THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SHORTCUTTING THE LINUX NETWORK STACK: EXPLORING LATENCY AND THROUGHPUT BENEFITS OF eBPF

MATTHEW S SICKLER
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Thesis Supervisor

John Hannan
Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

In the age of growing demand for data, there is also a parallel demand for the data to be provided instantaneously. Quick access to information has become a paramount pillar of the information age. As a result, system architects have invented new methods of speeding up their servers to handle requests faster than ever before. As hardware itself also quickens, some engineers turned to another source of potential slowdown, the operating system (OS), to cut corners. The creation of the Berkeley Packet Filter and, later, the Extended Berkeley Packet Filter (eBPF, though often shortened to just BPF) allow for just that. Combined with the eXpress Data Path (XDP) toolset, packet latency and throughput from client to server achieve better rates than relying upon the kernel to handle packets in many scenarios. This thesis aims to explore avenues of performance benefit and the potential applications and limitations of eBPF.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iv

ACKNOWLEDGEMENTS v

1. INTRODUCTION 1

    1.1 History of Packet Filters 2
    1.2 Purpose 4

2. BACKGROUND 5

    2.1 Packets 5
    2.2 The Network Stack 6
    2.3 User Datagram Protocol 8
    2.4 The Linux Network Stack 10
    2.5 The Kernel and Kernel Modules 11
    2.6 The Inner Workings of eBPF 13
    2.7 eXpress Data Path and eBPF 16
    2.8 BPF Compiler Collection Toolset 17

3. EXPERIMENTS 18

    3.1 General Setup 18
    3.2 Verification of eBPF Non-Concurrency 20
        3.2.1 Description and Purpose 20
        3.2.2 Method 20
        3.2.3 Results 22
        3.2.4 Analysis 24
    3.3 Effect of eBPF Demultiplexing on Latency and Throughput 26
        3.3.1 Description and Purpose 26
        3.3.2 Method 26
        3.3.3 Results 27
        3.3.4 Analysis 30
    3.4 Effect of Time on eBPF 'Bouncing' Latency and Throughput 32
        3.4.1 Description and Purpose 32
        3.4.2 Method 32
        3.4.3 Results 33
        3.4.4 Analysis 35
    3.5 Effect of Hit Rate on eBPF Cache Latency and Throughput 37
        3.5.1 Description and Purpose 37
        3.5.2 Method 37

3.5.3 Results 39
3.5.4 Analysis 41

4. CONCLUSION 43

4.1 Future Work 45

APPENDIX 46

BIBLIOGRAPHY 51

LIST OF FIGURES

Figure 1: Simplified OSI Network Layers Model (Kurose & Ross)..6

Figure 2: IPv4 UDP Packet Diagram (Slay) 9

Figure 3: Layers of the Linux Network Stack (Buse) 10

Figure 4: Kernel Module Interactions (Zakharov et al.) 12

Figure 5: Internal Mechanisms of eBPF (Gregg)..14

Figure 6: XDP and the Network Stack (Šabić) 16

Figure 7: VM Setup Commands 19

Figure 8: Baseline Socket Latency Comparison 22

Figure 9: eBPF Socket Latency Comparison 23

Figure 10: BPF vs Baseline Demultiplexing Latency Comparison 28

Figure 11: eBPF vs Baseline Return to Sender Latency 34

Figure 12: Comparison of Cache Methods on Latency 39

Figure 13: Comparison of Cache Methods on Throughput 41

Figure 14: Baseline Socket Throughput Comparison 46

Figure 15: BPF Socket Throughput Comparison...47

Figure 16: BPF vs Baseline Demultiplexing Throughput Comparison 48

Figure 17: BPF vs Baseline Return to Sender Throughput..49

LIST OF TABLES

Table 1: Latency Results of Experiment 4 49

Table 2: Throughput Results of Experiment 4 50

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my thesis advisor and mentor Anand Sivasubramaniam for his flexibility and vision with this thesis. I must also thank Adithya Kumar for his vital help in getting the research off the ground as well as his great aid throughout. I thank Amey Deotale for aiding in talking through difficult roadblocks and aiding in a faster virtual machine setup methodology. I also thank all those who gave great support during this endeavor, friends, and family alike.

This thesis is dedicated to my late grandfather Merlin R. Kister, the first of many Penn State graduates in my family who encouraged and inspired my love for problem solving and learning from a young age.

1. INTRODUCTION

An individual living during the information age that began in the mid to late twentieth century could never have anticipated the speed and volume of information that flows across the internet today. Studies have shown that internet users on e-commerce websites demand near instantaneous results or the companies risk losing out on potential business (Monaghan). The video game industry, now built on online gaming has similarly exploded with both popularity and need low latency servers to keep players happy. As optimizations for server response time have increased during this era of cloud hosting and computing, so too have efforts to seek new areas where latency can be shaved off. One such area that has not seen a great deal of exploration is kernel shortcutting. Kernel shortcutting within the realm of latency reduction is the process of skipping numerous steps within the existing kernel network stack, instead having a user-created function decide what to do with the incoming information directly. It is important to note that Linux is by far the dominant kernel for the top one million servers globally (96.3%), so even a small shortcut that can be found for the Linux kernel can create great impact in the industry (Galov). This thesis will explore one such method of shortcutting the Linux kernel through the use of the Extended Berkeley Packet Filter (eBPF or shortened to just BPF) alongside the eXpress Data Path (XDP) toolset.

1.1 History of Packet Filters

The first packet filters came out of the late 1980's and were simple firewalls that monitored network traffic and accepted or rejected packets based on an access control list. These basic packet filters came from the Digital Equipment Corporation whose paper first released in 1987. Based on a simple matching system, these firewalls were very rudimentary in nature and would not allow the level of control that is seen within modern packet filters (Ingham & Forrest).

Between the years of 1989 and 1990, engineers at AT&T developed the first stateful packet filter; one that could maintain connection knowledge. This allowed the firewalls (coined circuit-level gateways by the AT&T team of the time) to track and filter on more specific criteria within a connection. For example, these filters could observe TCP handshakes to ensure that a connection is valid and not a result of a SYN flood attack (Ingham & Forrest). Even still, these circuit-level gateways did not offer the customizability that some engineers desired in their packet filters.

Though still lacking in total user customizability, it was not until 1992 that the first major step towards user-written packet filters would be made with the release of the original (known now as the classic) Berkeley Packet Filter (cBPF). The classic Berkeley filter was able to attach switch statements onto a socket in the Linux kernel based on user-written machine level instructions. Being given access to a switch statement, two jump instructions for true or false, and a 32-bit general register space, users could, for the first time ever, write their own customized filter function. This filter was also the first of its kind to reside within the kernel and can take advantage of viewing application-layer level packets. Despite this customizability, it

was still heavily limiting for advanced users to take advantage of and still acted primarily as a basic firewall as a result (Schulist et al.; “BPF and XDP...”). However, it was with the creation of cBPF that a key divergence in the realm of packet filtering occurred: one path led to “pure” firewalls, and the other path led to the more customizable packet filters that this thesis attempts to explore.

It would not be until 2014 that the BPF toolset would expand its capabilities to allow users to shortcut the network stack while running user-provided code to potentially reduce latency. This extension of cBPF is aptly named the extended Berkeley Packet Filter (eBPF or just BPF) and it will be thoroughly discussed in the background section of this thesis. Of course, with BPF originating from University of California Berkeley, a competing research institution of the Massachusetts Institute of Technology (MIT) also has been continuously developing its own method of extending a kernel.

MIT’s Exokernel operating systems (ExOS) eliminates the waste of kernel abstractions and allows users and user applications to directly implement these abstractions for the sake of speed and efficiency. The ExOS is a UNIX-like kernel that focuses purely on self-protection and validation. By only needing to keep the kernel stable, it allows applications and libraries the ability to reduce overhead and directly access low level features. The Cheetah project within ExOS specifically had seen server throughput improvements of up to eightfold on the smallest of request sizes. Despite the promising speedups shown by the Exokernel project, it struggles with increased levels of complexity, especially with regard to resource management (MIT Laboratory; Aarabhi). Forcing an application to manage these explicitly can save on overhead, but certainly loses out on fast development turnarounds that the industry currently demands.

1.2 Purpose

This thesis takes aim at exploring the ability of eBPF to shortcut the Linux kernel and produce latency and throughput benefits to common server scenarios. First, it attempts to verify the non-concurrency of eBPF code to confirm that it cannot multi-process packets entering a machine. Then, it will determine any possible benefits from using multithreading to have one eBPF program act as four concurrent userspace server applications. From there, it establishes a baseline level of latency and throughput differences between a traditional userspace server and an eBPF kernel-level server intercept through simple packet bouncing. Finally, it takes aim at determining the potential advantages in latency and throughput when eBPF is used as a cache when compared with an equivalent userspace cache.

By first verifying the non-concurrency of eBPF, it removes the possibility of race conditions existing when the eBPF program communicates with its loader daemon. Looking into the potential benefits of demultiplexing requests along with spawning threads as a response could reduce the number of sockets in use on a server and further speedup server applications which call APIs to generate responses. The simple client-server packet bouncing experiment server latency reduction will be directly applicable to a simple server which receives and responds to consistent requests with consistent responses. Finally, the creation of a cache is a proof of concept that the stateful eBPF is able to hold data, demultiplex a request, and respond accordingly while still saving time over the traditional userspace application.

2. BACKGROUND

The following background section's purpose is to either catch up an experienced reader with the pertinent information related to this thesis specifically, or to enable a newer reader to have the necessary information to follow the research and data retrieved to best understand and interpret its results and conclusions. An experienced reader may find it more useful to skim this section until a new concept catches their eye, but a newer reader may find themselves needing to look deeper into some areas.

2.1 Packets

To begin this background section, it is imperative that the reader understand the most basic and fundamental piece of networking, that being the packet. A packet is simply a bite sized piece of information that is sent between two computers (machines). A packet therefore has a source, the machine which sent the data, and its destination, the machine to receive the data. A packet is made up of a few different sections of information, but most generally and importantly are those of the payload, its raw data that is the reason for the packet being created and sent, and headers, which contain important metadata about the source and destination of the packet. This thesis focuses on the transmission of IPv4, user datagram protocol (UDP) packets, which are types of internet protocols which will be further discussed in the following sections.

2.2 The Network Stack

Before jumping into individual protocols, it is important to understand the infrastructure that is enabling the transportation of packets that this thesis is based upon. The mechanisms that build the infrastructure known as the internet are what make up the network stack. The network stack is comprised of layers, of which there are five of importance: the application layer, the transport layer, the network layer, the link layer, and the physical layer. Though all important, the most imperative to know for the purposes of this thesis are the application, transport, and network layers.

