OPERATING SYSTEMS DATA TRANSFER OPTIMIZATION

Dennis Rodríguez R.

Thesis Submitted in Partial Fulfillment of the
requirements for the Degree of
Master of Computer Science

Department of Computer Science

INSTITUTO TECNOLOGICO DE COSTA RICA
2018

www.tec.ac.cr

TEC | Tecnológico de Costa Rica

Programa de Maestría
en Computación
Teléfono: 2550-2402

APROBACIÓN DE PROYECTO

"OPERATING SYSTEMS DATA TRANSFER OPTIMIZATION"

TRIBUNAL EXAMINADOR

Máster Ignacio Trejos Zelaya
Profesor Asesor

Máster Luis Carlos Loaiza Canet
Profesor Lector

Máster Luis Carlos Solano López
Profesor Externo

Dr. Roberto Cortés Morales
Coordinador del Programa
de Maestría en Computación

TEC | Tecnológico de Costa Rica
Maestría en Computación

Junio, 2018

Approval

Title:	**Operating System Data Transfer Optimization**
Examining Committee:	**Chair: Dr. Roberto Cortés Morales** Program Coordinator **Tutor: Ignacio Trejos Zelaya** Master of Science in Computation **Luis Carlos Loaiza Canet** Master of Science in Telematics **Luis Carlos Solano** Master of Science in Electrical Engineering
Date Defended/Approved:	June 2018

Ethics & Copyright

I, DENIS RODRIGUEZ-RODRIGUEZ with IDN° 112200751; declare that the dissertation “Operating System Data Transfer Optimization” has been developed completely by myself, having full respect for any intellectual property of the people who have developed the concepts and using the appropriate citation of the authors detailed completely in the references. This dissertation is presented only for the Costa Rica Institute of Technology in partial fulfillment of the requirements for the degree of Master of Computer Science.

I declare that the content of this document and its scope is authentic, and I take full responsibility for it.

San José, Costa Rica, June 2018.

Abstract

Balancing algorithms challenge the state of the art on how data exchanges as messages between programs that execute in the kernel and the applications running on top in user space on a modern Operating System. There is always a possibility to improve the way applications that rely on different spaces in an Operating System can interact. Algorithms must be placed in the picture all the time when thinking about next-generation human interaction problems and which solutions they require. Artificial Intelligence, Computer Vision, Internet of Things, Autonomous Driving are all data-centric applications to solve the next human issues that require data to be transported efficiently and fast between different programs, no matter whether they reside in the kernel or in user space. Chip designs and physical boundaries are putting pressure on software solutions that can virtualize and optimize how data is exchanged. This research proposes to demonstrate - via experimentation techniques, designs, measurement and simulation - that in-place solutions for data optimization transfer between applications residing in different Operating System spaces can be compared and revised to improve their performance towards a data-centric technology world. Specifically, it explores the use of a simulated environment to create a set of archetypical scenarios using an experimental design which demonstrates that PF_RING optimizes data messages exchange between Operating System kernel and user space applications.

Keywords: data; operating systems; pf-ring; pf-packet; virtualization; simulation; emulation; performance analysis; experimental design; algorithms; code complexity; embedded systems; SIMICS; QEMU.

Dedication

This work stands on its own thanks to a supportive family and amazing academic tutors. Computer Science is a passion and a way of living, without loved ones around this would not be possible.

Acknowledgments

To tutor and lecturer, Ignacio Trejos Zelaya who shares the greatest passion in computer science and education who has contributed to teaching the best Costa Rican computer scientists producing brilliant software developers shaping the country’s future in new technologies.

Table of contents

Approval ... iii
Ethics & Copyright ... iv
Abstract ... v
Dedication ... vi
Acknowledgments ... vii
Table of contents ... viii
List of tables ... xi
List of figures ... xii
List of acronyms ... xiv
Preface ... xv

Chapter 1. Introduction ... 1
1.1. Fundamentals ... 1
1.2. Problem Generalities ... 3
1.3. Justification ... 6
1.4. Problem Statement ... 8
1.5. Research Objectives ... 9
1.5.1. General Objective ... 9
1.5.2. Specific Objectives ... 9
1.6. Problem Scope and Limitations ... 9
1.7. Motivation ... 11
1.8. Thesis Structure ... 13

Chapter 2. Literature Review ... 14
2.1. Linux Operating System Fundamental Architecture ... 15
2.2. Data Transmission ... 19
2.2.1. TCP/IP Characteristics ... 19
2.2.2. Data Transmission ... 20
2.2.3. Data Reception ... 22
2.2.4. Interruption Process When Receiving Data ... 23
2.2.5. PF_PACKET Protocol ... 24
2.2.6. PF_RING protocol ... 25
2.3. Performance Benchmarks ... 30
2.3.1. Factors that Affect Performance ... 30
2.4. Experimental Design ... 32
2.4.1. Factorial Experiments ... 34
Factorial Experiment Concepts ... 35
2.4.2 Statistical Hypothesis Testing ... 38
2.5. System Simulation and System Emulation Technologies ... 39
2.5.1. Emulations vs. Simulation ... 39
2.5.2. What is SIMICS? ... 39
2.5.3. What is QEMU? ... 45

Chapter 3. Design 48
3.1. Problem Definition 48
3.1.1. Hypothesis 48
3.1.2. Approach 49
Solution Space 49
Direction 50
Prototype 51
3.2. Moving to kernel-space 51
3.2.1. Linux Kernel Modules 51
3.2.2. Linux Kernel Threads 52
3.2.3. Data Transmission from Kernel-Space 53
3.3. Architecture Overview 55

Chapter 4. Methodology 56
4.1. System Under Test Boundaries 56
4.2. Evaluation Methodology 57
4.3. System Services 58
4.4. System Workload 59
4.4.1. Simulated Network Traffic 59
Traffic Characteristic. 59
Data Rate. 60
Time Length. 61
4.4.2. Application Data Tx/Rx 61
Data Object Creation 61
Query Workload. 61
4.5. System Performance Metrics 62
4.6. System Parameters 62
4.6.1. Applied Algorithm Capturing Method 62
4.6.2. Device Selection 63
4.6.3. Filtering 63
4.6.4. System Specifications 63
4.7. Controlled System Factors 64
4.7.1. Data Processing Method 64
Kernel Module 64
AppDataPfRingLogger 64
4.7.2. Virtual Device 65
4.7.3. Test Workloads 66
Data Size 66
Data Rate 66
4.7.4. Application Data Workloads 67
4.8. Experimental Technique and Simulation Environment 67
4.8.1. Evaluation and Simulated Environment 67
4.8.2. Experimental Technique 69
Data Drop Rate 69

CPU Utilization 70
Memory Utilization 70
Data Retrieve Delay 70
4.9. Experimental Design 70
4.9.1. Experiment 0: empirical baseline 71
4.9.2. Experiment 1: PF_PACKET System Enabled 72
4.9.3. Experiment 2: PF_RING System Enabled 72
4.9.4. Methodology Summary 73

Chapter 5. Analysis 74
5.1. Results and Analysis of Experiment 0 74
5.1.1. Experiment 0 Overview 74
5.1.2. Data Drop Rate 77
5.1.3. CPU Utilization 79
5.1.4. Memory Utilization 80
5.1.5. Summary Analysis 80
5.2. Results and Analysis of Experiment 1 82
5.2.1. Experiment 1 Overview 82
5.2.2. Data Drop Rate 85
5.2.3. CPU Utilization 87
5.2.4. Memory Utilization 88
5.2.5. Summary Analysis 88
5.3. Results and Analysis of Experiment 2 88
5.3.1. Experiment 2 overview 89
5.3.2. Query Metrics 91
5.3.3. Data Drop Rate 92
5.3.4. CPU Utilization 93
5.3.5. Memory Utilization 93
5.3.6. Summary Analysis 94
5.4. Overall Analysis 94

Chapter 6. Discussion 96
6.1. Conclusions 96
6.1.1. PF_RING Kernel-Space Capability in an OS 96
6.1.2. Reduce Data Drop Rate When Using PF_RING in a User-Space App 97
6.1.3. Reduce CPU Utilization When Using PF_RING in an OS 97
6.1.4. Reduce Memory Utilization When Using PF_RING in an OS 98
6.1.5. On Research Methods and Tools 99
6.2. Engineering Significance 100
6.3. Future Work 101
Appendix A. AppDataPfringLogger changelog 102
Appendix B. Simics Configuration Script 103
Bibliography 106
Vita 108

List of tables

Table 2. 1 Two by two factorial experiments 36

Table 4. 1 Listing of each configuration test containing three tests every five repetitions. 72

Table 5. 1 Experiment 0 hypothesis testing of dropped packets.....78
Table 5. 2 CPU Utilization hypothesis testing in Experiment 0. 79
Table 5. 3 Dropped Packet Rate hypothesis testing on Experiment 1 86
Table 5. 4 CPU Utilization hypothesis testing on Experiment 1 87
Table 5. 5 Query SUT performance performed on Experiment 2.....92
Table 5. 6 Data dropped packet rate hypothesis testing on Experiment 2 92
Table 5. 7 CPU Utilization hypothesis testing in Experiment 2. 93

List of figures

Figure 1. 1 Data transmission between applications on a Linux OS. 5

Figure 2. 1 Fundamental Linux Operating System Architecture. 18
Figure 2. 2 Linux Operating System Architecture. 18
Figure 2. 3 TCP header structure. 20
Figure 2. 4 Data packet transmission through TCP/IP layers from kernel application. 20
Figure 2. 5 Data packet transmission through TCP/IP layers to kernel application. 22
Figure 2. 6 softirq interruption process when receiving data. 23
Figure 2. 7 Data flow differences between PF_PACKET and PF_RING. 28
Figure 2. 8 Ring buffer socket in PF_RING architecture. 29
Figure 2. 9 Pathways of an incoming data packet through the OS from Kernel to the application. 33
Figure 2. 10 Operating System “Build A” and “Build B” differences. 34
Figure 2. 11 Simics is running VxWorks Operating System. 41
Figure 2. 12 Simics architecture overview. 42
Figure 2. 13 QEMU system architecture 45
Figure 2. 14 Linux is running on top of QEMU on a Windows host. 47

Figure 3. 1 Architecture overview modeling kernel to user implementing data transmission operation using PF_RING. 55

Figure 4. 1 System conditioned under test diagram. 56
Figure 4. 2 Quick Start Platform running Linux. 68
Figure 4. 3 QSP running Linux on top of a simulation running Simics. 68

Figure 5. 1 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. Dropped packets (%). Blue = PF_PACKET, Red = PF_RING. 75
Figure 5. 2 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. CPU utilization (%). Blue = PF_PACKET, Red = PF_RING. 75
Figure 5. 3 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. Memory utilization (MB). Blue = PF_PACKET, Red = PF_RING. 76
Figure 5. 4 Summary graph of Experiment 0 SUT key metrics when packet size levels impact data drop when using pf_ring. 76

Figure 5. 5 Summary graph of Experiment 0 SUT key metrics when packet size levels impact CPU utilization when using pf_ring.....77

Figure 5. 6 Data packets are written to dev/null on Experiment 0 comparing pf_packet and pf_ring. 78

Figure 5. 7 Data that relates data drop rate and CPU Utilization. On the first graph, the Y-scales differ to visually show the effect of dropped data has on CPU Utilization. It is composed of a single trial but gives the 'all trials' trend. .81

Figure 5. 8 Graph compares CPU Utilization using PF_RING showing no data drops while AppDataPfRingLogger has the same data drop rate as in the first graph.....82

Figure 5. 9 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger. Dropped packets (%). Red = PF_PACKET. 83

Figure 5. 10 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger.CPU utilization (%).Red = PF_PACKET. 84

Figure 5. 11 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger. Memory utilization (MB). Red = PF_PACKET. ..84

Figure 5. 12 Data size impact depicts the rate of dropped packets. 85

Figure 5. 13 Data size impact depicts the CPU utilization. 85

Figure 5. 15 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. Dropped packets (%). Red = PF_RING.....89

Figure 5. 16 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. CPU utilization (%). Red = PF_RING. 90

Figure 5. 17 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. Memory utilization (MB). Red = PF_RING.....90

Figure 5. 18 Key query metric (query delay) for PF_RING kernel module.....91

List of acronyms

AD	**Autonomous Driving** is an automatic driver that controls a vehicle.
ASIC	**Application-specific Integrated Circuit** is an integrated circuit customized for a particular use rather than intended for general purpose.
API	**Application Program Interface** is a set of routines, protocols, and tools for building software applications. Specifies how software components should interact.
BIOS	**Basic Input/Output System** is a non-volatile firmware used to perform hardware initialization during the booting process (power-on startup) and to provide routine services for operating systems and programs.
DPI	**Deep Packet Inspection** is a form of computer network packet filtering that examines the data part (and possibly also the header) of a packet as it passes an inspection point, searching for protocol non-compliance, viruses, spam, intrusions, or defined criteria to decide whether the packet may pass or if it needs to be routed to a different destination, or, for collecting statistical information that works at the Application layer of the OSI (Open Systems Interconnection model).
ITCR	**Costa Rica Institute of Technology** is a government-owned Engineering and Management post-secondary education institution founded in 1971.
IoT	The **Internet of things** is the network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and network connectivity which enable these objects to connect and exchange data using Internet protocols.
NIC	A **Network Interface Card** is a network hardware adapter in the form of an add-in card that fits in an expansion slot on a computer motherboard. It provides the interface between a computer and a network.
OS	An **Operating System** is system software that manages computer hardware and software resources and provides common services for computer programs. All computer programs, excluding firmware, require an operating system to function.
PF	**Packet Filtering**. Data travels on the Internet in small pieces; these are called packets. Each packet has certain metadata attached, like where it is coming from, and where it should be sent. The easiest thing to do is to look at the metadata. Based on rules, certain packets are then dropped or rejected. All firewalls can do this. It is done at the network layer.
QEMU	**QEMU** (short for Quick Emulator) is a free and open-source hosted hypervisor that performs hardware virtualization (not to be confused with hardware-assisted virtualization).
SICS	Swedish Institute of Computer Science
SIMICS	**Simics** is a full-system simulator used to run unchanged production binaries of the target hardware at high-performance speeds. The SICS originally developed Simics and then spun off to Virtutech for commercial development in 1998. Intel acquired Virtutech in 2010 and Simics is now marketed through Intel's subsidiary Wind River Systems.
SoC	**System on a Chip** is a full system as an integrated circuit that integrates all components of a computer or other electronic systems.
SUT	**System Under Test** refers to a system that is tested for correct operation. It is the test object. The term is used mostly in software testing.
TCP/IP	**Transmission Control Protocol/Internet Protocol** is a suite of communication protocols used to interconnect network devices on the Internet. It is in use as a communication protocol in a private network.
VP	**Virtual Platform** in computing, *virtualization* is the act to create a virtual version of something on a virtual computer hardware platform.

Preface

Data transmission between interconnected devices nowadays demands speed and reliability. Wire-speed packet capture and transmission using commodity hardware have challenges on performance in a world where every human being may be connected through smart devices, exchanging images, documents, music, voice and other data for doing business with each other. Relying upon hardware development cycle is not fast enough to comply with user demands. Thus the implementation of innovative software algorithms that will break hardware barriers or optimize hardware capabilities is essential for technology advancement. The high-level objective of this research is to propose alternative ways for discovering and analyzing algorithms that will impact directly on data transmission in an always-connected world.

Chapter 1. Introduction

1.1. Fundamentals

Data and access to information are critical for each person in the world. As of this writing, 51% of the global population has Internet access. Most of them access via smart mobiles or personal computers. By 2021 the average smartphone user will plow through 8.9 GB of data per month [CERWALL, 2016]. Artificial Intelligence (AI), Computer Vision, Autonomous Driving, Internet of Things (IoT) will all require redesigned computer systems, which is impacting how Computer Science (CS) will solve software optimization problems to handle the considerable amount of data that will be required for fulfilling the needs of an average person.

Investigation towards optimizations in CS is mostly related to algorithms in the field of Analysis of Algorithms and Data Structures. The use of one algorithm versus another, their comparison and their results in real life scenarios are related to computer architecture and performance analysis. CS contributions in the field of simulation are also fundamental to Theoretical CS in experimentation and formal methods [RIVEST, 1990].

If a person wants to go from city *{R}* to city *{Z},* there are many ways to do it. He or she may take a flight, go by bus, by train or by any other transportation means that might be considered. It all depends on availability and convenience and how it suits his or her needs. Similarly, in CS there are multiple algorithms to solve a problem. When there is more than one algorithm to solve the same problem, we need to select the “best” one depending on the usage scenario. Performance Analysis creates organic results based on requirements to select the best-suited algorithm, from multiple ones, to solve a problem. Algorithm performance is a process of making objective judgments on algorithms – mostly through analysis of quantitative data. Performance analysis is concerned with predicting the resources that are required by an algorithm to perform its task.

In general terms, the substance of an algorithm is to provide the exact solution to a problem that is understandable and easy to implement, with finite and known memory and time requirements. Setting metrics related to the use of memory, speed of execution, complexity and even implementation details is required to compare algorithms. In this research, factors are reduced to evaluate an algorithm for space complexity and time complexity.

Today most data come from systems related to the World Wide Web or interconnected via the Internet. Inter-networked physical devices connected and communicating to each other will be around 30 billion objects by 2020 [HWANG, 2013]. Imagine the amount of raw data that such number of devices can generate and that will need to be processed by an endpoint device as fast as possible, so that a user does not have to wait. Depending on the medium, these data traverses the Internet, where most personal computers and smartphones connect to wireless access points utilizing a wireless network card. A card of this type is embedded in smartphones, access points, tablets, laptops, and personal computers. All these data are taken from the physical medium and processed by embedded software that receives the data and transforms it in such a way that the device 'understands' it and shows it in an easy way for a human brain to interpret. All this process involves a means to move data from *{R}* to *{Z}.*

This research intends to characterize an experiment environment with archetypical scenarios involving two algorithms known as PF_PACKET and PF_RING. Both algorithms are incorporated in alternate builds of an otherwise identical Operating System distribution. The only difference between the archetypical scenarios is the algorithm used to perform data transmission at the kernel level. A high-level objective of the investigation is not only to understand whether replacing one algorithm for another is feasible but also to establish whether there is a performance improvement regarding storage complexity (memory usage) and time complexity (processing cycles).

The importance of the study relies on generating similar workloads -- described in terms of volume and stress -- to orchestrate and define synthetic situations of common scenarios that push the boundaries of the mentioned algorithms. The study leverages the use of an experimental design method to objectively compare the complexity of the alternate algorithms.

An additional added value of this research is the creation of a simulated environment that serves as a medium to execute the designed experiments. The simulated environment uses a Full System Simulator known as SIMICS to decouple factors which may affect the observable system or the results. An analysis of the reasoning behind which Full System Simulator to choose between QUEMU and SIMICS is another added value of the investigation.

1.2. Problem Generalities

Computer network equipment contains embedded microprocessors that use specialized software to provide the capabilities needed to transmit user data. ***Firmware*** is code that implements the algorithms necessary for the data to arrive from one place to another [e.g., *{R} -> {Z}*] in an embedded device. Firmware is a program at the lowest level, close to the hardware, that establishes the logic for controlling the electronic circuits of any type of device. It is strictly related to the hardware or electronics of a device, so it is software that interacts with the electronic components taking care of the control to execute the instructions correctly.

Network equipment today uses a variety of computer architectures. For example, wireless access points typically use a configuration whose sole purpose is to transmit data as quickly and efficiently as possible. Such equipment uses dedicated Central Processing Units (CPUs) or Application Specific Integrated Circuits (ASICs) and devices with low power consumption. Devices primarily use computer architectures such as PowerPC, ARM or x86. There are also two different types of standard wired and wireless network cards in the industry,

provided by companies such as Intel, Qualcomm, Atheros or Broadcom. A commonly used Embedded Operating System (EOS) is a reduced open licensed GPL Linux distribution known as BusyBox. A simplified Unified Extensible Firmware Interface (UEFI) flavor such as seaBIOS (uncomplicated legacy BIOS) is also part of the tools used in industry standard embedded devices.

Advances in network data packet switching, the transmission of data packets from the analog medium for its digitalization and processing, tend to focus on hardware improvement. This puts the physics of electronic circuits to the limit. However, attention is switching towards software and extending its capabilities as a solution to many of the problems and limitations of physical circuitry. At companies such as Intel, for example, all their designs for the latest generation chips go hand in hand with the development of software to impact the market with value-added solutions for companies and end consumers. In technology jargon it is often quoted that “hardware without software is like a body without a soul.” Today, software is the value added on top of the chips. Still, the technology industry does not sell software; it sells solutions.

To understand the world around us, designers and engineers model the environment and represent what they need with mathematical equations that follow approximately - but faithfully - the behavior of reality. In order to understand the impact of one component versus an alternative on the performance, availability and reliability of a Network Operation System, this research leverages virtual platforms (VP) –pieces of software simulating pieces of hardware--. A model of the data flow between resident programs from user space and the Kernel can help their efficiency. Fast data transfer brings a better user experience when surfing the Internet, listening to a song or watching a movie by streaming.

Virtualization platforms (VP) are hardware models that live in virtualization software such as Simics [SIMICS, 2016] or QEMU [QEMU, 2016]. The emphasis of this research is the implementation of the PF_RING algorithm in substitution of PF_PACKET in an x86 VP that runs seaBIOS, kernel v4.6, and a custom BusyBox.

The methodology uses programs written in a high-level programming language such as C/C++ in order to execute unbiased and reproducible experiments to pass network packages (or any data type) in frames of the user space to the Kernel space performing a factorial analysis that provides objective measures of performance regardless of the (physical) hardware on which the algorithm works.

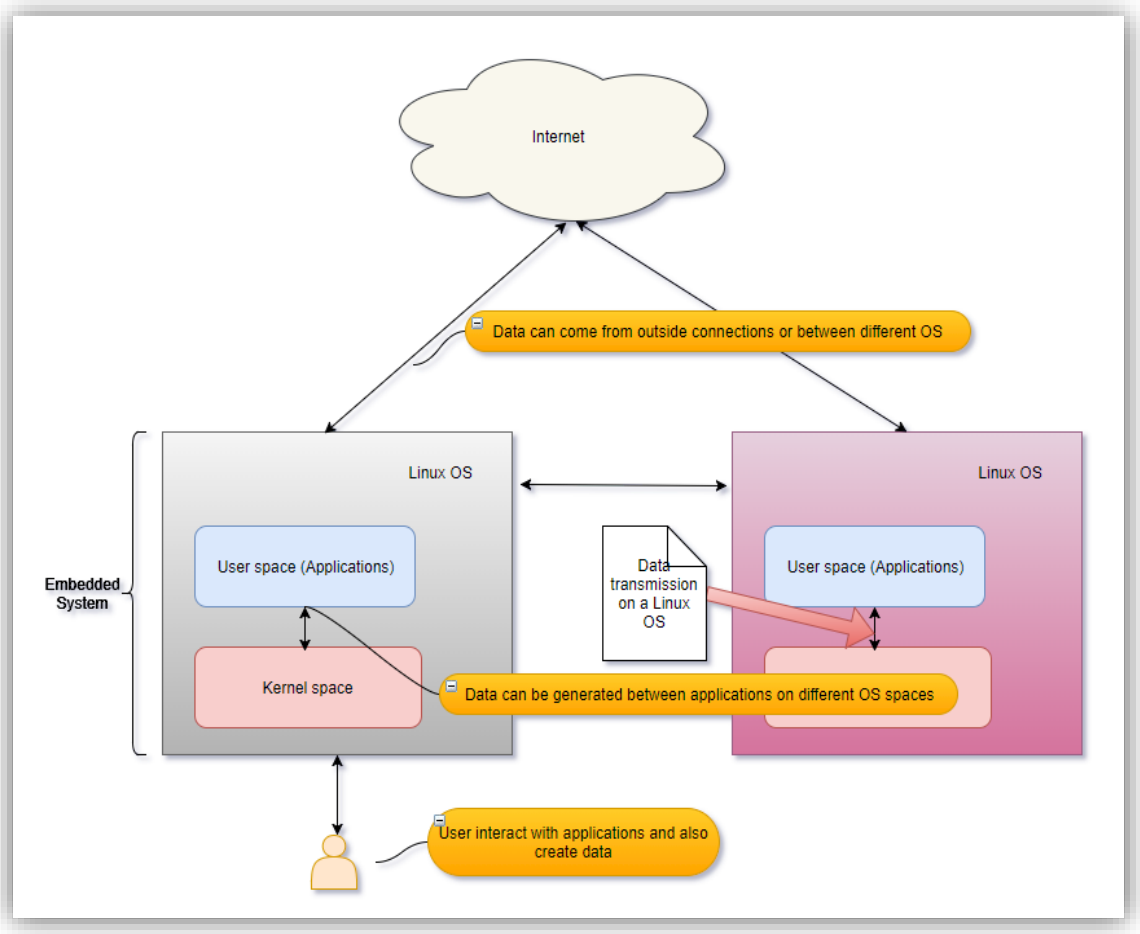

Figure 1. 1 Data transmission between applications on a Linux OS.

When a packet reaches the wireless network card of an embedded system, the Linux kernel takes the data and transfers it across the entire internal network segment stack in the OS to process it and thus sends the information to a user module or software application which runs in the user space. These copies of data from kernel space to user space become expensive in regards to CPU processing cycles, memory utilization, among other metrics. A copy of the complete package

has to be made in order to have it processed by the user application, typically using the system call interface or other types of interfaces [KELLER, 2008]. Figure 1.1 shows where the research problem is contextualized. The data transmission of OS components between user space and kernel space is the subject of this investigation.

Among the improvements made since version 2.6.32 of Linux Kernel, there is the ability to pass packets from kernel to user space using the PF_PACKET technique (also known as a raw socket) [MCCANNE, 1993]. The difference concerning the default network segment of the kernel is that PF_PACKET makes a copy of the packet buffer before it enters the network segment of the Kernel and sends the entire package to the application in the user space. These copies are always done for each one of the packets. Currently, these copies are executed through the system call interface which causes the OS to overflow with a significant amount of data traffic by continually interrupting the system for each copy of information frames.

