for 20 source-destination streams. The vertical rows represent the destination nodes for a traffic stream, while the horizontal rows represent the source nodes. The table entry where a horizontal and vertical row meet is the performance and (hop count) of that traffic stream from the respective source to destination. An x in the entry represents that there is no stream with that source and destination pairing. All bandwidth results in a vertical row indicate competition between receiving streams on the destination MSNGR component. All bandwidth results on a horizontal row indicate competition between sending streams on the source MSNGR component. The results demonstrate that the system performance scales well with a larger number of traffic streams. However, the variance from different links is even greater due to increased competition when multiple streams are sent or received from the same node. Similarly, performance is lower when a receiving node handles traffic from multiple sources. An example is the traffic stream from node 10 to node 23. This route has a bandwidth of 22.5 Mbps with one hop from source to destination. However, the bandwidth from node 7 to node 21 is only 20.5 Mbps for

one hop. The only difference between these two routes is that node 21 receives two traffic streams while node 23 only receives one.

3.2 Network Coverage

In a MANET, nodes cannot always communicate with each other, since the network topology relies on distance, communication medium, and environmental factors. In this section, we introduce the concept of network coverage. In this context, network coverage is defined as the percentage of nodes that can be reached by a central master coordinator component. Figure 2.3 demonstrates the architecture of this node.

In the following tests, the number of PHYs within a mobile node will be varied, limiting the number of remote connections that can be made from a mobile node. The number of routing endpoints will also be varied. These iterative tests aim to find the minimum configuration of components such that full network coverage is maintained. The more components within a system, the more resources utilized. Therefore, minimizing the number of components within the system will allow larger environments to be emulated with similar network coverage. Table 3.2 shows the iterative test results to determine the minimum number of components needed so that full network coverage is maintained.

Table 3.2: Network Coverage

PHY per Node	PHY per Endpoint	PHY per MC	Coverage (T/F)
4	0	1	T
3	0	1	T
2	1	1	T
1	4	4	F
1	3	3	F
1	2	2	F
1	1	1	F
0	0	0	F

Table 3.2 shows the results from a 32-node simulation. Three components are deployed in the environment: mobile nodes, routing endpoints, and master coordinators (MCs). Each of these components is tested with varying numbers of PHYs per node to allow remote connections. The master coordinator must have enough PHYs to connect to every routing endpoint. Additional PHYs can be used to connect to remote mobile nodes to expand connectivity. The column containing the number of PHYs in the MC excludes those allocated for connections to routing endpoints. Similarly, the column containing the number of PHYs in an endpoint excludes those allocated to the master coordinator. The number of PHYs within a routing endpoint constrains how many nodes can be connected externally. Fewer PHYs in a routing endpoint indicates fewer connectivity capabilities. The number of PHYs in a master coordinator indicates how many external connections can be made to mobile nodes.

The data shows that the minimum configuration of PHYs for full coverage contains two PHYs per mobile node, one PHY per endpoint, and one PHY per MC. Thus the mobile nodes can connect to two other mobile nodes, the endpoints can connect to one node, and the master coordinator can connect to one other node. The results show that the routing endpoints help expand network coverage and allow fewer PHYs to be used in each mobile node. This demonstrates a trade-off between memory utilization and performance. The routing endpoint approach decreases the total number of containerized components at the cost of performance. Each message sent from the master coordinator using this approach will need to go through extra hops from source to destination, impacting total bandwidth. The difference in memory and resources becomes more apparent in larger sized emulation environments. For example, in a 32-node environment eliminating two PHYs per node yields a reduction of 64 components.

3.3 Network Resiliency

Our final performance evaluation involves network resiliency, which is defined as the amount of network coverage maintained in the presence of link failure. In MANET environments, link failures can happen due to hardware/software issues, environmental factors such as transmission distances, and noise. A robust MANET should still be able to provide coverage in the presence of link failure. Recent research has shown significant interest in simulating and evaluating algorithms and protocols relevant to MANET in the context of wireless communications. These efforts include link repair in Ad-hoc On Demand Distance Vector (ADOV) algorithms [33] and failure prediction in MANET based systems using statistical methods [25]. These efforts aim to increase state of the art in MANET based systems by increasing network coverage during navigation of various topologies and predicting potential failures for more efficient routing.

In this section, we quantify network resiliency in the emulation environment when one or more links fail using a variation of ADOV for route exploration. The emulation environment supports link jamming to simulate link failure, which allows a user to disconnect any active link. When a link is jammed, the SC will attempt to find a connection to another PHY on the same or a different node to preserve network connectivity. In these simulations, we assume that only one PHY is rendered useless when a link is jammed, giving another PHY an opportunity to initiate a connection.

Network resiliency evaluations were performed for a 32-node emulation environment. The six-node system has insufficient connectivity to provide meaningful results in the case of link failure. In our experiments, links are jammed in a randomized fashion. The evaluation baseline assumes zero jammed links and 100% network coverage. The test starts with four links jammed, and each successive experiment iteration assumes an additional four jammed links, up to 24 links total. Table 3.3 shows the incremental coverage data. The first column shows the number of links deleted, the

Table 3.3: Network Resiliency

Number of Links	Coverage	Links Recovered
0	100%	0
4	96.5%	1
8	82.7%	4
16	58.6%	8
24	44.8%	10

second column shows achieved coverage, and the third column shows the number of links that were recovered by dynamic re-routing using new PHY connections.

The table shows that coverage does not decrease linearly. This result indicates that the MANET framework is a self-healing network, in which damaged links can be re-purposed for other connections to maintain coverage and network performance. The amount of recovered links increases significantly when the number of jammed links exceeds 20%. This result is attributed to how negotiation is handled; when a link is rendered inactive, one PHY is allowed to connect to another unused node. If only one link breaks, it is unlikely that its PHY can be re-purposed because the pool for available links is small. When more links are broken, the collection of available links increases, and the probability of re-initiation increases considerably. It is important to note that links within a MANET typically do not have uniform importance. It is often the case that one link rendered useless could have a drastic effect on coverage while a different link could have none. The results demonstrate that the MANET emulation environment can handle some link failures but cannot recover from all of them due to the non-uniform nature of link importance.

CHAPTER 4

SILICON OPENFLOW SWITCH

4.1 OpenFlow Fundamentals

The emulation environment uses an OpenFlow capable virtual switch called OVS to connect inter-node communication links. Figure 2.5 shows the external OVS instance and its relation to the emulation environment. The switch allows software to configure how traffic is forwarded from port to port. This functionality enables the decoupling of the data and control planes. In this chapter, the characteristics of OpenFlow capable switches are discussed and an implementation in digital hardware is presented.

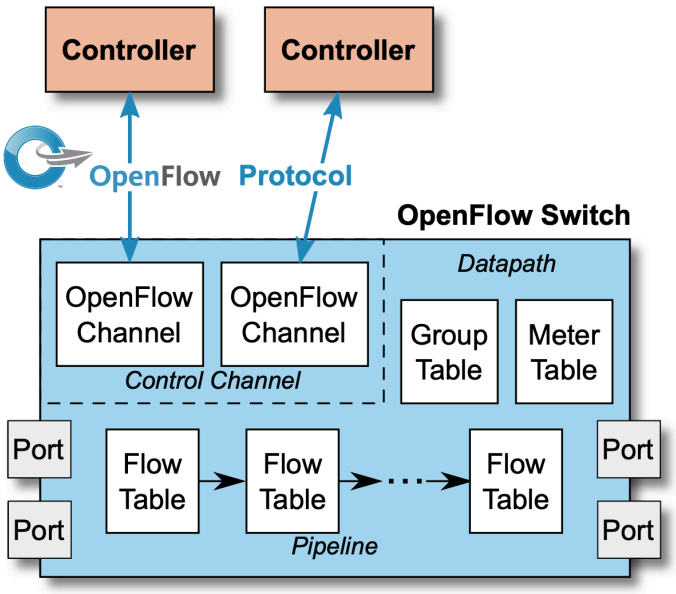


Figure 4.1: Open Flow Architecture [28]

Figure 4.1 depicts a switch that can support OpenFlow functionality. The architecture includes a control channel and a data path. The control channel allows an OpenFlow controller to modify the hardware's switching capabilities. The data path is compromised of N levels of flow tables which hold information on how packets should be processed. For example, a flow table can hold an entry that specifies that all TCP packets to be forwarded out of port 2. Similarly, a flow table can also hold an entry that specifies that all ARP packets should be dropped. The OpenFlow controller is typically located outside of the switch. The configuration channel is generally interfaced through a secure TCP socket. In this work the OpenFlow control port is accessed via a PCIe bus to allow user space software to control and manage flow entries and packet statistics. An OpenFlow switch's data path contains a group table and a meter table to store internal statistics on how flow tables are accessed. For example, statistics that specify how many times a flow entry has been hit or how many packets have been processed on a specific port can be stored in local Static Random-Access Memory (SRAM).

The OpenFlow protocol specifies how packets should be parsed and classified. Packets with Virtual Local Area Network (VLAN) tags are classified differently from those without tags. Packets at different layers of the Open Systems Interconnection (OSI) [10] model also must be handled differently. ARP packets reside on level 2 (L2) and do not have an IP level header. TCP packets have both IP and transport-level headers. Therefore, an ARP packet is classified and output in fewer cycles than a TCP or UDP packet. A packet with L4 fields can be classified in two cycles while L2 fields are done in one, assuming a 256-bit bus width.

4.2 Switch Architecture

A preliminary version of an OpenFlow switch has been developed to support packet offloading for matching, classification, and forwarding. Here, we describe the

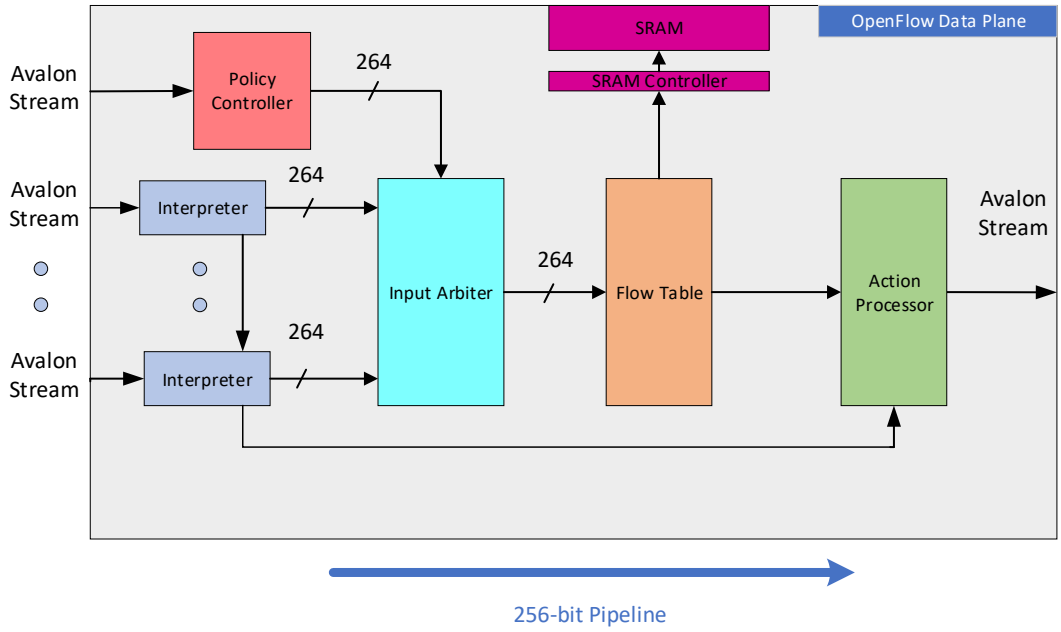


Figure 4.2: OpenFlow Pipeline

hardware data path written in RTL that supports the OpenFlow protocol. Figure 4.2 shows a block-level design for the OpenFlow pipeline. The diagram shows stream interfaces for data transfer. These interfaces are currently implemented using the Avalon Stream protocol. The protocol is a standardized bus specification in Intel FPGA systems that allow multiple writes from software-based sub-systems to be translated into a data stream for high performance applications [19] .

In this design, there are N Avalon stream interfaces allowing network traffic to be processed externally. Each interface has a TX (transmit) queue and RX (receive) queue that reside inside the interpreter modules. The interpreter module parses incoming Ethernet frames and constructs a match-field tuple, which is utilized during packet lookup and classification. The match-field tuple is a 264-bit tag that contains various fields from the packet which helps provide a unique identification. At the top left of the figure, an Avalon Stream interface to a policy controller is shown. The policy controller receives messages from the OpenFlow controller to add/delete/modify

flows or perform packets' actions. The policy controller receives messages from the host machine through the PCIe bus through a first-in-first-out (FIFO) queue. The PCIe bus provides an Avalon Stream interface for data communication, which allows a convenient interface mechanism to this design. The policy controller operates like the interpreter, except that it constructs a match-field tuple to write into the flow table instead of a lookup. These outputs go into an input arbiter that utilizes a round-robin scheduling policy without time slices to allow fair access into the data pipeline.

The data pipeline includes a flow table, a content addressable memory that initiates an action based on packet match fields. Typical actions specified by the flow table include packet drop, packet forward, and packet modification. After an action has been retrieved, the packet is forwarded to an action processor, where the corresponding action is performed. A local SRAM is utilized as a buffer. The data plane's width is 256-bits wide, which is constrained by the bus width of the PCIe IP provided by Intel.

4.3 Header Parsing

The Ethernet interpreter design reads and parses incoming Ethernet packets. These packets encapsulate a complete frame that can consist of L2, L3, or L4 headers of the OSI model. Header matching must support the following protocols from the OpenFlow specification: IPV4 [16], ICMP [15], ARP [31], TCP [17], UDP [21], and VLAN [5]. Figure 4.3 shows a block diagram of the Ethernet interpreter module.

In the figure, there are two FIFO memory elements. The RX FIFO is used to store incoming data from the Avalon stream interface. The TX FIFO is used to re-buffer the data as the parser reads from the RX FIFO. The Avalon stream interface has a data bus, a write enable signal, and a full signal, which allows back pressure so useful data will not be overwritten if an overflow occurs.