Layers	Description	Example Protocols
Application	Supporting Network Applications	HTTP, SMTP
Transport	Process-Process Data Transfer	UDP, TCP
Network	Routing of Datagrams from Source to Destination	IPv4, IPv6
Link	Data Transfer Between Neighboring Nodes	Ethernet, WiFi
Physical	Physical Signals	N/A

Figure 1: Simplified OSI Network Layers Model (Kurose & Ross)

The application layer is the top-level layer and final destination for our packets. It is also thought of as the user layer. This layer manages sending the packet data that has arrived on a

machine to the user applications that requested it and it enables a user to interact with the network through protocols such as those of HTTP requests. These protocols are often invoked when loading a webpage online and open a socket to facilitate this communication (Kurose & Ross). The protocols in this layer are higher level and have many details abstracted down the network stack.

The transport layer is one such abstraction of the application layer and its primary purpose is to facilitate communication between two machines. Most importantly, it enables this communication through two primary protocols: transmission control protocol (TCP) and user datagram protocol (UDP). TCP is a reliable protocol, meaning that it ensures machines will eventually end up receiving all the packets meant to be delivered (as long as there is a route available through the internet) and lost packets will be sent again until received. The downside of TCP is its slower initial connection and setup phase, which is not found in that of UDP. While UDP lacks the reliability of TCP, it requires no setup phase and enables packets to be sent quickly and freely. Later in this background section, this protocol will be further dissected. The main additions of these protocols and this network layer is the inclusion of ports in a packet header. Ports are similar to that of a real-life oceanic port in that it is a place for packets to arrive and be checked into a process running on a machine. Any running piece of computer software can be assigned to a port and be able to receive the data of packets that arrive to it (Kurose & Ross). Of course, TCP and UDP both rely on the abstractions of the further underlying layers of the network stack in order for the packet to arrive at such a port, the next level of this abstraction being the network layer.

The network layer is fundamental in routing packets from one machine to another through a network. Importantly to this thesis, every machine connected to a network holds a

unique address, called an internet protocol (IP) address. There are two versions of IP, IPv4 and IPv6, this thesis will only work with IPv4 packets and the only important difference for the sake of this thesis is the size and format of the shorter IPv4 source and destination header bits (see Figure 2). IP addresses are appended to the header section of a packet when it enters the network layer. These addresses will be important for this thesis in the calculation of a packet checksum as well as verification of the sender of an incoming packet (Kurose & Ross). For the purposes of this thesis, this is all that must be known about the network stack itself.

2.3 User Datagram Protocol

This thesis exclusively uses UDP for its trials and tests for its speed and ease of use. This section of background aims to take a deeper look into a UDP packet header (as a full IPv4 packet) and some of the key components that will be broken down by the software explored in this thesis. As UDP relies on the network stack layers beneath it, it is important to note that this is not simply a UDP packet, rather it is a UDP IPv4 packet (see Figure 2).

IP Bit Offset	0-3	4-7	8-13	14-15	16-18	19-31
0	Version	Header Length	DSCP	ECN	Total Length	
32	Identification				Flags	Fragment Offset
64	Time to Live		Protocol		Header Checksum	
96	Source Address					
128	Destination Address					
UDP Bit Offset	0-15				16-31	
160	Source Port Number				Destination Port Number	
192	Length				Checksum	
224	Payload					

Figure 2: IPv4 UDP Packet Diagram (Slay)

The important pieces of this packet header are the IP source and destination, the port source and destination, and the checksum sections. All of these sections will require some action on the part of the programmer in this thesis. While the sources and destinations are self-explanatory, being where the packet was sent from (the source) and where the packet is attempting to finally reach (the destination), the checksum in this section may not be so. The checksum denotes a method for machines that receive UDP packets to ensure the full packet arrived from transit as it was intended to be sent, meaning it was likely not corrupted or modified along its journey from source to destination. It is a sixteen bit, one's complement, hash-like representation of the IPv4 packet's headers and data section calculated by taking the one's complement sum of the header information from the IPv4 header, followed by the UDP header, and finally the data which gets padded with zeros if its length does not fall on a multiple of sixteen. This checksum must be calculated correctly if a machine is to receive this packet and

treat it as a valid incoming piece of information, otherwise the packet will be thrown out ("User Datagram Protocol...").

2.4 The Linux Network Stack

Packets are accepted into a machine through its so-called network stack. The Linux network stack (see Figure 3) has seven layers with the top-most layer being the application layer, where user applications can request information from the network, and the bottom-most layer being the physical network card that accepts signals from Wi-Fi or a physical cable.

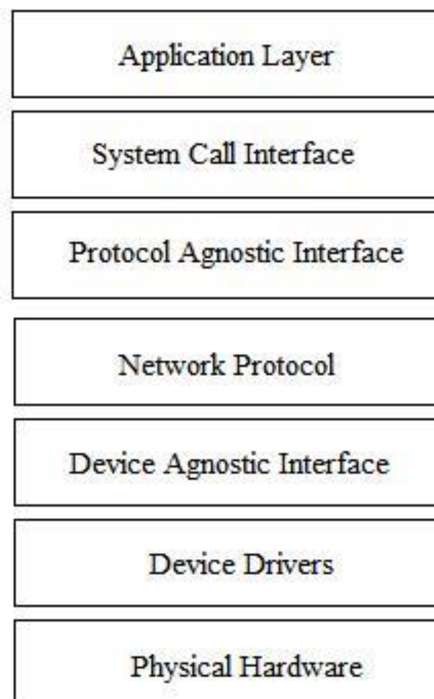

Figure 3: Layers of the Linux Network Stack (Buse)

Between these layers lie important processing to get the packet directed to the proper socket with the proper formatting to be interpreted for the user application. The system call interface layer is responsible for translating a user network request into a system call that enables the protocol agnostic interface layer to create a socket to send the request out to the network. Before being sent, the networking protocol layer sets up the outgoing or incoming packet's headers. Finally, the packet data is translated in the device drivers layer and is ready to be sent to the network card, the physical hardware layer. From there, the computer-external protocols take action, and the packet is sent. This process is, of course, reversed for incoming packets (Buse).

2.5 The Kernel and Kernel Modules

The kernel is simply a fancy name for the operating system of a computer. It represents the software that is always running to manage and maintain a computer. The kernel is what manages the network stack and controls and processes the packets within it. Inside of the kernel is space for advanced users to create their own programs that can be embedded to modify or create new behaviors on a level that is not as abstraction-heavy as userspace. These programs are also known as kernel modules, and they are what prevents an OS from being completely monolithic (Salzman Jay et al.). In other words, the modules allow extensions of functionality to be added on without modifying the original compiled OS binary or being forced to recompile or reboot for each new addition made.

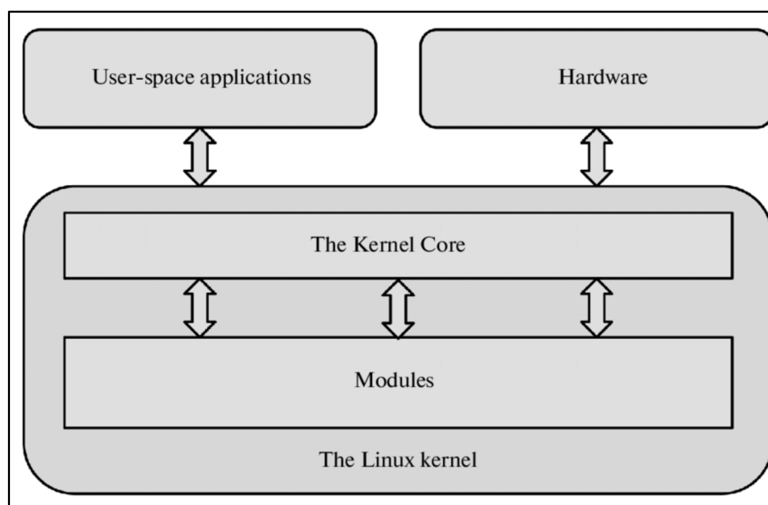

Figure 4: Kernel Module Interactions (Zakharov et al.)

Programming within a kernel module tends to be quite limited. Firstly, there are few languages that can be compiled compatibly for the Linux kernel, the fastest and most widely used being the C programming language (which is used in this thesis). Despite C's abundance of code libraries, almost none of these functions are available for programming in the kernel. In fact, only a select handful of basic libc functions were transferred into kernel module functions available for use. This problem is due to the fact that unlike a userspace program, kernel modules cannot have their symbols linked from a library during compilation, instead fully relying on the kernel and its system calls to resolve symbols (Salzman Jay et al.). It should be noted that in the case of eBPF, it may appear as though a programmer has access to extensive function calls through the BCC library, but it is important to understand that these functions are all implemented from the base functions provided with a standard kernel module within the eBPF source code.

2.6 The Inner Workings of eBPF

The history section of the introduction began to explain what packet filters and cBPF were; this section aims to expand upon that information and discuss eBPF in further detail. As mentioned, eBPF is an extension upon cBPF with many important differences and upgrades. Firstly, eBPF has access to ten, 64-bit registers, instead of only two 32-bit registers, greatly increasing efficiency within modern architecture. Additionally, it increased the overall stack size to 512 bytes, which allows for programs to have more wiggle room in terms of complexity (Fleming; "What is eBPF?..."). In terms of complexity, a recent update to eBPF increased the complexity limit, which used to be set to a mere 4096 instructions. Currently, eBPF programs allow for complexity up to one million instructions along with function chaining (Monnet). Furthermore, eBPF adds the ability to access and manipulate map data structures, and, if that was not enough, bounded loops were also introduced. Though perhaps the most important change for the sake of this thesis was the inclusion of dynamic interactions with a userspace daemon – specifically, the sharing of common data structures (Fleming; "What is eBPF?..."). The enabling of this communication between a userspace program, which has access to all the library calls a programmer could desire and the eBPF kernel module, is the real force that drove this thesis into being; it allows for complex, yet quick programs for servers. Of course, an eBPF program would never come to fruition without compilation.

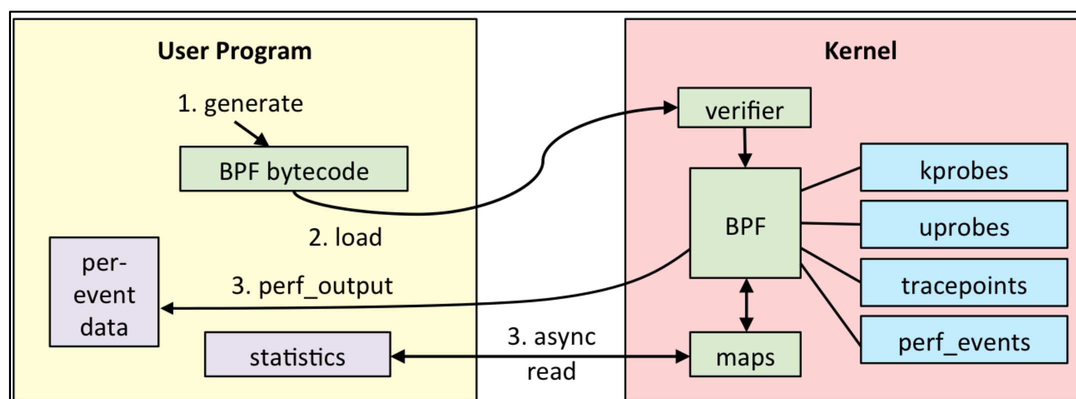

Figure 5: Internal Mechanisms of eBPF (Gregg)

The eBPF compiler is a just-in-time (JIT) compiler that functions in two steps. First, a verifier will run on the program, which will be discussed in the next paragraph. Then, once verified, the compiler moves onto the code generation stage. This stage translates the C program into architecture-based machine code (x86, Arch64, etc.). Translating the code was described as "one-to-one" from the C code by the architects of the JIT compiler and is straightforward in consideration. After translating the code, it is left with some symbols that must be taken care of in the linking phase. Though it does not allow library calls, the compiler must work with calls to map objects as well as calls to other eBPF programs which must be resolved (Wang). However, certainly the more interesting part of compilation is that of the verifier.