1.3. Justification

When a person is at an airport, restaurant, ice cream shop, train station, hotel, home, or even at a football stadium; the chances are high that he or she will wish to connect to a wireless network. Nowadays, in urban settings, it is assumed that the wireless network service or the 802.11n/ac standard [IEEE, 2015] is available to all those who wish to stay connected at any time and location. Studies indicate that wireless traffic from mobile devices will grow by 80% by 2018 [WOOD, 2013]. The world trend suggests that growth of Internet of Things (IoT), Artificial Intelligence (AI), Augmented Reality and Big Data Analytics will cause an explosion of data that will escalate the need for information processing, increasingly better, more robust and efficient, algorithms. The situation suggests that data processing speed on embedded wireless access devices or controllers will be determinant to keep up with high data demand.

Data Packet Inspection (DPI) [PORTER, 2005] consumes a high amount of CPU cycles depending on how much information extracted from the network packets – metadata - is exchanged between systems. The fewer data packets to be sent to the OS kernel[1] the better, so they can reach the application directly in the user space for further processing or presentation to a human user. It is better to process the packets for a service in the application that uses the data, instead of having to perform this in the kernel space and wait for it to send them to the user space program. Good utilization of system resources enables the EOS to take care of other priority tasks.

In the case of wireless devices such as smartphones, it is vital to explore hardware and software alternatives that expeditiously bring packets where they are needed, using more efficient algorithms that require fewer processor cycles and less memory. The aim is to diminish or avoid the use of resources that are already very scarce in an Embedded System. The proposal is to use a software component (the PF_RING algorithm) to pass data directly from the wireless network card (which is being managed in the EOS Kernel space using drivers) towards a user-space application - the program that consumes this data. Data transfer efficiency is becoming especially necessary to meet the extreme demand on wireless networks caused by the explosion of IoT and Big Data. The technology community needs to explore new algorithms for embedded devices, otherwise wireless communcation technologies will become a bottleneck in embedded OS that will be unable to accurately process larger number of network packets in the future. Communications performance will constrain and impact the markets for IoT, AI, Autonomous Driving Systems (ADAS) and Big Data Analytics / Business Intelligence.

[1] Specifically, to the Kernel's network stack segment.

1.4. Problem Statement

Connected devices are growing faster than anyone can imagine. By 2025, about 80 Billion devices will connect to the Internet [IDC, 2014]. In contrast, approximately 11 Billion devices are connected to the Internet today. By 2020 it will triple to 30 Billion, and five years later it will reach the mentioned number of devices. The total amount of digital data created worldwide is expected to increase from 5 zettabytes[2] to 44 zettabytes by 2020 [IDC, 2014].

The research reported in this paper is concerned with decreasing data processing time and memory consumption via algorithms that reside in an Embedded OS. The challenge is to create a set of archetypical scenarios that will simulate what happens when data comes into a system running an OS with one algorithm or the other. The research has aimed to determine a set of experimental design techniques that will allow the measurement of observable variables in software running on a simulated hardware and network environment.

The focus is on validating the integration of a new algorithm (PF_RING) within an Embedded OS and obtaining objective results to establish whether there is a consistent improvement in the transmission of data packets from the user space to the Kernel space and vice versa.

The research addresses replacing the PF_PACKET, component commonly used in Linux Kernels (within Embedded OS), with the PF_RING alternative, providing objective measures of performance and reliability. A methodology has been designed to run sets of experiments that use a virtual hardware simulation that allows to take measurements to compare PF_RING against PF_PACKET. The primary approach is to systematically observe different realistic scenarios and under which conditions there is an improvement in the copying of data between the kernel space and an application in the user space.

[2] 1 zettabyte = 10^21 bytes.

To improve the data transmission efficiency of commodity network systems, this research addresses data exchange in Operating Systems architectures leveraging from data capture algorithms from kernel to user space when reading and writing data between applications.

1.5. Research Objectives

1.5.1. General Objective

Determine the impact on system performance of two algorithms for data transfer that reside in different spaces of an Operating System.

1.5.2. Specific Objectives

1. Design a platform for technical experimentation on operating systems, software components, virtual computing, and networking elements.
2. Create an application that resides in user space to allow Kernel module usage with different algorithms for data transmission.
3. Propose a set of experiments to measure the use of resources by the PF_RING and PF_PACKET algorithms.
4. Execute a set of tests to collect precise performance measurements of both data transmission algorithms.
5. Analyze the obtained results to enable objective quantitative comparisons between the PF_RING and PF_PACKET algorithms.

1.6. Problem Scope and Limitations

The scope of the problem is enabling the use of the PF_RING technique [DERI, 2001] within a stable Kernel within a representative Linux-based Embedded Operating System targeted at Intel x86 architectures. The software will be run on top of an Intel x86 architecture model available on a virtual platform simulator (such as Simics). The research is limited to perform measurements according to the methodological definition and experiments (described later) by implementing an

application in the user space that allows to objectively compare PF_PACKET against PF_RING to establish whether there is an improvement in the exchange of data from the Kernel space to the user space that will impact system performance in network environments positively.

It is beyond the scope of the research to modify the Kernel OS, to update it or to change its configuration in such a way that the behavior of the OS varies with the methodological tests that will be executed. As explained in the methodology, some factors are considered outside the scope of the research whether they can affect the results positively or negatively.

The research approach is oriented to assessing alternative algorithms in pursuit of conclusive results derived from the methodological approach defined in this document. In face of continuous improvements in hardware or models for the x86 architecture that influence the speed at which packets are captured, it is considered as a factor outside the scope of the investigation, and this is considered in the experimentation method. Architecture models can vary, improve, and models on the virtual platform simulator can impact efficiency or reliability when a simulation is performed, but the methodological approach isolates these factors so as to remain unbiased. This investigation does not aim to create new data packet transfer or inspection algorithms; the intention is rather to establish whether an existing alternative surpasses the one commonly used in embedded system applications.

This research project synthesizes concepts and techniques from several areas of Computing such as: operating systems, advanced computer networks, algorithms, simulation, software engineering, probability, and statistics.

The research will be limited to the use of a virtual platform hardware model to simulate normal conditions and to obtain results regardless of undetermined factors in the factor analysis study. Also, the project is developed specifically in the firmware embedded in an x86 architecture. The development of any deliverable is subject to the above as well as the available hardware models that can be

simulated on a virtual platform on top of which the Unified Extensible Firmware Interface (UEFI), BIOS and Embedded OS can be run.

The following is delivered:

- A methodology to evaluate the performance of PF_RING compared to PF_PACKET in a simulated computer architecture model.
- A set of experiments that test different usage scenarios.
- An application expressed in a programming language that allows the tests to be performed following the methodological approach described herein.
- A set of experimental results.
- Analysis of results.
- Conclusions and recommendations.

The Kernel module implementing PF_RING is not delivered since this is subject to GPL licenses, but the code is provided so that the experiments can be replicated elsewhere.

1.7. Motivation

Data transmission from user space to the kernel has an associated cost regarding the resource utilization of an OS. There is little work on the Linux Kernel to perform this task more efficiently. Often reliance is placed on new hardware developments, but the software that controls and uses them is not efficient enough to exploit all capabilities.

Incorporating changes to a system that is in use or called into production by customers around the world is risky. A simple patch to the kernel can cause a company to lose millions of dollars in sales and customer losses. Before making a code change to a product or developing a new solution, the industry researches new technologies and creates Proofs of Concepts (PoC) that demonstrate their feasibility or otherwise; one such situation is management-related techniques of

data packets in the kernel space. This research's objectives will set the basis for discerning the following:

- Have a functional PoC with PF_RING implemented for data transmission packets between applications residing on the kernel space and the user space in a simulated environment.
- Feasibility of implementing PF_RING in the Linux Operating System.
- Use the PoC to apply the methodology for obtaining results and conclude whether PF_RING is superior to PF_PACKET.

By completing the objectives, the research will lead to answering the question: *Is it possible to make data traverse efficiently the Operating System stack between kernel space and the user space by using the PF_RING algorithm?*

Meeting the objectives of the research expectation involves answering the following questions:

- Is it possible to integrate a kernel module that implements PF_RING in the kernel code base of an OS in a simulated environment?
- Is it possible to change the paradigm in which data packages are copied between the Kernel and the user space by implementing the PF_RING technique?
- Is it possible to replace default or old algorithms in the Linux Kernel to make a way for new implementations for data transfer from one space to another, improving the resource utilization?
- Does the use of factorial analysis and hypothesis testing help to obtain conclusive results of PF_RING as an algorithm to improve data transmission between applications residing in different OS spaces?

1.8. Thesis Structure

Chapter 2 provides background information about data transmission at the Operating Systems level, architecture, data, PF_RING, PF_PACKET, and simulation. Chapter 3 delves into the details of designing data transmission experiments within an OS. Chapter 4 provides the methodology used to conduct the performance analysis of the system. Chapter 5 presents the results and analysis of the system performance, and Chapter 6 identifies the conclusions drawn from the performance analysis and indicates future recommendations in this research field.

Chapter 2. Literature Review

Software design processes and project management [RUPARELIA, 2010] in many cases do not allocate sufficient time to investigate in-depth a particular area of interest. Frequently there is no allowance for implementing a core solution and to conclusively demonstrate whether or not a research proposal is feasible in technical and economic terms. In large technology corporations, many projects are explicitly contingent on "time to market" (TTM). Several types of research are left out or not fully realized because of the priorities of reaching the market with a product as soon as possible.

Marketing departments have an increasing interest on what users are doing on social networks, how much time they spend connected to the Internet, the sites they visit, and which applications or which networks they use.

Implementation of PF_RING relates to "Deep Packet Inspection (DPI) [PORTER, 2005]". It takes each of the network packets or the network traffic data that passes through an embedded system, inspects the data components contained in the package in detail, and applies machine learning techniques [MOHRI, 2012] and heuristics. Quick classification processes help determine whether certain "flows"[3] correspond to an application, web service, particular service, among others. Such applications or services can be Facebook, Twitter, Instagram, Google, or Skype, among others. DPI allows to create user profiles, to determine the time that a user dedicates to a service; information can be collected and shown in dashboards allowing businesses to make strategic decisions.

It is of interest to unveil how the Linux Kernel in an embedded system can take the incoming data from the media and pass it through the network card to the driver and then to the application in the user space. Understanding the efficiency with which DPI is being applied in data-driven industries will shed light into improving the way the algorithms bring data to user applications. The use of DPI requires

[3] Set of data packets.

many data copies, via system calls, from the kernel to the user space making it costly for an application to send an interrupt to the system every time it requires the data back. PF_RING aims to reduce on expensive system calls; thus, an improvement in resource utilization is expected by changing the data transfer paradigm through the hardware and OS software stack to place the data where it is needed.

Historically, the common way to adopt a new technique of exchanging packets between the kernel and the user space has been to upgrade the OS Kernel from one version to another. The latter contains an accepted improvement on this topic by the Linux developer community. However, this is not feasible when times to market are becoming shorter, and companies need to release their products and services. PF_PACKET comes in the latest versions of the Linux kernel by default; in the last few years there have been no significant efficiency improvements in the way data is copied to the kernel stack.

Nowadays PF_PACKET supports DPI. However, an alternative technique called PF_RING has recently appeared. Its software architecture promises an improvement in data exchange efficiency between resident applications either in the kernel or at user level [DERI, 2001]. PF_RING is not implemented in the latest stable versions of the Linux Kernel or in an open source distribution of an OS in the Unix family.

It is fundamental to understand how Linux architecture works to explain how an algorithm (or software component) might be better than another one.

2.1. Linux Operating System Fundamental Architecture

Linux is one of the most used Operating Systems [FINLEY, 2018] solutions from smart devices to autonomous driving using it. It is widely used for embedded devices, and it is running in a wide variety of hardware. Linux is a stable, reliable and complete computing platform when compared to other commercial operating systems available [WIKIPEDIA, 2018]. Linux, currently, is increasingly being used

in businesses as a back-end server. The number of applications for Linux is growing and have reached the critical mass where it is changing how we humans interact and communicate.

Characteristics that sets Linux apart are:

Multiuser Capability: a capability of Linux OS where the same computer resources – hard disk, memory, others; are accessible to multiple users. That means every user has its own command terminal. A terminal will consist of at least a monitor, keyboard, and mouse as input devices. Client/Server architecture is an example of the multiuser capability of Linux, where different clients are connected to a Linux server. The client sends a request to the server with particular data and the server responds with the processed data or the file requested, the client terminal is also known as a "dumb" terminal.

Multitasking*:* Linux can handle more than one job at a time, say for example a user has executed a command for sorting a huge list and simultaneously typing in notepad. It is managed by dividing the CPU time (cycles) by the implementation of scheduling policies and context switching.

Portability: Portability is one of the leading features that made Linux so popular among users, but portability does not mean that it is smaller in file size and can be carried on a flash drive or any portable storage device. Instead, portability means that Linux OS and its applications can work on different types of hardware in the same way. Linux kernel and application programs support installation even on very minimal hardware configurations.

Security: Security is a significant part of any OS, for organizations/users who use the system for confidential work. Linux does provide several security concepts for protecting their users from unauthorized access of their data and system. Linux introduces security concepts such as:

- **Authentication:** Assigning first level authentication such as passwords and individual user's login names to ensure that only the correct person can obtain access to their work.

- **Authorization:** At the file level Linux has authorization in the form of reading, writing and executing permissions for each file to decide who can access a file, who can modify it and who can execute it.

- **Encryption:** It encodes a file into an unreadable format that is also known as *ciphertext* so that its content will be safe even if someone succeeds in opening it.

Communication: Linux has an excellent feature for communicating between users, it can be within a single main computer network or between two or more computer networks. Users can exchange mail and data through networked machines.

In general terms the Linux System Architecture has the following layers:

- **Hardware layer**: Hardware consists of all peripheral devices (RAM/ HDD/ CPU).
- **Kernel**: A core component of the OS that interacts directly with hardware and provides low-level services to upper layer components.
- **Shell**: An interface to the kernel, hiding the complexity of kernel's functions from users. Takes commands from a user and executes kernel's functions.
- **Utilities**: Utility programs that give the user most of the functionalities of an OS.

The fundamental view of Linux architecture can be summarized in two levels: user and kernel spaces as shown in Figure 2.1.

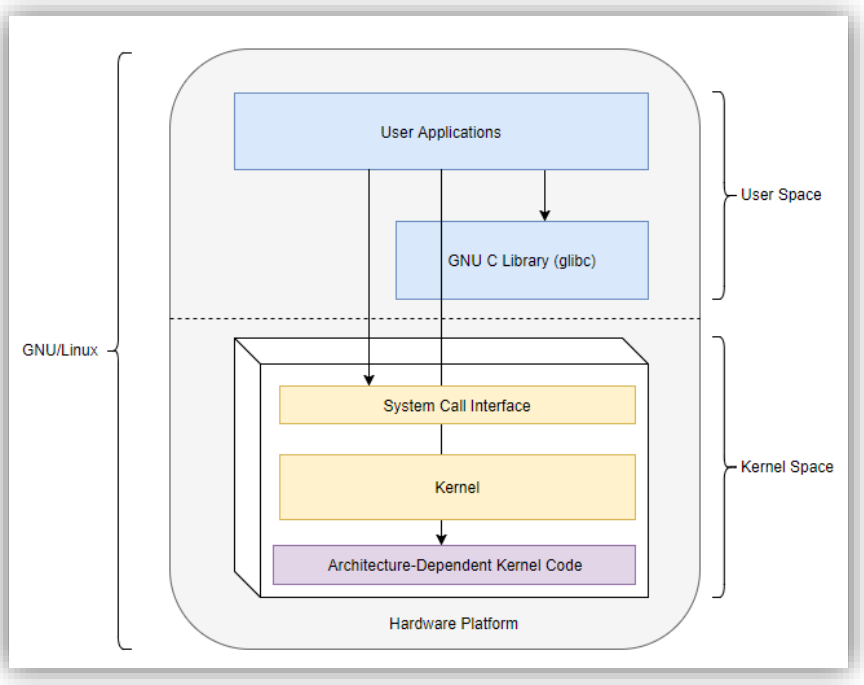

Figure 2. 1 Fundamental Linux Operating System Architecture.

A more fine-grained view of the architecture can be seen in Figure 2.2 below, where applications can reside in user space but also other programs can run in the kernel.

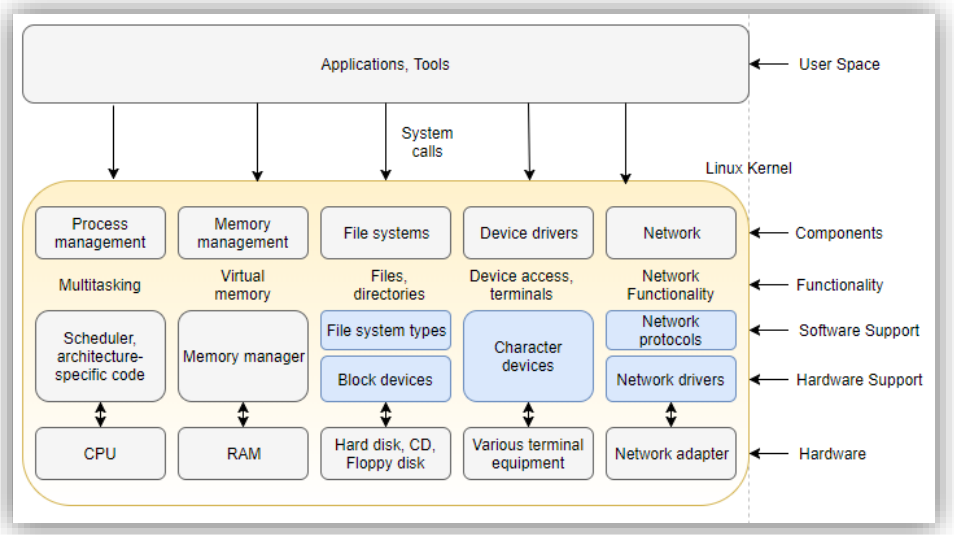

Figure 2. 2 Linux Operating System Architecture.

2.2. Data Transmission

Nowadays is impossible to imagine the Internet without the Transmission Control Protocol/Internet Protocol (TCP/IP) which is widely known as connection-oriented. All network services that have been developed use TCP/IP. Understanding how data is transferred through the network is fundamental if one wishes to improve data transmission performance.

2.2.1. TCP/IP Characteristics

TCP/IP was designed as a protocol to transmit information quickly while maintaining order in the data and without losing them on the way. Below the main characteristics of the protocol:

- **Connection-oriented:** First, a connection is made between two points (local and remote) and the data is transmitted. The TCP identifier is a combination of addresses between these two points having the following information in a flow: `<ip_src_addr, prt_src, ip_dst_addr, prt_dst>`
- **Bidirectional byte flow:** Bi-directional communication is done using a byte stream.
- **Order in the delivery:** A data receiving point receives the information in the order in which the issuer sent it. For this, a 32-bit integer is used.
- **Reliability:** The sender must receive a response known as acknowledge (ACK) from the receiver after sending the data.
- **Data control:** An issuer sends the largest amount of data that the receiver can handle. The receiver sends to the sender the number of bytes it can receive (size if used from the buffer, time window).
- **Congestion control:** A congestion window is used to prevent the network from becoming congested. Algorithms are used as TCP Vegas, Westwood,

BIC, and CUBIC. Normally these algorithms are implemented on the issuer side.

Figure 2.3 shows the structure of a TCP frame that consists of the above characteristics. [WIKIPEDIA, 2018]

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Figure 2. 3 TCP header structure.

2.2.2. Data Transmission

For the transmission of data, it must pass through several layers from the OS, as it can be seen in Figure 2.4 [SCHULTZ, 2011].

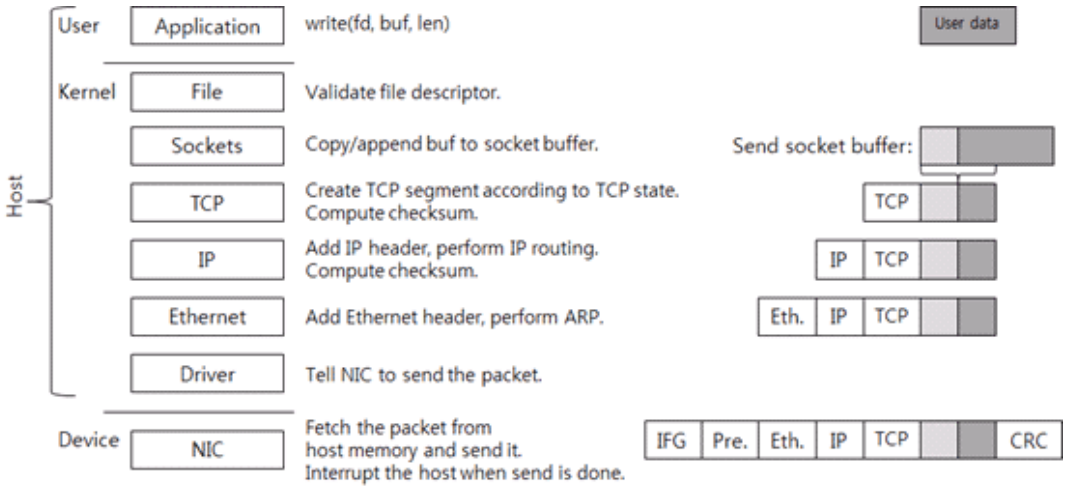

Figure 2. 4 Data packet transmission through TCP/IP layers from kernel application.

In general terms, it is very important to note that these layers are divided between the following two areas: user space, kernel space.

The system CPU performs most of the tasks that are executed in the user space and the kernel. In an Embedded System, some tasks must be executed in the space of the kernel but are mainly performed by a software driver of a network device or a specific device instead of the CPU. That can be a System on Chip (SoC) that is responsible for sending and receiving network packages through the medium --cable or air.

In Figure 2.4, it is observed that an application creates the data to be sent and a system call is made through the `write()` function. In addition to this, previously in the OS, it creates a File Descriptor which is a socket (a type of file in the OS). When the system call is made, basically what is done is a context switch area, now moving to the kernel space.

The socket in the kernel has two buffers: one that is `send_sckt_buff` and one for reception `rcv_sckt_buff`. When a system call is made, the user space data is copied to the memory used by the kernel space and is added to the end of the `send_sckt_buff` to send the information in the correct order.

There is a data structure (`struct`) connected to the socket for memory control block called TCP Control Block (TCB) that includes the data required to process a TCP connection. The data in that structure are the status (`LISTEN`, `ESTABLISHED`, `TIME_WAIT`), reception window, congestion window, sequence number, re-shipment clock, among others.

The data (payload) include information that is stored in the `send_sckt_buff`. The maximum size of the data are values given by the previous values in the TCP data structure. The sum code for error verification is calculated, and the frame is sent to the Internet Protocol (IP) layer that is responsible for adding the routing information that the packet carries. After this, the data packet sent requests the Network Interface Card (NIC) card.

When a data packet is sent or received, the NIC generates interrupts (Message Signaled Interrupts, MSI) to the CPU. Each interrupt has a `msi_id` interrupt number that the OS serves with priority so that a callback function is recorded in the code to handle the interruption.

2.2.3. Data Reception

Now, to know how data is received it is necessary to observe Figure 2.5 [SCHULTZ, 2011]. Reception is the process that is executed when a package enters the kernel space.

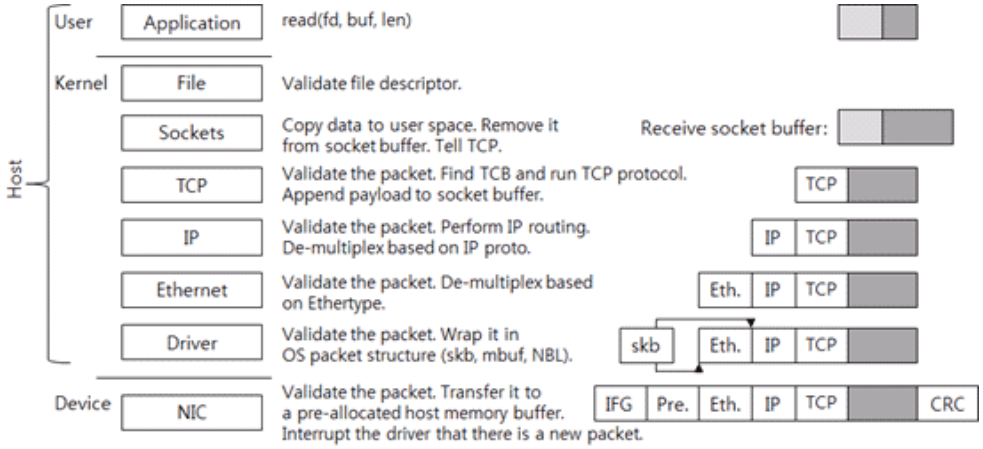

Figure 2. 5 Data packet transmission through TCP/IP layers to kernel application.

As it is observed, first the data packet arrives at the NIC in charge of validating the package by code correction of errors and sends the package to a buffer in the Kernel memory space which is a structure in the OS generally `skb_buff`. It is important to note here that the NIC can reject a package depending on several factors defined by the configuration of the card.

Using the data in `skb_buff` is how the information is carried through the IP and TCP layers. The latter is where the source IP address with its respective port is evaluated and where the packet with the destination IP and port is going. Thus, by

the `rcv_sckt_buff` corresponding to a file descriptor, the data is copied to the user space.

2.2.4. Interruption Process When Receiving Data

The process that is performed in a CPU-level interruption is complex. However, all that is needed to understand it is to identify a differentiator or determinant regarding efficiency when processing data packets in an embedded system. Figure 2.6 [SCHULTZ, 2011] shows the process that is performed in an interruption at the function level in the OS.