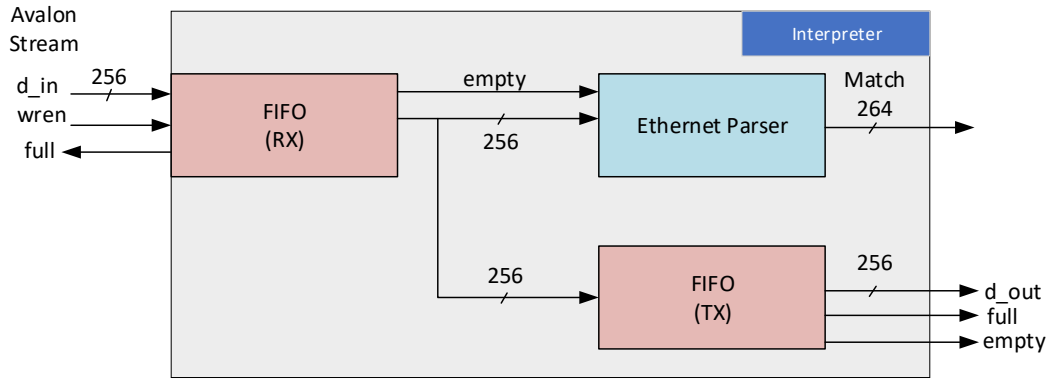


Figure 4.3: Ethernet Header Parser in the Interpreter

The Ethernet parser module within the interpreter sequentially reads 256-bit payloads and interprets the fields within them. The parser is a state machine that constructs a 264-bit match field based on packet header. The FSM determines packet type and specifies appropriate packet processing actions.

4.4 Flow Table

The flow table design includes three groups of memory elements. An OpenFlow switch performs two types of matching: exact and wild card. An exact match requires that all 264 bits of the input match an entry within the table. The exact match is implemented using a Content Addressable Memory (CAM). The wild card match allows a mask to be provided, which signifies which bits can be ignored in the match process. A Ternary Content Addressable Memory (TCAM) must be utilized to extend the memory to handle don't care bits to achieve this functionality. The TCAM is a special type of CAM that allows matching on three input fields: '1', '0', and 'X'. For example, if only the port the packet came on is of interest, a mask can be provided to disregard all other fields. In this configuration, all packets would be matched on the specified input port regardless of the other fields. Figure 4.4 shows the flow table

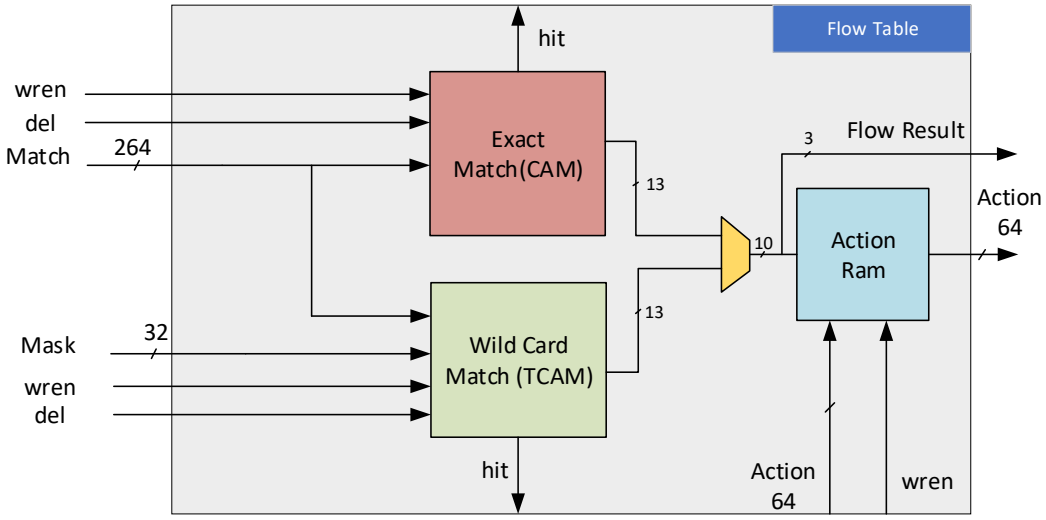


Figure 4.4: Flow Table

general architecture. Pipeline stages and most of the control logic are omitted for simplicity.

The CAM and TCAM memory elements handle packet lookup when incoming messages are parsed from the OpenFlow or Ethernet Interpreter modules. Both memory elements can handle entry writes and deletes. This functionality allows entries to be added, deleted, or modified when an OpenFlow message is received from the controller. If a received packet matches one of the CAM entries, the hit flag is asserted, and the corresponding address of the match is given to the priority encoder. The priority encoder determines which address is given to the action RAM, which holds information about how packets are handled when a match occurs. If both the TCAM and CAM have a match, the priority encoder will use the CAM address. The CAM's also specify the flow result signal based on the input port of the incoming packet. In this case, the flow result signal is 3-bits indicating a total of 3 input ports. This can be generalized to an N-bit signal for a total of N input ports. The action RAM then fetches the action associated with the matched packet. The action

Random-Access Memory (RAM) stores where the packet should be forwarded based on the hit flow entry along with an action, if any. Each valid CAM address has a corresponding entry in the action RAM. The action processor then performs data movement adhering to the received action.

4.4.1 FPGA Content Addressable Memory

Modern FPGAs do not contain CAM cell memory, although they can be emulated using logic [24]. The CAM key can be used as an address into RAM. The data at the location will consist of a '1' if there is a valid entry. The drawback of this implementation is memory utilization when using large keys. In this design, a 264-bit key must be used for the CAM. This means that we must have a total of 2^{264} locations in memory, which no FPGA can support. To address this issue, multiple smaller memories can be utilized. The 264-bit key can be split up into K different RAMs, in which the number of address bits for each is P. The equation shown below indicates the total number of memory bits required to realize a CAM in an FPGA using this approach. Each RAM has 2^P locations, and there is a total of $264/P$ total RAMS in parallel. Then each location has a width proportional to the number of entries within the CAM.

$$\frac{264}{P} * (2^P) * CAMEntries$$

A bit-wise AND can then be performed on the outputs of the RAMs and then converted into an address using a simple priority encoder. The encoder translates the resulting one hot encoding into a valid address for lookup. The basic architecture of the CAM can be found in Figure 4.5. The same concept can be applied to the TCAM, except at most 2^P writes are needed for each address specified in the TCAM mask. For example, a mask of all zeros indicates that all bits are don't cares, which translates to valid data at every RAM location. An FSM based model is used to

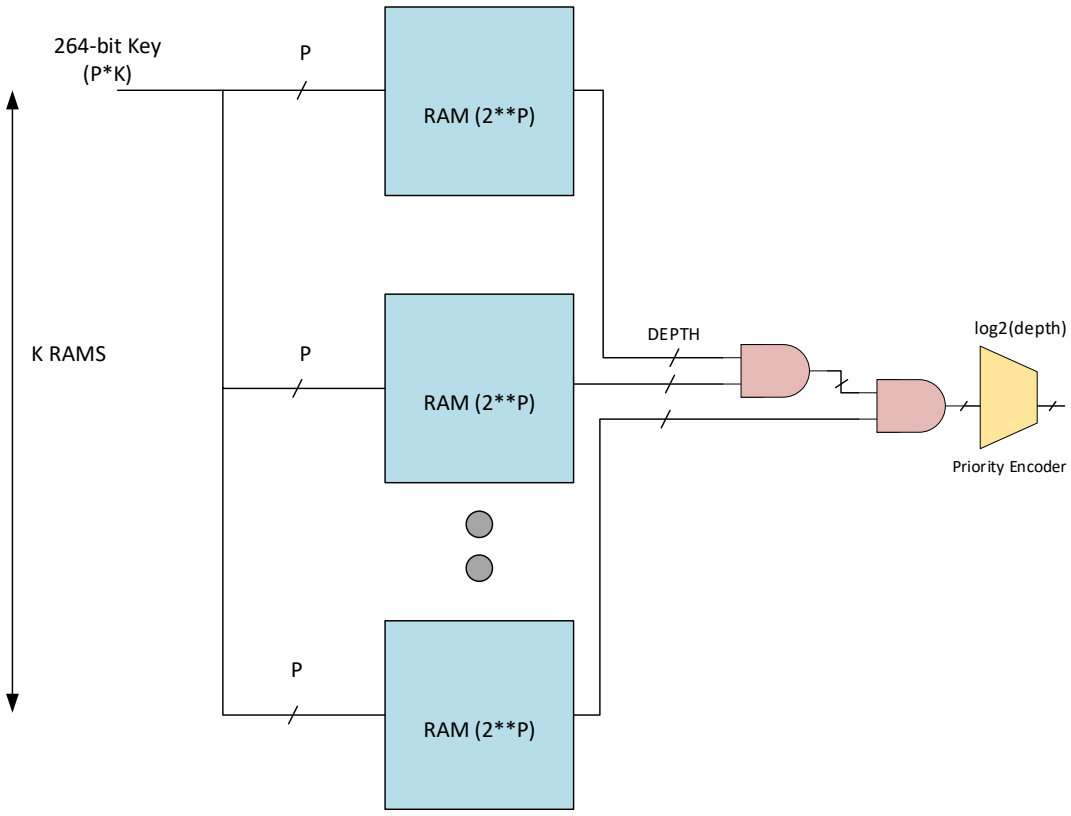


Figure 4.5: CAM FPGA Emulation

perform sequential writes based on an input mask to determine don't care encoding [22].

4.5 Action Processor

The action processor module performs packet processing options supported by the OpenFlow protocol. These actions include setting, modifying, or removing network packet fields and forwarding or dropping packets. In the current implementation, the module is used for forwarding traffic, dropping traffic, and setting MAC addresses. Figure 4.6 shows the architecture for the action processor.

The port inputs are connected directly to the TX queues located in the interpreter module. The flow result and action input signals indicate which input port has a

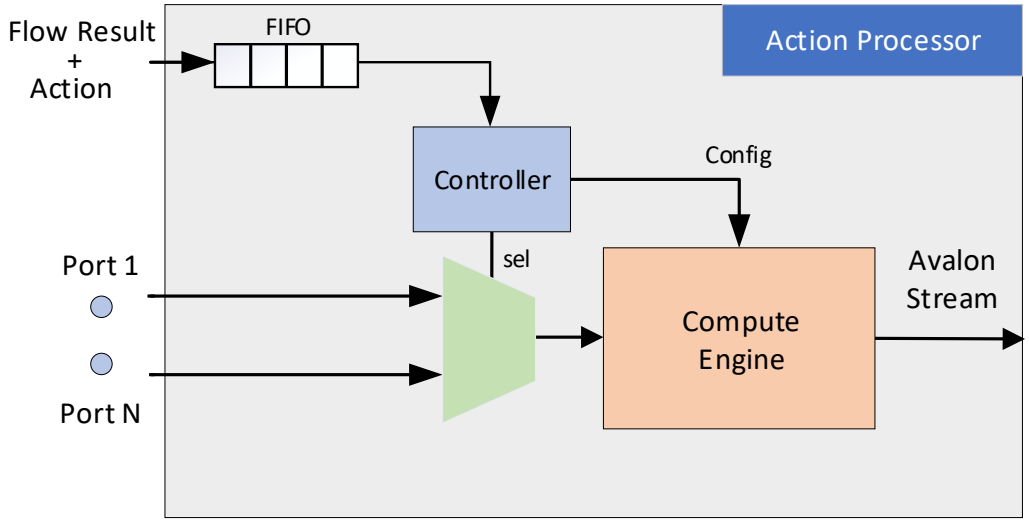


Figure 4.6: Action Processor

packet and the action needed to be performed, respectively. For example, if the flow result indicates no flow match, the packet is dropped, and the action field is ignored. Conversely, if the flow has a valid match, the port is connected to the compute engine through the multiplexer, and the action is performed. The controller facilitates the data transfer between the TX queues and the compute engine. The compute engine is configured based on buffered flow results and action data provided by the flow table. The Avalon Stream output interface connects to downstream logic used to transmit packets over PCIe which will be discussed in Chapter 5.

CHAPTER 5

PCIE SR-IOV VIRTUAL NETWORK ACCELERATOR

Processor-based networked virtualized systems are flexible for prototyping and algorithm exploration. However, there is an inherent limitation in system performance due to the threaded nature of CPU resources. In our software-only emulation environment, processes utilize CPU cycles and incur large overheads for context switching. The server currently used for testing includes 20 CPU cores, each with two threads per core. When running a 32-node simulation with four PHYs, 188 user-space processes, each with multiple threads, are created. This does not consider the processes for the virtual infrastructure such as OVS, Docker, and Linux based primitives. Therefore, context switching must frequently occur to guarantee fair access to resources. Figure 5.1 shows the CPU utilization using the htop command during a simulation run.

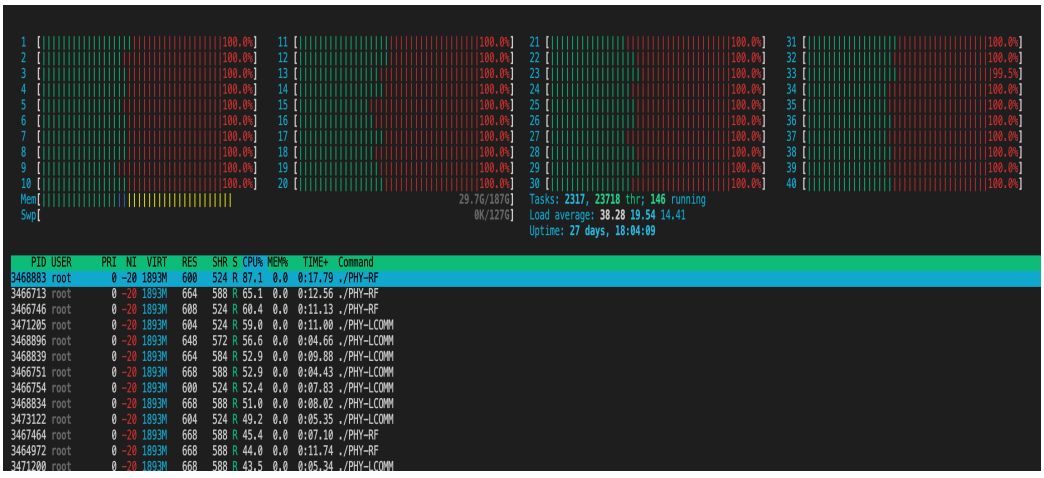


Figure 5.1: CPU Utilization

The CPU utilization shows that all cores and virtual threads are running at high capacity, demonstrating the emulation environment's CPU-bound computation.