The eBPF verifier is a fascinating and tricky piece of software designed to keep the kernel safe. To do this it follows a series of checks which can become complex quickly depending on the complexity of the program it verifies. First, it checks the control flow. Importantly, it looks for jumps that are out of range of all the eBPF programs' designated memory regions. It also checks for calls to functions that are undeclared or undefined. Second, the verifier runs a static check on the program. In this phase, it will check for references to out of

bound or unaligned stack memory addresses. Additionally, it will verify that there is no potential for divide by zero errors and that there are no illegal shift amounts for registers (those which may cause overflows). Third, the verifier runs status checks based on value ranges and does so by dynamically observing and tracking the flow of data during execution. Looking at the bouncing packets experiment eBPF source code on the referenced GitHub (see the Experiments section), one can see many if-statements checking the bounds of a packet header struct to satisfy this phase of the verifier. This phase ensures, amongst others, that pointer arithmetic stays within the bounds of the original allocation for that pointer. In addition to checks on ranges of values, it also does similar checks for register values. The last part of this phase checks for potential of stack corruption through overflows or mismanagements of pointers. Finally, the verifiers last round of checks is made to ensure that the complexity is within predefined limits and there exist no infinite loops nor unreachable instructions. In order to do this, it walks the code and stores the execution states and register status as it runs through possible executions. It forms a graph during this phase that should be a directed acyclic graph in order to ensure no infinite loops are possible. Furthermore, for sake of efficiency, graph pruning (when a branch of the graph is no longer considered by the verifier) takes place when execution and register states match that of a prior graph element. This process is, as one can imagine, slower and more intensive the more complex a user’s program is. Once finished, the compiled machine code is injected into the Linux kernel attached to the specified device and will run whenever the associated action is taken (Wang).

2.7 eXpress Data Path and eBPF

Within eBPF came a potential for fast packet handling, which is exactly what the engineers behind the eXpress Data Path (XDP) framework for eBPF understood. By eliminating all context switching, interrupts, and the entirety of the Linux network stack following the physical hardware layer, XDP enables eBPF to immediately intercept packets from the network card.

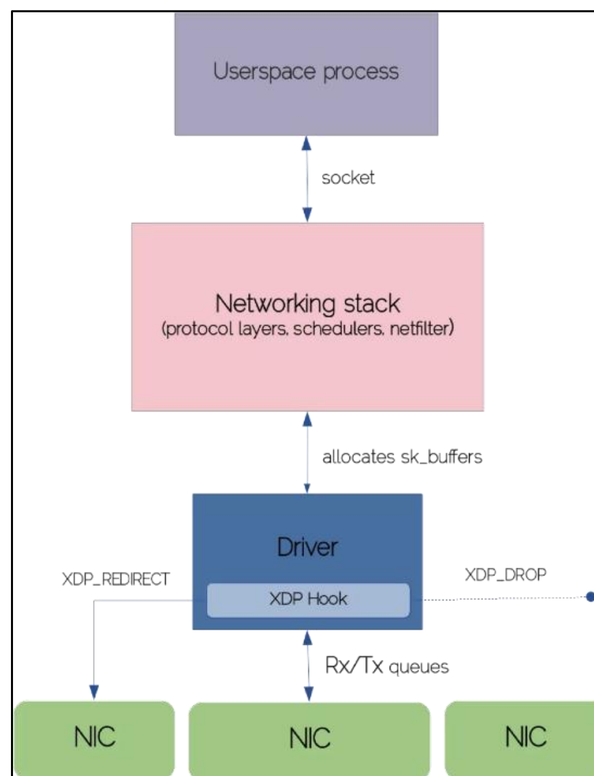

Figure 6: XDP and the Network Stack (Šabić)

In fact, XDP enables eBPF to run at such a low level, it requires the programmer of the eBPF program to create what is, in essence, its own device driver to manipulate and work with the incoming packets. With this addition of work does come an increase for potential speed,

which is precisely what this thesis is attempting to find within the toolset. Some of the major operations that XDP enables within eBPF is the dropping of packets (XDP_DROP), the passing of packets back into the traditional network stack (XDP_PASS), the redirecting of packets to another network card on the network (XDP_REDIRECT), and the bouncing of a packet back to the network card from whence it came (XDP_TX). Of course, the eBPF program is able to modify packets and perform other operations before these different XDP operations are called (Šabić; "EBPF XDP...").

2.8 BPF Compiler Collection Toolset

The BPF Compiler Collection toolset (BCC) was created in an attempt to make development more productive when programming with eBPF. It includes a plethora of toolsets and built-in functions and wrappers that allow an engineer to have a nice layer of abstraction from some of the messier parts of writing a kernel module, especially the lack of true library support. BCC also provides a userspace daemon that is able to compile and execute eBPF code and interact dynamically with it as it runs. This toolset gives eBPF the power to communicate through shared resources and, in some ways, act as a userspace level program while maintaining the high speeds found in an XDP, eBPF program ("BPF Compiler..."). Overall, this toolset was nice to use, though still limiting and lacking in some annoying ways (such as the lack of a UDP checksum calculation function). It should be noted that the toolset did enable some otherwise impossible tasks that can be found in experiment two.

3. EXPERIMENTS

The following subsections of this chapter will be broken down into subsections of a description and purpose, an experimental method, the results found, and finally analysis for the series of experiments conducted. For all source code used to implement these experiments, please navigate to the thesis GitHub page found at github.com/mzs6333/eBPF.

3.1 General Setup

The following set of experiments were conducted with the same initial setup. Two Virtualbox virtual machines (VMs) were used on a Windows 10 host machine with a total of four CPU cores and sixteen GB of RAM. Each VM was proportioned with four GB RAM and one CPU core each. Both VMs were running on Ubuntu 20.04.3 LTS with kernel version 5.13.0-30-generic. It is strongly recommended to install the Virtualbox Guest Additions disk onto the VMs for ease of programming between host machine and guest VMs. See Figure 7 below for the commands to be run on a fresh VM in order to run the experiments performed in this thesis. These VMs will be run on a single host to simulate traditional networking within a more controlled environment, and it is important to note that despite being virtual machines, the Linux network stack still operates as it would on a normal computer. In other words, a virtual machine shares its physical resources with the host, but it maintains kernel independence from the host. It is also

recommended for further experimentation that Wireshark be installed on the VMs in order to verify the proper functionality of eBPF programs (not shown in the commands below).

```
apt-get update && apt-get install -y \
    python3-dev -y \
    netcat -y \
    cmake -y \
    build-essential -y \
    flex -y \
    bison -y \
    qperf -y \
    git -y

export PATH="/server/clang+llvm-13.0.0-x86_64-linux-gnu-ubuntu-20.04/bin:$PATH"

apt install -y libedit-dev libllvm7 llvm-7-dev  \
    libclang-7-dev zlib1g-dev libelf-dev        \
    libfl-dev python3-distutils python

git clone https://github.com/iovisor/bcc.git
mkdir bcc/build
cd bcc/build
cmake ..
make
make install
cmake -DPYTHON_CMD=python3 ..
pushd src/python/
make
make install
popd
```

Figure 7: VM Setup Commands

3.2 Verification of eBPF Non-Concurrency

3.2.1 Description and Purpose

This initial, brief experiment was conducted to verify that eBPF packet interceptions were not run concurrently, but rather one call following another. From the design presented by the eBPF documentation and the understanding of packet queuing within the Linux network stack, the expected behavior is for eBPF to run once per packet and one packet at a time, exiting the function call before the next packet can begin its processing. Therefore, this study is expecting results to show varied latency between sockets for the baseline testing and a consistent latency between sockets for the eBPF testing. This expectation comes from the fact that machines are able to context switch and therefore dual process while performing certain operations. In the case of the baseline testing, while awaiting certain operations, the machine could context switch to another process and therefore introduce a sense of concurrency while performing the actions found within the sockets. In the case of eBPF, if it is not concurrently run, then it must run through the full action of the "socket" before sending its reply without any context switching possible, meaning that every socket should see the same average RTT latency.

3.2.2 Method

This study starts with the general setup described at the beginning of this section. In order to verify the non-concurrency of eBPF packet processing, one must first create a scenario where

multiple packets destined for multiple sockets are “concurrently” entering the network card. To achieve this feat, one VM acted as multiple clients to achieve rough concurrency of packet sending and one VM acted as a server that would receive packets and perform varied time-consuming operations on packet data before replying.

For the baseline tests, the client VM created four different processes that would send a UDP packet and await a reply from the destination before sending the next. With these four processes running in parallel, the host should observe what appears to be somewhat concurrent packet reception. The host similarly runs four processes that each have a socket that is awaiting a packet and will perform an action for some length of time that is different for each process, but consistent within each process. For example, socket one may run a loop for one thousand iterations and socket two may run an identical loop for only 500 iterations. Each client process tracks the total packet interactions with the host and reports them at the end of each thirty second test interval. Following the completion of fifteen tests, averages for RTT latency and throughput were calculated and recorded.

For the eBPF tests, the client VM acted identically as it had during the baseline testing. However, instead of having four processes running on the server VM, it ran the eBPF program alongside its BCC python daemon. All operations performed by each process on the host now had a place within the eBPF code. The eBPF program used a switch statement on port number to determine which actions to take to mirror the original baseline host actions identically. For example, if the baseline host process for socket one ran a loop for 600 iterations upon receiving a packet, so too would the eBPF program if it processed a packet destined for socket one. Following the mimicking of the original host processes, the eBPF code uses XDP to send a modified packet response back to the sender. Again, the client processes track their total packet

interactions with the server during thirty second interval tests. Following the completion of fifteen tests, averages for RTT latency and throughput were calculated and recorded. Refer to the GitHub page referenced at the start of this section for source code implementation.

3.2.3 Results

The results of the baseline 30 second latency testing are presented in Figure 8 below and will be summarized further following the figure.