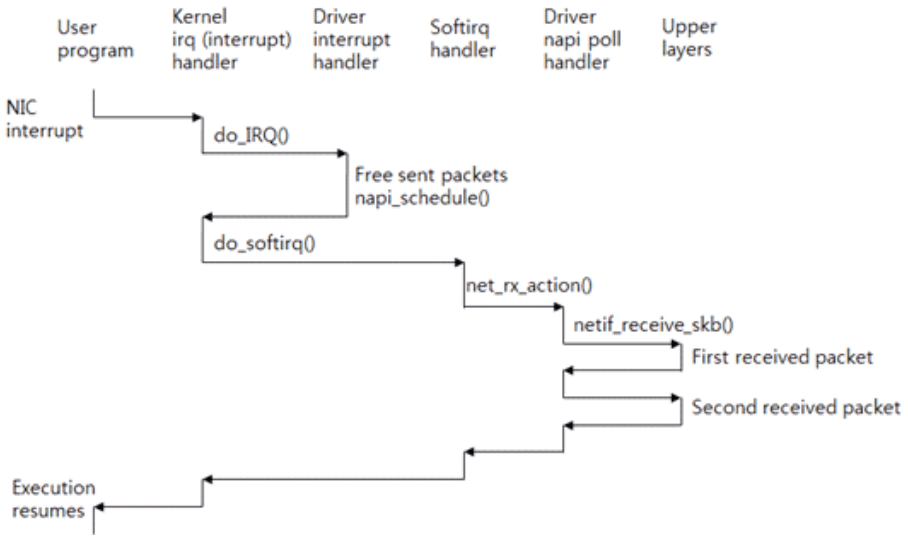

Figure 2. 6 softirq interruption process when receiving data.

Assuming that a program - in the user space - is being executed on the `CPU[0]`, at the moment when the NIC receives a packet and generates an interrupt for the `CPU[0]`; the function `do_IRQ0()` that handles the interruption in the kernel (called irq) is executed. The handler uses the unique `msi_id` interrupt number and then calls the `napi_schedule()` function to process the received packet. This last function calls the interrupt `do_softirq().` Thus the softirq function context is executed in a different but similar thread while blocking any other software

interruption but any other interruption by hardware is kept open (non-maskable interruption).

Then, the reception of the package is handled through the `net_rx_action()` function. This function calls the `poll()` request function which in turn calls the `netif_receive_skb()` function that sends the received data packet one by one to the user space.

It is important to note, as a point of interest for the investigation, that an interruption to the system takes place for each packet. This means high and intensive CPU use to send data from one space to another in an OS. As a testing method, it is normal to check that the CPU is running softirq and its memory usage, especially the Resident memory (Resident Set Size, RSS).

2.2.5. PF_PACKET Protocol

When a socket is opened with the standard call `sock=socket(domain, type, protocol)`, the domain (or protocol family) to be used with that socket must be specified. The commonly used families are PF_UNIX, for local communications on the same machine and PF_INET for communications based on IPv4 (PF_INETv6 for IPv6). Also, it must specify the type of socket and the possible values depending on the previously selected family. Some common values for the socket type when using PF_INET, for example, can be `SOCK_STREAM` (typically used with TCP) and `SOCK_DGRAM` (associated with UDP). The socket type influences how the kernel handles packets before being passed to the application. In this way, it is also a must to specify the protocol that will manage the packets.

In recent versions of Linux kernel (after 2.0) a new family of protocols or domain known as PF_PACKET was introduced. This family allows an application to send and receive packets dealing directly with the network card driver, thus avoiding the usual administration of the kernel network stack (IP / TCP or IP / UDP processing). That is, each packet is sent through a socket and goes directly to the Ethernet

interface, and any packet received through the interface goes directly to the application in the user space [INSOLVIBILE, 2001]

The PF_PACKET family supports two types of sockets: `SOCKET_DGRAM` and `SOCKET_RAW`. The first allows the kernel to add and remove the Ethernet headers while the latter allows the application to take over the complete control of the network header. The protocol field in the `socket()` function must match the Ethernet identifiers defined in the `</usr/include/linux/if_ether.h>` a library that represents all registered protocols that can be sent in an Ethernet frame [INSOLVIBILE, 2001].

From Linux documentation: *"Packet sockets are used to receive or send raw packets to the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.*

The socket_type is either `SOCK_RAW` for raw packets including the link level header or `SOCK_DGRAM` for cooked packets with the link level header removed. The link level header information is available in a common format in a sockaddr_ll. The protocol is the IEEE 802.3 protocol number in network order. The `<linux/if_ether.h>` include file for a list of allowed protocols. When the protocol is set to htons (`ETH_P_ALL`), then all protocols are received. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel."

2.2.6. PF_RING protocol

In simple terms, PF_RING allows packets on a single interface to be segmented across multiple threads or cores, allowing for more efficient packet processing. Data packets are inspected at a much lower level than traditional packet sniffers and engines, therefore reducing resource cost and increasing overall efficiency.

Passive data packet capture is necessary for many activities including network debugging and monitoring. With the advent of fast gigabit networks, packet capture

is becoming a problem even on PCs due to the poor performance of popular operating systems. The introduction of device polling has improved the capture process quite a bit, but it has not solved the problem. PF_RING proposes a new approach to passive packet capture that, combined with device polling, allows data to be captured and analyzed using the NetFlow protocol at almost wire speed on Gbit networks using a commodity system or standard OS.

It is a new logic for data packets exchange that dramatically improves the capture of network packets and is mainly characterized by the following:

- It is possible to implement it in the latest stable Kernel versions of Linux. Available for Linux kernels 2.6.18 and higher.
- It is independent of the drivers, network card drivers, or the Internet Protocol.
- It works in the Kernel space.
- It allows to specify filters and to use Berkeley Packet Filters (BPF) [BERKELEY, 2018].
- It provides content inspection as described for DPI.
- As of version 4.X, PF_RING can be used with vanilla kernels (i.e., no kernel patch required).
- PF_RING-aware drivers can increase packet capture acceleration.
- It works for 10 Gbit Hardware Packet Filtering using commodity network adapters.
- User-space DNA (Direct NIC Access) drivers for extreme packet capture/transmission speed as the NIC NPU (Network Process Unit) is pushing/getting packets to/from user space without any kernel intervention. Using the 10Gbit DNA driver, user can send/receive at wire-speed any size packets.

- Libzero for DNA for distributing packets in zero-copy across threads and applications.
- Kernel-based packet capture and sampling.
- Libpcap support for seamless integration with existing pcap-based applications.
- Ability to specify hundreds of header filters in addition to BPF.
- Content inspection, so that only packets matching the payload filter are passed.
- PF_RING plugins for advanced packet parsing and content filtering.
- Ability to work in transparent mode (i.e., the packets are also forwarded to upper links so existing applications will work as usual).

It is a prominent replacement to PF_PACKET and was introduced in 2001 by Lucas Deri [DERI, 2001]. Deri found that the Linux network stack introduces several bottlenecks that cause loss of data packets when transmitting packets from the medium [BRAUN, 2010]. The proposed new architecture was developed to eliminate bottlenecks, especially when the size of the data is small, what causes many interruptions to the CPU, context changes, memory sharing, among other consequences, as it was mentioned above. PF_RING is a modification that allows copying packages in a ring, completely forgetting to use the standard logic of the Linux Kernel.

PF_RING differs from PF_PACKET mainly in the number of steps for the exchange of data between the user space and the Kernel. Figure 2.7 [BRAUN, 2010] shows this difference in detail.

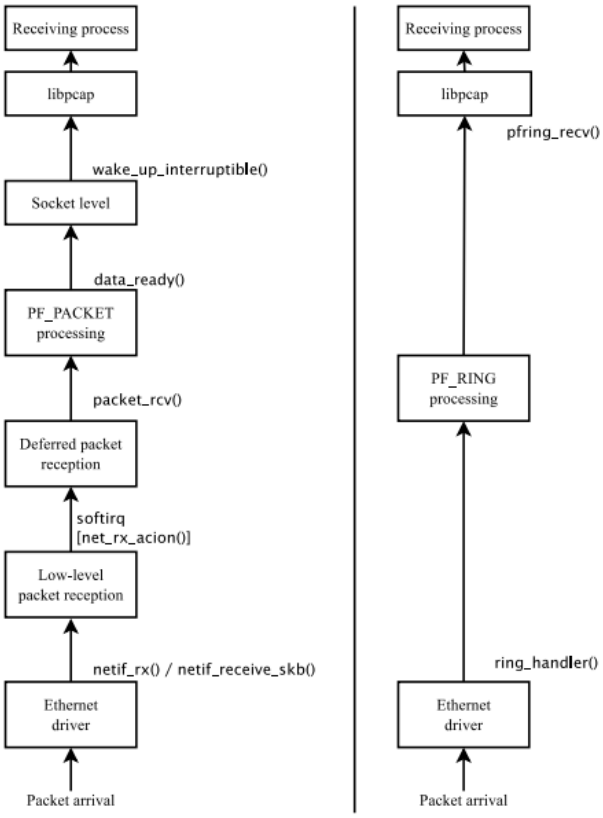

Figure 2. 7 Data flow differences between PF_PACKET and PF_RING.

As shown, the exchange of data with PF_RING requires fewer steps, system calls or processor cycles, which suggests better efficiency in terms of using OS provided resources.

Its architecture uses kernel rings (sockets) to exchange data between user space and kernel as shown in Figure 2.8 [DERI, 2001].

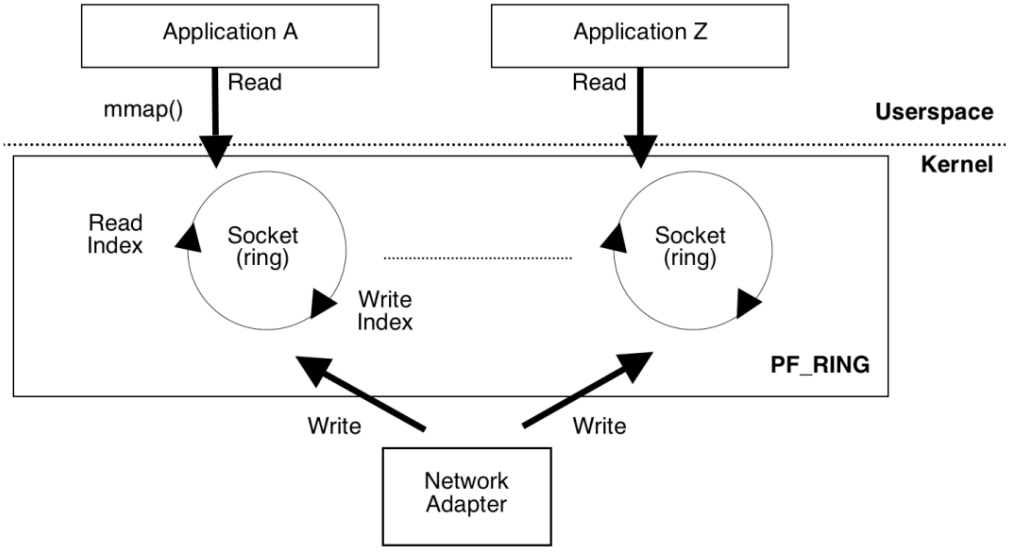

Figure 2. 8 Ring buffer socket in PF_RING architecture.

The advantages of a ring buffer located into a socket are:

- Data not queued into kernel network data structures.
- `mmap` primitive allows user space applications to access the circular buffer with no overhead due to system calls as in the case of socket calls.
- Even with a Kernel that does not support device polling under strong traffic data conditions, the system is usable because of the limited time necessary to handle an interrupt compared to regular data handling.
- Packet sampling is simple and effective when implementing, as sample data do not need to be passed to upper layers then discarded as it happens with conventional `pf_packet` applications.
- Multiple applications can open several `pf_ring` sockets simultaneously without cross-interference (slowest application does not affect fastest

It is important to note that applications need to be re-compiled or be `ring/mmap-aware`.

2.3. Performance Benchmarks

It is necessary to perform measurement tests to evaluate the performance of one algorithm over the other. In a built-in system many variables and situations can occur; therefore, it is necessary to reduce as many as possible unnecessary factors.

2.3.1. Factors that Affect Performance

Among the factors that can affect performance benchmarks are:

- The driver of an embedded system NIC card.
- The type of embedded system processor, 32b or 64b.
- The bandwidth of the memory in the system.
- Kernel version.
- Kernel configuration.
- The amount of data that you wish to send to the embedded system.
- The size of the packets and the frequency of arrival of the data.

Clearly the hardware and the card handler can be an important factor when capturing data from the medium. For this research, the factor is isolated or dismissed since the embedded system is ideally modeled with the best hardware specifications. Also, the idea of using PF_RING is to isolate from this factor and take it into account as a dominated factor. Also, as a value added to isolate hardware, a hardware simulator is used that will allow to inspect the problem without depending on hardware factors.

The speed of the processor and the architecture can be decisive when processing packages since the ability to perform as many instructions in the shortest possible time when receiving a package must be considered. This factor is discarded since

it is assumed that the processor and architecture model will have the best specifications for the tests. Also, it is expected that with new architecture models in a virtual platform, the results will depend on the focus of the research in PF_RING.

The speed with which the NIC and the processor can access memory for writing and reading is important for each packet as this eventually affects performance. This factor is important, and part of the research consists in showing that when applying the PF_RING technique the way in which the packages are copied results in an improvement over PF_PACKET.

On the software implementation side, the Linux Kernel version impacts performance because the version in the embedded system may be old while new versions are constantly being developed. However, it is a controlled factor because the tests are going to be performed with a recent stable kernel. Thus, for the tests, this factor is obviated in the investigation since it is not a dependency given that what changes is the implementation factor of the algorithm in the Kernel. The way in which the Kernel of the system is configured can affect the results since it can cause unnecessary overload causing the system to react slower. It is assumed that the Kernel is compiled with a standard configuration and an improvement in the configuration can be evaluated during the investigation. Also, as part of the investigation, it is possible to find errors in the NIC driver embedded in the modeled device, so this can cause inefficiencies that are beyond the scope of the project.

Other factors that are not controlled when measuring the effectiveness of an improvement can include the load handled by the system and the variety of captured packages. There must be a sufficiently varied set of packages to simulate the implementation of the research in a normal use environment. Also, as mentioned, a quite important variable factor is the size and frequency of data used to obtain the results. Depending on the type of packet, it is possible to simulate 1.4 million (64 bytes) of data per second or as few as 81000 (1522 bytes) per second. RFC2544 provides the definition of a methodology that helps define the number of

packets, the size, and the frequency and even suggests experiments to make appropriate measurements.

On this research, certain data flows of a certain size are defined in such a way that enough consistency is obtained in the results to make a simple comparison (following the provisions of RFC2544). In the same way, the idea is to take data about real packages and explore the behavior of PF_RING in an OS.

The objective of this study is to find the best technique for passing packets from the kernel space to the user space (even vice versa) that results in the best performance regarding the use of resources such as memory, system calls, and interruptions to the CPU. It can be measured in Megabits per second (Mbps) or the number of data packets per second per Gbps (pps/Gbps). It should take into account the minimum size of an Ethernet packet that includes 7 bytes of pre-scope, 1 byte for the delimiter, 64 bytes of the Ethernet frame and the frame separator (12 bytes). Also, the historical data of CPU utilization is used.

2.4. Experimental Design

The environment setup for the research needs to be performed on a platform that enables experimentation. Experiments can be placed in two of three possible pathways: pf-packet and pf-ring. Figure 2.9 [SCHULTZ, 2011] details the path that a packet takes from a NIC in the Kernel space to the user space.

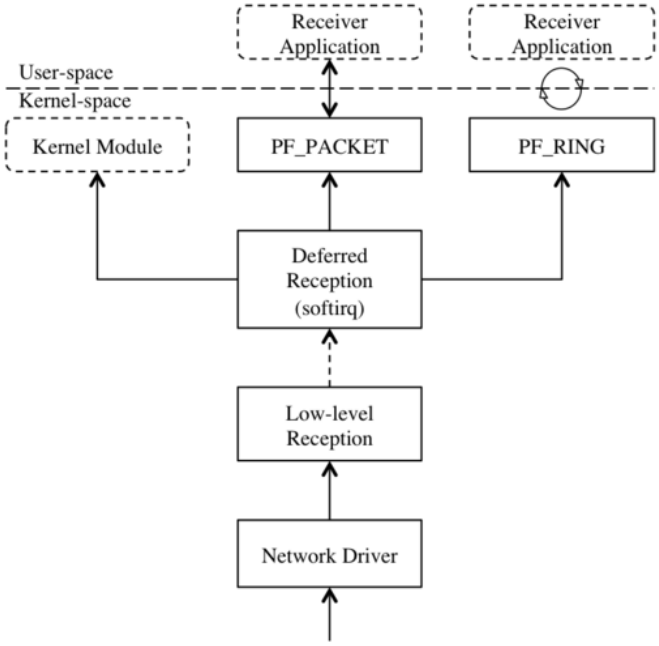

Figure 2. 9 Pathways of an incoming data packet through the OS from Kernel to the application.

An OS is built as a binary with a specific configuration that supports pre-defined hardware and software platforms and applications. A workspace with an OS “Build A” and “Build B” is set to be able to support a specific real hardware platform. More interestingly, a binary “build A” and “build B” of an OS is built to be executed in a simulation environment. “Build A” is PF_RING OS enabled and “Build B” is PF_PACKET OS enabled. Figure 2. 10 There are no other differences between builds except the implemented algorithm so they are comparable.

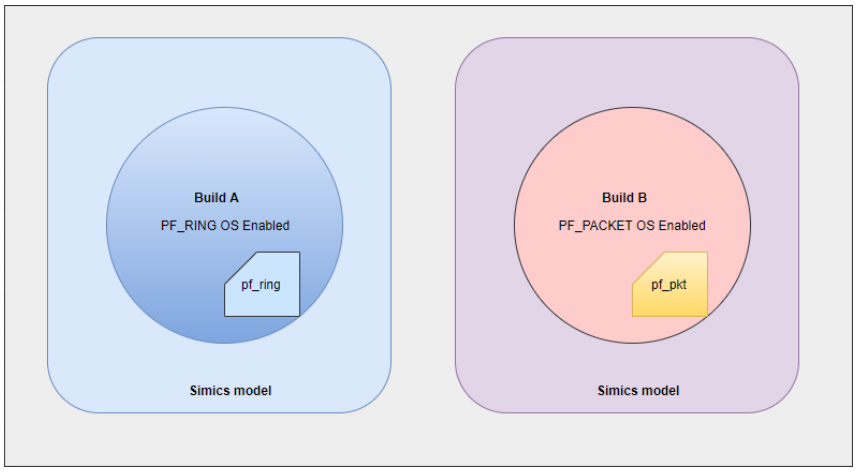

Figure 2. 10 Operating System “Build A” and “Build B” differences.

What is more important in the workspace is the creation of a kernel module in an OS that runs in a Full System Simulation tool running on top of a hardware model that implements the PF_RING technique instead of PF_PACKET. That is, the data packets are no longer going to be treated by the regular algorithm of a Linux Kernel module directly for data handling, nor by PF_PACKET but by a module that implements the PF_RING logic.

Additionally, an application that interacts with the PF_RING ring will be implemented in the user space, receiving packets and sending packets to the Kernel module through the libpcap library [see the project at http://www.tcpdump.org/] or an implementation that even allows to omit it. In this way, it is expected to be able to perform measurements such as speed (in Mbps) or "throughput" (in pps / Gbps) in the copy of packets from one space to the other in the user/kernel areas of the OS.

2.4.1. Factorial Experiments

On multiple occasions, it is interesting to know the influence of two or more factors on a response variable. For example, in the study of the behavior of a computational algorithm it is interesting to know if the influence on the state of two

parameters affects the speed to find the response variable that calculates the mentioned algorithm. In this type of case, it is appropriate to use a factorial experiment, which means that each treatment is defined by the combination of the factors between the input parameters of the algorithm.

Factorial experiments are defined as those in which two or more main factors are compared or studied simultaneously, including different levels or modalities of each factor. Normally, variance analysis is used as a statistical technique to analyze the effect of two or more independent variables (factors) on a response variable.

In factorial experiments, treatments are formed by combining each level of one factor with each of the levels of the other (or of the others, if there are more than two). This type of experiment also makes it possible to evaluate the effects of the interactions. It is said that between two factors there is interaction if the effects of a level of one factor depend on the levels of the other. In other words, the response of one factor is influenced differently by the levels of the other.

The existence of interactions indicates that the effects of the factors on the response are not additive and therefore the effects of the factors cannot be separated.

Factorial Experiment Concepts

Factors are characteristics that involve two or more different modalities, variants or levels [LINCOLN, 2014] and can be:

- **Qualitative**: Those in which the levels define or express a particular modality of the characteristics of the factor; each level has an intrinsic interest or is independent of the other levels. These factors respond to the characteristics of the qualitative variables. E.g., different types of packages (TCP, UDP, etc.).

  Factor: variety of packages (V), Levels: {v1, v2, v3, ...}

- **Quantitative**: Those whose values correspond to numerical quantities, that is, values inherent to a quantitative variable. Ex: quantity of packages.

  Factor: packages (N), Levels: {n0, n1, n2, ...}

To symbolize the factors, the use of the capital letter linked to the name of the factor and that letter (which can be uppercase or lowercase) with a numerical subscript for the levels has been generalized. It is also possible to use a capital letter for the factor and other letters for the levels that replace the names.

In a factorial experiment, the treatments result from the combination of the levels of a factor with the levels of the other factors. For example, if 3 N factors are combined with two levels (n1 and n2), the resulting treatments are N1n1, N1n2, N2n1, N2n2, N3n1, N3n2.

The interesting thing about the factorial experiments is that they can be applied to different designs: completely randomized, blocks, latin-squares. The complete Factorial Experiments include - for balancing reasons - all possible combinations between the different levels of the factor involved in the experiment. For example: if we assume the simplest factorial experiment: two factors a and b, each with two levels 1 and 2, we obtain: a1 and a2; b1 and b2. The possible combinations are four. And the respective treatments are identified as a1b1, a1b2, a2b1, a2b2. Thus, the structure shown in Table 2.1 can be built.

Table 2. 1 Two by two factorial experiments

	a levels		
b levels		a1	a2
	b1	a1b1	a2b1
	b2	a1b2	a2b2

As the number of factors and levels increases, the number of treatments increases significantly and with it the difficulty of choosing the appropriate design.

Factorial experiments generally provide more complete information than common experiments since they allow the study of the main factors, the combinations of all the levels and the interaction of the factors. In factorial experiments it is common to talk about "treatment structure" indicating that treatments are formed by combinations of factors [LINCOLN, 2014].

Interaction is the reciprocal effect of 2 or more factors or the modification of the effect of one factor by the action of another or others. The study of the interaction between the factors is one of the important characteristics of factorial experiments.

The possibility of joint studies of two or more factors with their corresponding levels makes factorial experiments very useful for exploratory research and as a previous step to subsequently concentrate attention on the aspects that may be of greater interest, according to the general conclusions that these experiments provide.

Among the advantages and disadvantages are [LINCOLN, 2014]:

• They allow the simultaneous study of two or more factors

• They allow studying the possible interaction between the factors involved, and consequently the effect or behavior of each factor in the different levels of the other factor.

• They are more efficient than simple experiments, where only one factor is studied.

• They also provide general results that make them useful in exploratory experiments.

• Since all the possible combinations of the different levels are included, they usually provide a large number of degrees of freedom for the experimental error, with the consequent advantage that this means.

In contrast to the above, as the number of factors and levels increases, the number of treatments increases for the whole experiment and in particular for each

repetition. With all this, the difficulty of adapting the most appropriate design to the experimental material increases and the cost of each repetition rises significantly.

Although not all the combinations between the different levels are of interest to this research, some of the experiment factors cannot be excluded for balancing reasons that the analysis requires [LINCOLN, 2014].

2.4.2 Statistical Hypothesis Testing

The purpose of statistics is to test a hypothesis. An experiment executes a series of reproducible steps to obtain coherent results. A Statistical Hypothesis or confirmatory data analysis is a hypothesis that is testable based on observing a process that is modeled via a set of variables. It is a method of statistical inference [HANZE, 2001].

Experimental data sets are compared, or any data sampling is compared, against a synthetic data set from an idealized model. A hypothesis is proposed for the statistical relationship between the two data sets, and this is compared as an alternative to an idealized null hypothesis that proposes no relationship between two data sets. A comparison is statistically significant if the relationship between the data sets would be unlikely close to the null hypothesis after calculating the probability. Hypothesis tests are used in determining what outcomes of a research would lead to a rejection of the null hypothesis for a pre-specified level of significance. The process of distinguishing between null hypothesis and alternative hypothesis is aided by identifying two conceptual types of errors and selecting parametric limits for these errors. One type of error is rejection of a true null hypothesis (false positive) and the other type is retaining a false null hypothesis (false negative). [HANZE, 2001].

The *p*-value is the probability that a given result (or a more significant result) would occur under the null hypothesis. For example, say that a fair coin is tested for

fairness (the null hypothesis). At a significance level of 0.05, the fair coin would be expected to (incorrectly) reject the null hypothesis in about 1 out of every 20 tests. The *p*-value does not provide the probability that either hypothesis is correct (a common source of confusion) [NUZZO, 2014].

2.5. System Simulation and System Emulation Technologies

2.5.1. Emulations vs. Simulation

When a system mimics an observable behavior to match an existing target, it is called emulation. Emulation does not accurately reflect the internal state of the system being emulated.

On the other hand, simulation requires modeling the underlying state of the target system as accurately as possible without affecting simulation speed. The result of a simulation is a model that describes system behavior and allows lets to observe its internal state. Ideally a simulation can look inside and observe properties that are possible to get out of the original target. In a real model simulation there are logical shortcuts taken for performance reasons, but at the end, the simulation is trustworthy enough to do experimentation that would be impossible to perform with an emulation.

2.5.2. What is SIMICS?

Simics has a direct impact on the product development process shifting time-to-market and improving quality. It enables software development to be early implemented in a virtualized platform enhancing the overall process of producing enterprise-ready software. Simics by itself is a very interesting simulation technology.

Simics is a simulator – a full system simulator program simulating a set of pieces of hardware working altogether. A virtual platform is a solution running in Simics

providing a virtual hardware solution as running software. It virtualizes embedded hardware in a different way that hypervisors do.

A hypervisor will expect an OS to target a particular virtual machine architecture, while a Simics virtual platform is a software simulation of a set of pieces of hardware logically connected working altogether. Simics is a tool for developing software substituting or eliminating the use of hardware. On the other hand, a hypervisor is a way to manage hardware at runtime. A hypervisor can run on top of a Simics simulation, or a hypervisor can be developed using Simics [WIND RIVER, 2010].

Simics has simulated from basic embedded boards with a single processor or SoC all the way up to servers or even clusters. A virtual platform is a model sufficiently complete and correct to fool the target software into believing it is running on real physical hardware and it is fast enough to be used for regular software development tasks. A “target” in a virtual platform is a model of the hardware being simulated.