Based on this information, the overall performance may be improved by offloading computation from the CPU to a custom hardware data plane. By offloading computation, the CPU will be freed up to work on other tasks , while packets are matched, classified, and transferred into user space memory. The htop results show that reducing CPU utilization could allow more processes to be quickly serviced allowing more concurrency to be exploited in the simulation.

Using the design from the previous chapter as a basis, we developed an end-to-end FPGA based acceleration stack for software-defined networking. The hardware-based OpenFlow switch designed in Chapter 4 was integrated into a PCIe acceleration engine using kernel-level drivers to abstract network operations. The following sections will discuss the design decisions involved at various levels of the HW and SW stack.

5.1 SR-IOV Virtualization Specification

SR-IOV [18] is a specification that supports efficient HW/SW co-design for scalable networked infrastructures that sits on top of traditional PCIe. The SR-IOV specification allows a device such as a network adapter to separate access to its resources through PCIe hardware functions [7]. There are two types of functions enabled by SR-IOV: Virtual Functions (VF) and Physical Functions (PF). The PF advertises the device's SR-IOV capabilities. It configures and manages all enabled VFs. Each VF is typically a memory region or network interface. Each PF and VF is assigned a unique requester ID so that each can initiate reads, writes, and Direct Memory Access (DMA) requests through the host PC software. The specification allows network traffic to bypass the software switch layer of the virtualization stack so that these environments can get near line-rate performance. Figure 5.2 shows an abstract representation of a SR-IOV network stack.

In this scenario, an application-specific process manages the PF while the network stack sends and receives data from the VFs. The PF manager that runs within the

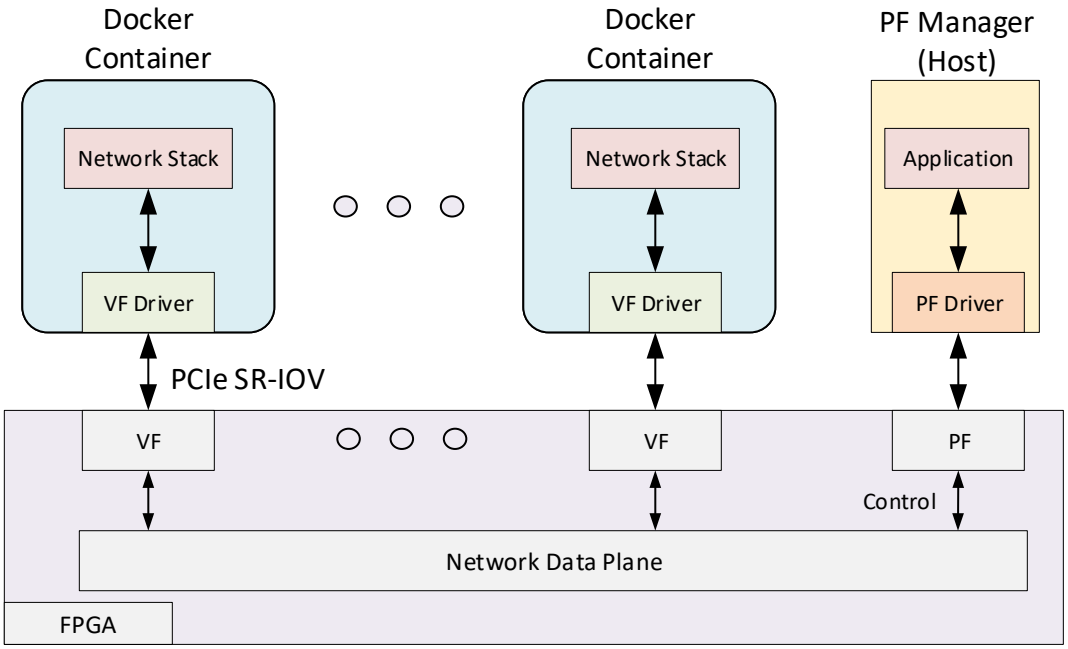


Figure 5.2: SR-IOV Architecture for Containerized Applications

host controls the SR-IOV system configuration and the compute engine in the physical hardware. For example, in a networked system, the PF manager would configure the switching substrate to the handle operation of N VFs. The Docker containers are virtualized operating systems that each gets their own dedicated PCIe function as if there were N physical hardware devices. The virtualization standard allows the host software to mimic how a system operates under the conditions mentioned earlier. Thus, each driver within a Docker container communicates with the hardware as if it has its own I/O device and associated memory resources. The configuration allows data to be transferred from the Docker container to the hardware very efficiently with minimal CPU overhead.

5.2 High-Level Acceleration Solution

To develop a general-purpose accelerator for scalable network interfaces, the PCIe SR-IOV specification was utilized to interface with a custom logic design on an Intel Stratix 10 FPGA [34]. Figure 5.3 shows an abstract representation of how the custom FPGA design interacts with the emulation environment through the facilities supported by the SR-IOV specification. The OpenFlow data plane from Chapter 4 was integrated into the custom HW solution to achieve packet classification capabilities.

Each network interface acts as a port on the external OpenFlow switch used in the emulation environment. The SR-IOV specification supports up to 2048 Virtual Functions (VFs) across all Physical Functions (PFs). Thus an SR-IOV enabled FPGA can support hardware offloading for virtual switches with up to 2048 ports assuming no other physical functions are utilized. Every configuration in our emulation environment can be supported by this offload infrastructure. The theoretical limit for hardware acceleration would be a total of 512 mobile nodes, where each node has 4 PHYs (2048 total ports). This count is subject to FPGA resources that are constrained by memory utilization for base address registers, descriptor tables, and configuration space for PCIe capabilities.

In this solution, each PHY within the emulation environment is connected to a VF through Linux network name spaces and virtual Ethernet interfaces. Each PHY can initiate a TCP-based socket connection using low-level Application Programming Interface (API) primitives. The VF kernel drivers initiate DMA transfers to the physical hardware data plane when the Ethernet device wants to send a packet. When the hardware transfer is complete, an MSIx interrupt will be triggered, allowing the VF driver to reallocate the DMA descriptor and memory space for new incoming packets. After matching and classification of a packet, the data plane will either drop or forward the packet to the correct memory space associated with its corresponding

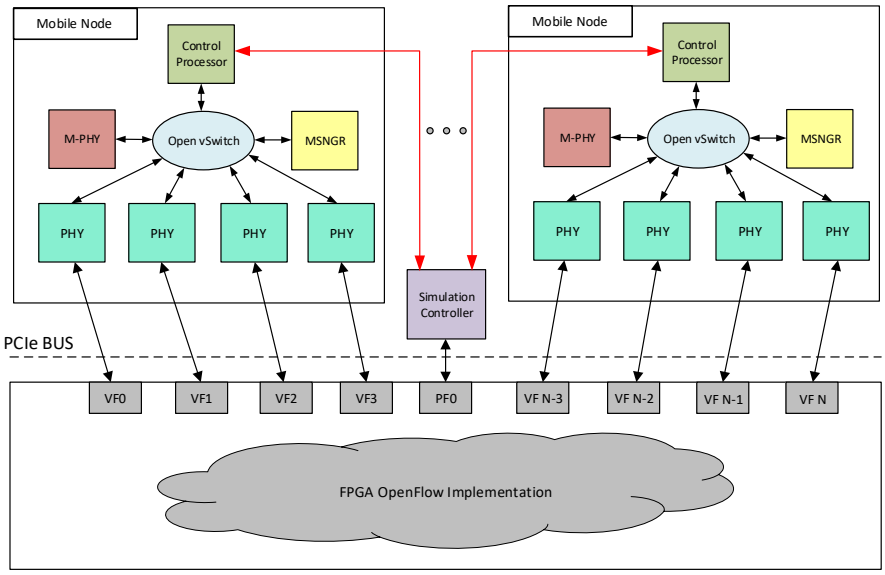


Figure 5.3: PCIe SR-IOV Accelerator

VF driver. The driver will then receive this packet through another MSIx interrupt and then transfer the data to the receiving PHY component's socket connection. This abstraction allows the emulation environment to operate and function in the same manner as the virtual switching solution.

The Simulation Controller can directly program the OpenFlow data plane through kernel-level drivers and basic user-space API calls on the host machine. The role of the simulation controller is to manage which mobile nodes can communicate with each other during a simulation as visibility changes as a function of time. A Linux character driver provides these capabilities, allowing user-space programs to communicate to the hardware devices on-chip memory through the kernel's file system. In addition, the abstraction allows flows to be programmed in the hardware using file I/O, which fits nicely into the existing emulation environment with few modifications. Unfortunately, the programming channel in out system is not compliant with commercial OpenFlow controllers since communication is not facilitated through TCP with SSL.

5.3 Linux Device Driver

A device driver acts as an interface between system-level software and the physical hardware data plane, such as a NIC or Graphics Processing Unit (GPU). Unlike userspace programs, device drivers can be dynamically loaded at run-time or during program execution using the modprobe utility [2]. The device driver allows system software developers to work at an abstract level without understanding the device's physical hardware and associated architecture. The kernel driver is the only piece of code that should interact directly with the hardware register interfaces to guarantee the correctness and atomic execution. There are three main types of device drivers: character, block, and network interfaces [9]. The virtual network accelerator design predominantly uses network drivers to interact with the Linux network stack. Thus, we will cover the operations of these types of interfaces exhaustively in the following sections.

5.3.1 Network Drivers and PCIe Support

In this system, network drivers are used to model the operation of a NIC. A network device driver creates an interface on the host machine that can be assigned an IP address, MAC address, and various packet processing options and protocols. The device driver will dictate how asynchronous events are handled, such as command-line utilities, packet receiving, and packet transmitting. The following initialization steps occur to set up a network device to transmit and receive packets over PCIe.

- Probe PCIe bus components for device ID, and function number. Allocate memory regions for each base address register configured by the hardware.
- Attach interrupts to PCIe hardware functions and provide associated event callbacks.
- Initialize network interface with name, MAC, IP, queue structures, and setup function callbacks for asynchronous events.

- Allocate DMA memory address regions and allocate ring-buffers for transmit and receive descriptors.

The SR-IOV specification allows for a different device driver to be attached to each VF or PF. The device driver attached to the PF will enable SR-IOV capabilities and will receive messages through a character driver to manage the hardware data plane. The PF will be written to and read from similar to how a file operates, through a stream of bytes. Each VF will be attached to a network device driver to facilitate socket communication from the upper-level network stack. The network driver provides the necessary abstraction to integrate hardware into a network infrastructure that spans the OSI model. These mechanisms allow less software intervention during packet transfer from source to destination allowing a reduction of CPU utilization typically leading to an increase in performance.

5.3.2 Packet Representation in the Linux Kernel

The Linux kernel provides many utilities that allow device driver developers to interact with the networking subsystem through hardware devices. These facilities are particularly beneficial for a NIC. Still, they are also used in numerous other applications such as hypervisors, virtual switching solutions, Docker, and packet sniffing utilities such as tcpdump and Wireshark. When a higher-level protocol such as TCP, UDP, or ICMP wants to send a packet over a virtual or physical interface, the kernel wraps the data in a socket buffer, which is the primitive data structure used in the networking subsystem. The name was derived from the convention of sockets, which are used to initiate and transmit data in user-space applications. The following is a partial definition of the socket buffer (sk_buff) structure from the Linux kernel source tree. The image highlights the memory regions associated with each field.

The most important fields of this structure definition are the core management of the packet data area. It is important to note that these data areas are allocated

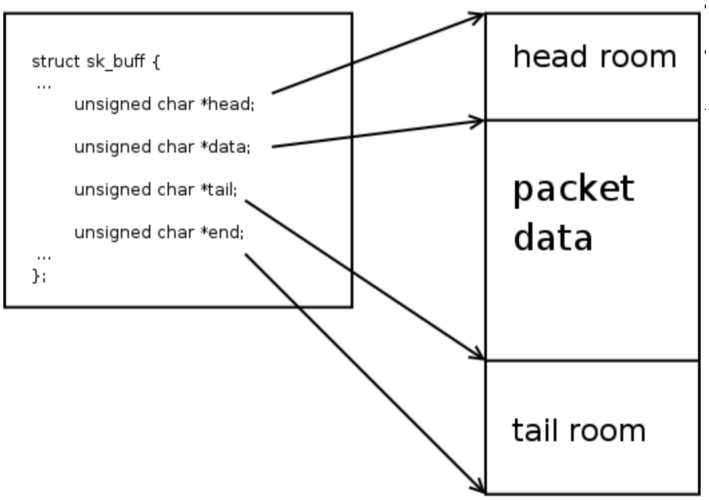


Figure 5.4: Socket Buffer Kernel Structure

contiguously as to make packet transmission and reception easier. There are ways to circumvent this action for system performance, which will be discussed in the future work (Chapter 7). The rest of the declaration is omitted for simplicity. The main memory management implementation of a sk_buff includes a total of four pointers. The four pointers include: head, data, tail, and end.

The head points to the beginning of the structure and usually is allocated in memory for alignment purposes. This area, known as the head room, is used to make sure IP level headers are aligned appropriately for kernel processing of network packets. When dealing with IPV4 packets, the headroom offset is typically 2 bytes. Unfortunately this causes DMA operations to be unaligned, which can incur additional performance penalties.