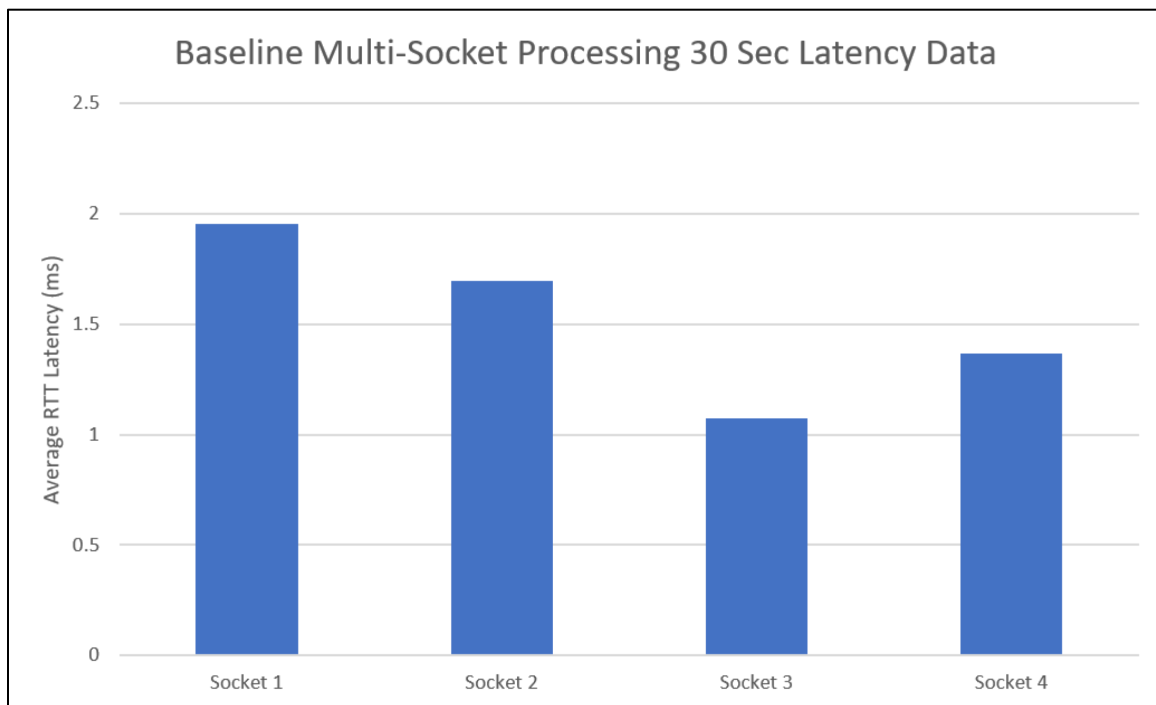

Figure 8: Baseline Socket Latency Comparison

The results for the baseline test are as follows:

- Socket 1 had an average RTT latency of 1.95606 ms with a standard deviation of 0.0468
- Socket 2 had an average RTT latency of 1.69781 ms with a standard deviation of 0.0297
- Socket 3 had an average RTT latency of 1.07569 ms with a standard deviation of 0.0548

- Socket 4 had an average RTT latency of 1.36592 ms with a standard deviation of 0.0232

There were no major outliers within the tests performed and the data produced lines up accordingly with the workload for each socket. It is very clear that the amount of work being done in each socket has a direct effect on the average RTT latency. One interesting figure is the higher standard deviation present with Socket 1's results, which will be examined further in the analysis section. For throughput test results see Figure 14 in the appendix – these results mirror that of latency inversely.

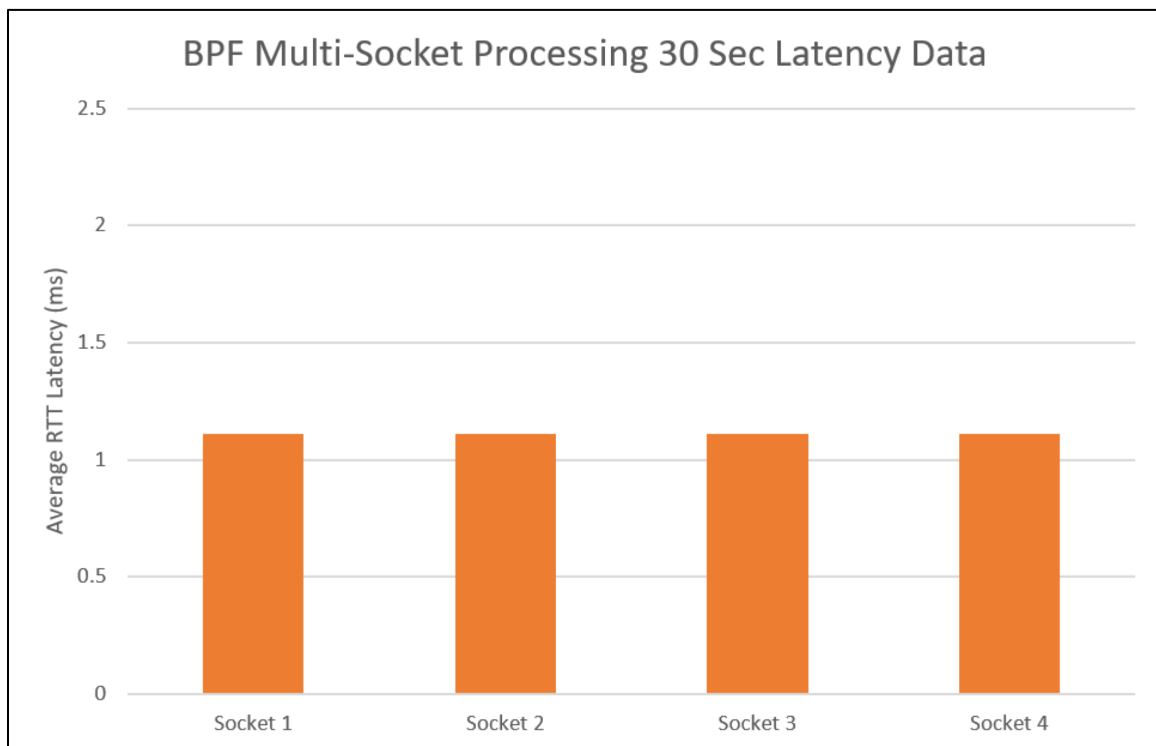

Figure 9: eBPF Socket Latency Comparison

The results for the eBPF test are as follows:

- Socket 1 had an average RTT latency of 1.11046 ms with a standard deviation of 0.0452
- Socket 2 had an average RTT latency of 1.10894 ms with a standard deviation of 0.0452
- Socket 3 had an average RTT latency of 1.08225 ms with a standard deviation of 0.0460

- Socket 4 had an average RTT latency of 1.10862 ms with a standard deviation of 0.0446

Similar to the baseline testing there were no major outliers in the data. It is very apparent that there is no major difference in latency for each socket when the same socket code is run embedded within eBPF. All data is incredibly consistent not only between sockets, but also between individual trials, seeing a near exact standard deviation for each socket. It is important to note that there was a mirroring of results from the first baseline trials in that, though slight, Socket 1 held the highest RTT latency, followed by 2 and 4, and ending with 3. Another similarity can be seen in that Socket 3 held the highest standard deviation across its trials. For throughput test results see Figure 15 in the appendix – these results mirror that of latency inversely.

3.2.4 Analysis

It was expected that the baseline tests would see a difference in RTT latency between its sockets due to concurrent processing of packets and their associated workloads, while there would not be a difference in RTT latency within the eBPF tests. This hypothesis has been confirmed by the results presented by the test. Through the confirmation of this hypothesis and the verification that eBPF is a non-concurrent process, it begs the question of whether or not it is possible to introduce some concurrency to eBPF. This question will be addressed in experiment two, though there are some other interesting talking points from this data.

Firstly, it is important to note that the lower the latency, the higher the standard deviation became. This topic was seen to be consistent over the course of these studies and will be further discussed in the conclusion section of this thesis. Secondly, there was a slight mirroring of

results found within the eBPF tests when compared with the baseline tests. This similarity can be explained as not an indication of concurrency, but rather as a direct result of experimental methodology. As the four client processes are running concurrently on the client VM, it means that the quicker they receive a response from the server, the quicker they can again send a packet in response. Even as eBPF is processing packets one at a time before sending a reply, if multiple replies happen fast enough, it is possible that the client machine will receive and process packets from the host out of order. In the case of the lightest weight workloads of the server, especially that of the low loop count workload for Socket 3, it likely means that the client had processed a packet associated with Socket 3 before another socket, despite the other socket actually receiving its reply from the server first. This difference would have to be very slight and would prefer sockets with quicker workloads, which is exactly what happened in this case. Therefore, this mirroring, especially with how slight it is, can easily be explained by race conditions occurring with the processing of replies from the server and is not a result of some concurrent behavior.

3.3 Effect of eBPF Demultiplexing on Latency and Throughput

3.3.1 Description and Purpose

This experiment built from the previous one and attempted to discover if it was possible and worthwhile to introduce pseudo-concurrency into an eBPF program. Furthermore, it was also aimed at discovering whether eBPF could, in the case of UDP communication, reduce, or even eliminate, the need for sockets actively listening on ports altogether. Instead of having lots of processes connected to sockets, it was theorized that an eBPF program could stand in for a nearly arbitrary number of sockets. It was hypothesized that an eBPF program that manipulates a set of data structures to act as userspace packet storage, could enable a daemon to spawn threads which each perform some operation and send a reply from userspace. Additionally, it was hypothesized that this minor shortcutting of the Linux network stack still may provide additional speedup when compared to a baseline test. An interesting side product of this experiment is determining the level of bottleneck the workload of each socket has on the overall latency. This side product will be shown by also measuring the non-concurrent baselines for each socket's workload.

3.3.2 Method

This study's method heavily mirrors that of the prior experiment. In fact, the baseline test setup is identical to that of the prior experiment with the only exception being the workload for each socket was increased and changed from loops to sleep commands to enable concurrency within

threads. Both baseline and eBPF tests were run for thirty second intervals as before. Refer to the GitHub page referenced at the start of this section for source code implementation.

For the baseline test setup, please refer to the prior experiment's method and reference the source code for the workload tweaks made. One addition to the prior baseline test is that each socket would also be measured individually (not concurrent with other processes running) to aid in determining the overhead of spawning threads in userspace and determine potential bottlenecks.

For the eBPF test setup, the prior experiment's eBPF program was built upon and modified heavily. Instead of performing the work then and there in the kernel upon receiving a packet, select packet information was now passed onto a corresponding "socket packet queue" map data structure. These data structures stored the information from the packet as well as its corresponding source port number in order to send the response to the proper socket when the userspace program is ready. In addition to changes to the eBPF program, large changes were made to the daemon file. While running, it would now loop and constantly check for new entries in each socket's queue. If there was a new entry, it would spawn a new thread which would perform the same work as the baseline test for the corresponding socket and send the proper reply from userspace. As before, averages for RTT latency and throughput were calculated and recorded.