From a software perspective, Simics simulation is like hardware. A Simics setup can load the same binaries that would be used on a physical target board or system, execute those binaries practically in the same way they would be executed in real hardware. A software stack includes everything from initial boot code to hypervisors, operating system, user-level application code. If a model is complete enough, there is no need to modify the target code to run on Simics.

Simics is a full software program, nothing else. It does not require special hardware, boards, or emulators to simulate. Simics runs on any personal computer, anywhere, at any time. Simics available binaries can run on Windows or Linux host machines. It is possible to send a Simics model anywhere across the globe even by e-mail. One of its main features is that it can replace hardware

boards for any global development team. Refer to Figure 2.11 for Simics running VxWorks overview [ENGBLOM, 2010]

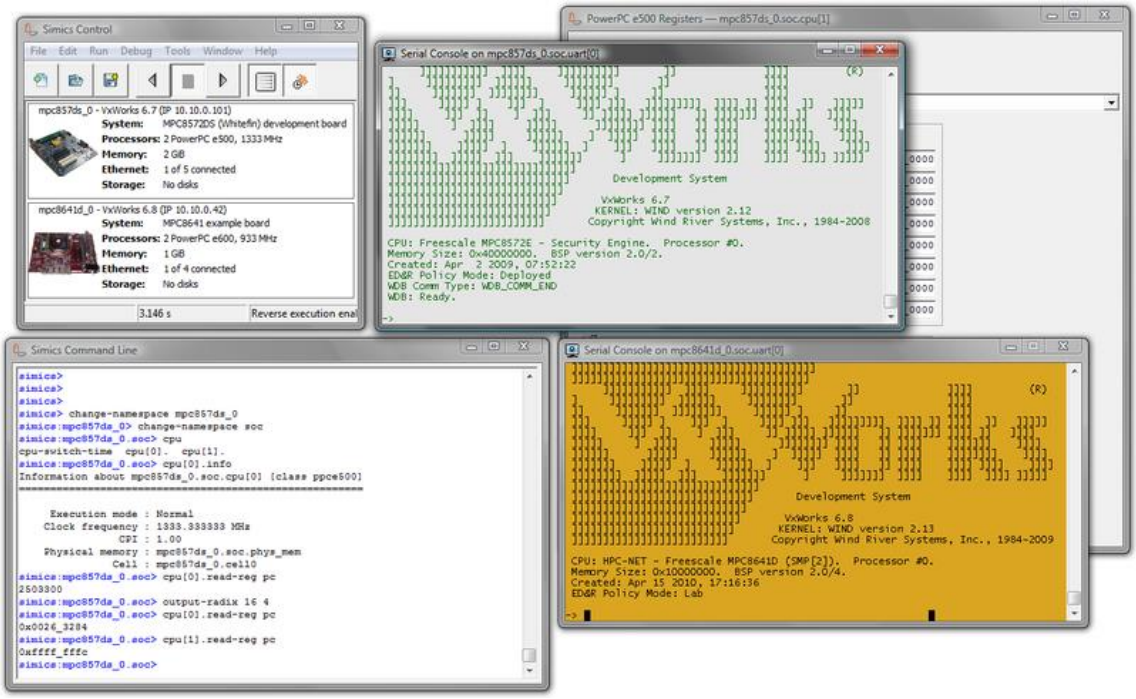

Figure 2. 11 Simics is running VxWorks Operating System.

Simics is a simulation of a target, board or any system hardware. The Virtual Platform contains models of processor cores, buses and other interconnections between devices. It contains memories, peripheral devices, and networks.

The main devices are the fast instruction-set simulators (ISS) which are used to execute ARM, DSP, MIPS, Power Architecture, SPARC, x86/IA or other processors in binary code. A full set of other devices modeled to let run an OS is also necessary. A full simulation on Simics includes models for memory-management units (MMU) and all the memories and devices found in the memory maps of the processors. Refer to Figure 2.12 for Simics architecture overview [ENGBLOM, 2010]

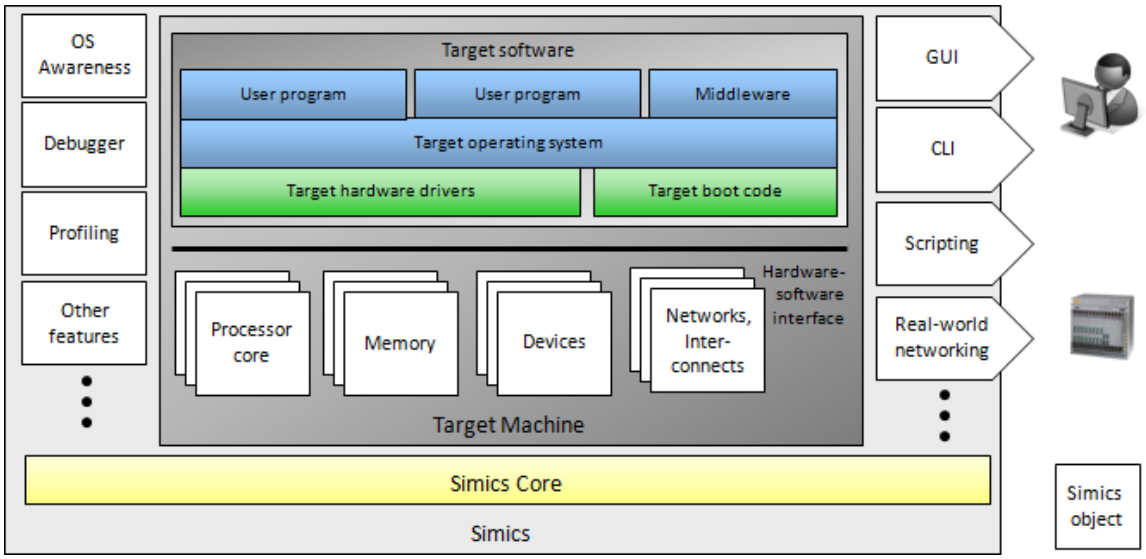

Figure 2. 12 Simics architecture overview.

When a microprocessor does a memory operation, a memory load or storage is performed and the access address is first translated through MMU getting back to a physical address. This physical address is used to build a memory transaction in Simics. The transaction is sent to a memory map, which determines what specific part of the target system the transaction is targeting. If it hits the memory, the memory content is read or modified depending on which operation comes with the transaction.

When a transaction hits a peripheral device, the simulation model for that device is called to determine the effects of the access. The device model has the logic in place to work on the access effect. It might send an interrupt to the processor, reconfigure the hardware registers, reset a board, set or start a timer, send a network packet, drive an IO pin or any other condition. When a transaction does not hit anything, the simulation gets an exception back to report the bus error to the software.

As a software application on a host machine, all can be summarized as a function call that calls other functions between the objects building up the simulation model. On Simics, everything is an object that uses functions to expose communication

for transactions related to the simulated system (like memory operations), internal communication in the same device model, maintenance, logging, tracing or other operations. In hardware, this style of simulation is known as transaction-level modeling. It is mainly focused on moving data around transactions, not on clocking, pins or low-level hardware concepts.

As it is designed, Simics can add new objects to a simulation at any given point in time. Objects can be reconfigured on how they are connected and their properties at run-time modified. It is one of the advantages of using a simulation to develop software or run experiments rather than a real system. Its flexibility makes it possible to simulate any hardware system and operations like adding or removing boards from a rack or changing network cables between boards.

Objects are built from classes loaded at run-time. Every simulation model is stored in its own .dll or .so file and can be loaded while the program simulation is running. There is no need to recompile anything to create a new configuration; all that is needed to load the required modules when Simics is running is already stored. For example, this is very similar to a virtual machine on Java or .net environments on how objects are compiled, loaded, connected (interfaces) and managed. It is not similar to static linking programmers use in C or C++ programs. Every change in a module can be compiled separately making it very easy to recompile after a change in a simulated target. It makes it easy to use Simics for experimentation and software development.

Device modules on Simics can be created in C, C++, Python or the Simics device modeling language DML. DML is a C-code generator that makes it easy to create skeleton code to build complex device models. Any other language can also be used as long as it can be linked to a C-language module.

To interconnect and manage modeled hardware, a virtual platform is created with a special object known as a component. It is logical to group hardware device models and make them work together. Components group devices, memories,

interconnects and processor cores as logical units corresponding to what we know as a chip, a SoC, ASIC, any board, racks or any scalable system. These objects can be reused and arbitrarily placed into another one, giving the ability to model any form of hardware design hierarchy. When the component hierarchy is studied, it is easy to understand the structure of a virtual hardware design.

Any processor, board, network, heterogeneous computer architecture and target OS can be modeled with Simics. Models can scale and even thousands of processors can be modeled, which makes it easy to configure experiment setups for research. A simulation can run even months of the target time. Heterogeneous simulations include different processor types and families: multi-core, single-core, from 8-bit to 64-bit, symmetric shared memory or asymmetric multiprocessing have been used to simulate. A model can be created by any user extending the hardware library available with any component object needed. The only requirement to build target models is to have Simics base binary and Model Builder.

On input/output (IO) to a simulated system, Simics has interactive sessions with serial consoles and graphical displays. It is also possible to connect simulated serial connections on Ethernet networks to real physical networks. The most common use cases are to keep target I/O with scripts of various forms of traffic generators. Compiled software binaries are the major form of input which Simics takes directly. There are many other domain-specific ways to develop Simics simulations.

Simics user-interface has a scriptable command-line that comprehends Python a simple GUI, an Eclipse GUI, and connections to debuggers such as gdb-serial. These interfaces make it simple to manipulate a target system and debug software running in a simulation without software being disrupted. It is the main advantage of Simics as development and experimental analysis tool.

Simics is built from a set of logically connected objects, including user-interface components, debugger connections, and command-line interfaces. Any object can use the Simics API and function calls into other objects as infinite possibilities to create. Therefore, it is very easy to extend, and it is flexible to experiment with many use cases. Any user with some programming knowledge can create modules and software that runs on those models. Simics capabilities on tracing, fault injection, debugging, remote control and even customer graphical user interfaces make it a great tool for research.

2.5.3. What is QEMU?

QEMU is a generic open source machine, user space emulator and virtualizer. QEMU stands for Quick Emulator. It is a software program which features allow it to emulate a complete machine in software without any need for hardware virtualization support. It uses a dynamic binary translation that allows good performance. QEMU can be integrated with Xen and KVM hypervisors to provide emulated hardware while allowing the hypervisor to manage the central processor unit. QEMU is able to run a hypervisor and can reach almost near-native performance for CPUs. It is a software that can run software on top of OS made for different machines (ARM or x86).

Figure 2.13 [QEMU, 2016] below represents a system architecture modeled in QEMU.

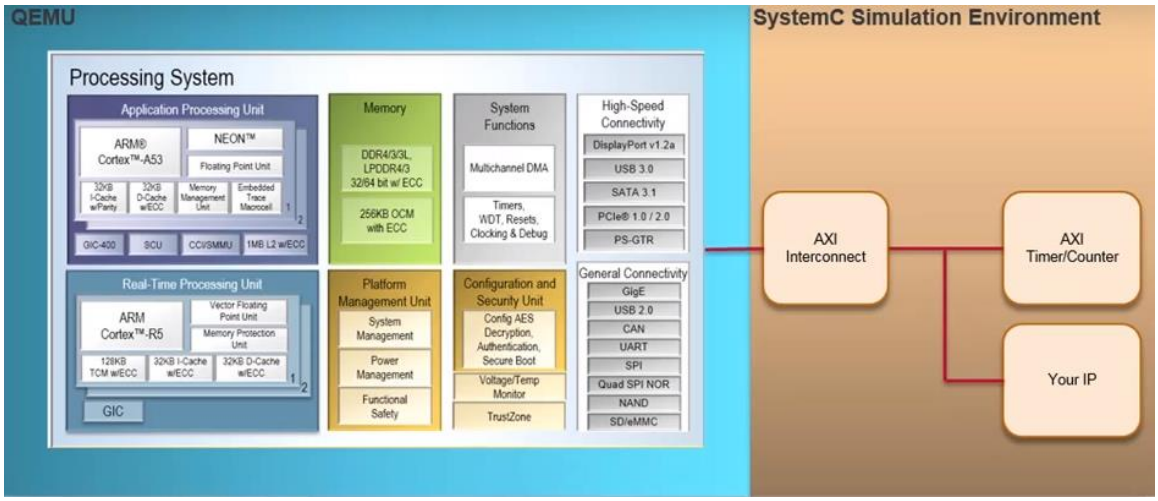

Figure 2. 13 QEMU system architecture

QEMU provides virtualization for Linux virtualization API. It allows binaries compiled against one architecture to be run on a host using a different architecture. It involves simple CPU and system call emulation.

QEMU is released under GNU General Public License, version 2. It is a multi-platform software that can be built in all modern Linux platforms, OS-X, Win32 or other UNIX targets. Its source code is maintained under GIT version control system: `git clone git://git.qemu.org/qemu.git`

QEMU has multiple operating modes: [QEMU, 2016]

- **User-model emulation**: it runs single Linux programs that were compiled for a different instruction set. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.
- **System emulation**: it emulates a full computer system, including peripherals. QEMU can boot many guest OS including Linux, Windows among others. Emulate different instruction sets.
- **Xen hosting**: QEMU emulates hardware, the guest execution is done within Xen and is hidden from Qemu.

Qemu can run a Linux Operating System on top of a Windows host machine as Figure 2.14 shows [QEMU, 2016].

Figure 2. 14 Linux is running on top of QEMU on a Windows host.

Chapter 3. Design

3.1. Problem Definition

The data transmission system at the OS level is computationally disadvantaged in keeping up with data networks or even same-system produced data rates. This is true for systems which are I/O bound that need to store, route or process data. As discussed in Chapter 2, there are efforts to apply performance improvements to get data from kernel space to user space and vice versa for analysis. Any data generated and transacted from one side to another is costly regarding system resources.

3.1.1. Hypothesis

In simple form, the objective of this research is to improve data transmission performance within an OS. There are several options and obvious approaches available. Top options include: 1) distribution of network traffic between multiple systems to offload data transmission, 2) increase hardware specifications (e.g., CPU speed, Memory size and speed, storage controller, storage media, etc.), or 3) identify opportunities to use existing hardware more efficiently. This research targets the latter, optimizations code will allow the first two options to scale to handle higher data throughput.

The removal of bottlenecks creates the opportunity to improve performance at every hardware and software level allowing user end applications to leverage higher performance without the cost of replacing existing hardware or supplementing it with additional more capable hardware.

Target improvements are:

- Reduce CPU utilization for data transmission within an OS.
- Reduce memory consumption of data exchange between kernel and user spaces.

- Reduce the rate of data packet drops under high data transmission.

Data transmission between applications residing at different levels in the OS is costly due to system calls; considering that, the goal of this research is to enable the PF_RING algorithm in the system to handle data transmission at theoretical maximum speed with minimum CPU and memory resource consumption, regardless of buffer sizes.

The hypothesis for the system using the PF_RING kernel space algorithm is that data transmission between applications is significantly greater with fewer system resources than when using PF_PACKET. It is hypothesized that the system with PF_RING with the same OS configuration as PF_PACKET can accomplish improved data transmission without impacting the OS negatively.

3.1.2. Approach

The following subsections provide a high-level roadmap of this research approach to the problem of transmitting data in an OS environment.

Solution Space

The first step was to determine the path for the bits from the point where a network card receives data in its driver's buffer to the point where an application takes it out of the buffer cache by reading it either from memory or directly by a storage controller. Ideally, a path is a data packet that starts from the network card and ends up in a disk or even CPU cache memory as the end point where an application can use it almost in no time, but this is beyond the scope of software programmability and more likely a hardware solution. To track every bit as it traverses through drivers, frameworks, APIs, queues, and applications is not an easy task. It is not even possible with closed source OS code or drivers. Current research uses an open-source OS with strong community support. The basis of the experiment uses a 64-bit version of Ubuntu 17. The open-source nature of Ubuntu Linux and community support provided the best possible chance to easily modify and plugin code. Linux Cross Reference (LXR) was an aid to find the path

data was traversing by allowing to navigate through the Linux kernel code in a completely cross-referenced environment. Most of the data transmission is well known in a Linux OS, thus the selection of Ubuntu for the research.

As mentioned in Chapter 2, a simulation environment decouples the OS and applications of any hardware or test board that allow to isolate and inspect experiment factors by break pointing into the code, reverse executing, fully pausing the simulation, registering and inspecting variables. Selection of Simics as a full system simulation using education licenses enables the use of hardware models in a way that they can be a controlled factor that is not determinant for the research.

Direction

Historically, performance has been achieved despite unnecessary layers of abstraction, context switches, and wasteful system calls. PF_RING points to a way to remove unnecessary bottlenecks between kernel and user space. The success of efforts done in this algorithm place the question “Could this algorithm be used instead of another to remove bottlenecks on data transmission between kernel and user space?”. The question leads to determine the most viable way to substitute PF_PACKET for PF_RING in an OS. Could data transmission within an OS via kernel memory mapping into the application space in the same way the upward copy is done perform better on a high data rate environment? The hypothesis is yes; however further investigation shows that memory mapping was already being used by the user-space libraries to get the most of the performance of the constrained interface between user-space and kernel-space [DIBONA, 2017]. Further attempts to optimize the existing use of memory mapping, combined with context switches of system calls, pointed to the use of better algorithms altogether. PF_RING functionality is relatively straightforward placing it inside the kernel, which means that any system call would be between kernel threads and more efficient.

Prototype

Kernel data transmission algorithm interaction is included in the design, development and comparative analysis of a PoC for this research. The goal of the experimental system is to demonstrate that using an algorithm that leverages on kernel-space capabilities improves the performance of data transmission between applications that reside at different spaces within the same OS. Performance metrics are discussed in Chapter 4, but it is important to understand that the rate of data messages dropped in transmission is a relevant metric. CPU utilization, memory utilization, and query response rates are also critical metrics although their importance gets reduced when data packets drop rate is high.

3.2. Moving to kernel-space

PF_RING kernel module builds on the ability to avoid penalties for a user-space to kernel-space copy and back, system calls and the overhead derived from context switches. PF_RING retrieves data packets from and to applications or network cards, provides filtering capabilities to retrieve data of interest, and writes the packets of interest to memory where applications make immediate use of them. Whether the application is TCPDUMP, Snort or a custom-built tool, each application provides fundamental capabilities to make use of algorithm features. Moving PF_RING into the kernel and creating an application that leverages from these features is a hurdle. The design and development of the application that uses the kernel module capability a not straightforward or community supported. An added value of this research is the use of the simulation environment to implement the integration of PF_RING on an OS kernel to observe detailed experimental factors.

3.2.1. Linux Kernel Modules

Documentation on understanding and implementing Linux kernel modules is widely available on published reference books and through the expert community. Linux kernel code has evolved vastly in recent years and approaches to OS

scheduling, threading, task management, interrupts and network I/O has changed on the recent mainstream code. A good amount of kernel code functionality is statically compiled. Device drivers, non-essential functionality, and other features are considered volatile and are provided as kernel modules. Linux modularity makes it easy to enhance OS functionality to meet changing industry demand.

The implementation of PF_RING protocol handler is an enhancement as a kernel module. Rather than changing the Kernel core or changing Linux kernel stack, PF_RING provides the capability to dynamically be inserted without altering standard behavior of the existing kernel core code or other user-space code that may depend on it. Putting this capability in place is part of this research providing dynamically loadable features preventing complex additional conditional logic to determine when to use the feature. It is used right when the module is inserted into the kernel and nonexistent when unloaded.

The capability above is key to the research experiments for implementing and evaluating kernel-level prototypes. The design and implementation of a kernel-level algorithm are performed by the PF_RING kernel module. The necessary flexibility to dynamically insert these capabilities into a plain vanilla Linux kernel is important to accurately measure performance analysis of a System Under Test (SUT) which will be discussed in Chapter 4.

3.2.2. Linux Kernel Threads

Kernel threads are background processes that act as agents for tasks that may typically be blocked. An example is a background process `bdflush` kernel daemon. It is responsible for the task of writing dirty pages from the buffer cache to the physical storage. `bdflush` is implemented as a set of threads that grow and shrink in a configurable minimum and maximum thread count boundary to match storage demand. The dynamic thread count behavior impacts the experimental design and performance evaluation detailed in Chapter 4 and Chapter 5 respectively. Any write to a physical device as drivers can be blocked, kernel

capabilities to respond to such situations are implemented either very carefully as a tasklet or as a kernel thread. A tasklet uses a `softIRQ` that might have scheduling pitfalls where the code is running with a high priority. An application written for this research has a function specifically tasked to read and write from a buffer on a ring.

On Linux there are different ways to create a kernel thread. For this research, the choice is to use `kthread_run()` macro. It creates a daemon and wakes up the thread. A daemon, in simple terms, removes all file descriptors from the thread (`stdin`, `stdout`, `stderr`), all signals are disabled, and the parent thread is changed to `kthreadd`, the kernel background thread default for the owner. Implementation on kernel module uses *`kthread_run()`* to create `pkap_thread`. Kernel threads do not have user-space connections, allowing fast and efficient context switching when executing them.

3.2.3. Data Transmission from Kernel-Space

When data comes from a network, the multiple variants libpcap library provides user-space applications access to packet capture capabilities. No standard libraries exist to capture data within the kernel.

Selection of PF_RING

The following factors are key to the selection of PF_RING as the algorithm of choice for this research:

1. Efficient
2. High Performance
3. Ability to packet filter.
4. It does not require changes to the Kernel core code.
5. Can be used by user-space applications or even kernel-space applications to control comparative performance analysis.

These factors make PF_RING stand out as a choice. Other libraries like libpcap require changes to the kernel core to share the ring at kernel-space. Netfilter plug-in is highly complex and cannot be used by existing user-space applications. A custom NIC device driver implementation is also out of the scope of this research and it is difficult to share for use with available user-space applications. PF_PACKET is the default alternative to PF_RING; however, there are known limitations and the subject of this investigation is proposing a substitute. As mentioned in Chapter 2, PF_RING offers characteristics like reasonable resource utilization, performance above or equivalent to libpcap implementation, and it is a dynamically loadable kernel module that can be inserted into a standard plain vanilla Linux platform without any changes to kernel core source code. It has a lightweight pfring API with essential ring creation features, and the libpcap-ring library provides an interface to interact with other widely used user-space applications. There is plenty example code to fully demonstrate library usage.

Sending and Receiving Data with PF_RING

PF_RING provides a ring buffer memory allocated in kernel space. When used by a user-space application, the pfring library maps the kernel-space memory into the application's memory. Section 2.1.2 describes how PF_RING performs data transmission. As part of the experiments performed in this research, transmission of packets was done by using the *pfsend* and *pfcount* applications.

3.3. Architecture Overview

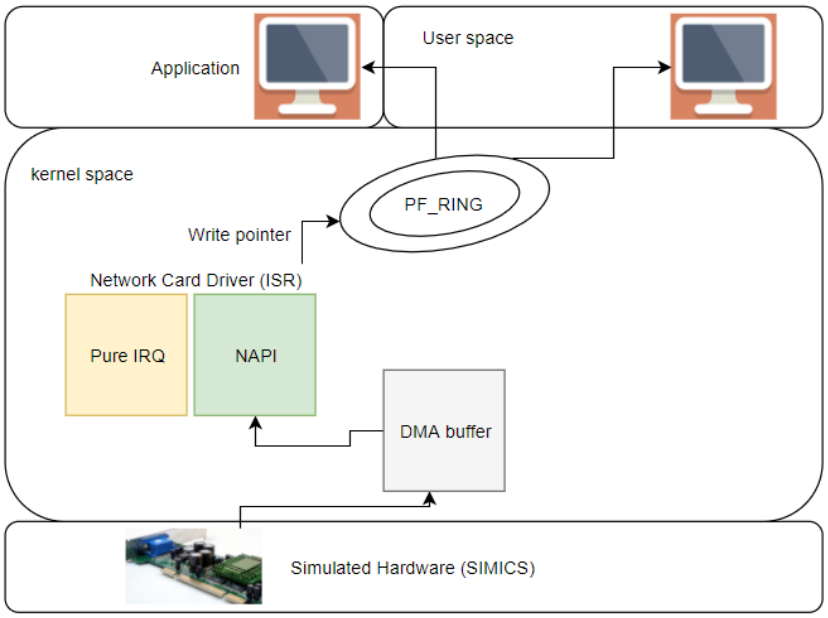

Figure 3. 1 Architecture overview modeling kernel to user implementing data transmission operation using PF_RING.

Section 3.2.3 describes the design for a system using PF_RING. The architectural overview shows how all pieces come together. Figure 3.1 above depicts the research architecture. Notice that the figure shows that the kernel handles data transmission and action. The application system in user-space is related to the ring through the `/proc` filesystem.

Use of PF_RING results in a reduction of memory footprint and copy count that is provided to user-space applications through pfring and libpcap. Other improvements can be made to the algorithm that are out of the scope of this research. pflib library copies the data from the ring before passing the copy to the application. Kernel module file writing is less expensive that avoiding memcpy() calls. As a proof of concept, PF_RING was built into the kernel as a module utilizing the built-in mechanism provided by PF_RING reporting status and health of the ring buffer and giving an insight of the data transmission process.

Chapter 4. Methodology

A presentation of the methodology used to evaluate the performance of PF_RING in an Operating System kernel module compared to a de-facto PF_PACKET algorithm used by user-space applications. This research evaluates four metrics: CPU utilization, memory usage, packet rate drops and data query delay from an application. Section 4.1 defines system boundaries, and section 4.2 presents system evaluation methodology. Section 4.3 describes the system simulation services provided by a system running on Simics. The workloads and performance metrics are described in Section 4.4 and 4.5 respectively. System parameters and factors are treated in sections 4.6 and 4.7. Finally, the evaluation technique and experiment designs used to test, analyze and interpret system performance are detailed in sections 4.8 and 4.9.