The data pointer indicates where the actual packet starts. This information includes L2, L3, and L4 protocol headers along with the application data that is passed through the network layer. When transferring packets to hardware, this pointer is provided to fetch data for proper classification and processing or to transmit over physical ethernet. The tail pointer indicates where the packet ends. Therefore, the

difference between the tail and header pointer is also defined as the packet length represented in bytes. The tail pointer also indicates the start of the tail room, which is again used for alignment purposes. The size of this segment is the difference between the tail and end pointer.

5.3.3 Packet Lifecycle in the Linux Kernel

A Linux network interface supports transferring and receiving packets abstractly. The device driver provides this abstraction and mitigates the need for the network stack to be involved in the underlying data transfer. When the network packet is received on an interface, the networking subsystem releases control of the data structure and passes it off the kernel driver associated with that interface. Once it releases control, the networking stack trusts that it will be delivered successfully to its final destination. The receiving end is the exact opposite; the networking subsystem has no context of where this packet came from and has enough information to process it successfully. This section will discuss the life cycle of a networking packet in the context of a hardware data plane and the hand-offs between hardware and software. Figure 5.5 demonstrates the procedures involved in packet transfer to and from software onto a hardware data plane. In this scenario, the packet processing hardware acts as a loopback interface. Data is forwarded from the RX channel to the TX channel of the same network interface. In a practical networking scenario, this is not common but is used for illustrative purposes to highlight the events that occur in packet transfer to and from hardware and their relationship in time. The red text demonstrates packet processing events, while the bold text indicates where the event is occurring. In a virtualized SR-IOV environment, there exists N network stacks and device drivers but only one physical hardware device.

The life cycle of a packet is split up into hardware and software control. The TX and RX ring descriptors are shared between both hardware and software and partially

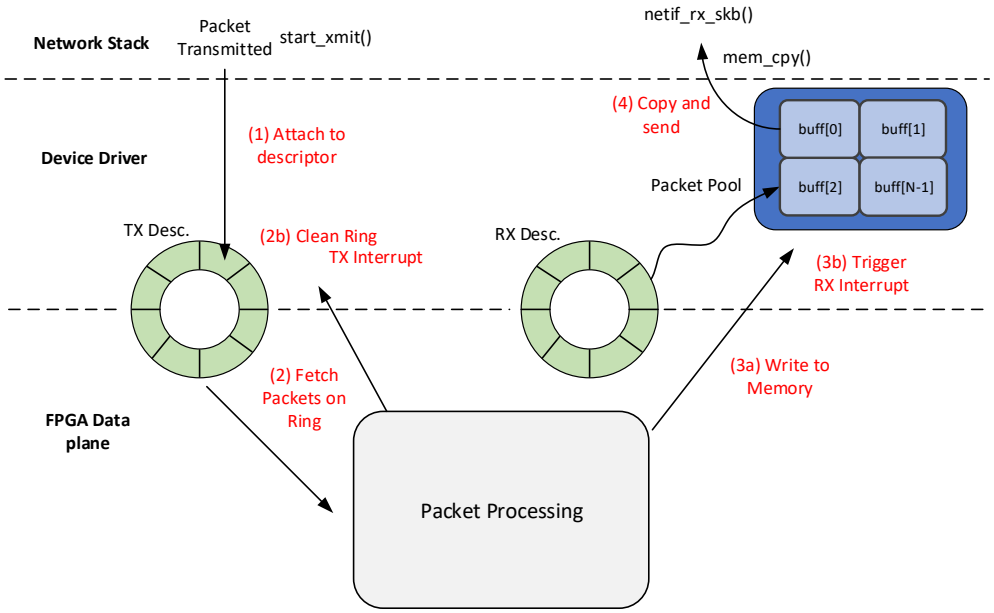


Figure 5.5: Linux Kernel Packet Lifecycle HW/SW

exist in both control planes. The descriptors are a circular type buffer with a head and tail pointer. The head and tail can only be incremented and will wrap after all locations are exhausted. The rings are shared with hardware using DMA coherent memory regions, which are consistent between HW and SW regardless of the state of the cache. The HW will only write to the head, and the SW will only write to the tail, so race conditions are avoided. This allows the software to know how much work has been completed in HW and vice-versa.

The network subsystem is represented simplistically by the appropriate API given to device drivers. When a packet is transmitted, internally the $start\_xmit()$ function is called. When a packet is received, the device driver hands off the packet through $netif\_rx\_skb()$. The hooks to the networking subsystem provide the proper abstraction for the integration of application-specific hardware. The packet processing engine can implement any network behavior without affecting any functionality up the stack.

The left side of the diagram consists mainly of the TX side of the network interface, while the right shows the RX side. When the network stack wants to send a packet, an asynchronous call is performed in the device driver and is passed a socket buffer. The driver will create a TX descriptor entry and notify the hardware of a new packet. The packet notification process is called a doorbell and is an expensive process. The packet processing element will then fetch an updated copy of the TX descriptor through DMA. When the HW receives the descriptor, it will fetch all available packets on the ring. When the packet is received, a TX interrupt will be triggered to allow the ring to be cleaned and deallocated. In this scenario, the packet processor will forward the traffic back out on the same interface. The packet processor will consume an RX descriptor entry and write the packet back into RAM at the address pointed to by the entry. After the DMA operation is complete, an RX interrupt will be asserted, notifying the driver that a packet is available. The driver will then copy the packet out of the pool into an intermediate buffer and will be handed off to the network stack. The numbers in the diagram indicate the order in which these events happen.

5.3.4 NAPI compliant drivers

The New Application Programming Interface (NAPI) is a capability of the network stack to provide a hybrid mode of operation for high bandwidth network interfaces. The hybrid approach uses both polling and interrupts to help efficiently handle bursty network traffic on an interface. When a packet is received in the driver, the software turns off interrupts and switches to a polling-based approach. This is known as interrupt coalescing, which allows the user to hold back hardware interrupts until a certain amount of traffic is consumed. A NAPI-compliant driver can specify the number of packets allowed to be processed before interrupts are re-enabled. The amount of work is known as the weight of the NAPI interface, and various values can be used depending on the traffic pattern. Typical values set in high-performance

networking range from 16 to 64. These weights are highly dependant upon the internal buffer space for TX/RX packet descriptors and the MTU for the given interface. Section 5.4.4 will discuss the API provided to the device driver to support NAPI compliance in hardware.

5.4 PCIe Hardware Implementation

5.4.1 PCIe Specification Overview

PCIe is a complex switching network that allows data to be transferred between endpoints and the root complex. The root complex device of a PCIe system connects the CPU and memory subsystem with the PCIe switch fabric. The switch fabric can be interconnected with one or more downstream switches. Each switch has its own flow control and forwards messages depending on the packet destination. A PCIe endpoint can be located downstream from multiple switches; therefore, it has variable latency for all operations. Each endpoint can be configured to try to enforce certain latency constraints but can not always be guaranteed. Figure 5.6 shows a sample PCIe subsystem with multiple endpoint devices and their relationship to the root complex. In our system, the FPGA resides on an L0 link attached directly to the root complex. This allows for the minimization of latency, which is critical in networked applications when input data is sparse.

Each endpoint device must follow a specific set of rules to not corrupt memory and cause fatal kernel exceptions. Mainly, a PCIe endpoint must respond to both memory reads and writes. In the case of a memory write, the endpoint will typically accept the data and store it within on-chip memory. These requests are known as posted requests because they do not require a response and are the simplest type of PCIe transaction. In some cases, the endpoint can send an error message in response to one of these requests, which is also considered a posted transaction. The other type of PCIe transaction is a non-posted request where the requester expects a reply or

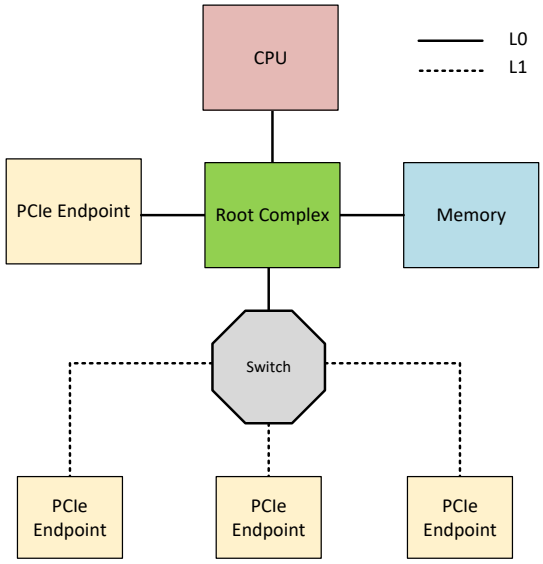


Figure 5.6: PCI Express Interconnect Network

acknowledgment. An essential non-posted transaction in any DMA capable endpoint is the read request. When an endpoint or root complex receives a read request, it must respond with a completion packet. The completion packet contains the requested data starting at the specified address with metadata indicating the returned data format. If the requester does not receive a completion in a configurable time frame, a timeout occurs, which can cause fatal exceptions depending on the context of the data transfer.

Two parameters affect the performance of an endpoint device when communicated directly to the root complex of the CPU and memory subsystem. The first is the Maximum Payload Size (MPS) and the second is the Maximum Read Request Size (MRRS). The MPS is a negotiated parameter that is managed by the operating system based on the smallest supported payload size of any path in the PCIe switching network [29]. The MRRS is the largest amount of data that can be read in a single transaction from an endpoint device. The MRRS is typically larger than the MPS but is not required to be so based on the PCIe base specification. The theoretical maximum achievable bandwidth on PCIe 3.0 with eight lanes is 64 Gbps which does not

consider the overhead associated with the protocol. The percentage of the theoretical bandwidth can be approximated by amortizing the cost of the protocol overhead. As the MPS of the PCIe network increases, so does the achievable bandwidth. Figure 5.7 shows the achievable bandwidth as a function of the MPS with both a 32b and 64b addressing scheme. These results take into account the overhead associated with PCIe Transaction Level Payload (TLP). The transaction level of PCIe provides data in packets with a protocol header indicating the type of transaction, alignment, and size.

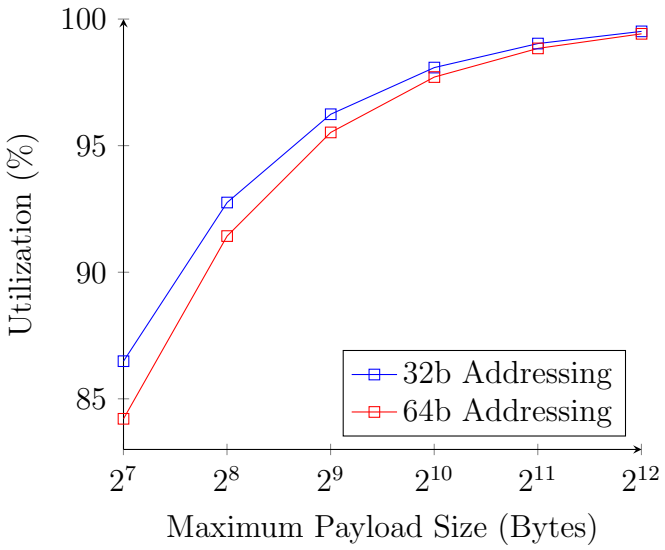
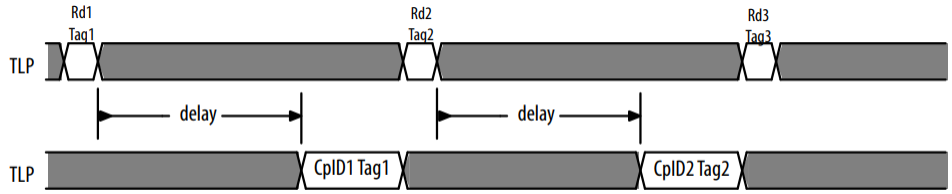


Figure 5.7: Theoretical PCIe 3.0 Bandwidth

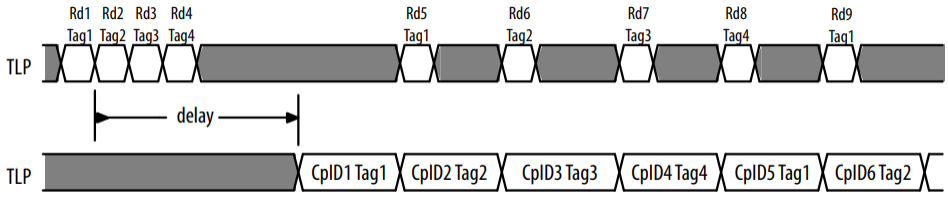
5.4.2 Read Requests for High Bandwidth Applications

PCIe endpoints are capable of achieving high bandwidth by performing DMA using non-posted read requests. This allows data transfer without CPU intervention. Due to the structure of the switching network, a read request can incur high latency. The root complex must first receive a memory read request and then form the response and send it to the requester. The process enforces a message to traverse the switching network from source to destination and then again from destination to

source. Additionally, each message may need to traverse multiple switches between the root complex and endpoint depending on the configuration. The high latency can be mitigated by sending pipeline read requests. In other words, requests are sent continually without waiting for the completion data. Figure 5.8 shows an example waveform comparing both a non pipelined and pipelined version of data transfer initiated by an endpoint.



(a) Low Bandwidth



(b) High Bandwidth

Figure 5.8: PCIe Read Request Comparison [20]

The top waveform shows the non-pipelined request transfer where only one request is initiated at a time. This configuration yields low bandwidth because the latency penalty is incurred on each payload transfer. The bottom waveform shows the pipelined approach where N number of transactions are sent before receiving any data. The second approach yields high bandwidth because the latency is only observed on the first transfer request. The second approach is the preferred method but is not as trivial to implement. Modern PCIe IPs support 256 unique outgoing requests. However, in practice, most DMA engines don't allow this many due to design difficulties.

When a PCIe endpoint initiates multiple outstanding read requests, the endpoint must keep track of them all using a unique completion ID. The PCIe root complex can return these completion requests in segments that span multiple transaction-level packets. Each segment of a single read request must be returned in increasing address order, but the order of the returned completion packets across multiple requests is not guaranteed. Therefore, read request $i + 1$ may be returned before read request i enforcing additional buffer space and control logic to handle out-of-order completions. Our approach uses pipelined read requests with a 32 entry windowing mechanism. The windowing mechanism limits the number of requests allowed before a response is received. The larger the windowing mechanism, the more difficult timing closure is. Section 5.4.3.1 will discuss the detailed design to handle the aforementioned challenges.