3.3.3 Results

The results of the baseline 30 second latency testing are presented in Figure 10 below and will be summarized further following the figure.

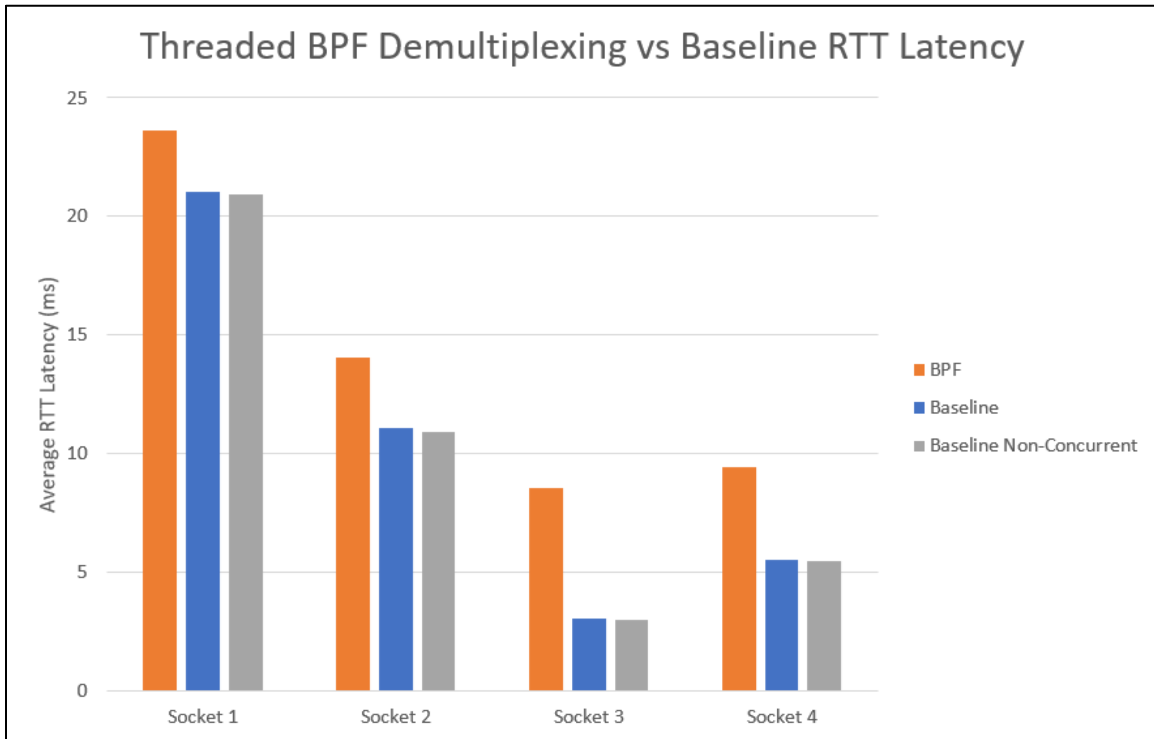

Figure 10: BPF vs Baseline Demultiplexing Latency Comparison

The results for the baseline (concurrent) RTT latency test are as follows:

- Socket 1 had an average RTT latency of 21.0104 ms with a standard deviation of 0.0422
- Socket 2 had an average RTT latency of 11.0745 ms with a standard deviation of 0.0406
- Socket 3 had an average RTT latency of 3.01217 ms with a standard deviation of 0.0275
- Socket 4 had an average RTT latency of 5.53901 ms with a standard deviation of 0.0269

There were no major outliers present in the data. There is again a clear connection between socket workload and average RTT latency. An interesting occurrence can be found when observing the difference in standard deviation between Sockets 3 and 4 and Sockets 1 and 2. For throughput test results see Figure 16 in the appendix – these results mirror that of latency inversely.

The results for the eBPF RTT latency test are as follows:

- Socket 1 had an average RTT latency of 23.5948 ms with a standard deviation of 0.0651
- Socket 2 had an average RTT latency of 14.0695 ms with a standard deviation of 0.0513
- Socket 3 had an average RTT latency of 8.51660 ms with a standard deviation of 0.0406
- Socket 4 had an average RTT latency of 9.44624 ms with a standard deviation of 0.0459

There were again no major outliers present in the data. It is clear from the data that there is a connection between the workload of a socket and average RTT latency within this test. For throughput test results see Figure 16 in the appendix – these results mirror that of latency inversely.

Comparing the two sets of concurrent data, it is very apparent that the eBPF latencies are higher than that of the corresponding baseline latencies across the board. The most extreme difference can be seen when comparing Socket 3's latency, where the results from the eBPF test's latency is 2.83 times greater. Furthermore, the eBPF test results saw higher standard deviation compared to the corresponding values in the baseline test.

The results for the baseline non-concurrent RTT latency test are as follows:

- Socket 1 had an average RTT latency of 20.9283 ms with a standard deviation of 0.0390
- Socket 2 had an average RTT latency of 10.9282 ms with a standard deviation of 0.0397
- Socket 3 had an average RTT latency of 2.95578 ms with a standard deviation of 0.0272
- Socket 4 had an average RTT latency of 5.47618 ms with a standard deviation of 0.0281

There were no major outliers present in the data. The average RTT latency for each socket collected non-concurrently is slightly lower than its concurrent counterpart. The standard deviations are very similar in nature between the two. Otherwise, this data very nearly mirrors that of the concurrent baseline test results.

3.3.4 Analysis

It was hypothesized that demultiplexing incoming packets using eBPF and concurrently sending responses through multithreading in userspace would yield performance benefits. This hypothesis was not supported by the experiment's results. The data presented by the experiment indicate that this methodology of introducing concurrent to eBPF has performance losses compared to the baseline tests. Furthermore, the results from the non-concurrent test seems to indicate that there could be some bottlenecking of the VM CPU occurring, which will be further discussed in the general conclusion. However, despite the worse performance, the eBPF test did successfully mimic the results of the concurrent baseline and only use one socket to send packets back instead of the four used in the baseline.

The hypothesis was not supported by the data likely due to the overhead involved with creating some concurrency. There are two major areas of slowdown that could be the culprit, one being the need to create threads, and the other being the BCC daemon itself. In the baseline tests, there is no need to spawn a new thread for each incoming packet. Through the creation of new threads, there is the introduction of context switching, and, especially in an already slow language like python, the slowdowns incurred by these context switches add up rapidly. Additionally, the BCC daemon itself in this experiment almost represents a modified form of the Linux network stack. To think that a python program is able to outpace kernel code written in C is a silly idea to begin with, and it is clear that the daemon is not able to keep up with the kernel. These two main slowdowns seem to explain the data well, but also would explain the smaller differences between eBPF and baseline latency when the latency was high to begin with due to socket workload as well as the large differences between the two when latency was lower. For example, when more packets enter the queues for Socket 3, which has a smaller workload, more

context switches will occur and will act as the bottleneck instead of the workload itself. Then, when considering Socket 1, there will be overall less packets entering the queues with a larger workload, meaning the bottleneck will still be in the workload section, making the difference in latency appear smaller.

Finally, there was a success in that the program did successfully create a concurrent system using eBPF interception and successfully turn a system of four concurrent socket programs into one multithreaded program. If the daemon itself could be written in C, or there is a way in the future where the data structures modified by the eBPF program can be referenced by a C program, it is possible that the overheads of multithreading in python could be minimized.

3.4 Effect of Time on eBPF 'Bouncing' Latency and Throughput

3.4.1 Description and Purpose

This study conducted attempts to determine three main ideas within eBPF. Firstly, it verifies that eBPF should have a faster response time than a traditional userspace program. Secondly, it will determine if there is any startup phase to using eBPF with XDP. Thirdly, it will determine if there are any changes in the benefits of using eBPF to bounce packets depending on the test runtime. It is hypothesized, due to the low-level access of the packets in the Linux network stack, that an eBPF program made to bounce packets will outperform the same program run in userspace. Furthermore, it is not anticipated that any startup phase is present within eBPF programs as there is no caching occurring that could create a slow start. Finally, it is expected that the benefits of eBPF will slowly add up over time, meaning that the longer a test runs for, the larger the difference between userspace and eBPF average latency will become.

3.4.2 Method

This study starts with the general setup described at the beginning of this section. In order to determine the benefits of using eBPF across time intervals, one must first create a server and client scenario in order to bounce packets back and forth to measure average RTT latency and throughput. This setup is similar in nature to the demultiplexing tests in that one VM will act as a client and the other will act as a server. The average RTT latency and packet throughput will be

calculated and recording following the tests at intervals of 15 seconds, 30 seconds, 45 seconds, and one minute for ten trials.

The client VM code will remain the same for both the baseline and eBPF tests and will simply open a socket and send and receive packets to and from the server for the duration of the test.

The server VM code will switch between a baseline version and an eBPF version. The baseline version is very similar to the client code in the reverse order. It simply accepts a packet and then creates a response packet with identical data and sends it right back before awaiting the next arrival. In the case of the eBPF program, it does the same exact thing from within its kernel module. All that must be done for the program to be successful is the swapping of header values before calling XDP_TX to send the packet back out the network card from whence it came. It should be noted that, in order for the packet to be valid, a new UDP checksum will need to be created manually, which is a large portion of the eBPF program. Refer to the GitHub page referenced at the start of this section for source code implementation.

3.4.3 Results

The results of the time dependent latency testing are presented in Figure 11 below and will be summarized further following the figure.