4.1. System Under Test Boundaries

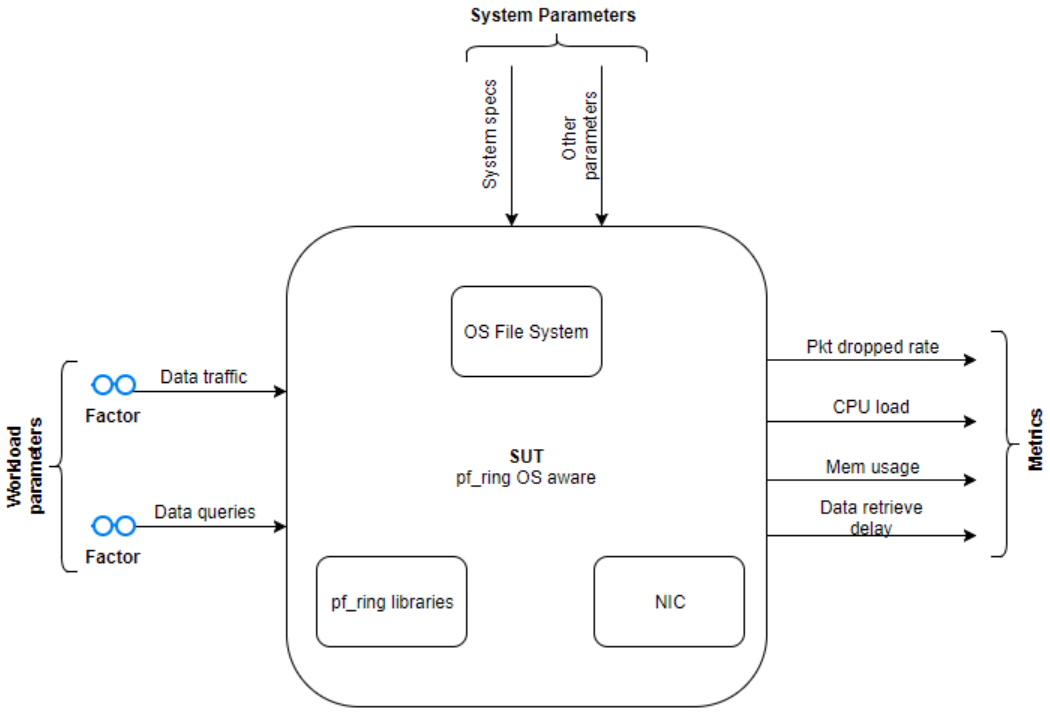

Figure 4. 1 System conditioned under test diagram.

Figure 4.1, shows the SUT of the evaluation system. PF_RING provides a straightforward capability, but being placed into an Operating System Linux kernel it may behave with unknown side effects derived from a general purpose OS, multitasking tasks, multiprocessing computer system (multiple cores simulation) meaning that there are a considerable amount of elements that affect the system. The only purpose of the SUT shown in Figure 4.1 is to clarify the components that have a direct influence on the performance of the data transmission algorithm and the system under study.

The essential part of the algorithm is the PF_RING library running as a kernel module on a simulated hardware environment running the Operating System unchanged binaries on Simics. Other components are the user-space application agent used to retrieve or write data on the ring. Workloads are included in the diagram such as data queries for data or simulated network traffic. Details on parameters are included in section 4.6.

Metrics used in the performance analysis are packet drop rate, CPU load, memory usage, and query delay. On a real-life system, data query latency or data being dropped are important metrics. Metrics and methods are described in greater detail in Section 4.5.

4.2. Evaluation Methodology

To measure a real system with PF_RING enabled on an embedded network device for performance evaluation during the experiment is out of the scope of this research but would be ideal. Results are verified and validated using analytic analysis in a simulation environment. The proposed system is interconnected and dependable on other systems. Accurate modeling of system behavior is ideal but also out of the scope of the investigation, but it aims to behave and facilitate performance measurement characterization. Interdependent systems of systems are highly complex but simulation with Simics provides observability.

To evaluate performance characterization of a PF_RING-enabled and a non-PF_RING system, actual metrics must be obtained for both kernel space capabilities and equivalent user-space applications making use of these features. On user-space, a tool based on open-source code was written to simulate network traffic capture data on a system using libpcap and to write them to pcap-formatted files on disk. This tool was also modified to use PF_RING-enabled libpcap.

The main difference the tool allows is to use PF_PACKET or PF_RING ring buffer. There is where the difference relies on the SUT. Both ways utilize the fwrite() function that uses buffering by default, generating fewer system calls when transmitting large chunks of data or with small data each time. The goal is to use also fewer CPU cycles and less memory as per the metrics detailed before. Additionally, the tool was modified to be able to write to dev/null as a control in the experimental trials.

4.3. System Services

The system allows for any given data, either coming from the network or produced from the application, to be written or retrieved from the ring on the kernel or an I/O storage device accurately. The system must provide determinism as a service for storage or read/write from the ring of the data of interest. In general terms, the system provides services like:

1. Providing an experimental environment that allows determinism.
2. Accurately allowing data disk I/O storage simulation that responds to queries from an application.
3. Being a simulated environment where system factors can be controlled and observable.
4. Running a real OS on a simulated system.

Given the overhead rate of the smallest data packets of required information data, the modeled system needs to be able to sustain theoretical write speeds to disk of 70MB/s for an average size of 765 Bytes that can be received by the host system.

4.4. System Workload

The SUT workload is determined mainly by a controlled factor such as the data traffic sent to the system. Data characteristics are determined by size and rate during performance evaluation. Query data workload test is part of the last experiment configuration to evaluate access to the ring from an application.

4.4.1. Simulated Network Traffic

Traffic Characteristic.

Data transmission is the principal observable element of system workload. The size and frequency of data packets introduce stress to the algorithm read/write functions. As part of this research, the workload is produced by pktgen – a Linux Kernel module — to create data streams synthetically. It uses the User Datagram Protocol bit-spitter to test the transmission and reception of NICs drivers and capabilities. This method of data generation is enough to stress the experimental systems as described in the following statements:

1. For the sake of packet size, data payload is not important nor is any protocol details above layer 4.

2. PF_RING algorithm removes the sk_buff from the kernel TCP/IP stack before the point where a given protocol can distinguish any data payload.

3. Protocol does not affect any PF_RING filtering capabilities such as Access Control List (ACL).

Pktgen setup can generate average packet size dropping below 700 Bytes. Several instances of pktgen systems can fully saturate a system data processing

with small packets. A single packet generation thread creates enough traffic types and rates to evaluate the algorithm Kernel module capability initially. A user-space application typically stores data sequentially either on a ring buffer or an I/O storage device. The focus on this research is to go beyond boundary limits, so testing scenarios are repeated while writing to dev/null. Doing this removes any bottleneck on a simulated storage hardware device to isolate this factor on performance data rates transmission.

Data streams are changed in size and data transmission rate, so it can test all evaluation scenarios. Data bit rate is a significant observable factor in the system workload, the number of data packets per second (pps) has more observability significance to CPU utilization metric than the data bit rate alone. As mentioned, a huge amount of small size data packets at a given rate can add greater system workload than lower quantities of big data packets at the same given rate. To evaluate system behavior, the size setting is the following: mini, macro and random. The mini-test is set to transmit only at 64B frames to impact performance on the SUT when the chain of functions on the algorithm handles a data packet buffer transferring from memory to another. The macro test is set to transmit to 1500B frames to impact SUT when moving data with minimum system resource spent setting up the data movement. Random is set to transmit an average distribution of data packets ranging from 64B to 1500B frames. The purpose of this settings is to experiment with system performance when the data input factor is varying.

Data Rate.

The packet data rate can be adapted by pktgen delay knob. A nanosecond level adjustment of delay in-between packets is proportional to the throughput of system traffic processing. Using a Real Time OS is out of the scope of this research, but a Linux kernel with these characteristics can provide high resolution of data processing timing.

Time Length.

In a simulated environment, 5 minutes real time could mean hours of simulation time. That time is enough to level up the SUT to a steady state even with the lowest trial data rates. Getting to a steady system state is important for valid measurement analysis. On a steady system state the test environment settles to a point where the Linux ring buffer cache is not a factor in system performance, and any query to the ring buffer can be satisfied on every task improving the performance of data transmission that represents real world behavior.

4.4.2. Application Data Tx/Rx

Data needs to be recognizable to be able to search for it and for the application to understand how to accomplish the search and response.

Data Object Creation.

Data can be seen as objects (marked packets) that are transmitted over a network or within a system that specifies information or addresses (memory location) considered as workload. Data can be objects that are beacons sent once at a time constantly to make sure the whole system is connected. These marked objects in complex systems are a noticeable workload increase, and the SUT is affected by them. These marked packets are identical to any other general traffic on an Internet Protocol network.

Query Workload.

The simulated test environment uses an application that has a background script exercising the SUT that starts data packet queries to the ring looking for marked objects to get a better grasp on what could happen in a real life connected device. The impact of this type of workload on the system is related to the SUT CPU resources consumed by additional tasks looking for objects with some type of metadata. This also represents a situation where read/write performance is impacted and dropped packet rate would also be affected.

4.5. System Performance Metrics

The system must use the kernel module PF_RING algorithm to store data in the ring buffer with a higher probability of success and low data drop rate. To accomplish this, the system must perform efficiently with the least overhead as possible. The following metrics are defined to measure system success:

- CPU utilization
- Data drop rate
- Memory utilization
- Data query delay response

These metrics are used because they are standard and universally used to compare between algorithms at high-level definition.

4.6. System Parameters

SUT properties are the system parameters that directly affect its performance. As a system of systems built in Simics running a general purpose OS the complete list of parameters is so wide that it is out of the scope of this study. From experience and subject matter expertise the list of factors is reduced and chosen for the experiments to those that will affect SUT performance in providing services the most. System performance, as already mentioned, is primarily affected in this research by its workload parameters. The following sections will describe system parameters for the SUT.

4.6.1. Applied Algorithm Capturing Method

The PF_RING algorithm determines whether the data should be passed to the application user-level space or if it can remain in the kernel for its processing inside

the SUT. The options are 1) user-space application or 2) pf_ring kernel module under study. Both the application and the kernel module use the same library to send/retrieve data packets from a NIC and filter them for interest.

4.6.2. Device Selection

I/O storage configuration of the system also impacts SUT performance in providing services. Any data that comes into the SUT also needs to be stored somewhere. Simics provide I/O simulation capabilities, options are 1) SATA hard Drive formatted to ext4 with 65MB/s write speed or 2) the dev/null device. Ext4 is selected for the experiments since it provides sequential write speeds like other Linux file systems. In this research it is not vital for storage to be permanent, and dev/null device is adequate for measuring SUT performance without getting too much impact on a physical storage device.

4.6.3. Filtering

Filtering is a system parameter. It is intentional for discarding some data defined by a pre-configured rule set. The SUT through PF_RING is capable of filtering data packets. As mentioned in Chapter 4 the ability to perform filtering is a key factor in the experiment design; however, for experimentation, all filtering is disabled for SUT performance analysis.

4.6.4. System Specifications

Hardware specifications for the host system impact SUT performance. Increasing CPU performance (adding cores or improving its specifications), memory, buses, platform architecture, storage system, NIC directly impacts the performance of the studied system. A model can be changed leveraging Simics to improve specs and tweak SUT performance; however, it is out of the scope of this research to control these factors. The specifications for the test system are constant to get consistent test results. Use of Simics is focused on the fact that it allows deeper observation into system states facilitating research reproducibility and observability.

4.7. Controlled System Factors

The following section is a presentation of the system and workload parameters to be used as factors during the test evaluation. Factors were selected through pilot experiments and experience knowledge. Factors are changed during each experiment. Experiment 1 and 2 are described in Section 4.9.

Factors in Experiment 1 are selected based on the impact on the SUT under controlled workloads to transmit and receive data with PF_PACKET. Factors in Experiment 2 are varied in the same way as in the previous experiment with the same workloads, but with PF_RING. Sections 4.9.2 and 4.9.2 outline the factors selected from the system and describe the workload parameters used during the experiments.

4.7.1. Data Processing Method

The method is the PF_RING implementation into the Kernel module where data packets traverse kernel-space to user-space and vice versa, or even when they do not. The impact of how the algorithm treats data and improves data transmission at different resident spaces is a main factor and thus the goal of this thesis.

Kernel Module

The PF_RING kernel module installed instead of the default solution on an OS is the proof of concept for a kernel level capability under test. It is a dynamically loaded Linux kernel module that implements a background process of a thread type. Details on how the solution in place is designed are located in Chapter 3.

AppDataPfRingLogger

The AppDataPfRingLogger application serves as a tool that provides functionality and performance threshold for the user-space system. The application resides in user-space. It is a libpcap-based data packet application which is a *custom* modification from DaemonLogger written by Martin Roesch. Modifications are

proposed only for this investigation specifically and do not pretend to substitute the original solution. The use PF_RING hooks provides an efficient functionality. AppDataPfRingLogger leverages the nature of the DaemonLogger base code for representing a user-space application.

AppDataPfRingLogger was modified to support dev/null as logical I/O device and compiled with a libpcap-1.8.1-ring library which is the PF_RING enabled version of the standard libpcap universal library. This application retrieves data packets from the same ring buffer as the PF_RING kernel module. It is an excellent focal point for the SUT. It uses the fwrite() call (through libpcap's pcap_dump() function) that uses the write() system call. It is handled by vfs_write() function which passes handling of this call to the filesystem's write() file operation. As discussed in Section 3.2, the kernel module uses the write() file operation directly. This guarantees the application write access to the data is identical to the one that uses the PF_RING kernel module.

4.7.2. Virtual Device

Although simulation system services provide an I/O storage model, a virtual device is used on the hosted OS. The /dev/null device:

- Discards all data written to it.
- Acknowledges with SUCCESS always.
- Provides End of File (EOF) reporting always.

This is selected as a method to extend beyond the ability of the simulated hardware adding capabilities of sequential block writing between 2.6GBps and infinite. Data cannot be verified on correctness since it is immediately disposed.

4.7.3. Test Workloads

Data Size

A key service of the algorithm running as a kernel module is to provide the data available in the ring buffer so an application can access it as fast as possible. The most impactful system workload is copying data coming from a NIC to the ring, handling it over to an I/O storage device or to an application for user layer. Data size is the main contributor to system stress, even more than the data bit rate. On a network, devices typically use 64B for stress testing. Developers need to make sure proper functionality under too many data packets being filled in the network. To simulate that in the experiments data size is controlled by the following criteria:

- Minimal size: 64B data frames
- Maximum size: 1500B data frames
- Random size: distribution from 64B to 1500B data frames.

Data Rate

Data rate is also significant as SUT workload. Low stress, stress high point, and unlimited theoretical stress of system experiments with rate will be controlled as follows:

- Baseline (b0): 100Mbps: Low-stress testing to explore SUT behavior under light workloads.
- Easy (b1): 140Mbps: Close to data processing routine handle packets per second (pps) limits. 1500B + Random (650Mbps): Maximum data rate an I/O storage device can handle when writing to it.
- Moderate (b3): 200Mbps: Above processing handling routines. 1500B + Random (700Mbps): Moderately above the maximum data rate the system can handle.
- High (b4): 340Mbps: well beyond the data rate processing handling routines can perform requiring 2 pktgen systems to create the desired rate. 1500B + Random (980Mbps): well beyond the rate the system can handle.

Experiment 2 configuration involves the SUT reaching a more realistic workload to approximate the system, so the data packet sizes are randomly selected between 64B to 1500B. Distribution of random sizes is normal, and average data size is 782B. On Experiment 3 levels are more granular to determine data query workload effect on the SUT: 100 Mbps, 225 Mbps, 350 Mbps, 575 Mbps, 650 Mbps, 700 Mbps, and 975 Mbps.

4.7.4. Application Data Workloads

Workloads are applied in every experiment. The application will query data from the ring buffer for experiment trial. Refer to Figure 2.8.

4.8. Experimental Technique and Simulation Environment

SUT evaluation is conducted thorough measurement of the system ability to place data on the ring buffer and it being read by the application. System analytical modeling and simulation are performed using Simics full system simulator with the objective to simplify and add determinism to experiments. A real-life network and a system-of-systems would be complex and unreliable. The development of the application and Linux OS with PF_RING running on top of a simulation add flexibility to observe and adapt functionality to new experiment environments or platforms.

4.8.1. Evaluation and Simulated Environment

As depicted in Figure 3.2, the environment to evaluate the SUT is a simulated Simics platform running a Linux OS with the following characteristics:

- Intel® Quick Start Platform x86 Simics simulator.

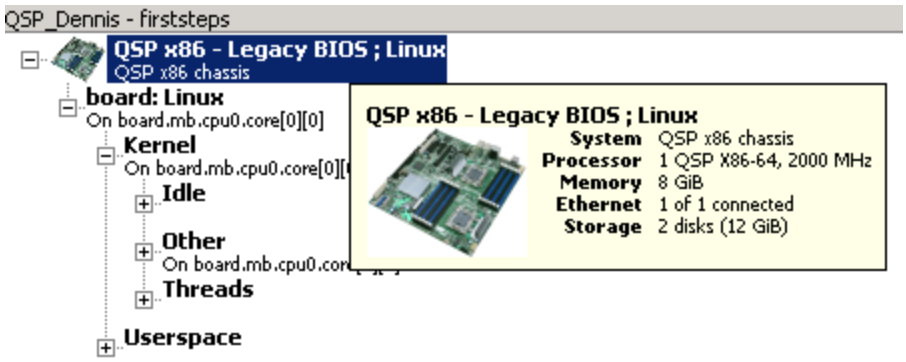

Figure 4. 2 Quick Start Platform running Linux.

- PF_RING version 6.6.0 revision XXXX
- libpcap-1.8.1-ring (distributed with PF_RING 7.0.0)
- pktgen single instance
- Linux genericx86-64 3.14.19-yocto-standard #1 SMP PREEMPT Mon Jan 25 10:29:05 CET 2016 x86_64 GNU/Linux

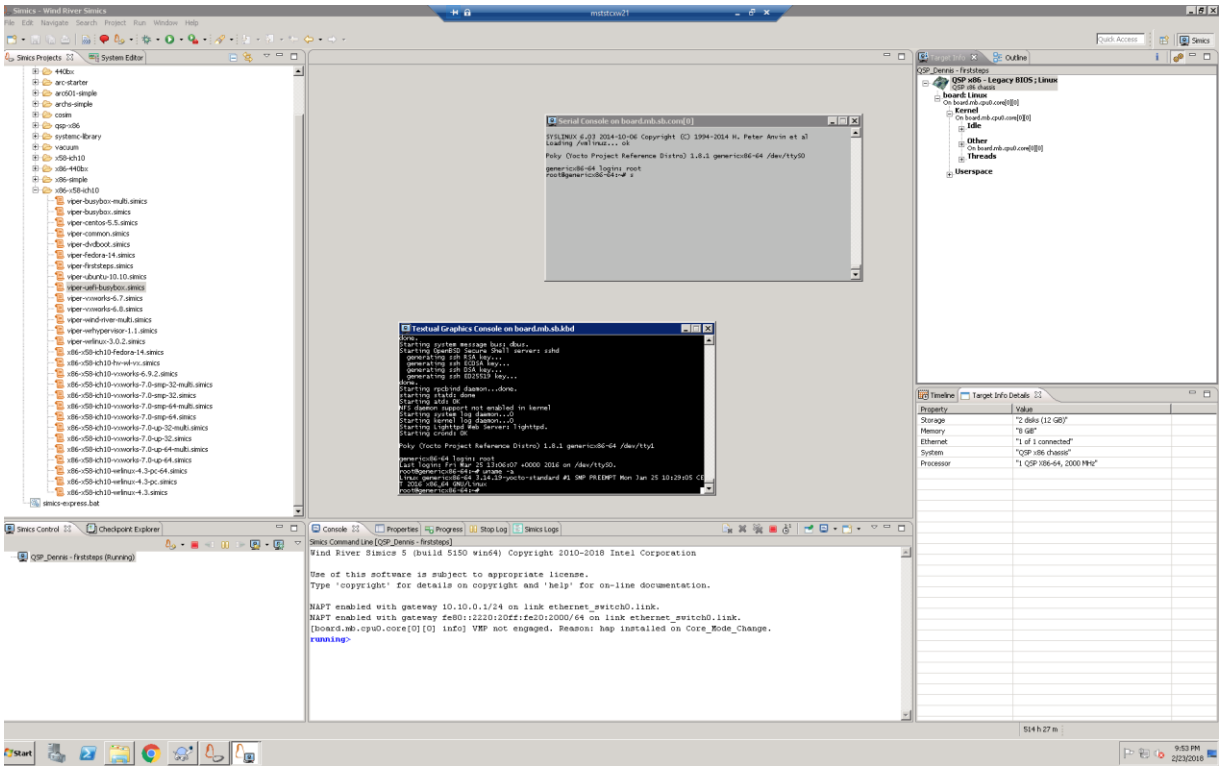

Figure 4. 3 QSP running Linux on top of a simulation running Simics.

4.8.2. Experimental Technique

This section describes the method used to get the data and the process used to analyze the information for each trial. The metrics selected are used to be able to compare design performance to the default traditional approach --that is to compare the effect of PF_RING kernel module on SUT performance with the AppDataPfRingLogger application. Every trial set experiment discussed in Section 4.9 provides a specific factor level combination. Each combination of factors has a workload configuration level which enables the metrics to be comparable to each other.

Data Drop Rate

"In this Internet age of network computing, one of the most critical quality attributes is system and network availability, along with reliability and security. Requirements for high availability by mission-critical operations have existed since society became reliant on computer technologies. In the Internet age, software code is distributed across networks and businesses increasingly share data, a lack of system availability is significantly increasing adverse impacts." [KAN, 2003]

As mentioned before, data drop rate is significant to SUT system performance. Success of the experiments depends largely on data being dropped or not. If drops are too high, there is no sense in running any other experiment. Data drop is bounded to the probability (*p*) of a packet being successfully captured on the ring:

$$p = 1 - P;\ where\ P = \frac{\#\ of\ pkts\ stored\ in\ ring}{\#\ of\ pkts\ transmitted} \quad (4.1)$$

Data drop rate is an important measure for being able to compare the pktgen of packets generated and stored in the ring. Each test workload will be run five times. The number of attempts for each test experiment configuration is established from the baseline. The SUT is based on a general-purpose operating system. Trial repetition is required to get consistent and reliable results although this can be ensured by a simulated environment.

CPU Utilization

SUT load is impacted by the data workload, and the data vary from the options described in Section 4.7. CPU utilization information is gathered by pidstat Linux utility, part of the systat system status monitoring tools suite. The pidstat will be configured to monitor the kernel thread collecting data every five seconds through the duration of the test. CPU utilization average for a set of settled factors is the main metric of interest; however, historical utilization changes give a clearer view of possible bottlenecks reacting to data traffic changes, the influence of other kernel tasks, etc.

Memory Utilization

Memory utilization data is gathered using the same metrics as CPU utilization. Refer to CPU utilization experimental method technique.

Data Retrieve Delay

In Experiments 1 and 2 the above metrics are used. Data retrieve is also monitored and analyzed. Response metrics of interest are cycle delay in seconds, number of queries completed successfully and percentage of the marked packets found.

4.9. Experimental Design

There are two experiments. Experiments 1 and 2 leverage from pilot testing to determine fixed factors (parameters), controllable factors, data gathering methods, simulation infrastructure and the base virtual platform configuration. Experiment selections are put under stress testing for normal and atypical workloads to simulate a real system connected to the Internet. Workload intention is to provide input data to the SUT to measure its performance.

4.9.1. Experiment 0: empirical baseline

This section describes experiment tests based on empirical purposes and methods. It is useful to determine controlled parameters but to better understand the SUT.

Configuration. A set of tests conducted to experiment on PF_RING configurations for slot count, filesystem formatting, kernel module installation and formatting options.

System Architecture. Determine Linux systems data packet generation and evaluate to ensure that the system will be fully exercised. Select to have a controllable and observable test environment. Key decisions are: whether to use a simulated data generator or an operating system utility, and which simulator provides a controllable and deterministic test environment that would best stress and provide the SUT environment.

Data Producer. Set of tests to be conducted to gather baseline configuration options of the packet generation tool. It is important to be able to find a configuration that would also be deterministic, reliable and easy to set for repeatability. Set configuration file format that just easily enables data speed, sizes, addresses, and duration of every test.

Profiling. When a test environment is set and it is possible to send data input to a SUT, workloads are sent with `oprofile` utility test suite to profile the PF_RING kernel module.

Measurement Tools. It is necessary to acquire data from each of the tests tools to get metrics, monitor the SUT, get the status of PR_RING ring buffer health and drop count, in addition to data transmitted by the generator and data being copied to the application from the ring. All this is provided by the SUT with the pf_ring library.

The experiment focuses on tweaking baseline variable results to set initial data test count, set data rate levels to observe SUT characteristics performance from baseline and data send to ensure the SUT reaches a stable state.

4.9.2. Experiment 1: PF_PACKET System Enabled

This experiment describes the SUT with PF_PACKET configuration running under the scenarios described in Table 4.1. After a stable repetition period, the SUT is exercised generating data packets that are received and written to dev/null over the course of 10 hours of simulation time.

Table 4. 1 Listing of each configuration test containing three tests every five repetitions.

T1/T2	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**
pkt_sz	min	max	rand	min	max	rand	min	max	rand	min	Max	rand
Mbps	100	100	100	140	652	650	200	714	700	340	985	975

T3	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
pkt_sz	rand	rand	rand	rand	rand	rand	rand
Mbps	100	100	100	140	652	650	200

4.9.3. Experiment 2: PF_RING System Enabled

This experiment describes the SUT with PF_RING configuration running under the scenarios described in Table 4.1. After a stable repetition period, the SUT is exercised generating data packets that are received and written to dev/null over the course of 10 hours of simulation time. The ring buffer, which is not available in the scenario described in Experiment 1, is monitored in this experiment.

4.9.4. Methodology Summary

This chapter discussed the methodology used to evaluate the performance of the system under various data stress scenarios. Experiments 1 and 2 target SUT performance characteristics and main kernel module internal features by applying workloads during the test. Statistical tests are used to study and analyze the effectiveness of PF_RING compared to the default algorithm on the operating system depending on the workloads.

Chapter 5. Analysis

This chapter presents and analyzes the results of the experiments as described in Chapter 4. Section 5.1 through 5.3 discuss system performance results and how the system under test behaves with data. Experimental data analysis is presented in Section 5.4.

5.1. Results and Analysis of Experiment 0

Experiment 0 gathers basic information about the system to give an empirical idea of how the system behaves under certain workloads. This experiment measures performance data to obtain the stable or normal operation performance status of the system under test. The SUT is configured as an OS with the PF_PACKET component and the same OS with the PF_RING component as the only difference, to create two experimentation scenarios. The metrics for this experiment are categorized by trial set. When executing the experiment, there are twelve trials divided into four different buckets of data rates. Instead of exploring the data of each experiment trial, Section 5.1.1 shows an overview of the key results of the experiment. The other sections detail key points of interest.