5.4.3 PCIe Endpoint Architecture

The main components of the architecture consist of an OpenFlow capable switch, a receive (RX) DMA engine, a transmit (TX) DMA engine, and a descriptor table. The RX engine is involved with transferring data from the host PC onto the FPGA so that the network processor can interpret and classify packets. The TX engine is then responsible for transferring the new data to the correct location in memory based on the classification result of the OpenFlow switch. Figure 5.9 demonstrates the notional architecture for the PCIe endpoint and the functional relationship of all block modules.

The endpoint configuration API acts as local storage for the entire device. The block module is implemented using on-chip memory and is partitioned to hold all registers for both VFs and PFs. When the endpoint receives a memory write in the PCIe TLP format, the request is converted into a byte addressable memory write that is compliant with the Avalon MM specification. The module also monitors specific

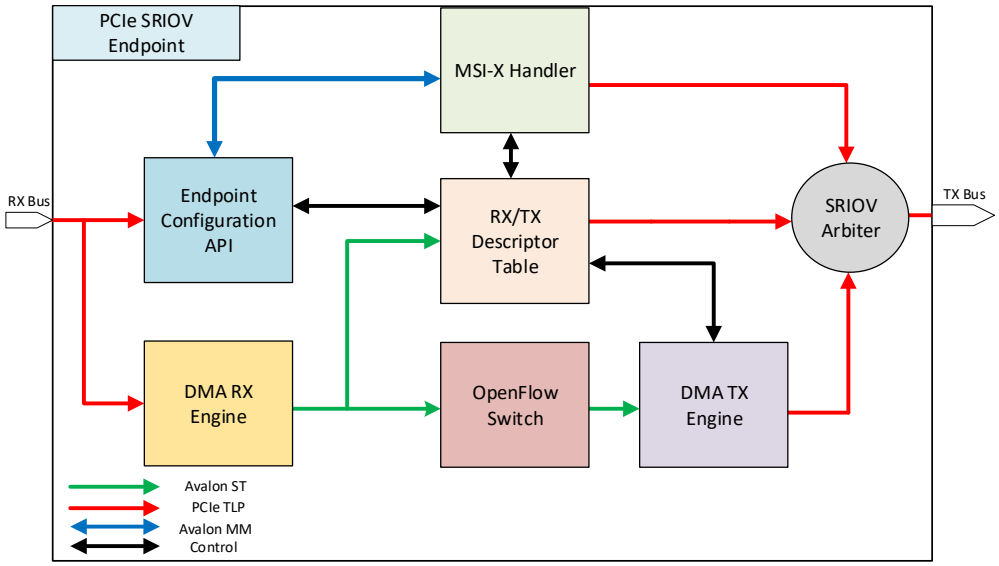


Figure 5.9: PCI SRIOV Architecture

addresses for DMA and OpenFlow programming operations. When either these occur, the module will configure downstream logic for correct operation.

When a packet arrives on the RX data bus, the receive engine strips all header information along with any invalid data that may exist depending on the return data format. The OpenFlow capable switch operates on streams of contiguous data, but the bus returns data in a PCIe TLP format. Therefore, the RX engine must convert between these formats for correct operation. The PCIe TLP format is provided in a stream interface with a protocol header indicating addressing and alignment information. Once the OpenFlow switch classifies a packet, the TX engine will convert from the stream format back to the PCIe TLP format for the TX bus. The descriptor table manages the memory locations for both the receive and transmit engine, a shared module between the two. The transmit descriptor is filled when the FPGA driver is loaded, and the receive descriptor is fetched when a packet needs to be sent. The descriptor table logic issues read requests in response to received RX descriptors. The number of outgoing packets for a descriptor is equal to the absolute value of the

$tail - head$ pointers. The descriptor setup allows the HW to bulk transfer packets to increase efficiency and decrease latency caused by multiple descriptor fetches. For example, if only one packet is transferred for each descriptor fetch, then the effective fetches per payload is two. Additionally, each packet would require reaching over the bus four times which would incur high latency penalties.

The data plane also tracks receive and transmit events. When an event occurs, the host software is notified that all outstanding packets have been received or that a packet is available for the network stack. This is done through the use of MSI-X interrupts. When an event is detected, a message is sent through the MSI-X handler. The MSI-X handler reads a unique memory location from the endpoint configuration API that contains a unique address and data word that the device driver negotiates. The data word is then written into that memory address over the bus, which triggers an interrupt. Section 5.4.4 will discuss this process more in-depth, as well as the features in use to make our device NAPI compliant.

Finally, an SR-IOV arbitration mechanism is used to share access to the transmit path. This module arbitrates control between multiple queuing structures while keeping track of the VF or PF requesting access. The requesting entity must be provided to the bus in a specific format for correct interrupt and memory translation on the host software.

5.4.3.1 Receive Engine

The receive portion of a DMA engine is more complicated than the the transmit since an endpoint can have multiple outstanding read requests at any given time. Therefore, the endpoint must track and manage out of order completions and provide an intermediate buffer mechanism. Downstream network logic only supports serial data streams which enforces data to be aligned correctly on the fundamental data boundary of the bus width which is 32 bytes. Thus, control logic must handle all

potential alignment scenarios of the completion data that is returned by the root complex. Figure 5.10 demonstrates the data path that is employed on the receive engine and the abstracted control logic dealing with alignment management.

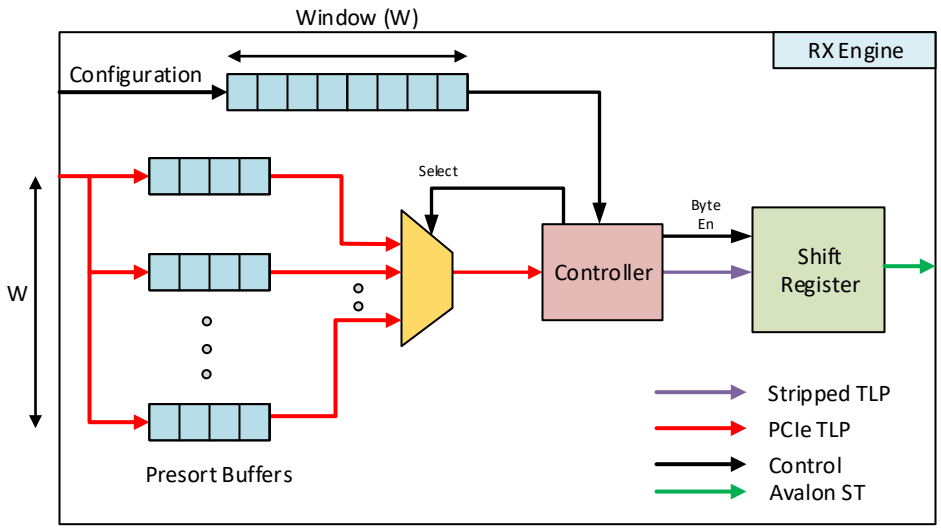


Figure 5.10: Receive DMA Engine

The input to the receive engine consists of W FIFO interfaces to help presort data based on the unique ID tag provided by the PCIe bus. This allows us to serialize the data into the order that it was requested. The command FIFO buffers read request metadata which is used in the serialization process. Each entry represents a packet segment with the corresponding tag that needs to be processed first. The output is used to configure the muxing logic so that our control circuit can interpret alignment information from the correct presort buffer. Each presort buffer holds the PCIe TLP header data to extract all valid data from the 32-byte bus. The control logic generates a byte enable signal that is used to write to a shift register. The shift register filters out all invalid bytes so that data can be packed correctly for the OpenFlow switch. When data is returned out of order by the PCIe bus, the presort buffers will hold data until the corresponding command FIFO entry is reached. Thus, in the worst-case,

each presort buffer must have enough space to store the MRRS supported by the endpoint. The current implementation supports a MRRS of 512B, which enforces a worst-case buffer size of 24 elements based on a 64B read completion boundary (RCB). The RCB is the root complex's memory alignment requirement when returning data to an endpoint.

The number of presort buffers is referred to as the pipeline window. The window is the amount of non-posted read requests initiated by the endpoint before a response is received. To manage outgoing tag IDs, we use a tag array that acts as a circular buffer. When all available tags are used, the tag array will apply back pressure to the DMA engine until completions are returned. Since we enforce the order of completions, tags are reallocated in the order they are used. The tag array always keeps track of the following tag that is going to be used. When all tags are exhausted, the tag array will default to reallocating tags as soon as a completion is returned through a FIFO interface.

5.4.4 System Memory and Programming Models

PCIe SR-IOV uses Intel Virtualization Technology for Directed I/O (VT-d) and the Input-output Memory Management Unit (IOMMU) to allow a single hardware device to appear as multiple. These technologies help map memory spaces and interrupts between virtual and physical hardware devices so that traffic can be forwarded to its corresponding location with the least amount of software intervention possible. In our system, the FPGA handles network operations for N virtual network interfaces through one PF. You can think of each physical function as its own type of acceleration infrastructure that's virtualized on the FPGA. In this work, the PF is used for the OpenFlow switch. Each VF and PF has its own memory space and registers within the fabric. The entire memory space is partitioned into $VF\ count + PF\ count$ segments, each with a 64-byte address space aligned on 32-byte boundaries. Figure

5.11 shows the address space along with the API that is used to interface with the accelerator solution. Each VF has the same address space. In each partition, only 96 bytes are shown, but there is an extra 32 bytes in physical hardware to make it aligned on a power of two. The specific alignment allows the memory to be indexed based on an integer representation of the PF or VF requesting access.

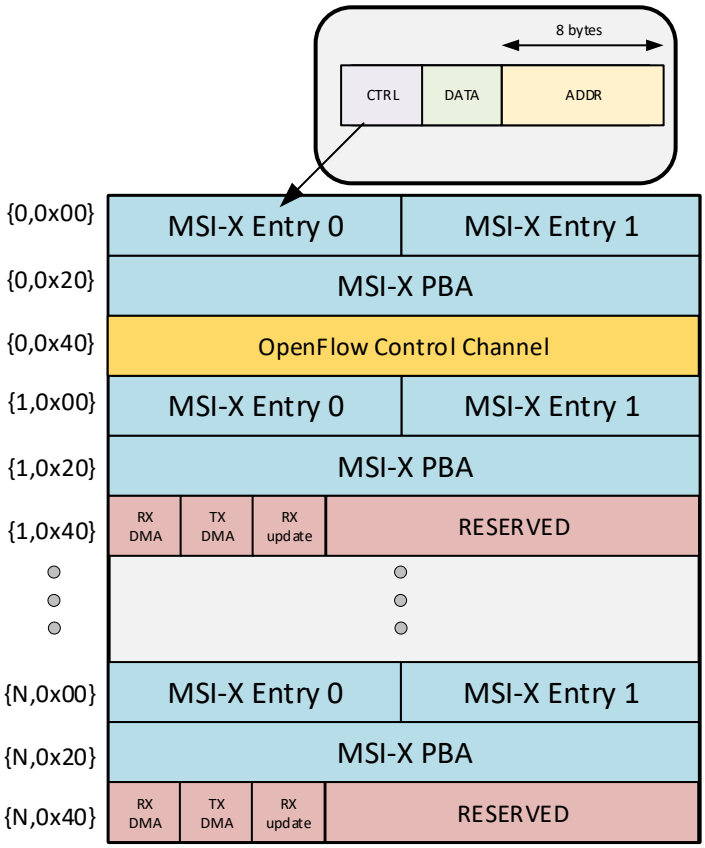


Figure 5.11: PCIe SR-IOV Address Space

The first partition is allocated to the PF, and is used as a control channel to the OpenFlow switch. Each section afterward is assigned to VFs in increasing order. Each VF has address space allocated for interaction with a DMA engine. When the network stack wants to send a packet, it writes to the 32-bit word associated with the transmit path. The base address of the packet descriptor is embedded in this memory

write so that the hardware can fetch it. Once the hardware receives the descriptor, all valid packets will be translated from host memory to the FPGA. The API for the receive path is only used at driver initialization time. The same procedure applies, except the descriptors are stored on the FPGA using on-chip memory and reused for the duration of the operation. The TX DMA engine utilizes these descriptor entries to write into system memory when a packet has a valid flow entry in software.

Both VFs and PFs have address space allocated for MSI-X generation. The first 32 bytes of each partition represent the MSI-X table, and the next 32 bytes represent the Pending Bit Array (PBA). The PBA is used to help negotiate between the Host SW and the HW so that interrupts are not lost or missed. The PBA is represented in a one-hot encoding, where each bit represents one vector in the MSI-X table. The MSI-X table consists of 16 bytes for each vector, where each vector is a unique interrupt generation source. Each vector has a separate address, data value, and associated interrupt callback in the host SW. When hardware wants to initiate an interrupt, it reads the address and data from the corresponding memory region and transmits a posted write request over the PCIe bus. The host SW will detect this event and dispatch an ISR to handle the event. In this system, interrupts are only used to notify of a packet arrival or departure. The upper 4 bytes of an MSI-X entry enable or disable a particular interrupt vector through its control field. This is necessary to support NAPI-compliant interfaces. When the device driver receives a packet, interrupts must be disabled so that polling can be used instead.

CHAPTER 6

EVALUATION OF PCIE SR-IOV VIRTUAL NETWORK ACCELERATOR

In this chapter, we will discuss the performance of the FPGA OpenFlow switch in various scenarios. First and foremost, the design will be evaluated on raw bandwidth capabilities for OpenFlow matching, classification, and forwarding to and from host memory. These tests will show the hardware transfer capabilities for both RX and TX DMA to and from containerized applications. We will then analyze how much of this bandwidth can be utilized by network protocols such as TCP/IP and UDP. The sending and receiving rates are regulated by the Linux networking stack based on packet size, congestion, internal queuing, and CPU utilization. Often these protocols cannot achieve the maximum bandwidth allowed by the underlying hardware. Afterward, we evaluate the hardware implementation with the existing Mobile Adhoc network emulation environment and compare results to those based on pure software approaches. Finally, we will evaluate application-specific acceleration of course grain computation on the edge of the network communication to reduce I/O bottlenecks.