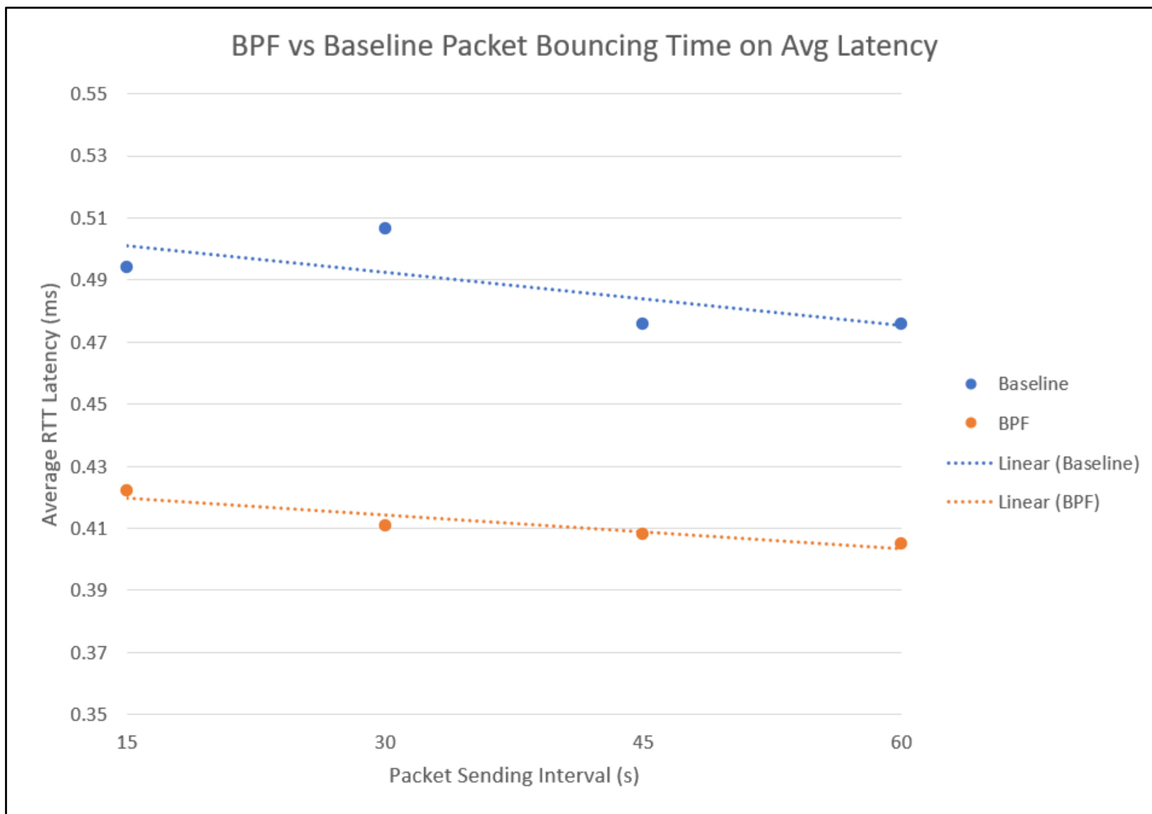

Figure 11: eBPF vs Baseline Return to Sender Latency

The results from the baseline packet bouncing RTT latency tests are as follows:

- The 15 sec. test saw an average latency of 0.49405 ms and a standard deviation of 0.0045
- The 30 sec. test saw an average latency of 0.50653 ms and a standard deviation of 0.0120
- The 45 sec. test saw an average latency of 0.47564 ms and a standard deviation of 0.0038
- The 60 sec. test saw an average latency of 0.47583 ms and a standard deviation of 0.0041

The results from the eBPF packet bouncing RTT latency tests are as follows:

- The 15 sec. test saw an average latency of 0.42226 ms and a standard deviation of 0.0026
- The 30 sec. test saw an average latency of 0.41093 ms and a standard deviation of 0.0011
- The 45 sec. test saw an average latency of 0.40822 ms and a standard deviation of 0.0034
- The 60 sec. test saw an average latency of 0.40505 ms and a standard deviation of 0.0017

There was an outlier present in this data, that being the results of the 30 second baseline test. It is clear from the data presented that something went awry in this test as its standard deviation was far higher than that of the other similar tests and its average latency does not follow the trend set by the rest of the baseline data and by the eBPF data.

Clearly shown by both the graph and the statistics reported above, the eBPF average latency was lower when compared to their respective counterparts. Another interesting comparison is that the standard deviation for the eBPF tests were also lower than their respective baseline tests. The data shown here appears to be negative and linear for both the baseline tests and the eBPF tests. The baseline test line of best fit had slope of -0.0006 and y-intercept of 0.5094 with an R^2 value of 0.5375 indicating moderate correlation. The eBPF test line of best fit had slope of -0.0004 and y-intercept of 0.4252 with R^2 value of 0.8765 indicating a strong correlation. The difference between the 15 second test and 60 second test for the baseline was 0.01822 ms while the difference between the 15 second test and the 60 second test for eBPF was 0.01721 ms. Additionally, the absolute value of the average difference between the baseline and the eBPF results was 0.07640 ms. For throughput test results see Figure 17 in the appendix – these results mirror that of latency inversely.

3.4.4 Analysis

It was hypothesized that running eBPF would have a beneficial effect on the average RTT latency of communication between a server and client. This hypothesis was supported by the data as a clear difference was found between the two result sets. Furthermore, it was also hypothesized that there would not be any startup time needed for eBPF to begin to outperform

the baseline tests (if it would outperform it at all). This hypothesis was also supported by the results as there were no major drops between the shorter length tests and the longer tests that could not be explained by the final hypothesis. This hypothesis was also supported by the linearity in the trendlines for both the baseline and eBPF datasets. The final hypothesis suggested that there would be an increase in difference between the baseline and eBPF testing. This hypothesis was not supported by the data found. Instead of an increase in differences between the latency of the baseline and the eBPF tests, the difference was found to be about consistent between test lengths. In order to verify this claim, longer tests of multiple minutes could be performed and graphed to determine if this relationship is constant, or if it would begin to vary as hypothesized.

There was an outlier present within the baseline test results whose potential cause will be further discussed in the general conclusion section dealing with CPU workloads. Overall, this study has verified the claims that eBPF will yield noticeable differences when compared with the baseline. An additional metric that was collected is the throughput, measured in packets sent per second. This metric is especially important for this experiment as the difference in number of packets sent can illustrate the potential benefits of using eBPF on a real server. Again, these results in throughput can be seen in Figure 17 in the appendix. This study's results pave the way for that of the next experiment, where we selectively bounce packets using a cache.

3.5 Effect of Hit Rate on eBPF Cache Latency and Throughput

3.5.1 Description and Purpose

This study attempts to determine if eBPF can be utilized effectively as a cache between a client and a server to cut down on latency more so than a traditional userspace cache. It is hypothesized that placing a cache inside of kernel space using eBPF will be more effective in reducing average RTT latency than a cache found in userspace performing the same operations. Of course, as the prior experiments determined, there is a reduction in latency when using eBPF compared to a userspace solution, though the question with using it in a cache form is when the extra work of setting up an eBPF cache would be a worthwhile investment. This study hopes to find the relationship between hit rate and latency benefits to aid in determining whether or not the development of such a cache is worth the upfront setup cost.

3.5.2 Method

This study starts with the general setup described at the beginning of this section. In order to test the hypothesis above, one must first create a scenario involving a client, a server, and an intermediate cache. The client setup builds from the generic client explained in prior experiments, however in this study it will be varying its packet data according to a dictionary of characters. The client will be made to choose between eight options of characters to ensure

consistent cache hit rates across trials. Apart from this data, the rest of the client file is the same as prior clients.

The baseline server setup will involve the creation of two paths. The first path represents a cache hit and will instantly send a reply to the client. The second path will act as a cache miss and simulate some work being done by the server, mimicking perhaps an API call or two before it generates its response to the client and sends its reply. Additionally, in the case of a cache miss when the cache is not full, the packet data and the response will be added to the cache in a first-come-first-served fashion. In the case of the eBPF server, this baseline server file is tweaked so that the first path does not exist, meaning that if there is a cache miss, eBPF will allow the packet to be normally forwarded to the userspace program which will simulate the same amount of work and send the reply from userspace.

The eBPF cache will exist in a layer between the base server and the client. It will check packets for a cache hit, which is when the packet data matches that of data in its map data structure. If it hits, it will pull its reply from the map and immediately send the reply as was done in experiment three. It if misses, it sends a flag to the daemon which updates its cache map before allowing the packet to enter userspace to generate the response from the base server. Additionally, a final dataset was generated that simulates the latency when the userspace cache always hits, but on top of this perfect userspace cache there is also an eBPF cache whose hit rate varies in order to visualize the latency difference made purely by the efficiency of eBPF.

All tests were performed for ten trials at thirty seconds with cache hit rates of 0%, 25%, 37.5%, 50%, 75% and 100%, which were enabled by manually changing the maximum cache sizes for both the baseline and eBPF caches. Refer to the GitHub page referenced at the start of this section for source code implementation.

3.5.3 Results

The results of the time dependent latency testing are presented in Figure 12 below and will be summarized further following the figure.

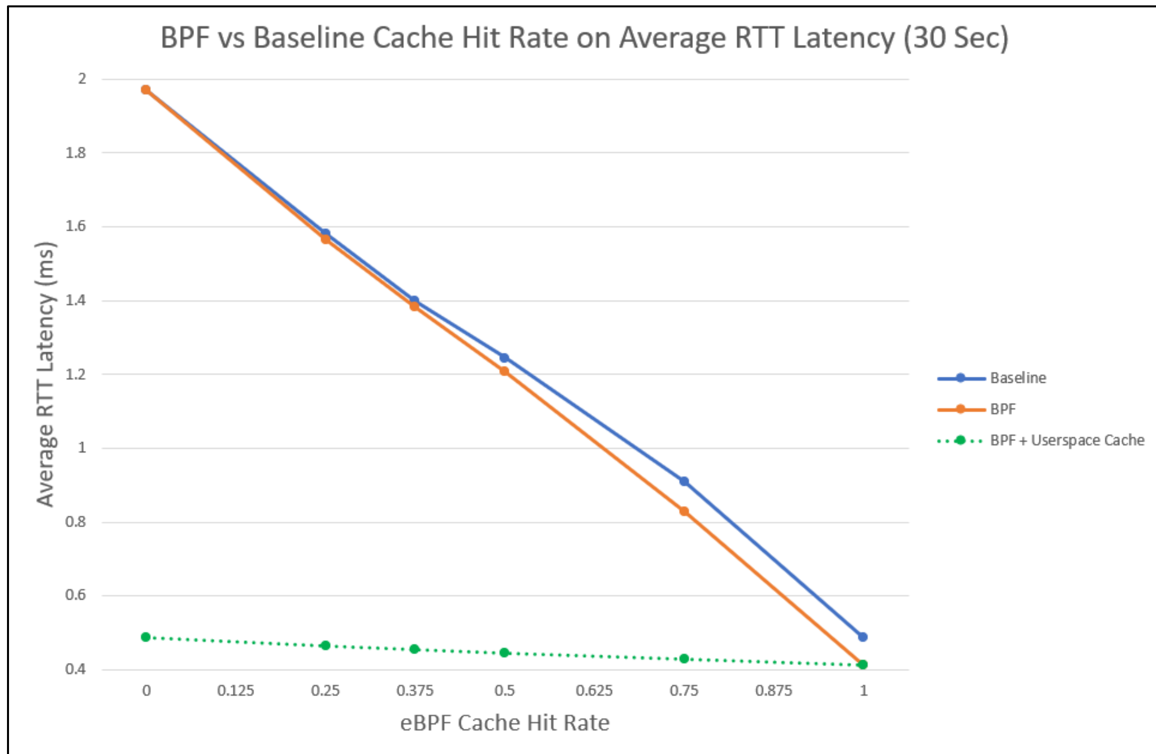

Figure 12: Comparison of Cache Methods on Latency

While there were no notable outliers present within the data collected, there is a variation from the overall trend present with the final data point for the baseline test. This variation will be further discussed in the analysis section. For sake of readability, a summary of exact values for the three sets of six data points will be omitted from this section and can be found in the appendix as Tables 1 and 2. The general trends for both the baseline and eBPF cache test results are negative as cache hit rate increases. The difference between the eBPF latency line and the

baseline latency line begin to increase noticeably at a cache hit rate of 50% and expands further at a 75% hit rate. The gap seems to stabilize at a 100% cache hit rate.