5.1.1. Experiment 0 Overview

The following graphs in Figure 5.1 show the key SUT metrics. All trial packets are randomized. From the packet size factor, random size is the most significant standard data traffic. When using minimum packet size (64 Bytes) the experiment provides information about system behavior with the worst-case scenario showing high data copying overhead. For the experiment trial with maximum packet size (1500 Bytes) system behavior had the best-case data copying overhead scenario.

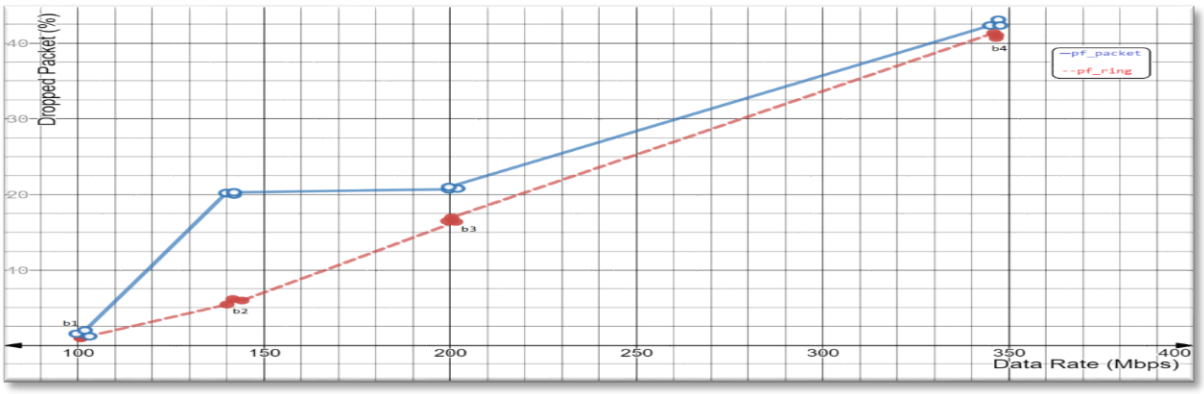

Figure 5. 1 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. Dropped packets (%). Blue = PF_PACKET, Red = PF_RING.

Figure 5.1 shows the key metric of dropped packets between PF_RING and PF_PACKET. At data rate level b2 PF_PACKET drops more data packets than PF_RING. At b4 level there is no difference between the two.

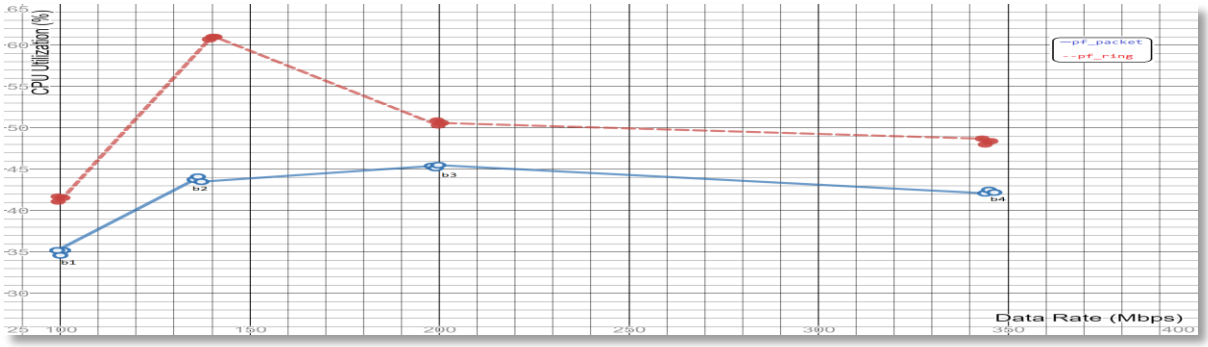

Figure 5. 2 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. CPU utilization (%). Blue = PF_PACKET, Red = PF_RING.

The inclusion of the results for trial sets with the lower sized packets introduces too much overhead and confusion to represent SUT behavior accurately.

Figure 5.2 shows the summary graph of the impact of the packet size on CPU utilization and drop packet rate when using PF_RING. There are no differences between measures on each data rate bucket in the graph due to the advantage of using Simics Virtual Platform to simulate the experiments. Because of that, the range of confidence is 98% on each data interval. At bitrate 3 (b3), there is a

statistical difference between experiment trials using random data size distribution and maximum sized packets, but the following descriptions are shown:

- Experiment trials between random and maximum data size packets behave similarly in data loss and CPU characterization.
- Experiment trials with the minimum packet size differ noticeably in both data loss and CPU characterization.

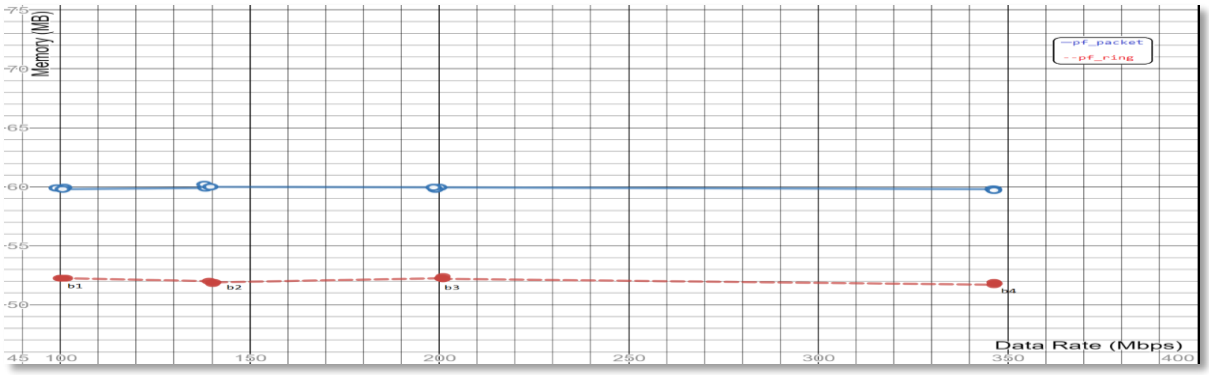

Figure 5. 3 Summary graph of Experiment 0 SUT key metrics between pf_packet and pf_ring. Memory utilization (MB). Blue = PF_PACKET, Red = PF_RING.

Figure 5.3 shows that PF_RING memory utilization does not change when the data rate is incremented. PF_PACKET memory utilization (60 MB) is higher than PF_RING (52 MB).

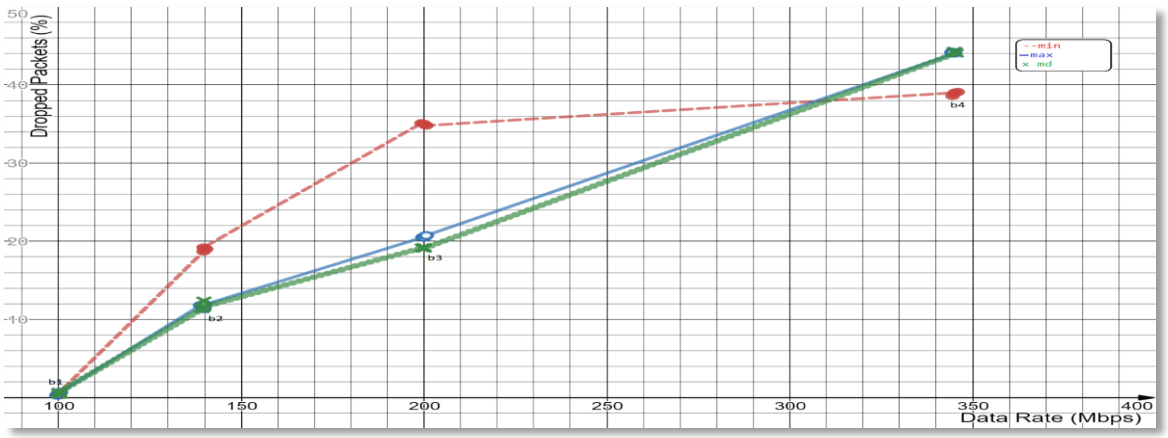

Figure 5. 4 Summary graph of Experiment 0 SUT key metrics when packet size levels impact data drop when using pf_ring.

Figure 5.4 shows the key metrics when the experiment is performed at different packet size levels. With PF_PACKET the behavior of the SUT performs the same when packet size is small or big. It drops the almost the same percentage when the data rate is high for all packet sizes.

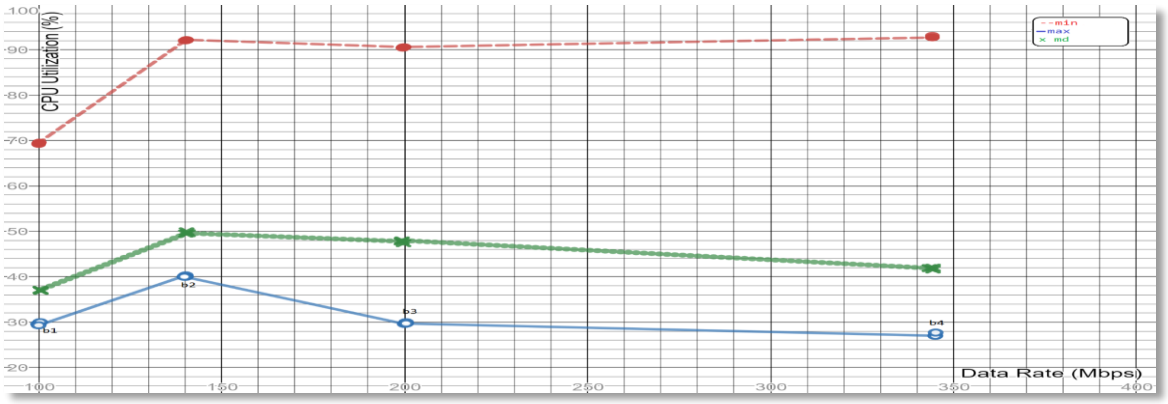

Figure 5. 5 Summary graph of Experiment 0 SUT key metrics when packet size levels impact CPU utilization when using pf_ring.

Figure 5.5 shows when the packet changes to the minimum size the CPU utilization rise to its maximum (~90%) no matter the data rate.

5.1.2. Data Drop Rate

Beyond data rate b1 where PF_PACKET or PF_RING are fully capable of zero packet loss, the PF_RING kernel module packet data drop rate is lower than PF_PACKET and even than AppDataPfDataLogger user application using PF_RING hooks. Table 5.1 below shows the results of the tests performed (hypothesis testing) on the data drop rate for each of the 4 distinct rata bitrate levels.

$$H_0: p\left(PktDropRate_{pf_{ring}}\right) = p\left(PktDropRate_{pf_{pkt}}\right)$$

$$H_0: p\left(PktDropRate_{pf_{ring}}\right) < p\left(PktDropRate_{pf_{pkt}}\right)$$

Table 5.1 shows that the null hypothesis needs to be rejected for every b-level (data bitrate level) that includes some data drop. The PF_RING kernel module is capable of transmitting more data to the buffer or even to disk than the AppDataPfRingLogger user-space application.

Table 5. 1 Experiment 0 hypothesis testing of dropped packets

Null Hypothesis	Scope	Estimate	p-value
p(*PktDrpRatePF_RING*) < *p*(*PktDrpRatePF_PKT*)	Experiment 0	0.08861	4.76E-79
p(*PktDrpRatePF_RING*) < *p*(*PktDrpRatePF_PKT*)	b1	0	N/A
p(*PktDrpRatePF_RING*) < *p*(*PktDrpRatePF_PKT*)	b2	0.16458	1.43E-82
p(*PktDrpRatePF_RING*) < *p*(*PktDrpRatePF_PKT*)	b3	0.09436	2.42E-08
p(*PktDrpRatePF_RING*) < *p*(*PktDrpRatePF_PKT*)	b4	0.06853	7.41E-14

Table 5.1 shows the *p* value. The p values for every experiment scope are less than 0.05% (5%) which indicates it is enough to claim that PF_RING drops less data at different data rate than PF_PACKET. The p-value on each data rate level show the results of this study are solid and repeatable.

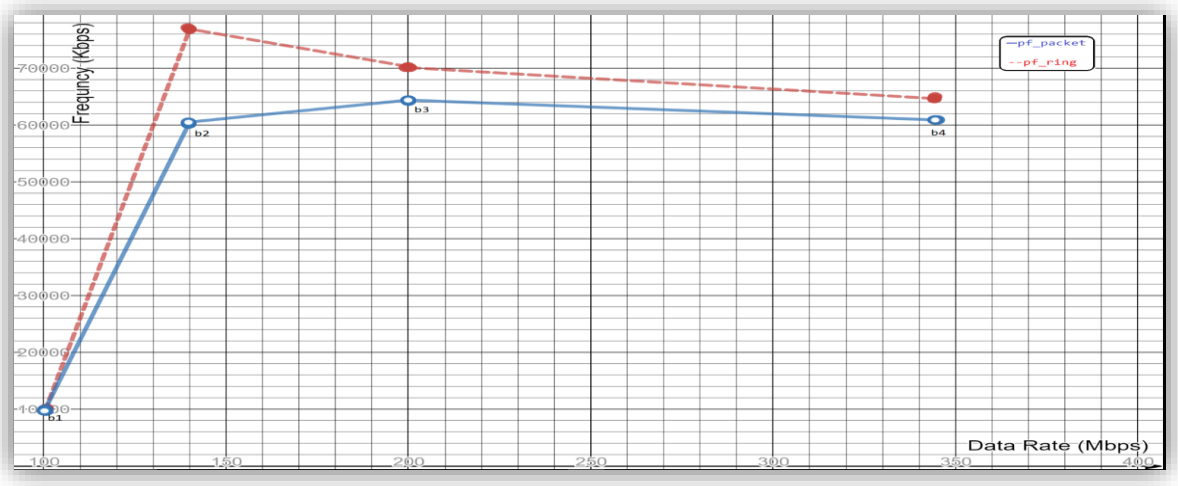

Figure 5. 6 Data packets are written to dev/null on Experiment 0 comparing pf_packet and pf_ring.

Figure 5.6 displays the differences in KiloBytes per second (KBps) between PF_PACKET and PF_RING. Due to multiple dependencies of the SUT, the actual

data rate written to disk varies as the network load changes; however; the data shows on average that the kernel-space implementation (PF_RING) is capable of transmitting 15-20% more KBps than PF_PACKET with a 98% of confidence interval (due to the simulation environment).

5.1.3. CPU Utilization

As can be observed in Figure 5.2's graphical summary of CPU utilization, the original hypothesis expressing that the PF_RING component reduces CPU utilization appears to be false.

$$H_0: p(CPU_{pf\_ring}) = p(CPU_{pf\_pkt})$$
$$H_A: p(CPU_{pf\_ring}) < p(CPU_{pf\_pkt})$$

Table 5.2 shows the results of the null hypothesis performed on CPU Utilization for the 4 different bitrate levels. Both tables show that the inverse of the original hypothesis is valid for all the cases but for b3 there is insufficient evidence to reject the null hypothesis. This metric shows a failure of the PF_RING kernel module to achieve one of the objectives (a reduction of the CPU utilization). The Summary analysis on section 5.1.5 examines the relationship between the metrics and provides an insight into the eventual effect on the SUT.

Table 5. 2 CPU Utilization hypothesis testing in Experiment 0.

Null Hypothesis	Scope	estimate	p-value
p(CPU*PF_RING*) < *p*(*CPUPF_PKT*)	Experiment 0	-7.85041	1
p(CPU*PF_RING*) < *p*(*CPUPF_PKT*)	b1	-6.90122	1
p(CPU*PF_RING*) < *p*(*CPUPF_PKT*)	b2	-16.76033	1
p(CPU*PF_RING*) < *p*(*CPUPF_PKT*)	b3	-3.67531	0.97152
p(CPU*PF_RING*) < *p*(*CPUPF_PKT*)	b4	-1.89512	0.94604

Inverse Null Hypothesis	Scope	Estimate	p-value
p(CPU*PF_RING*) > *p*(*CPUPF_PKT*)	Experiment 0	-7.85041	1.53E-31
p(CPU*PF_RING*) > *p*(*CPUPF_PKT*)	b1	-6.90122	5.07E-14
p(CPU*PF_RING*) > *p*(*CPUPF_PKT*)	b2	-16.76033	4.60E-39
p(CPU*PF_RING*) > *p*(*CPUPF_PKT*)	b3	-3.67531	0.02839
p(CPU*PF_RING*) > *p*(*CPUPF_PKT*)	b4	-1.89512	0.05396

The first part of the table demonstrates that the null hypothesis cannot be rejected while the second part shows that the inverse null hypothesis is invalid on the data rate b3 level. There is not enough change after evaluating the null and alternate hypothesis to reject that PF_RING CPU Utilization outperform PF_PACKET key metric.

5.1.4. Memory Utilization

The memory utilization shown in Figure 5.3 depicts the measurable memory utilization throughout all the experiments. The PF_RING utilizes 52 MB of memory and the PF_PACKET 62 MB of memory. This does not vary with the packet size, bit rate or any runtime variable on the simulated virtual platform. The use of PR_RING ring buffer --sized when initialized-- makes up the most of the memory utilized by the kernel module. As much as the snaplen[4] or the number of ring slots were changed the same memory usage change was seen. Any memory consumption outside the ring, PF_RING saves memory by not buffering writes to disk and not copying the data from the ring to use it. Section 3.1.1 explained that one objective of the PF_RING PoC is to reduce memory utilization and Section 4.5 detailed memory utilization as a system metric. The objective was for the driver to make decisions to reduce memory footprint. As seen in the experiments, PF_RING memory utilization is consistently 17% less than PF_PACKET, indicating that the objective to reduce memory usage has been achieved.

5.1.5. Summary Analysis

As per the experiment, system metrics indicate that PF_RING kernel thread drops fewer data packets, consume more CPU resources and use less memory than PF_PACKET and AppDataPfRingLogger user-space application. While experiment indicators show some information about the impact of the PF_RING

capability at kernel-space, they do not provide the full picture without probing the dependencies between the data. In particular, note the relationship between data dropped packets and CPU utilization. Figure 5.7 and Figure 5.8 provide the common interaction of both two metrics. The graph data set from Experiment 0 at data rate b2 is an example. Although one single trial was performed for simplicity, the relationship between the two metrics represents all trials executed that resulted in dropped packets. Any increase in the data drop rate reduces CPU utilization. This relationship exists due to data being retrieved from the PF_RING ring buffer. Dropped packets never reach the buffer thereby reducing the workload at any given data rate. The effect of the dropped data is similar to the effect of filtering data though dropped data as an intentional choice. The experiments are not focused on capturing the data accurately but on the trade-off of CPU Utilization for improving data speed transfer aligned with the aim of this research. Experiment 0 reveals that PF_RING SUT option uses a greater percentage of CPU resources even when the difference cannot be attributed to missing data.

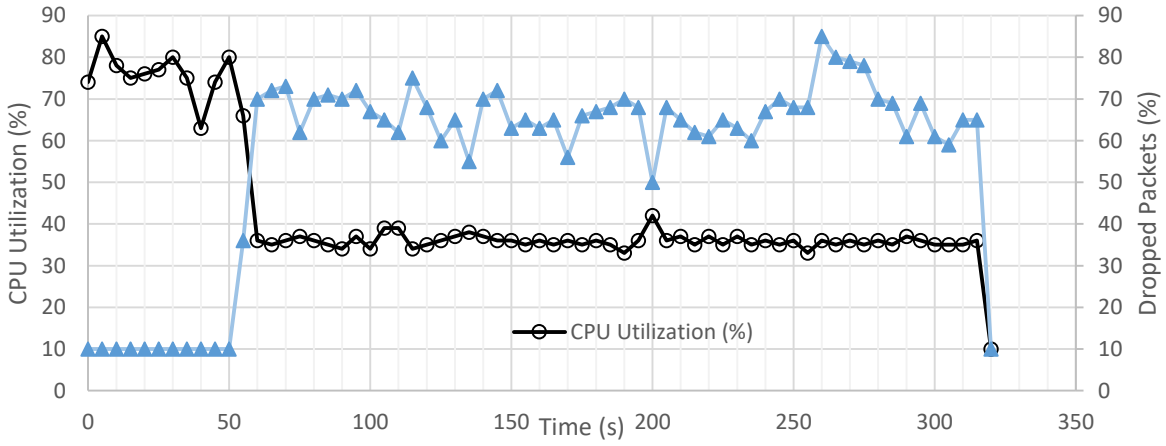

Figure 5. 7 Data that relates data drop rate and CPU Utilization. On the first graph, the Y-scales differ to visually show the effect of dropped data has on CPU Utilization. It is composed of a single trial but gives the 'all trials' trend.

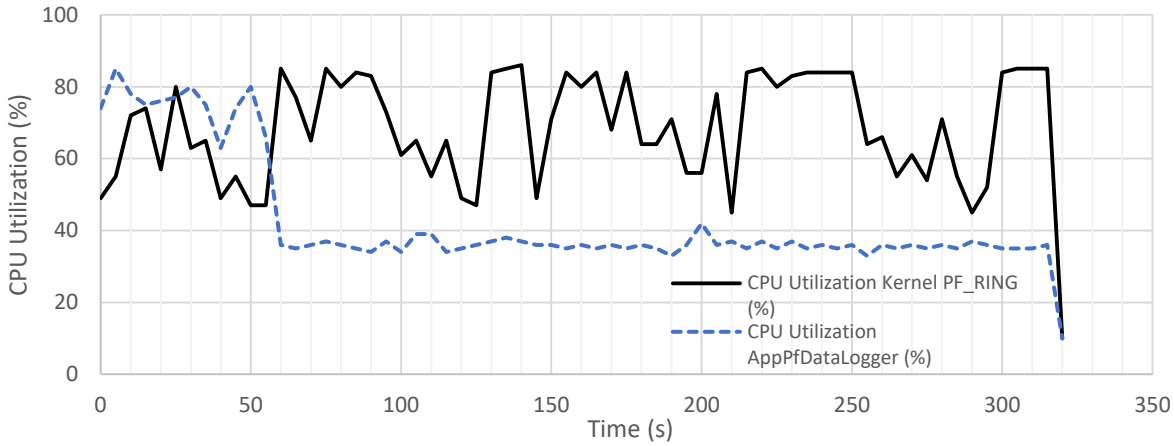

Figure 5. 8 Graph compares CPU Utilization using PF_RING showing no data drops while AppDataPfRingLogger has the same data drop rate as in the first graph.

5.2. Results and Analysis of Experiment 1

Experiment 1 gathers system metrics for the SUT under PF_PACKET while storing the data captures to the dev/null device. The SUT is an OS configured with PF_PACKET. This experiment measures performance data for the capture and storage process alone. No data queries were performed for this experiment during this phase of testing. Again, metrics for this experiment 1 were categorized by trial set. There are 12 trials, each representing 5 independent trials for several combinations of data bit rates and packet size. Section 5.2.1 presents an overview of the key results of the experiment, and subsequent sections detail the characterizations of interest.

5.2.1. Experiment 1 Overview

The three graphs in Figure 5.9, Figure 5.10 and Figure 5.11 below summarize the key metrics for all trial sets where the data packets are randomized for PF_PACKET SUT configuration. While not perfect, the SUT configuration is the most representative data size distribution of standard Internet traffic. Trial sets use the minimum packet size (64 B) which gives insight into SUT behavior with the worst-case overhead to data transmission ratio. Trial set with maximum data packet size (1500 B) does give an insight on system behavior with the best-case

overhead-to-data transmission ratio. The same rationale applied to this experiment as the one explained in Experiment 0 for general exclusion of trials with minimum or maximum sized data packets from the overall summary. The graphs depict the influence of data size on SUT performance during Experiment 1 and try to provide the best possible assessment and performance indication beyond the limits of what a real network system could provide. Using a Simics Virtual Platform simulation where the SUT runs, introduces a specific separation from real world testing and thus the inclusion of minimum and maximum data size packets is useful for revealing certain computational characteristics of the SUT configuration.

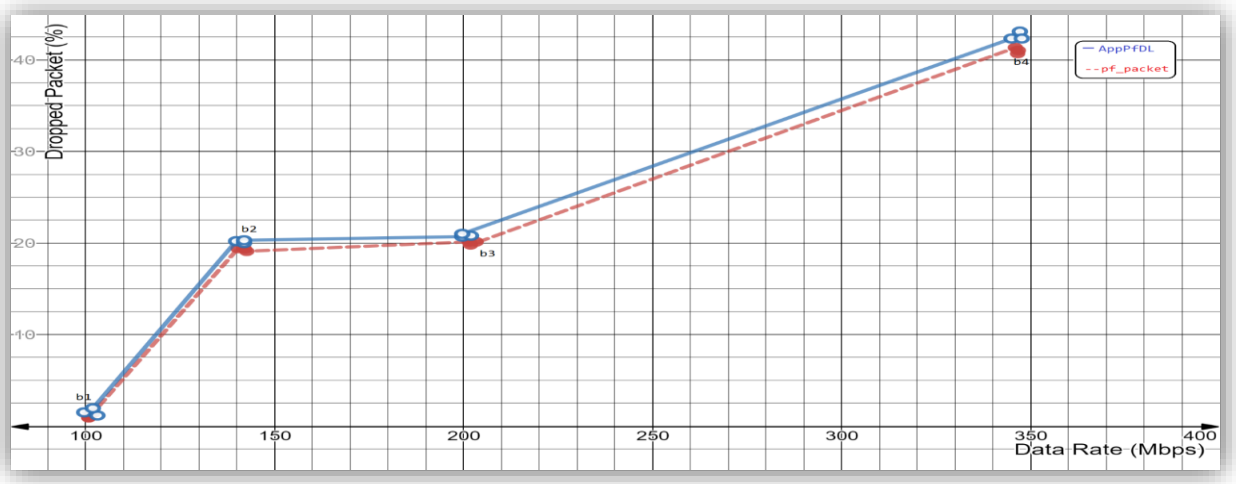

Figure 5. 9 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger. Dropped packets (%). Red = PF_PACKET.

Figure 5.9 shows the key metric of dropped packets between PF_PACKET and the user space application. On each data rate the amount of data dropped is the same.