6.1 Performance for Containerized Applications

6.1.1 Bandwidth Capabilities of Direct Memory Access

In this section, we look at the memory transfer speed between two VFs to help quantify our DMA performance capabilities. These two VFs act as separate I/O devices that can be passed through to separate Docker containers. These tests circumvent the Linux network stack to send packets to the FPGA as fast as possible

directly. This allows us to quantify the total transfer rate supported by the hardware without worrying about protocol-specific abstractions. In this scenario, packets are fetched from the hardware, classified by the OpenFlow protocol, and transferred back to host memory in rapid succession. Figure 6.1 shows the total bandwidth for increasing transfer sizes up to the maximum of 1MByte. The maximum transfer size is dictated by the maximum TCP/IP segment and the number of entries we allow on our receive descriptor. The receive descriptor has 16 entries, and the maximum supported TCP/IP segment is 64KB

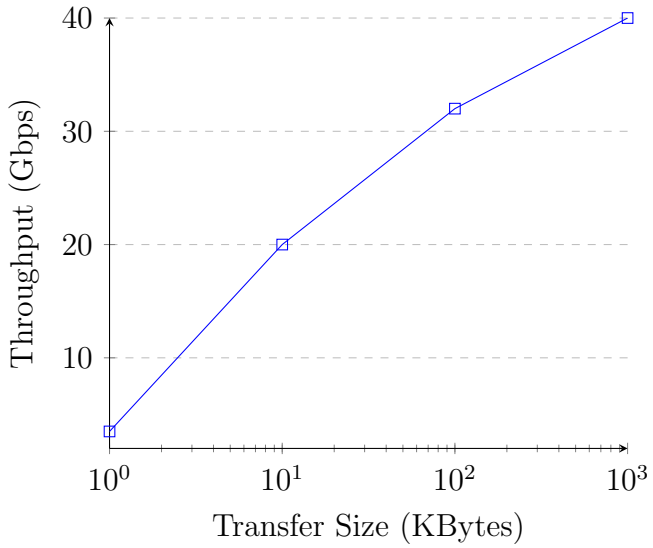


Figure 6.1: DMA Throughput for OpenFlow

The above results show the effect of a hardware PCIe endpoint. When the transfer size is small, the throughput is also low, meaning the hardware is underutilized. It is necessary to make two separate DMA transfers for a single payload, one for the descriptor and one for the packet. Thus, it is necessary to reach over the bus on two different occasions and incur four latency penalties since each read request requires two PCIe switch traversals. The lowest packet transfer of 1KB achieved 3 Gbps, while the highest of 1MB achieved 40 Gbps. Most of the performance degradation of small transfer sizes is due to the latency involved in a single transfer. In our design, we

can only achieve 62.5 percent of the maximum allowed bandwidth of PCIe 3.0. The lower utilization is attributed to the overhead associated with the TLP, flow control tracking, and external back pressure provided by the application. The backpressure limits our transmission speed based on the number of credits a link partner can accept from our design. If the root complex or downstream switch cannot process TLPs as fast as provided, credits will inevitably run out, causing performance degradation. A credit is a fundamental unit of data based PCIe flow control used to determine a link partners available buffer space. In the current implementation, a credit represents 4 bytes of data.

6.1.2 Performance Quantification of Network Communication

The emulation environment discussed in Chapter 2 uses the external OpenFlow switch to facilitate communication between two mobile nodes using the TCP/IP protocol. Thus, TCP/IP will be the primary evaluation criteria for comparison between the SW and HW approaches. This section will provide an in-depth comparison between the SW and HW approaches and their respective bandwidth and latency measurements with various traffic patterns. In this scenario, we create two mobile nodes connected through both mediums; the OVS instance and the FPGA. The bandwidth is calculated using the iperf utility [12], while the latency is calculated through the ping utility [4]. The cubic congestion control algorithm [13] is used in both cases, which is the default on newer Linux kernel distributions. Figure 6.2 demonstrates the bandwidth and latency measurements of both switching solutions using a single core of the processor. All results are generated by using the maximum allowed packet size of 64KB. We show the total bandwidth of two separate processors running OVS, with different clock speeds. The 2GHZ processor is a Intel Xeon, while the 4GHZ is an Intel i7. The FPGA bandwidth is the same on either processor because it is a

bottleneck between memory transfers. We will discuss memory-bound computation limitations at the end of this chapter.

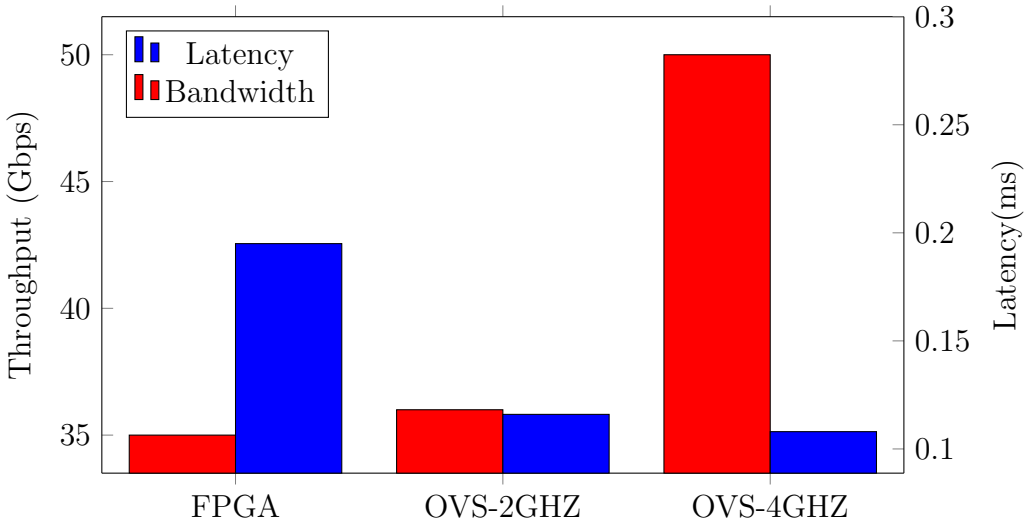


Figure 6.2: Performance Comparison between OVS and FPGA

The results show that the FPGA transfer speed can keep up with the virtual switching solution when using large multi-core processors such as those on a server. However, when using fast processors with a small number of cores, the transfer rate is much faster. When OVS transfers large packet sizes, it is highly efficient, and higher clock speed processors can move memory quicker. The ping results show similar discrepancies. When a packet is processed on the FPGA, the hardware must fetch the packet and a descriptor resulting in a longer delay between packet processing events. In ping, packets are not sent quickly enough to mask the latency, resulting in a higher latency in most cases than the pure software approach.

OVS has highly efficient bandwidth when transferring large payloads using cached OpenFlow actions but tends to be slower when using smaller payload sizes. Using smaller packet sizes, the FPGA has a much closer comparable bandwidth to the OVS instance running at 4GHZ. If we go one step further and offload fragmentation to the FPGA, we see that the FPGA is more efficient. Fragmentation in this context is when

the CPU must break up large IP packets into fragments to pass through links with a smaller maximum transmit unit. Figure 6.3 demonstrates the transfer bandwidth as a function of the packet size for both virtual and physical switching solutions in both the fragmented and non-fragmented cases.

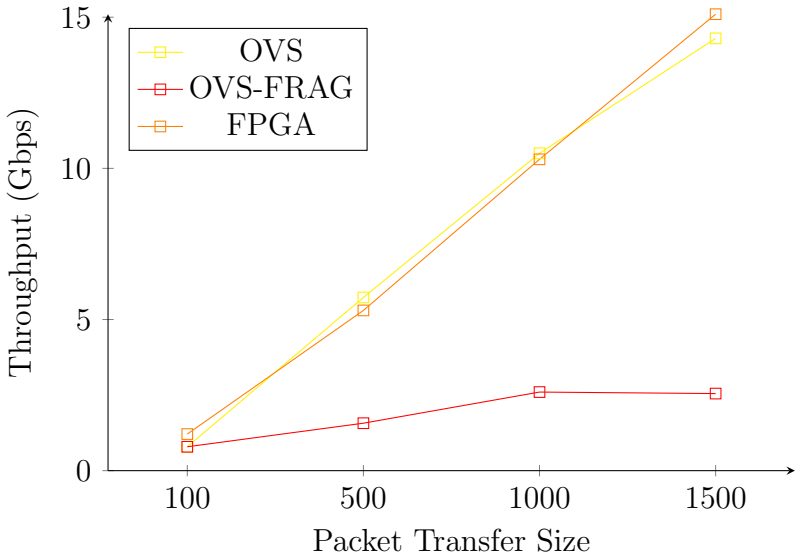


Figure 6.3: Small Payload Bandwidth Comparison between Mobile Nodes

The data demonstrates that the FPGA performs similar to OVS, with no fragmentation performed in software. Offloading the packet to hardware maintains bandwidth well and, in some cases, exceeds it, meaning that latency is not too substantial. For example, in the case of 100 byte packet transfers, the FPGA exceeds OVS by 53% with a bandwidth of 1.21 vs. 0.79 Gbps, respectively. When performing fragmentation in software, the performance significantly decreases, but the FPGA can maintain line rate since the FPGA can bulk transfer up the theoretical TCP/IP stream limit. The results highlight the benefit of an FPGA when enough CPU-intensive processing is offloaded. Course grain computation offloading will be further explored in Section 6.3 with a virtual scenario that requires parallel DSP processing.

6.2 MANET Emulation HW/SW Performance

In this section, we evaluate the performance of the FPGA in the MANET emulation system. We analyze two scenarios: one with the OVS instance and the other with the FPGA. The environment will consist of six Mobile Nodes, each with one PHY device that supports three concurrent connections. We simulate each scenario using 64KB image data as the transmission payload to help quantify the capable bandwidth. The bandwidth of each stream will be measured separately and cumulatively. The cumulative bandwidth will help compare the two switching solution's network capacity with all CPU cores in full utilization. Figure 6.4 shows the respective bandwidth of each traffic stream along with their source and destination. The legend signifies the comparisons between virtual switching and physical switching scenarios. The data is average across five different test runs.

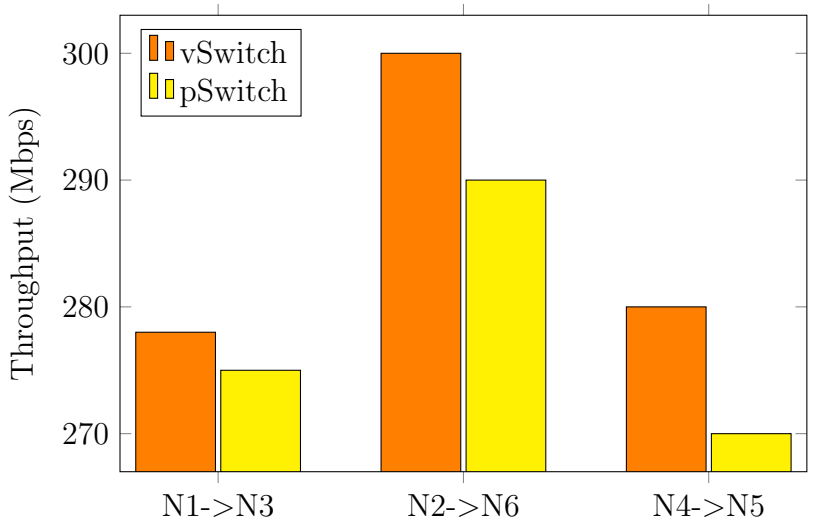


Figure 6.4: MANET Emulation FPGA vs OVS

This scenario demonstrates that even with significant CPU overhead, the virtual switching solution still has higher bandwidth in the average case. However, the FPGA can sometimes match SW performance due to the non-deterministic nature of the PCIe bus latency. For example, if the latency is low over a specific test run, then

the performance can be closer to SW. The CPU, on the other hand, is much more deterministic and gets similar results across multiple runs making it slightly outperform the FPGA. The latency overhead for offloading, in this case, is not justified due to the low amount of computation being migrated to HW.

Open vSwitch uses a very efficient first-level caching mechanism called a microflow [30]. The microflow provides $O(1)$ time packet lookup when matching to an OpenFlow entry. When a packet identifier exists in the microflow, the kernel processing module performs the cached OpenFlow actions. This process is entirely handled in the kernel, making it very efficient and is known as the fast path of OVS. On the other hand, the slow path of OVS is handled in userspace, which incurs high latency penalties for packet classification and processing. The slow path is when OVS consults its OpenFlow pipeline, which has many CPU-intensive operations to handle robust classification and matching. The OVS OpenFlow pipeline needs to be consulted whenever a packet is received which is not present in the cache.

The microflow cache makes packet classification cheap for long-lived transport-level connections, such as TCP and UDP. However, when flows or socket connections are changed frequently over a short period, the microflow cache has a high miss rate, making OVS default to the OpenFlow pipeline to decide a packet's corresponding action. In these tests, we will test the efficiency of both solutions when using short-lived TCP/IP sockets. In each case, we will connect a socket and send a payload size of N bytes of data and then repeat the process over one second. The same test procedure is also performed for sending two packets per TCP/IP connection. The number of packets per second is tabulated to measure the cost of an OpenFlow table lookup in software vs. the FPGA. This is analogous to frequently changing connections between mobile nodes in our emulation environment, which will happen if nodes move quickly over a short period based on our provided visibility information. For example, a UAV swarm communication network has fast movement between mobile nodes, whereas

a satellite network is fixed for long periods. Table 6.1 shows the results from the repeated TCP/IP connection test with both one and two packets per connection at varying payload lengths. The entry in each cell represents the number of packets per second in thousands.