It is important to note the green data set, which, as described in the method, is the latency when a userspace cache is at 100% hit rate and the eBPF cache is varied according to the x-axis values. These values perfectly align themselves between the final two data points of the baseline and pure eBPF cache tests. This dataset is also negatively linear, albeit with a much shallower slope.

The data for the packet throughput results can be seen below, as they indicate an interesting trend that gives further insight into the comparison of these tests. Through the lens of throughput, the results appear almost exponential in nature, with the throughput really ramping up the closer it gets to a 100% hit rate. Unlike before, the green data set is now the only obviously linear set of data points and still is set between the final data points of the baseline and eBPF cache results.

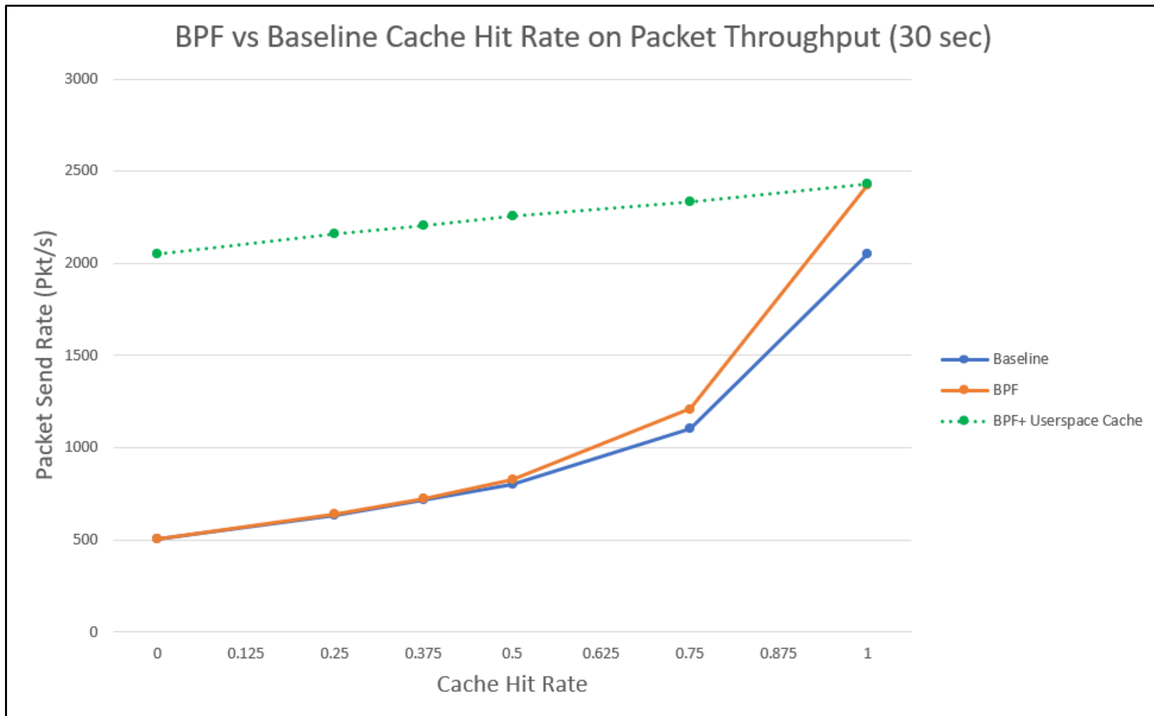

Figure 13: Comparison of Cache Methods on Throughput

3.5.4 Analysis

It was hypothesized that implementing a cache inside of the kernel using eBPF would yield performance gains in terms of latency and throughput. The data collected from this study is in partial agreement with this hypothesis. While there were noticeable improvements for the eBPF cache with larger cache hit rates, the reality is that not all caches can achieve such high hit rates due to infeasibility of storing all possible responses. Moreover, the lower cache hit rates did not see any real improvements in (though certainly did not detract from) latency. Thus, this data gives great insight into determining whether or not it would be beneficial to implement such a system based on the expected cache hit rate.

The trends in data collected are likely a direct result of the amount of work simulated by the server. The more work done by the server, the closer the results should get to a curve that appears exponential or at least quadratic in nature as there is considerably less time spent on generating responses when cache rates are high. On the other hand, if the amount of work is very small, it is likely that the curves generated will appear closer to linear as the cache hit saves very little time, which can be clearly seen within the green dataset where the server workload is effectively zero apart from overhead of running python in general. Again, there was some variation from the overall trend in the baseline data especially, which will be further explained in the general conclusion.

4. CONCLUSION

This thesis was conducted to explore the ability of eBPF to shave fractions of seconds off of client-server interactions through shortcutting the Linux network stack. It explored non-concurrency verification, concurrent demultiplexing, simple packet bouncing tests, and kernel caching through the creation of client-server experiments. In the non-concurrency verification study, it was determined that eBPF was not a concurrently run program, instead each packet triggered the program to run with one instance at a time until completion. Within the concurrent demultiplexing study, it was found that, while it is possible to introduce concurrency within eBPF, it comes with an overhead cost to make it slower than the baseline but could save on sockets if necessary. Through its packet bouncing and caching tests, it found that eBPF will provide beneficial decreases in RTT latency and increases in throughput over the traditional userspace implementations.

In nearly all studies conducted, there were some similar sources of error possible within the findings that must be discussed. As these tests were run on VMs within one host machine, they were subject to the possibility of interruptions and CPU maxing. Interruptions are anything that takes the focus of the CPU away from the test that is currently being run. These would appear as clear outliers in the data collection process and required the data collection to take place when machines were in a stable state. CPU maxing, on the other hand, come as a result of the server not having enough work to do when receiving a packet, making the bottleneck of data

collecting not the work performed on a packet, but rather on general packet processing overhead. If the bottleneck is not found in the server workload, it would likely be the case that on a more powerful (or less powerful) machine, the results could vary from what was found in this thesis. In the case of a more powerful machine, it is likely that it finds eBPF has even more beneficial gains when performing the packet bouncing and high cache hit rate experiments. On the other hand, a less powerful machine likely sees smaller differences in these studies. Finally, CPU maxing is likely the cause of the single outlier and the increase in variation of experiment results due to increased chances of small context switching by the kernel having large impact on the data collected.

Ultimately, it is the judgement of the author that until there is greater ease in producing eBPF programs, the use case for setting up an eBPF system on a server is niche at best. In theory, if setup properly in a data center environment, it would enable greater throughput and thus cut down on the number of machines required to process the same number of packets. However, the main drawback from this theory is that each machine running in a database is often running a number of virtual machines on a docker-like system. As I discovered in the setup phase of this thesis, any sort of virtualization that uses kernel virtualization (like Docker and WSL), meaning it uses the kernel of the host and not its own, is not natively compatible with eBPF, as it requires its own network stack and access to the network card to function ("Docker vs Virtual Machine..."). Furthermore, even if all machines were running VMs with their own kernels, the cost of writing personalized drivers for each server functionality would be incredibly high at the current stage of eBPF development. With that said, there is a great possibility for latency reduction within eBPF, and to ignore its potential completely would be a waste. There is plenty left to discover and consider for the future work with eBPF.

4.1 Future Work

With this thesis being exploratory in nature, there exist numerous areas of research to continue exploring the potential benefits and possibilities using eBPF. Firstly, all tests conducted worked with quite small payloads. Determining the effects of differing payload sizes is desirable as average packet sizes vary largely from application to application and discovering these effects present a unique challenge with an eBPF program that must stitch portions of packet data together if larger than the maximum single packet size. Additionally, the bandwidth gains of using eBPF should be explored and their experiments may mimic similar procedures as the ones explored here, simply using more concurrent clients to flood the packet queues. A large item that proved a challenge to this thesis is the work with TCP packets instead of only UDP. Being able to handle TCP packets and the corresponding handshaking protocol would be a beneficial future area of research. The creation of an eBPF load balancer through further demultiplexing is another area of exploration possible on top of this work. As described in the previous section, eBPF is not natively compatible with kernel virtualization tools like Docker. Some additional effort may be worth investigating if there is truly no compatibility possible to run eBPF with Docker, or if a workaround is possible. Finally, further investigation into whether or not packet data can be demultiplexed and sent to an open socket as if it naturally propagated the network stack should be done. If this is possible, it could quickly change the results of experiment two into something beneficial for latency.