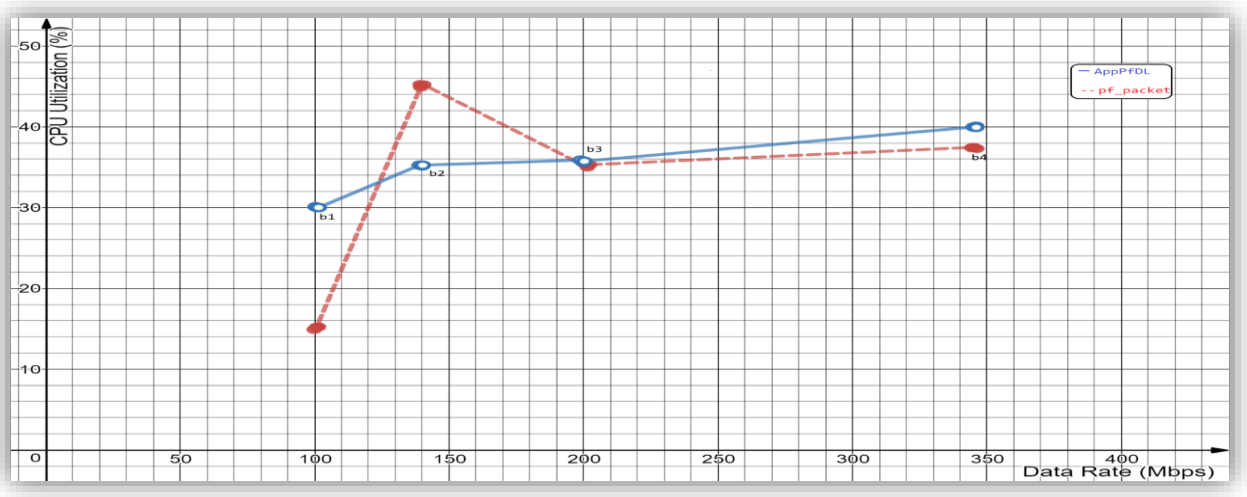

Figure 5. 10 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger.CPU utilization (%).Red = PF_PACKET.

Figure 5.10 shows the key metric of CPU utilization between PF_PACKET and the user space application. CPU utilization has same value (40%) at high data rates in the user space application.

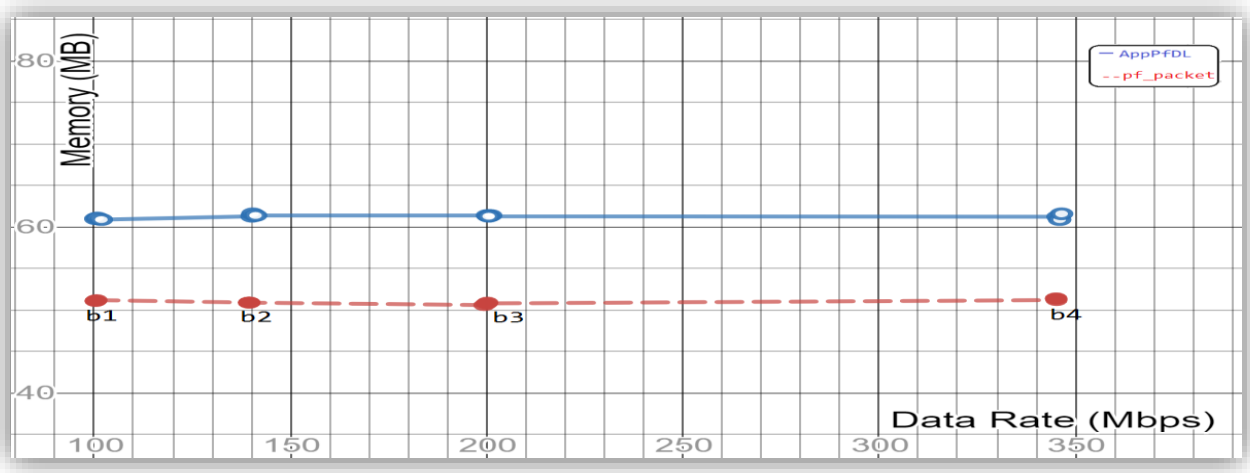

Figure 5. 11 Summary graph of Experiment 1 SUT key metrics between pf_packet and AppDataPfRingLogger. Memory utilization (MB). Red = PF_PACKET.

Figure 5.11 shows the key metric of Memory utilization between PF_PACKET and the user space application. PF_PACKET Memory utilization is lower than AppDataPfRingLogger.

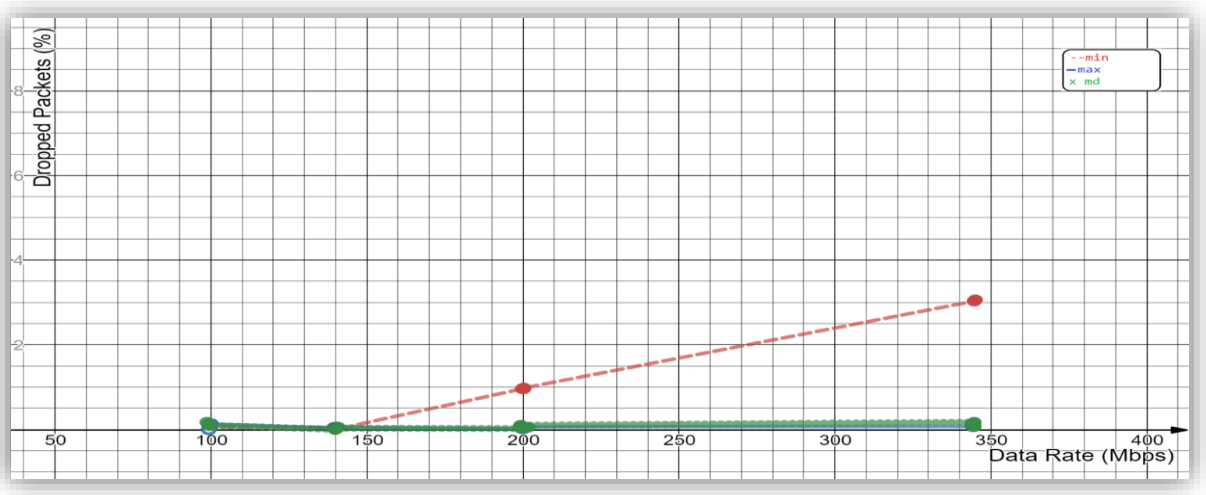

Figure 5. 12 Data size impact depicts the rate of dropped packets.

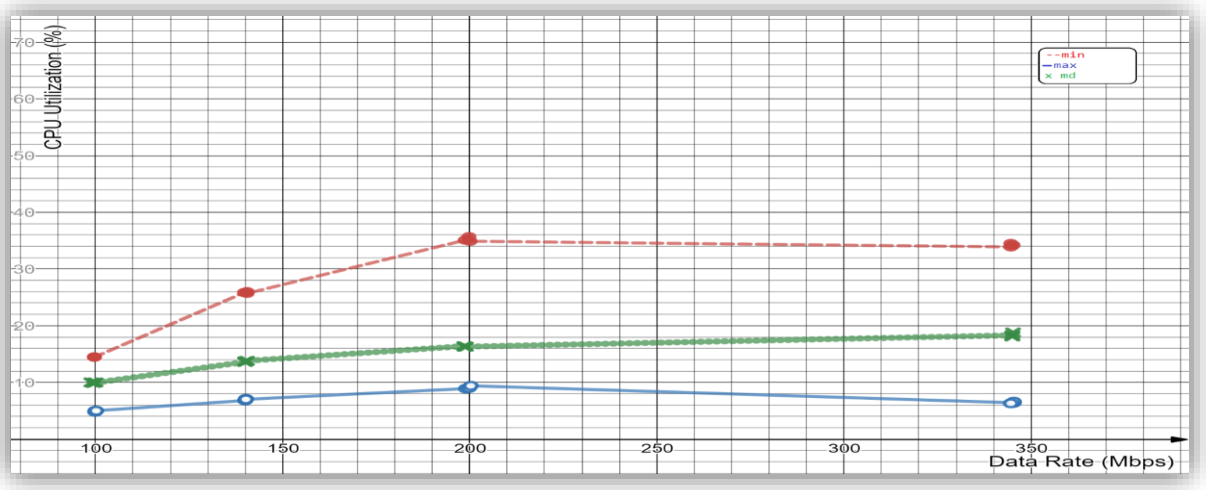

Figure 5. 13 Data size impact depicts the CPU utilization.

Figure 5.13 shows that when the data size is small and the data rate is high there is an increment of the amount of data loss when PF_PACKET is enabled. Figure 5.14 shows that CPU utilization is higher when using small data size.

5.2.2. Data Drop Rate

Unlike Experiment 0, where the data drop rate differences are evident after data rate level b1, the use of the dev/null virtual device for writing data instead of simulated disk maintains very low data drop rates until much higher data rates

come in. This result is due to the fact that every write is virtually non-blocking in the simulation. Though all write activity even to dev/null is a potential blocking call (from a software perspective) assuming a real OS where the file system delivers the data bytes to the bit bucket as fast as it gets them; the simulation removes one of the most significant delays on the SUT. The data bytes never hit any buffer cache, and therefore the data written to dev/null does not produce any dirty virtual page. The bdflush kernel daemon does not increase CPU usage for any write function call. It is for these reasons that Experiment 1 is used for comparison with PF_RING and identification of SUT processing behavior.

The results of the tests performed on the data drop rate for each of the 4 different data rate levels reveal no statistically significant data packet loss for any data bitrate when considering experimentation trials with randomly selected data sizes or maximum data packet sizes; therefore, Table 5.3 shows the results of the null hypothesis performed on the data packet drop rate trial set with minimum data size for each of the 4 different data bitrates of the SUT configuration selected.

$$H_0: p\left(PktDropRate_{pf\_pkt}\right) = p\left(PktDropRate_{appPfDL}\right)$$

$$H_0: p\left(PktDropRate_{pf\_pkt}\right) < p\left(PktDropRate_{appPfDL}\right)$$

The table shows that the null hypothesis is valid. When scoped by level, the null hypothesis cannot be rejected with 98% confidence interval (data whiskers on a deterministic simulation) for trials on data rate b2 and b3. For practical matters, the difference between methods is only visible at data rate level 4 (b4), and estimations are below 4%. Yet data rate levels at b4 are 340Mbps for the experiment trials with minimal size packets, the 4% result translates to 650,000 packets and 45MB of data per second.

Table 5. 3 Dropped Packet Rate hypothesis testing on Experiment 1

Inverse Null Hypothesis	Scope	Estimate	p-value
p(*PktDrpRatepf_pkt*) < *p*(*PktDrpRateAppPfDL*)	Experiment 1	0.01508	4.14E-12
p(*PktDrpRatepf_pkt*) < *p*(*PktDrpRateAppPfDL*)	b1	0	N/A
p(*PktDrpRatepf_pkt*) < *p*(*PktDrpRateAppPfDL*)	b2	0.00021	0.24550

p(*PktDrpRatepf_pkt*) < *p*(*PktDrpRateAppPfDL*)	b3	-0.00600	0.92221
p(*PktDrpRatepf_pkt*) < *p*(*PktDrpRateAppPfDL*)	b4	0.05349	2.36E-18

5.2.3. CPU Utilization

As shown in the graphs for CPU Utilization from Figure 5.10, there is evidence to accept original hypothesis, which states that the kernel module of the SUT application would reduce CPU utilization when using PF_PACKET.

$$H_0: p(CPU_{pf\_pkt}) = p(CPU_{appPfDL})$$
$$H_A: p(CPU_{pf\_pkt}) < p(CPU_{appPfDL})$$

Table 5.4 shows the results of the tests performed on the SUT configuration selection when PF_PACKET is enabled in the OS for each of the 4 different data rate levels, compared to the user-space application that is not able to use any PF_RING hooks. The table evidently indicates that the PF_PACKET utilizes significantly less CPU than the AppDataPfRingLogger user-space application. This is between 20%-35% less, at 98% confidence interval. Since PF_PACKET does not buffer any write, it is not penalized the way AppDataPfRingLogger is set in this simulated solution. To accomplish the work AppDataPfRingLogger must copy the data to a stream buffer and perform the expensive copy from the user-space to the kernel-space without any PF_RING hooks even though the data packets are immediately discarded by the file system. The reason the PF_PACKET requires less CPU in this specific experiment simulation has no practical value in real-world performance; however, the data point to write efficiency within the kernel-space and indicate CPU resource utilization in the SUT is handling the data packets to the file system in a non-blocking scenario.

Table 5. 4 CPU Utilization hypothesis testing on Experiment 1

Null Hypothesis	Scope	Estimate	p-value
p(*CPU pf_pkt*) < *p*(*CPU AppPfDL*)	Experiment 1	26.02810	0
p(*CPU pf_pkt*) < *p*(*CPU AppPfDL*)	b1	19.56981	0
p(*CPU pf_pkt*) < *p*(*CPU AppPfDL*)	b2	24.55692	1.57E-283
p(*CPU pf_pkt*) < *p*(*CPU AppPfDL*)	b3	22.07353	1.20E-77
p(*CPU pf_pkt*) < *p*(*CPU AppPfDL*)	b4	35.65992	0

5.2.4. Memory Utilization

As per Section 5.1.4, memory utilization is static in this experiment too and does not depend on the non-controllable factors or any workload levels. Memory usage is not changed for PF_PACKET and, in the case of writing data to dev/null, the additional memory and associated copies used by the AppDataPfRingLogger user-space application both memory and CPU resources consumption is higher without any major impact.

5.2.5. Summary Analysis

Given the deterministic and simulated nature of Experiment 1, the data results and analysis apply to SUT capability closest to real-world system capability. Both the data and the graphs show that a kernel-space data transmission capability PF_PACKET can derive more benefit from faster data transmission and faster storage than a user-space application. The PF_PACKET OS build is compared with a user-space application. Their capabilities are helpful to determine where the improvement is located at the SUT.

5.3. Results and Analysis of Experiment 2

Results of Experiment 2 gather the information of the SUT configuration with PF_RING kernel module while comparing data transmission with a user-space application. This experiment measures performance of the data transmitted, stored in dev/null and with a data query process. The key metrics for this experiment are again categorized by an experimental trial set. Same trial sets as previous experiments for the sake of comparison with 4 independent trials for 4 data bit rates divided into buckets (b1-b4). This scenario is an OS build analogous to the one used for Experiment 1, but with PF_RING kernel module enabled instead of PF_PACKET (which is disabled). The experiment relates closely to representing real-world workload and network data traffic of a system. The SUT received the data packet sizes configured to be randomly selected for all the sets. Figure 5.15, Figure 5.16, and Figure 5.17 depict an overview of the key results for the

experiment while Figure 5.18 is a representation of an overview of the key query-based (delay) results for the experiment to show differences between the kernel-space applications versus the user-space ones.

5.3.1. Experiment 2 overview

The graphs shown in Figure 5.15, Figure 5.16, and Figure 5.17 is a summary of the key metric results for Experiment 2. In this experiment focus is on PF_RING capabilities to be compared against a user-space application while capturing enough data to be comparable with Experiments 0 and 1. All experiment trial sets use randomly sized packets, ranging from 64 Bytes to 1500 Bytes. During trials at all the 4 data bit rates, the AppDataPfRingLogger application does a query process on the SUT repeatedly searching for marked packets of interest on the ring buffer. Experiment focus is on the effect of query delays on the SUT key metrics.

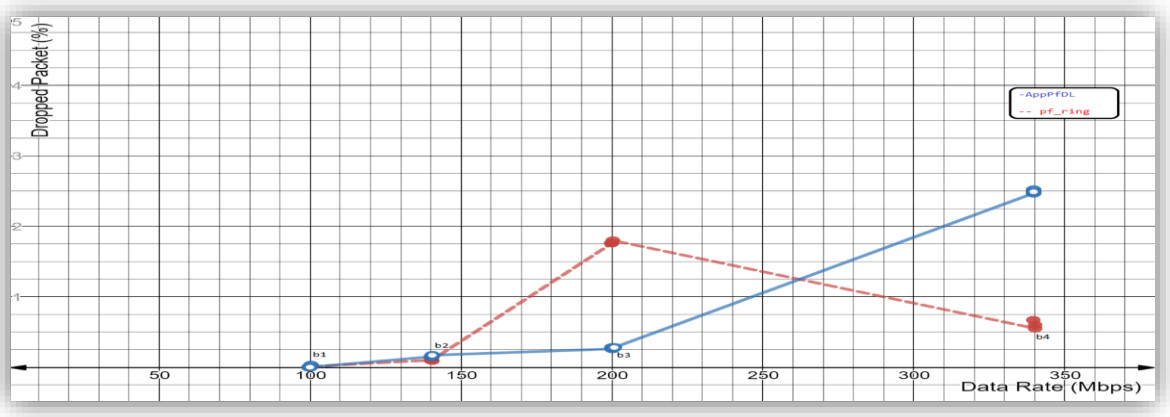

Figure 5. 14 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. Dropped packets (%). Red = PF_RING.

Figure 5.15 shows the key metric of dropped packets between PF_RING and the user space application. A result that stands out is that at higher data rate PR_RING data loss is low.

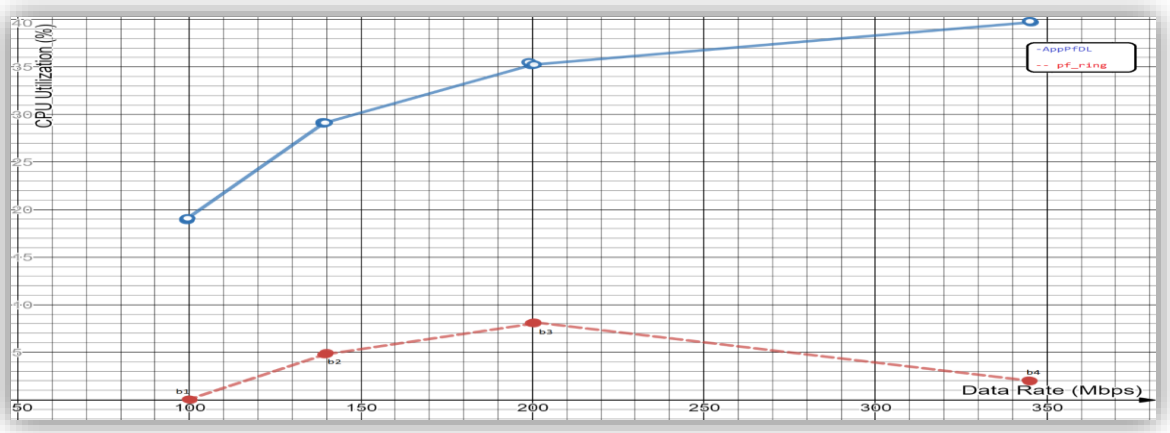

Figure 5. 15 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. CPU utilization (%). Red = PF_RING.

Figure 5.16 shows the key metric of CPU utilization between PF_RING and the user space application. PF_RING CPU utilization is low compared to the user space application at the same data rate.

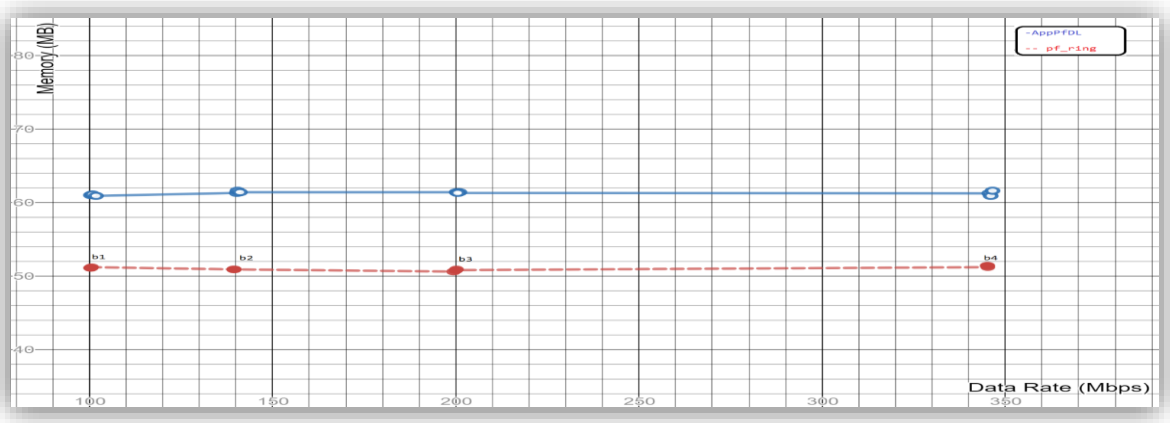

Figure 5. 16 Summary graph of Experiment 2 SUT key metrics between pf_ring and AppDataPfRingLogger. Memory utilization (MB). Red = PF_RING.

Figure 5.17 shows the key metric of Memory utilization between PF_RING and the user space application. PF_RING Memory utilization is lower by 19% (52 MB) than AppDataPfRingLogger (62 MB).

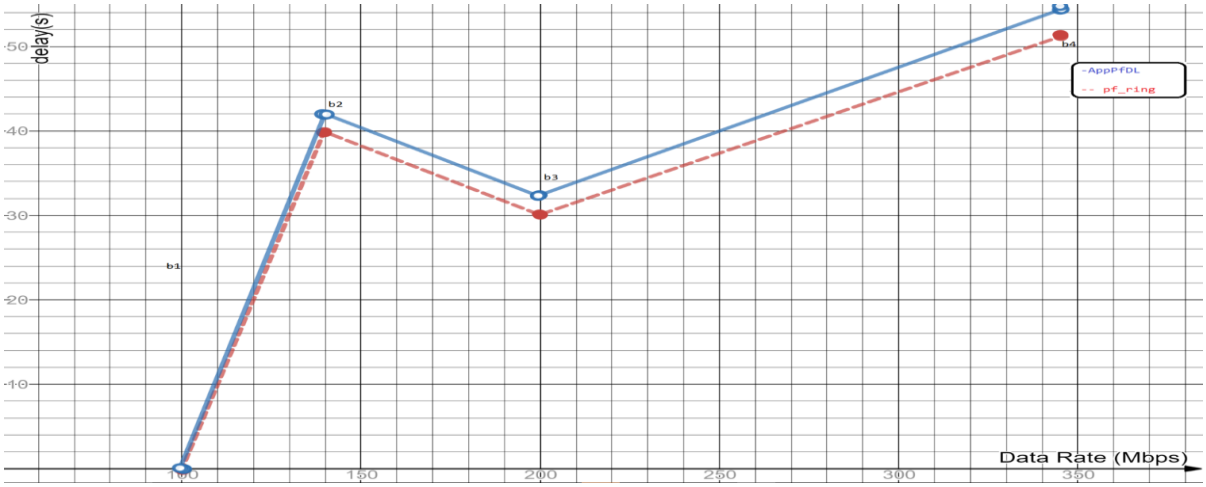

Figure 5. 17 Key query metric (query delay) for PF_RING kernel module.

Figure 5.18 shows the key metric of query delay between PF_RING and the user space application. Query response by PF_RING OS enabled to the user space application is almost the same at high data rates.

5.3.2. Query Metrics

The way the queries are performed on the SUT records the number of queries completed, the time delay for the response and the number of data packets marked as found. Only results for data query delay are presented in this experiment. The workloads are combined to exercise the SUT. Figure 5.18 shows an overview of query delay indicating the following:

- As the data rate level increases, the number of queries answered decreases and the delay in the query responses increases proportionally.

The goal of this research with respect to time delay is that data at the kernel-space of the PF_RING capability responsiveness satisfy live queries for the data being captured in the ring. Therefore, satisfaction of this goal is achieved by the lack of results that show a rejection of the null hypothesis:

$$H_0: p\left(QryDelay_{pf\_ring}\right) = p\left(QryDelay_{appPfDL}\right)$$

$$H_A: p\left(QryDelay_{pf\_ring}\right) > p\left(QryDelay_{appPfDL}\right)$$

Table 5.5 shows test results for the query delay performed on the PF_RING/OS enabled experiment for the above metrics and data bit rate settled by the SUT configuration. In this case, there is insufficient evidence (p value is bigger than 0.05) that the null hypothesis can be rejected; therefore, the table indicates that the objective of the query performance is better than a user-space application or compared to PF_PACKET when using a kernel-space implementation.

Table 5. 5 Query SUT performance performed on Experiment 2

Null Hypothesis	Scope	estimate	p-value
p($QryDelay_{PF\_RING}$) > p($QryDelay_{appPfDL}$)	Experiment 2	-2.0023	0.33484

5.3.3. Data Drop Rate

The data drop rate describes the reliable behavior of the SUT. It also describes whether the system is performing well under high data volume or SUT stress. In Experiment 2, the relationship between data drop rate and CPU Utilization is as shown in Figure 5.15. As the data rate increases, the data drops decrease compared to the user-space application. Compared to Experiment 1, its behavior is better: under the same bit rate levels (b4), using PF_RING less data is dropped as compared to PF_PACKET.

Table 5.6 shows the results depicted in Figure 5.15. It is a test performed on data drop and data bit rate for the SUT selection using PF_RING. There is no evidence to reject the null hypothesis indicating that SUT selection does not impact data drop rates. While the AppDataPfRingLogger data dropped rate is estimated to be less than 0.5% better on the b3 level than PF_RING it is not conclusive on all bit levels shown in Figure 5.15.

Table 5. 6 Data dropped packet rate hypothesis testing on Experiment 2

Null Hypothesis	Scope	Estimate	p-value
p(*PktDrpRate pf_ring*) < *p*(*PktDrpRateAppPfDL*)	Experiment 2	-0.00964	0.8276

A potential reason for the comparatively better performance of AppDataPfRingLogger at the user-space is the PF_RING hooks write buffering. Since the PF_RING kernel modules use the memory slot directly for writing the file system, the writes are not blocked and are quickly consumed by the free rings in the algorithm. This effect is leveraged by the user-space application as the ring is freed quickly by a simple `memcpy()` providing another buffer and reducing the cost of transferring data to userland.

5.3.4. CPU Utilization

Table 5.7 shows the results of testing performed on CPU Utilization by the SUT with PF_RING on Experiment 2. There is strong evidence to accept the null hypothesis indicating that the kernel-space SUT configuration with OS PF_RING enabled reduces CPU utilization when data traffic workloads are present. The estimated reduction of the kernel-space SUT is at 5% with a 98% confidence due to the deterministic simulation when running the experiments. The 5% utilization difference is significant compared to the PF_PACKET solution. This indicates that a kernel-space improved algorithm incurs in a reduced contention of system resources in this scenario.