One	SW	HW	Increase (%)
100B	6.0	19.6	226
1000B	5.8	18.2	213
10000B	5.6	16.8	200

Two	SW	HW	Increase (%)
100B	11.4	23.8	108
1000B	11.1	23.5	111
10000B	9.4	20.8	121

Table 6.1: Repeated TCP/IP Connections (Kpps)

In both cases, we use a single OpenFlow table inside of the pipeline for fair comparisons. The latency of SW-based classification grows linearly as a function of the depth of tables within the pipeline. However, adding additional tables in HW would only add a few more cycles of latency in the nanosecond scale, assuming a 250Mhz clock. Thus, the relative speedup would grow significantly as the OpenFlow pipeline deepens and becomes more complex.

The results in Table 6.1 demonstrate that the HW solution is quicker when using short-lived transport connections. Mainly when using TCP/IP due to the increased overhead and 3-way handshake. The table on the left has a 66.67% microflow cache miss rate, and the table on the right has a 50% miss rate. This includes the syn and ack packets required for the TCP/IP handshake. When the miss rate is higher, the HW performs better compared to the SW approach by 2-3x. The HW approach has no concept of a cache miss rate because every packet must traverse the hardware, which is not always ideal. The future work section in Chapter 7 will address solutions to this issue to make the system more efficient.

Table 6.1 test results show that the FPGA-based OpenFlow pipeline is more efficient than the software. Therefore, traffic patterns that traverse the OVS slow path are more efficient on the FPGA and increases as a function of the pipeline

complexity. However, in most cases, where the cache miss rate is low, the SW will perform better because computation on the CPU is minimal. Due to the minimal CPU computation of the OVS microflow cache, it is difficult to justify sending the packet to the FPGA and back just for a simplistic lookup table and corresponding forwarding action when actions are cached.

6.3 Edge Application Specific Acceleration

A virtual scenario is constructed using the Mobile Adhoc emulation infrastructure discussed in previous chapters to demonstrate the benefits of edge-specific acceleration. In this scenario, two mobile nodes communicate, simulating UAVs in a swarm communication setting. The nodes' transmit audio data is used to make decisions on current flight paths. The audio data is sampled at high fidelity and contains noise that needs to be filtered to sense and actuate correctly. The simulated mobile nodes are connected through the network substrate.

In the first scenario, the filter computation is performed in SW within the mobile node. The filter operation is performed on the data first and then sent over the network to the other node. In the second scenario, the filter computation is performed on the FPGA on the edge of the network using a pipelined parallel approach. In both cases, a 4-tap FIR filter is utilized with an arbitrary impulse response. The following equation is used to realized convolution over the finite impulse response $H[k]$. The impulse response can be chosen to implement any filter and is irrelevant to the performance results of our experimentation.

$$X(conv) = \sum_{n=0}^{N-1} \sum_{k=0}^{3} X[n] * H[n-k]$$

The input $X[n]$ represents the audio stream and each entry within the stream is represented as a 32-bit fixed point integer with an amplitude ranging from 0 to 1. In

the physical hardware, a technique called super-sampling is used, which allows you to process data faster than the on board clock frequency by processing N samples at a time. The PCIe transfer rate is 64 Gbps at peak performance therefore eight 32-bit integers must be processed concurrently as to not enforce back pressure. However, it is nearly impossible to successfully build this many concurrent samples without timing violations, thus we utilize four samples per cycle as opposed to eight. Figure 6.5 shows an example design of super-sampled FIR filter.

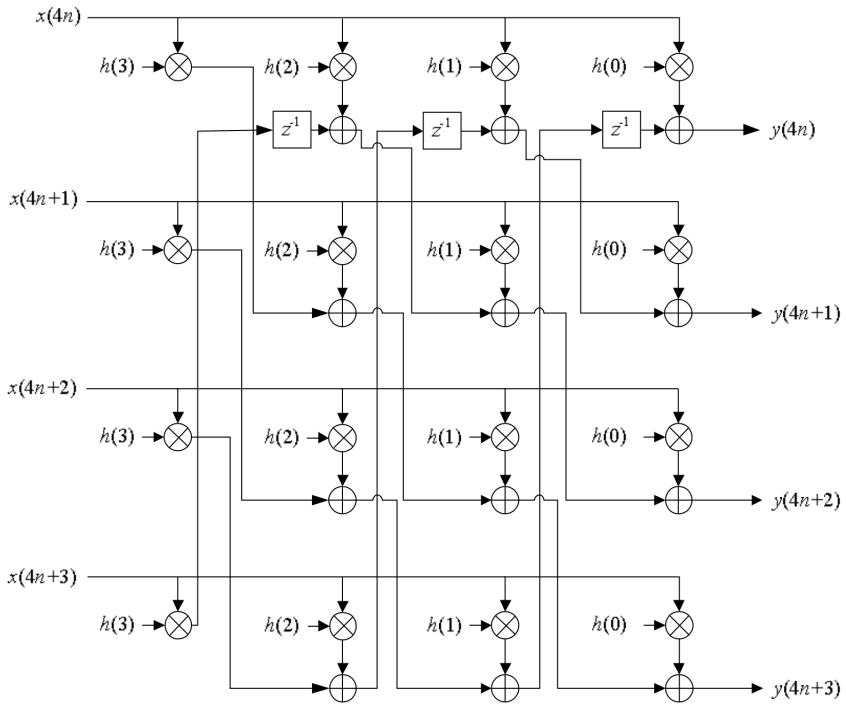


Figure 6.5: Parallel FIR Filter Transposed [14]

The transposed filter bank turns a Single Input Single Output (SISO) system into a Multi Input Multi Output (MIMO) system, which allows an increased sample rate without the need to increase the system clock. The above block design supports a 32 Gbps sample rate in our current system. The critical path of the design increases as a function of the parallel samples per cycle. This phenomenon is due to the cascaded nature of the filter bank. Every filter subsection is fed to the next stage through a chain of adders. The critical path can be seen from $X(4n)$ to $Y(4n + 3)$. The

path contains a chain of four multipliers and adders, which will grow with concurrent samples. However the area grows quadratically. For example, if we increase the number of samples by a factor of two, we will need two times the filter subsections, and each of them will need two times the number of adders and multipliers. Consequently, the number of pipeline registers will also increase linearly with concurrent samples.

We can integrate this hardware building block into the OpenFlow switch by layering it on top of the protocol. For example, the filter operation can be used as a customized action that will be applied on a stream of data if it matches an entry in the OpenFlow table. In addition, the programmability of an OpenFlow switch allows us to apply custom acceleration functions to a robust set of possible packet configurations. For example, the filter can be applied to packets with specific IP addresses, ports, MAC addresses, protocols, and ingress ports or any combination with wildcards.

The filter computation can be easily layered over a packet-based protocol such as UDP. In this case, TCP/IP is not an ideal protocol since packets can split on arbitrary boundaries. The networking stack controls this action based on network congestion and server acknowledgment configurations. Since a user at the application level cannot control these boundaries, the filter computation will have inconsistent holes causing undesired results. To configure the computation on the network data plane, a user could specify a flow rule as follows:

```
fpga_util add-flow in_port=1,actions=FIR,output:6
```

The rule shows that all packets on ingress port one will have the FIR filter computation performed on the UDP data payload. The payload starts at the 55th byte of data after ethernet, IP, and transport headers. Afterwards, the packe t will be forwarded to egress port six and delivered to the containerized application. The same rules could be applied numerous different ways, which demonstrates the flexibility of

these types of infrastructures. Figure 6.6 shows the bandwidth achieved as a function of filter sample length for both the SW and HW approaches.

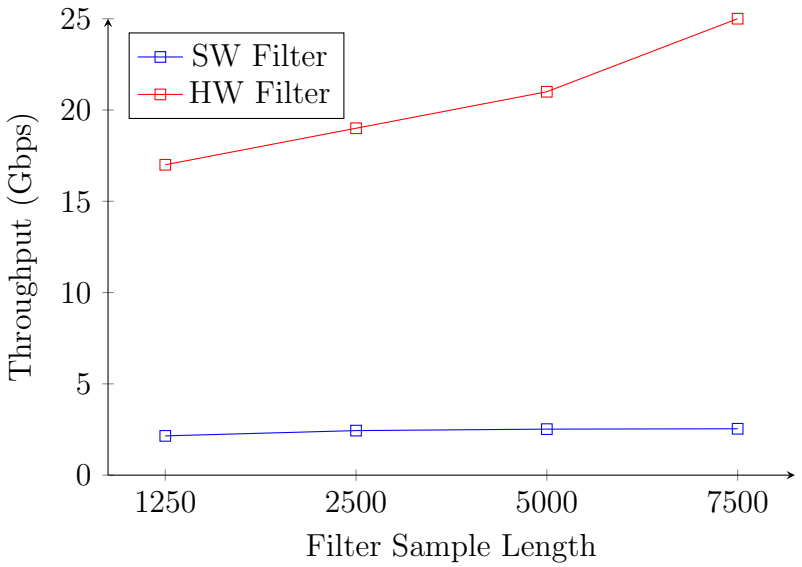


Figure 6.6: Edge Acceleration Speedup over UDP Protocol

The data shows that the HW filter is significantly quicker than the software. By offloading course grain computation, we can reduce I/O bottlenecks by performing the computation on the edge. The achieved bandwidth decreases as a function of filter sample length because the smaller the packet, the less utilized the network data plane is. This is due to the DMA engines and how the efficiency changes when dealing with smaller bulk transfers. However, the SW approach maintains similar performance irrespective of filter sample length. This is because the computation speed on the CPU grows as a function of input size. Thus, data can be sent faster with a lower filter sample length, increasing network link utilization. The maximum speedup between the two approaches is 10x, which is at higher filter sample lengths. The relative improvement would continue to grow with added offload complexity. For example, if we wanted a more accurate filter, we could add additional taps to the hardware resulting in more multiplies and adds per clock cycle. However, due to

the sequential nature of the CPU, the additional adds and multiplies would not be overlapped in SW, resulting in even lower link utilization.

6.4 Limitations for Memory Bound Computation

In our current system, the throughput of the network switch is bound by memory transfer speeds. The PCIe 3.0 spec [29] is an older specification from 2010, but OVS has improved significantly in recent years. Thus, comparisons are difficult to make fairly since the specification in use is outdated. This section discusses the newer PCIe specs available and in use for high throughput data center workloads. We will then make approximations on the relative speedup we could achieve based on our current DMA efficiency of the bus.

Many of the newer FPGAs now support PCIe 4.0, such as Intel Agilex and Stratix DX, and Xilinx Ultrascale+. These cards also support the Cache Coherence Interconnect for Accelerators (CCIX) base standard, making it ideal for high throughput workloads in modern data center environments. PCIe 4.0 provides a 2x improvement in total throughput from the previous version and requires 2x the operating clock for the endpoint applications. Timing closure is often difficult at these frequencies, depending on the application, but can be run slower if needed for correct operation. Intel Agilex particularly supports PCIe 4.0 x16, which would provide a 4x improvement in total throughput over the current design. Figure 6.7 shows the achievable bandwidth with different PCIe configurations varying lane width and specification version.

The configuration to the far left is the current system implementation which resides at a maximum bandwidth of 64Gbps. The speedup can be achieved by upgrading to a newer specification version, increasing the lane width, or both. The easiest from a design standpoint is upgrading to a more recent specification. This allows for the speed to increase without changing the design. However, it would introduce a

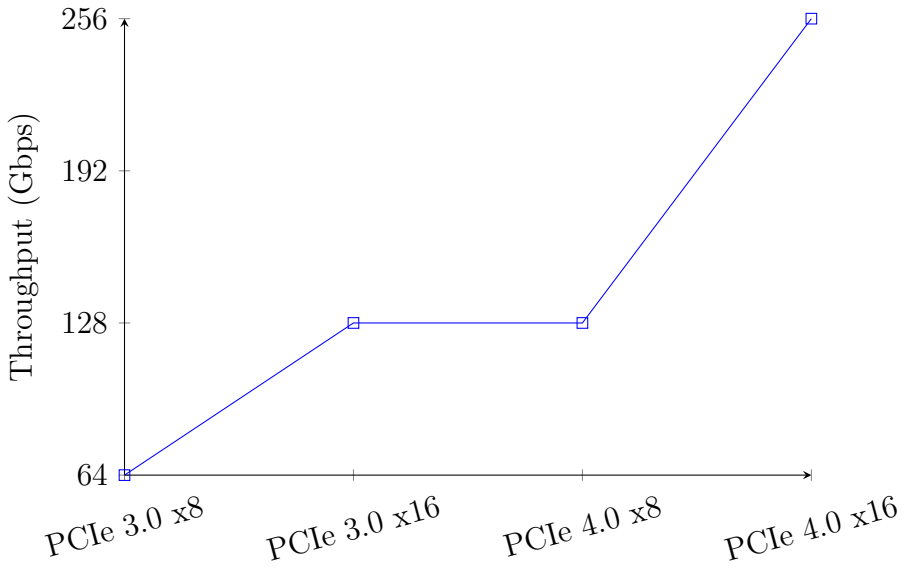


Figure 6.7: PCIe Spec Comparison

new design challenge for timing closure. Thus, pipeline registers would need to be added to compensate for a higher frequency. Luckily this is not too difficult since the design does not contain any significant data, control, or structural hazards for pipelined designs. This also means the design could be compensated without adding any complicated logic between pipelined register stages, maintaining the design size.

An increase in lane width would increase throughput proportionally but would be much more challenging to implement. This is because the lane width increase would enforce the data path's width to increase by a factor of two. The addition would change the network processor design and a portion of the RX and TX DMA engine block modules and increase the hardware cost substantially. Therefore, it is much more feasible to upgrade the specification for performance increases, leaving more room for improvement in the future. Based on the current DMA efficiency, if we upgrade the specification to PCIe 4.0, the throughput improvements should be around 70+ Gbps out of 128 Gbps. In addition, the design change would make the FPGA and host software tightly coupled, providing an acceleration solution that exceeds OVS's current performance for single core network applications. The theoretical speedup

over OVS would be approximately 40%, assuming a large multi-core server processor is used.

CHAPTER 7

CONCLUSION

In this thesis work, we presented a scalable MANET emulation environment that handles various use cases and topologies utilizing SDN and Docker. We demonstrated the environment's performance during simulations and highlighted key metrics of interest in MANET environments, such as coverage, fault tolerance, and bandwidth. The environment supported rapid prototyping using a graphical user interface to manage abstraction and display real-time traffic. SDN is very flexible and customizable; however, it utilizes high CPU utilization and can become problematic in large-scale emulated systems. To this end, we developed an open-source FPGA based OpenFlow pipeline using PCIe SR-IOV with efficient DMA engines. The results show that it is hard to justify utilizing a hardware network substrate when transferring data on the same host due to latency constraints, but transferring between hosts on a physical LAN is improved greatly through this approach. However, performance speedup can be achieved on the same host by offloading application-specific computation on top of the OpenFlow pipeline on the edge of the network stack. Thus, this work contributes a configurable IP that can be used in our MANET infrastructure and numerous other NFV tasks, which need a flexible network data plane with high throughput on and off-chip memory transfers.

7.1 Future Work

7.1.1 Scatter Gather I/O

The Linux kernel provides a hardware enhancement that allows reduced CPU utilization when transmitting packets over an interface. The enhancement reduces the copying required for large packets across multiple protocol headers. When a hardware device supports scatter-gather I/O, the kernel provides a linked list of packet fragments that make up an entire payload. These packet fragments are given to the hardware to assemble for proper operation. The fragments can be split on any boundary and can span across multiple layers of an Ethernet frame. Reducing the CPU utilization when sending packets can increase the system's scalability when emulating extensive network infrastructure. When a network interface does not support scatter-gather I/O, the networking subsystem must copy all protocol headers into one contiguous buffer so that the hardware can bulk transfer with proper packet header locality. Better performance could be achieved if the network hardware could be modified to accept a scatter-gather list of packet fragments and combine them contiguously instead of the host CPU. The modification would allow the network stack to provide packets to the hardware quicker and achieve a higher link utilization.

7.1.2 FPGA Offload with OVS

The results generated in this thesis work show that for the majority of traffic patterns, the OVS instance worked more efficiently, specifically on long-lived transport-level connections. OVS uses a microflow and megaflow cache [30] that all operate in the fast path. Thus, any cached packets can be processed very quickly with minimum CPU utilization and overhead. This makes an OVS hardware offload more useful as opposed to a complete replacement. Ideally, the FPGA design could be integrated with OVS and Data Plane Development Kit (DPDK), where expensive CPU-intensive OpenFlow pipeline operations could be offloaded to the hardware. In response, the

hardware could send back a unique marker for a particular flow entry so that the software could cache all subsequent packets making forwarding more simplistic. DPDK is a valuable package that works with OVS, which uses poll mode drivers in userspace to process more packets per second. However, it drastically increases CPU utilization since network interfaces are polled constantly for new events to be processed. An OVS offload engine would provide the best utility for OpenFlow pipeline acceleration and cached OpenFlow packet transfers. Unfortunately, this type of system requires a deep understanding of the OVS code-base to create hooks for HW/SW integration, which was out of scope for this thesis work.

7.1.3 Distribute Computation Across Hosts

The current system limits emulation to a single host. Each mobile node is virtualized using Docker containers in one computing environment. This limits scalability to the number of resources on one machine. However, multiple hosts could be used in one emulation as long as the network substrate supports this. The FPGA could use its external QSFP connectors to bridge communication between separate hosts through Ethernet. Thus, each host can communicate to any mobile node connected to the FPGA by using the programmable rules supported by OpenFlow. Each compute host could realize N mobile nodes, and the computation could be distributed to increase scalability. Each compute host could be directly plugged into the central server with the FPGA to support distributed emulation. Figure 7.1 shows an example system leveraging two SR-IOV enabled FPGAs to support distributed MANET emulations. The IP developed in this work is customizable and can easily be adjusted to support other FPGAs. Different compute hosts could utilize any FPGA that supports SR-IOV with minor modifications to the IP core.

In this example, we have three compute hosts running MANET sub-networks. The red and blue paths indicate routes enabled by both software virtualization and

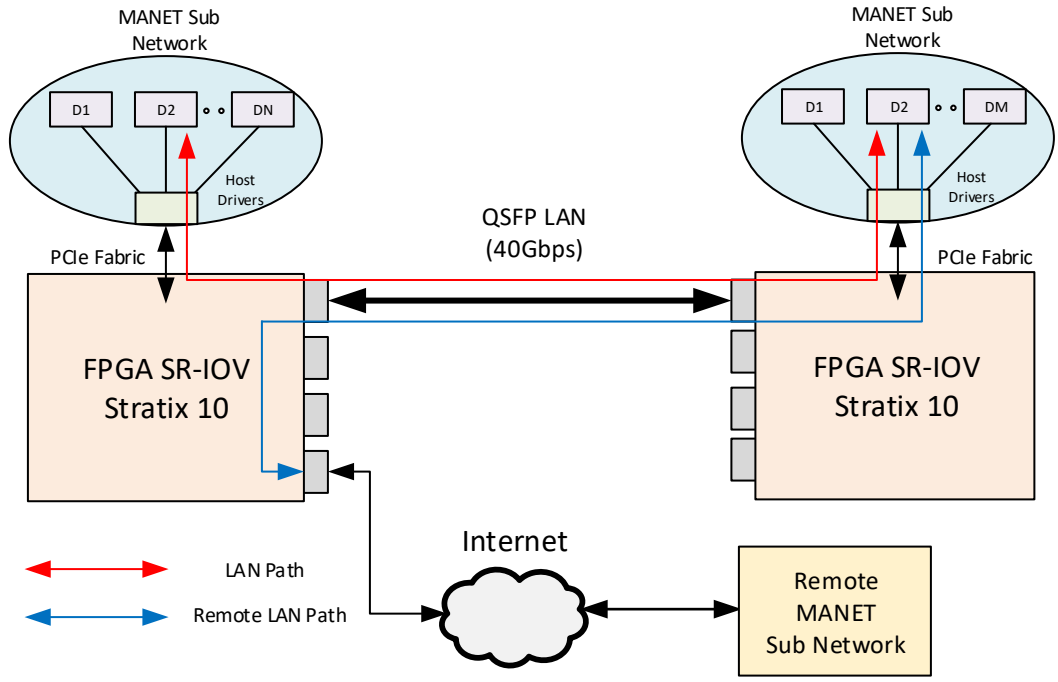


Figure 7.1: Distributed Emulation with SR-IOV enabled FPGAs

physical hardware. The red path connects Docker containers within each MANET sub-network through a Local Area Network (LAN) that goes through two OpenFlow pipelines for programmable routing. The blue path connects two docker containers through indirect routing. The remote MANET sub-network is not connected directly to the destination; thus, an intermediate network path is created within one of the FPGA's OpenFlow pipeline to forward the traffic across the LAN. Many complex routing topologies can be made between sub-networks as long as the control plane of all FPGAs are correctly synchronized. This enforces a distributed flow control methodology on top of our existing approach.

The custom switching solution would allow the QSFP network connector to achieve line rate at 40 Gbps, which is a significant improvement over using OVS as a bridge to an external NIC. In a non-SR-IOV system variant, each container would need to traverse the internal Docker network bridge into OVS through a virtual interface.

Then the packet would need to be classified, and the action performed and sent to the NIC through more virtual network infrastructures. Furthermore, each packet that is forwarded externally must traverse the FPGA regardless. Therefore, we achieve OpenFlow classification, matching, and forwarding for free without any additional latency penalties while reducing CPU utilization on each compute host. Our MANET emulator, hardware design, and device drivers developed in this work provide a path to these types of para-virtualized systems to enable efficient HW/SW co-design for SDN-based systems.

7.1.4 Interconnected CPU and FPGA

The current approach has high latency due to the PCIe switch traversals necessary for DMA. However, the latency could be reduced by utilizing an embedded CPU hardened on the same fabric as the FPGA. The modification would allow DMA to be performed with much lower latency and higher bandwidth. Now the throughput from host memory to the FPGA logic would be bound by the DDR transfer rate. Modern DDR4 memory has peak transfer rates of 256 Gbps, which is far higher than the current system implementation. Although, without PCIe, the SR-IOV protocol would be unavailable and we would need to implement our own virtualization layer on top. Thus, bandwidth would degrade due to translations between the different network interfaces in host SW making it difficult to get line rate performance. Performance benchmarks would need to be conducted in order to quantify these design changes. The changes would only be beneficial if the reduction of latency is more substantial than the efficiency enabled by the SR-IOV protocol.

BIBLIOGRAPHY

[1] Empowering App Development for Developers. https://www.docker.com/.

[2] Modprobe(8) - Linux man page. https://linux.die.net/man/8/modprobe.

[3] Open vSwitch. https://www.openvswitch.org. Accessed: 2020-12-04.

[4] Ping(8) - Linux man page. https://linux.die.net/man/8/ping.

[5] IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-1998* (1999), 1–214.

[6] Akyildiz, Ian F., Wang, Xudong, and Wang, Weilin. Wireless mesh networks: a survey. *Computer Networks 47*, 4 (2005), 445–487.

[7] Aviviano, Amy. Overview of Single Root I/O Virtualization (SR-IOV) - Windows drivers.

[8] Chowdhury, N., and Boutaba, R. Network virtualization: state of the art and research challenges. *IEEE Communications Magazine 47*, 7 (2009), 20–26.

[9] Corbet, Jonathan, Rubini, Alessandro, and Kroah-Hartman, Greg. *Linux device drivers*. OReilly, 2010.

[10] Day, John, and Zimmermann, Hubert. The OSI reference model. *Proceedings of the IEEE 71* (01 1984), 1334 – 1340.

[11] Docker. Docker Official Images, DinD. https://hub.docker.com/_/docker.

[12] Gueant, Vivien. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[13] Ha, Sangtae, Rhee, Injong, and Xu, Lisong. Cubic. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[14] Hou, Baolin, Yao, Yuancheng, and Qin, Mingwei. Design and FPGA Implementation of High-speed Parallel FIR Filters. *Proceedings of the 3rd International Conference on Mechatronics, Robotics and Automation* (2015).

[15] Information Sciences Institute. Internet Control Message Protocol RFC.

[16] Information Sciences Institute. Internet Protocol RFC.

[17] Information Sciences Institute. Transmission Control Protocol RFC.

[18] Intel. PCI-SIG Single Root I/O Virtualization (SR-IOV) Support in Intel Virtualization Technology for Connectivity.

[19] Intel FPGA. *Avalon Interface Specifications*. pp. 40–52.

[20] Intel FPGA. PCI Express High Performance Reference Design, Dec 2018.

[21] J. Postel ISI. User Datagram Protocol RFC.

[22] Jiang, W. Scalable Ternary Content Addressable Memory implementation using FPGAs. In *Architectures for Networking and Communications Systems* (2013), pp. 71–82.

[23] Kumar, Vijaya G., Reddyr, Vasudeva Y., and Nagendra, Dr M. Current Research Work on Routing Protocols for MANET: A Literature Survey. *International Journal on Computer Science and Engineering 2*, 3 (Nov 2010), 706–713.

[24] Kyle Locke Xilinx. Parameterizable Content-Addressable Memory.

[25] Lahyani, I., Makki, W., and Chassot, C. Failure Prediction for Publish/Subscribe System on MANET. In *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (2012), pp. 98–100.

[26] Liu, B., Yuan, B., Dai, H., Zhao, H., Yu, J., and Bhuyan, L. Improving the throughput and delay performance of network processors by applying push model. In *2012 IEEE 20th International Workshop on Quality of Service* (2012), pp. 1–4.

[27] Naudts, B., Kind, M., Westphal, F., Verbrugge, S., Colle, D., and Pickavet, M. Techno-economic Analysis of Software Defined Networking as Architecture for the Virtualization of a Mobile Network. In *2012 European Workshop on Software Defined Networking* (2012), pp. 67–72.

[28] Open Networking Foundation. OpenFlow Switch Specification.

[29] PCIe Express. *PCI Express® Base Specification*, 11 2020. Revision 3.0.

[30] Pfaff, Ben, Pettit, Justin, Koponen, Teemu, Wang, Alex, Jackson, Ethan J., Zhou, Andy, Rajahalme, Jarno, Gross, Jesse, Casado, Martin, Amidon, Keith, and et al. The Design and Implementation of Open vSwitch. *USENIX Symposium on Networked Systems Design and Implementation* (May 2015), 117–130.

[31] Plummer, David C. An Ethernet Address Resolution Protocol RFC.

[32] Sarawi, Shadi, Anbar, Mohammed, Abdullah, Rosni, and Hawari, Ahmad. Internet of Things Market Analysis Forecasts, 2020–2030.

[33] Shah, K. A., and Gandhi, M. R. Performance evaluation of AODV routing protocol with link failures. In *2010 IEEE International Conference on Computational Intelligence and Computing Research* (2010), pp. 1–5.

[34] Terasic Inc. DE10-Pro SX Edition FPGA Development Kit User Manual.

[35] The Qt Company. Design & Development Framework for Cross-platform Applications. https://www.qt.io/product.

[36] Ubuntu. Enterprise Open Source and Linux. https://ubuntu.com/.

[37] Woesner, H., Greto, P., and Jungel, T. Hardware Acceleration of Virtualized Network Functions: Offloading to SmartNICs and ASICs. In *2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)* (2018), pp. 1–6.

[38] Xingtao, L., Yantao, G., Wei, W., Sanyou, Z., and Jiliang, L. Network virtualization by using software-defined networking controller based Docker. In *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference* (2016), pp. 1112–1115.

[39] Yazar, Alper, Erol, Ahmet, and Schmidt, Ece Guran. ACCLOUD (Accelerated CLOUD): A novel FPGA-Accelerated cloud archictecture. *2018 26th Signal Processing and Communications Applications Conference (SIU)* (2018).

[40] Zhang, H., Wang, Y., Qiu, X., Li, W., and Zhong, Q. Network operation simulation platform for network virtualization environment. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)* (2015), pp. 400–403.