Table 5. 7 CPU Utilization hypothesis testing in Experiment 2.

Null Hypothesis	Scope	Estimate	p-value
p(CPU pf_ring) < p(CPUAppPfDL)	Experiment 2	3.39733	2.90E-08

5.3.5. Memory Utilization

As shown in Figure 5.17, the memory utilization indicates the kernel-level buffering benefits mentioned in Section 3.2. The memory used by AppDataPfRingLogger is 62 MB which is higher than the PF_RING kernel module. Overall this indicates that, in terms of memory, both Experiments 1 and 2 show that kernel-space solutions memory utilization is somewhat static and not dependent on system factors if implementations are correct. SUT selection can increase buffering by initializing PF_RING's ring buffer with more slots - which may not be comparable between experiments.

5.3.6. Summary Analysis

On Experiment 2 PF_RING's implementation certainly impacted the data drop rate when the data bit rate increases: the number of data drops is reduced. Although not conclusive due to other bit rate levels where the data drop rate is higher, overall, the hypothesis is accepted and conclusive in certain data bit rates submitted to the SUT with PF_RING. The Operating System with PF_RING enabled performs better when compared with Experiment 1 results. On CPU utilization, Experiment 2 demonstrates better performance than AppDataPfRingLogger with an estimated reduction of 5% of the kernel-space SUT configuration and comparatively better versus CPU Utilization on Experiment 1 SUT (PF_PACKET configuration).

5.4. Overall Analysis

To recapitulate, the key aspects of the PF_RING kernel module during the experiments are:

- The PF_RING kernel module improves data transmission performance to user-space either by:
    - Transmitting significant amount of data packets, with increased CPU utilization due to the higher data bit rate levels (b3-b4).
    - Transmitting the same percentage of data packets but with reduced CPU utilization when writing to the virtual simulated storage device.
- The PF_RING kernel module algorithm uses significantly less memory; which is relevant in today's embedded devices. On other types of applications, computing power takes priority on top of memory utilization, in which case our results are less relevant.

- The PF_RING kernel algorithm does not impact user-space queries negatively, responding adequately and improving the system compared to a user-space application using PF_PACKET.

Chapter 6. Discussion

This chapter presents the overall conclusions reached from the research. Section 6.1 considers each research objective and determines whether it was met. The significance of the research results is presented in Section 6.2. Finally, Section 6.3 mentions possible directions for related research.

6.1. Conclusions

6.1.1. PF_RING Kernel-Space Capability in an OS

The first goal of the research reported in this thesis was to study the possibility of replacing a de-facto kernel-level data transmission component (PF_PACKET) by an alternative (PF_RING) to determine whether this impacts performance. The design and build of an experimental setting were completed using Simics Simulation models that allowed an objective comparison of basic functions of a PF_RING OS-enabled capability against another build of an PF_PACKET OS-enabled.

To accomplish the goal, a PF_RING kernel module was built in a Linux Operating System to expose the ring functionality so that the memory could be mapped and addressed by other kernel modules, and also provide the necessary functions to select the ring from a user-space application or from the kernel module itself. With the PF_RING kernel module in place, it was possible to access the ring buffer itself, initialize it and have it recognized by other kernel modules in the kernel-space. With PF_RING enabled, it was possible to consume the ring´s memory in slots in the same way pfring's library functionality does it in user-space. The experiment enabled to extend the role of a typical Linux kernel module beyond its traditional role, so it was able to access the ring through user-space application and to write the data to it or any other simulated virtual devices. The experiment proceeded similarly with PF_PACKET, using a suitable algorithm architecture to be able to compare it against PF_RING.

The PF_RING algorithm as a form of a kernel module was loaded into a Linux system and tested against simulated real network traffic. This demonstrated the ability to transmit data between kernel space and user space and vice versa, decide which data to store or to write into the simulated virtual device and maintain data lifecycle through live network simulation tests, thus accomplishing the first objective of the research.

6.1.2. Reduce Data Drop Rate When Using PF_RING in a User-Space App

The research subject through the experiments was the goal to improve upon the performance characteristics of the PF_PACKET data transmission algorithm by replacing it with PF_RING, employing the kernel-space data transmission capability in place while using a user-space implementation to leverage it. The specific metric for improvement, in this case, was the reduction of the data drop rate during simulated virtual time running random network traffic. Testing of the system with live simulated network traffic revealed that the PF_RING kernel module reduced the data drop rate by an average of 9.0% with 98% confidence, using Simics as a deterministic environment and experimentation vehicle. In cases where the user-space application was using PF_RING, the experiment scenario produced a favorable reduction in CPU utilization compared with PF_PACKET.

The practical use of PF_RING and a user-space application that could hook with the algorithm implementation at kernel-space were demonstrated on a system simulating real network data traffic, thus satisfying the conditions for meeting the research objective.

6.1.3. Reduce CPU Utilization When Using PF_RING in an OS

The goal to reduce the resource demand of the system while using PR_RING by reducing CPU utilization was another research subject. Reduction of CPU utilization was possible due to the removal of bottlenecks, removing multiple memory copies and leveraging the transition from user-space to kernel-space and

vice versa to write data either to disk or virtual devices. However, from the experiments, CPU utilization reduction was observable when writing to virtual devices (dev/null) and when data packets were not so small in size. At those times the average utilization was at only 4%, with 98% confidence, according to measures taken on the Simics simulation environment.

As discussed in Chapter 5, CPU utilization is dependent on different factors, one of which is data drop rate. By reducing data drop rate by 9% on average, data transmission freed resources to handle the additional workload. During the test scenarios, the PF_RING kernel module utilization increased to 8% on average with a 98% of confidence. The increment on CPU utilization is an acceptable trade-off when the data transmission accuracy is more important in higher data network traffic situations. This was proven true when the data drop rate decreased in a higher proportion than the growth in CPU utilization.

After considering the behavior of the SUT and the benefits of building an OS with PF_RING kernel module enabled, CPU utilization for situations with high data transmission and low data drop rate impact, it is concluded that the proposed system confirms the research hypothesis.

6.1.4. Reduce Memory Utilization When Using PF_RING in an OS

The goal to reduce the resource demand on system memory utilizing PF_RING built within an OS was another research subject. Memory utilization reduction was possible due to PF_RING kernel-space module direct use of the ring memory slots in the call of any user-space application or even between other kernel modules.

The kernel-space PF_RING implementation did not use any stream buffering since the data packets were transmitted directly to the ring, which was memory mapped. On Simics, the memory was instantiated as an object since Simics sees any device memory mapped in the system, including memory. According to the experiments, file system writes within the kernel space appear to be cheaper than any additional memory copies required on normal user-space applications. By avoiding data

copies while using the PF_RING ring (which is not available in PF_PACKET), the kernel implementation reduced the memory for every task by 10MB which corresponds to a 17% reduction with a 100% confidence using a simulation. The static memory reduction accomplished the research goal.

6.1.5. On Research Methods and Tools

The Full System Simulator experiments performed on top of Simics made it possible to create a testing environment that was easy to use and enabled the handling of repetitive trials on experiments. Setup and configuration required time at the beginning but it was only done once on this investigation. This allowed for every new experiment setup to be as simple as launching the program and start running the simulation, which allowed hardware-free experimental tools. A Full System Simulation environment facilitated the isolation of the hardware factor reducing any dependencies on physical measurement equipment or laboratory availability while being able to launch Simics on any host at any place and time. In addition, having an isolated experimental environment was possible, thus preventing exogenous perturbations from introducing errors to the experiments and allowing systematic data gathering for statistical analysis.

As an academic tool, Simics provides full-featured free licenses for researching new technology architectures and software. The ability to control experimental factors and isolate observable variables was key for this research and sets a precedent for future work on many computer science research arenas.

Factorial analysis and statistical hypothesis testing permitted to introduce a methodological framework that can be ported to examine similar or related hypotheses. A systematic series of experiments allow to gather results with high confidence, which in turn permits to conclude objectively. This is a framework that can be ported to future work and similar research.

6.2. Engineering Significance

This research provides the Costa Rica Institute of Technology and industry interested in exploring and innovating on IoT, Embedded Devices and Networking appliances with an improved method for efficiently passing data from Kernel space to user space in an (Embedded) Operating System. An Operating System with PF_RING enabled benefits the overall security of the information, its integrity, its availability, its consistency, its reliability, and data transmission efficiency in high-density data traffic networks. Any improvement in the performance of data transmission reduces the processing footprint and increases the effectiveness of the data to be available in any application. The simulation experiments, with the PF_RING algorithm enabled on Linux Operating System improves the performance of the data transmission process, and provides indications that it scales better than PF_PACKET in highly-connected and dense networking data traffic - with increased performance in memory usage, less data drops and shorter data query delays.

PF_RING Linux kernel module presents a stable, high-performance application that can transmit data with high reliability making it available at a ring buffer or even written to an I/O disk. This research provides a unique demonstration of improved data transmission on high-density network data traffics where the principal goal is to have the data available for consumption by a user space application as fast and reliable as possible, and it sets a path for future research to begin addressing the performance issues of networked devices facing higher amounts of data traffic coming - beyond just capturing some data and sending it to be processed and displayed by an application dashboard.

Even though the PF_RING OS enabled data drop rate reduction of 9% may seem small when compared to PF_RING, that percentage delta could translate to losses that approach almost 100 Mbps on a Gigabit Ethernet Link. Malicious methods to attack and obfuscate a network connected device are to flood it with an overwhelming amount of harmless data to slow the device and breach the secured

network. When network flood traffic increases, the probability of key data packets being lost is high and the connected devices processing power and its value decrease. In light of a decreasing data drop rate, 9% is a significant achievement of this research.

A simulation environment that provides determinism and makes it easy to apply experimental statistical methods offers the academic community with a baseline for future investigations using Simics as a tool to reproduce experiments performed in this research or to apply the lessons learned in other computer science research areas.

6.3. Future Work

Although not all possibilities are listed, the following subsequent research topics are suggested:

- Scale the PF_RING Operating System build from just one simulation to 1000+ simultaneous simulations running on a powerful server, to study scalability and cluster architectures.

- Modify the PF_RING kernel module to add a robust filtering capability for IPv6 data traffic, controllable by the application or from the kernel space.

- Evaluate performance of other data transmission algorithms, such as Zero Copying (ZC), against PF_PACKET and PF_RING, using an experimental methodology analogous to the one performed in this research.

- Leverage PF_RING capabilities built in the Operating System located in the kernel space to build a set of indexes of marked data packets to improve data query response and reduce delay.

- Improve PF_RING kernel module implementation by adding crash avoidance before the operating system reaches a failure state.

Appendix A. AppDataPfringLogger changelog

```
Index : daemonlogger − 1.8.1/ daemonlogger.c
===================================================================
−−− daemonlogger − 1.8.1/ daemonlogger.c ( revision 88)
+++ daemonlogger − 1.8.1/ daemonlogger.c ( revision 89)
@@ −216,4 +216,8 @@
static int maxpct ;
static int prune_flag ;
+/*
+ * Add log to null virtual device
+ */
+ static int logtonull = 0;
static char * interface;
@@ −415,4 +419,11 @@
{
time_t currtime ;
+ // Virtual device
+ char * nullfile = "/dev/null " ;
+
+ i f ( logtonull == 1) {
+ return nullfile;
+ }
+ / / End of added code
memset( logdi r , 0 , STDBUF) ;
@@ −1137,4 +1148,5 @@
printf ( " −u <user name> Set user ID to <user name>\n " ) ;
printf ( " −v Show daemonlogger version\n " ) ;
+ printf ( " −X Log to /dev/null\n " ) ;
}
@@ −1153,5 +1165,5 @@
while ( ( ch = getopt ( argc , argv ,
− " c : df : Fg : hi : l :m:M: n : o : p :P: rR : s : S : t :T: u : vz "))!= − 1)
+ " c : df : Fg : hi : l :m:M: n : o : p :P: rR : s : S : t :T: u : vXz "))!= −1)
{
switch ( ch )
@@ −1300,4 +1312,7 @@
prune_flag = PRUNE_OLDEST_IN_RUN;
break ;
+ case 'X ' : /* Added log to null case* /
+ logtonull = 1;
+ break ;
default :
break ;
```

Appendix B. Simics Configuration Script

```
if not defined host_class { $host_class = "jsl" }
## Search paths should uses %simics% to make sure it will on package
add-directory (lookup-file "%simics%/targets/x86-jsl/")
add-directory (lookup-file "%simics%/targets/x86-jsl/images/")

$pmc_image_folder = (lookup-file -query "%simics%/targets/x86-jsl-
extensions/wb/pmc/images/")
if ($pmc_image_folder != FALSE) {
    add-directory $pmc_image_folder
}
$ish_image_folder = (lookup-file -query "%simics%/targets/x86-jsl-
extensions/wb/ish/images/")
if ($ish_image_folder != FALSE) {
    add-directory $ish_image_folder
}
$hda_image_folder = (lookup-file -query "%simics%/targets/x86-jsl-
extensions/wb/hda/images/")
if ($hda_image_folder != FALSE) {
    add-directory $hda_image_folder
}

## Common images TODO: include to Common.include
$common_images = (lookup-file -query "%simics%/common_images/")
if ($common_images != FALSE) {
    add-directory $common_images
}

## Common include for images
$common_include = (lookup-file -query "%simics%/targets/Common.include")
if ($common_include != FALSE) {
    run-command-file $common_include
}

## Internal definitions
if not defined wb       { $wb         = NIL }
if not defined wb_force { $wb_force   = NIL }
if not defined swm      { $swm        = NIL }

# Loading release helper module
load-module release-helper

### Software one script settings

if ($swm == NIL) {
    # default SW config
    run-command-file "%script%/jsl-sw.include"
} else {
    # user defined SW config
    # look if it is user config
    $user_swm = (lookup-file -query $swm)
    if ($user_swm != FALSE) {
        run-command-file $user_swm
    } else {
        # look if it is one of the base
        $base_swm = (lookup-file -query "%simics%/targets/x86-jsl/"+$swm)
        if ($base_swm != FALSE) {
            run-command-file $base_swm
        } else {
            interrupt-script "SW MANIFEST FILE WAS NOT FOUND"
        }
    }
    # What is not redefined will be taken from default (???)
    run-command-file "%script%/jsl-sw.include"
}

### WB CONFIGURATION one script settings
```

```
if ($wb != NIL) {
    if ($wb_force) {
        rh-load-wb $wb -f
    } else {
        rh-load-wb $wb
    }
} else {
    echo "INFO: No WB configuration is specified, running just base"
}

### WB models disabled by default
if not defined gfx_wb_enable    { $gfx_wb_enable   = FALSE }
if not defined csme_wb_enable   { $csme_wb_enable  = FALSE }
if not defined hda_wb_enable    { $hda_wb_enable   = FALSE }
if not defined ish_wb_enable    { $ish_wb_enable   = FALSE }
if not defined pmc_wb_enable    { $pmc_wb_enable   = FALSE }
if not defined punit_wb_enable  { $punit_wb_enable = FALSE }

@wb_keys = ["csme", "punit"]
@wb_suffix = "_wb_enable"
@for w in wb_keys:
    if conf.sim.env[w + wb_suffix]:
        location = os.path.join("%simics%", "targets", "x86-" + simenv.host_class + "-
extensions", "wb", w)
        f = SIM_lookup_file(location)
        if f:
            print "Add WB folder " + f
            #SIM_run_command("add-directory " + f)
            conf.sim.simics_path += [f]
        f = SIM_lookup_file(os.path.join(location, "images"))
        if f:
            print "Add WB image folder " + f
            #SIM_run_command("add-directory " + f)
            conf.sim.simics_path += [f]

### Check all images there
if not defined sxp_image        { $sxp_image = "" }
if not defined cd_image         { $cd_image  = "" }

if not defined required_files { $required_files = [] }
$required_files += [ ["BIOS ROM image",      $bios],
                     ["Dummy ROM image",            "vbios.stub"],
                     ["Board ID",                   "empty.brdid"],
                     ["Board ID",                   "karkomx0.brdid"],
                     ["ENH VGA GOP driver image",   "QemuVideoDxe.rom"],
                     ["ENH VGA BIOS Rom image",     "enh_accel_vgabios.bin"]]

if ($pmc_wb_enable != FALSE) { $required_files += [ ["PMC ROM", $pmc_bootstrap_image] ] }
if ($punit_wb_enable  != FALSE) {
    if (lookup-file -query "%simics%/targets/x86-jsl-
extensions/wb/punit/images/"+$punit_pcode_path) {
        add-directory -prepend (lookup-file "%simics%/targets/x86-jsl-
extensions/wb/punit/images/"+$punit_pcode_path)
    }
    $required_files += [ ["PUNIT ROM", $punit_pcode_img] ]
}

if ($ish_wb_enable  != FALSE)   { $required_files += [ ["ISH ROM", $ish_rom_image] ] }
if ($hda_wb_enable  != FALSE)   { $required_files += [ ["HDA1 ROM", $hda_Fw1File] ]  }
if ($hda_wb_enable  != FALSE)   { $required_files += [ ["HDA2 ROM", $hda_Fw2File] ]  }
if ($csme_wb_enable != FALSE)   { $required_files += [ ["ME ROM", $me_mia_bios] ]    }

# BASE MISC FILES
if ($bios2 != "")               { $required_files += [ ["BIOS2 (second component)",
$bios2] ] }
if ($disk_image != "")          { $required_files += [ ["Disk image", $disk_image] ]
}
if ($cd_image != "")            { $required_files += [ ["CD image", $cd_image] ]
}
if ($sxp_image != "")           { $required_files += [ ["SXP image", $sxp_image] ]
}
```

```
$missed_files = ""
foreach $file in $required_files {
    try {
        if not (lookup-file $file[1]) { throw; }
    } except {
        echo ("File not found. " + $file[0] + " : " + $file[1])
        if $missed_files != "" { $missed_files += ", " }
        $missed_files += $file[1]
    }
}
if $missed_files != "" { interrupt-script "Some files have not been found: " +
$missed_files }

### Print SW configuration
echo "INFO: SW CONFIGURATION"
# OS image
if ($disk_image != "") {
    echo ("INFO: OS $disk_image = " + (lookup-file $disk_image))
} else {
    echo ("INFO: NO OS. No $disk_image is specified. Using empty image")
}
# BIOS
echo ("INFO: BIOS/IFWI $bios = " + (lookup-file $bios))
if ($bios2 != "") {
    echo ("Using $bios2 (second component) = " + (lookup-file $bios2))
}

if ($pmc_wb_enable   != FALSE) {echo ("INFO: PMC ROM  $pmc_bootstrap_image = " + (lookup-
file $pmc_bootstrap_image))}
if ($punit_wb_enable != FALSE) {echo ("INFO: PUNIT ROM $punit_pcode_img    = " + (lookup-
file $punit_pcode_img))    }
if ($ish_wb_enable   != FALSE) {echo ("INFO: ISH ROM $ish_rom_image        = " + (lookup-
file $ish_rom_image))      }
if ($hda_wb_enable   != FALSE) {echo ("INFO: HDA1 ROM $hda_Fw1File         = " + (lookup-
file $hda_Fw1File))        }
if ($hda_wb_enable   != FALSE) {echo ("INFO: HDA2 ROM $hda_Fw2File         = " + (lookup-
file $hda_Fw2File))        }

if $hda_wb_enable {
      $external_hda = TRUE
      $r2s_enable = TRUE
}

### CONFIG SET UP, let's run everything

# Base
run-command-file "%script%/jsl-system-pre.include"

# All PRE work for integrating WB components
run-pre-wb

### This is only ONE instantiation
instantiate-components

# Base
run-command-file "%script%/jsl-system-post.include"

# All POST work for integrating WB components
run-post-wb

sim->handle_outside_memory = TRUE
```

Bibliography

1. Berkeley Packet Filters. (2018). Linux Socket Filtering aka Berkeley Packet Filtering (BPF). Kernel.org. Retrieved from https://www.kernel.org/doc/Documentation/networking/filter.txt, Date April 218.
2. Braun Lothar; Didebulidze, Alexander; Kammenhuber, Nils; Carle, Georg. (2010). *Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware*. ACM SIGCOMM conference on Internet measurement. Technische Universität München. NY. ISBN: 978-1-4503-0483-2
3. Cerwall, Patrick. (2016). On the pulse of networked society. Ericsson Mobility Report. EAB-16:006659 Uen, Revision A © Ericsson AB.
4. Collins, L. M., Dziak, J. D., Kugler, K. C., & Trail, J. B. (2014). *Factorial experiments: Efficient tools for evaluation of intervention components.* American Journal of Preventive Medicine.
5. Deri, Luc. (2011). *Improving Passive Packet Capture: Beyond Device polling*. NETikos, Pisa, Italy.
6. DiBona, Chris. (2017). *Open Sources: Kernel Space and User Space*. *Chapter 84* Safari Books. Retrieved from https://www.safaribooksonline.com/library/view/open-sources/1565925823/ch09s04.html, Date April 2018.
7. Engblom, Jakob. (2010). *What is Simics, really?*. Wind River Simics Blog Network. Retrieved from http://blogs.windriver.com/engblom/2010/04/what_is_simics_really.html, Date: April 2018.
8. Finley, Kilnt. (2016). *Linux took over the web. Now it's taking over the world*. Wired Business. Retrieved from https://www.wired.com/2016/08/linux-took-web-now-taking-world/, Date: April 2018.
9. Hazewinkel, Michael. (2001). *Statistical hypothesis, verification of Encyclopedia Mathematics.* Springer Science + Business Media B.V. / Kluver Academic Publishers. ISBN 978-1-55608-010-4.
10. Hwang, Jong-Sung; Choe, Young Han. (2013). Smart Cities Seoul: a case study (PDF). ITU-T Technology Watch. Retrieved from https://www.itu.int/dms_pub/itu-t/oth/23/01/T23010000190001PDFE.pdf, Date: October 2016.
11. IDC. (2014). *Data Growth, Business Opportunities, and the IT imperatives*. Retrieved from https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, Date: July 2014.
12. IEEE; 802.11 wireless protocol standard. (2015). Retrieved from https://en.wikipedia.org/wiki/IEEE_802.11ac, Date: March 2018.
13. Insolvible, Gianluca. (2001). *The Linux Socket Filter: Sniffing bytes over Network.* The Linux Journal, Issue #86. Mayo 2001. Retrieved from http://www.linuxjournal.com/article/4659#navigation, Date: August 2014.
14. Kan, Stephen H. (2003). *Metrics and models in Software Quality Engineering*. Chapter 13, p359.

15. Keller, Arianne. (2008). *Kernel space – User space Interfaces*. Retrieved from http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html, Date: June 2014.
16. McCanne Steven; Jacobson Van. 1993. *The BSD packet filter: a new architecture for user-level packet capture*. USENIX Association, Berkeley, CA, USA. Retrieved from http://www.tcpdump.org/papers/bpf-usenix93.pdf, Date: April 2014.
17. Mohri, Mehryar; Rostamizadeh, Afshin; Talwalkar, Ameet. (2012). *Foundations of Machine Learning*. MIT Press, Boston, MA, USA. ISBN 9780262018258.
18. Nuzzo, Regia. (2014). *Scientific method: Statistical errors.* Nature. p506.
19. Porter, Thomas. (2005). *The Perils of Deep Packet Inspection*. Security Focus. Retrieved from http://www.symantec.com/connect/articles/perils-deep-packet-inspection, Date: June 2014.
20. Qemu. (2016). *Qemu hypervisor*. Retrieved from https://en.wikipedia.org/wiki/QEMU, Date: March 2018.
21. Qmet201*.* (2014). *Factorial designs.* New Zealand Lincoln University. Retrieved from https://library2.lincoln.ac.nz/documents/Factorial-Design.pdf, Date: December 2014.
22. Rivest, Ronald L. (1990). Cryptology. Handbook of Theoretical Computer Science. Elsevier.
23. Ruparelia, Nayan B. (2010). *Software Development Lifecycle Models*. Hewlett Packard Enterprise Services. ACM Volume 35, Issue 3. New York, USA.
24. Schultz, Michael. (2011). *A measurement study of packet reception using Linux*. Washington University in St. Louis. Retrieved from http://www.cse.wustl.edu/~jain/cse567-11/ftp/pkt_recp/#pfpacket, Date: July 2014.
25. Simics. (2016). *Wind River Simics Full System Simulator*. Retrieved from https://en.wikipedia.org/wiki/Simics, Date: March 2018.
26. Streubel, Jennifer. (2003). *What is Computer Science?.* Retrieved from http://www.cs.bu.edu/AboutCS/WhatIsCS.pdf, Date: September 2014.
27. Wikipedia, the free encyclopedia. (2018). *Comparison of Operating Systems*. Retrieved from https://en.wikipedia.org/wiki/Comparison_of_operating_systems, Date: April 2018.
28. Wikipedia, the free encyclopedia. (2018). *Internet protocol suite*. Retrieved from https://en.wikipedia.org/wiki/Internet_protocol_suite, Date: April 2018.
29. Wind River. (2010). *Wind River Hypervisor Product Note*. Retrieved from https://www.windriver.com/products/product-notes/wind-river-hypervisor-product-note.pdf, Date: February 2018.
30. Wood, Rupert. (2013).*Research Forecast Report; Wireless network traffic worldwide: forecast and analysis 2013 – 2018*. Analysis Mason. Retrieved from http://www.analysysmason.com/Research/Content/Reports/Wireless-traffic-forecasts-Oct2013-RDTN0/samples-TOC/, Date: July 2014.

Vita

Dennis Rodriguez attended the Costa Rica Institute of Technology. He graduated with an Electronics Engineering *LIcenciate* (Post-Bachelor's) Degree in February 2010.

His first engineering experience was working as an Embedded Software Development Engineer at Hewlett Packard in the Research and Development Department, where he contributed to the Networking division products from Wireless to Wired solutions. His contributions spanned in record selling products for about eight years with the company. After his tenure, he decided to move to Intel Corporation to lead and develop the next virtual platform software models running in Wind River Simics Full System Simulation.

On the personal side, Dennis has a lovely family which captures all his attention in his spare time. Although he is a Software Development Manager, his coding skills and passion remain alive learning new programming languages.