APPENDIX

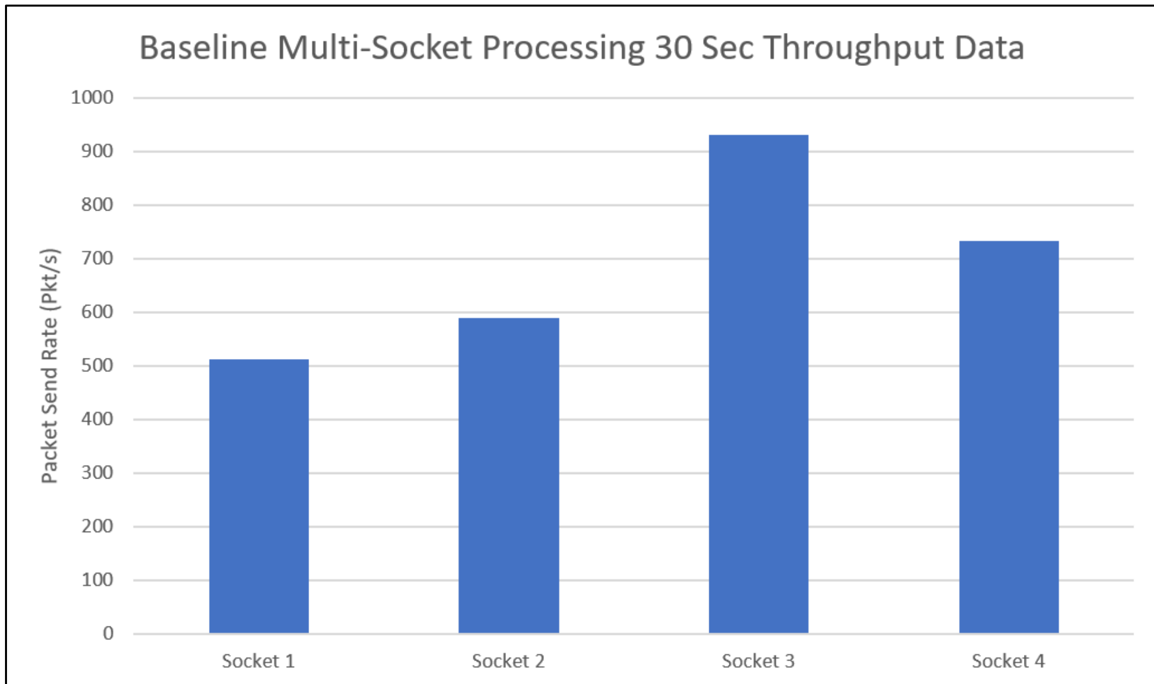

Figure 14: Baseline Socket Throughput Comparison

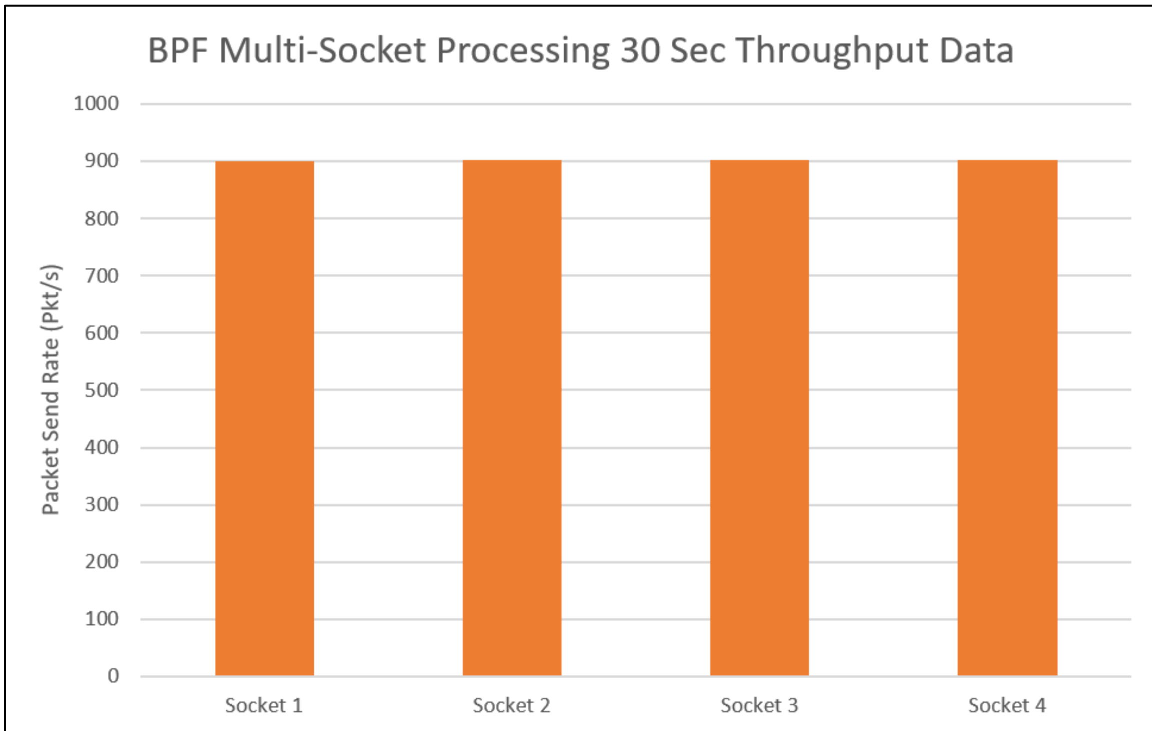

Figure 15: BPF Socket Throughput Comparison

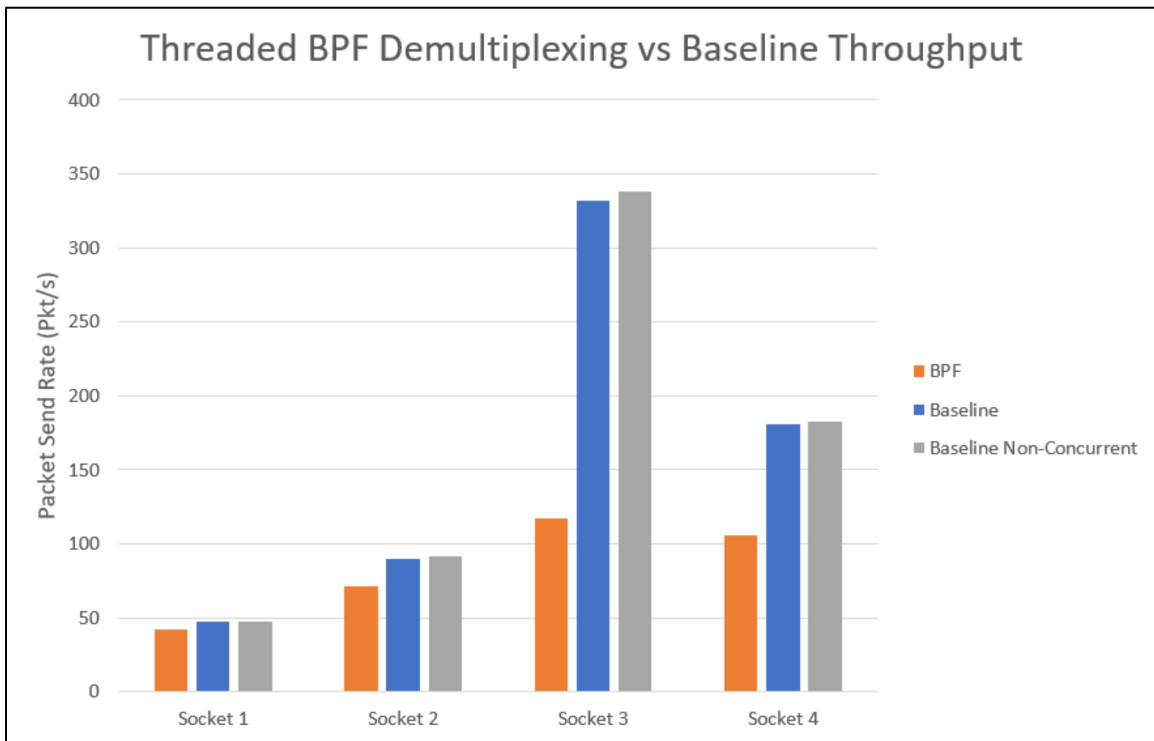

Figure 16: BPF vs Baseline Demultiplexing Throughput Comparison

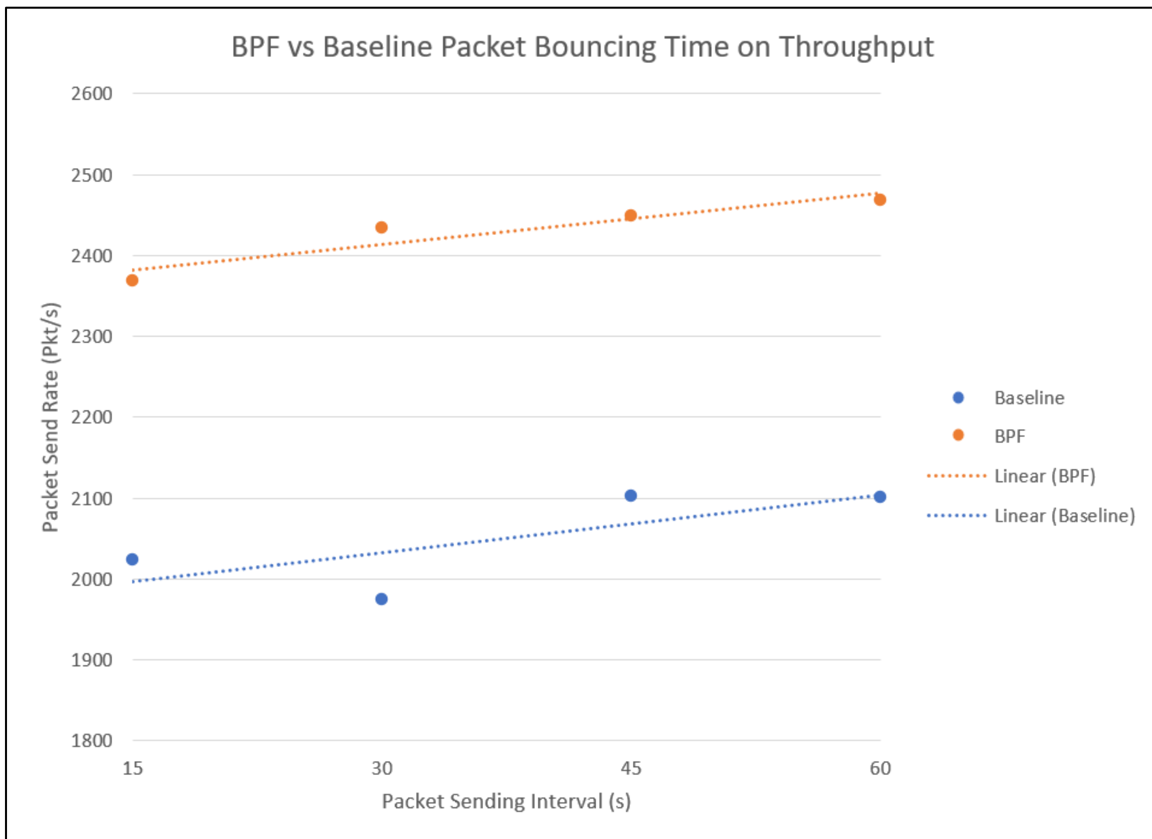

Figure 17: BPF vs Baseline Return to Sender Throughput

Latency Results (ms)	Hit Rates					
	0	0.25	0.375	0.5	0.75	1
Baseline	1.97054689	1.58002844	1.39801482	1.24469965	0.90729831	0.48735477
BPF	1.97054689	1.56546787	1.38412136	1.20716574	0.82679704	0.41293761
BPF + Baseline	0.48735477	0.46312591	0.45396081	0.44353241	0.42828754	0.41152376

Table 1: Latency Results of Experiment 4

Throughput Results (Pkt/s)	Hit Rates					
	0	0.25	0.375	0.5	0.75	1
Baseline	507.473333	632.9	715.3	803.406667	1102.17333	2051.89333
BPF	507.473333	638.786667	722.48	828.386667	1209.48667	2421.67333
BPF + Baseline	2051.89333	2159.24	2202.83333	2254.62667	2334.88	2429.99333

Table 2: Throughput Results of Experiment 4

BIBLIOGRAPHY

Aarabhi, Vithusha. "Exokernels : An Operating System Architecture for Application Level Resource Management." *Medium*, 2 Sept. 2017, https://medium.com/@vithushaaarabhi/exokernels-an-operating-system-architecture-for-application-level-resource-management-32d0daaeeab0.

BPF and XDP Reference Guide. Cilium, https://docs.cilium.io/en/latest/bpf/.

BPF Compiler Collection (BCC). Iovisor, https://github.com/iovisor/bcc#readme.

Buse, Jarret. "Linux Network Stack." *Linux.Org*, XenForo, 18 Sept. 2013, https://www.linux.org/threads/linux-network-stack.9065/.

"Docker vs. Virtual Machine: Where Are the Differences?" *DevOps Conference*, 23 Nov. 2017, https://devopscon.io/blog/docker/docker-vs-virtual-machine-where-are-the-differences/.

"eBPF XDP: The Basics and a Quick Tutorial." *Tigera*, https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/.

Fleming, Matt. "A Thorough Introduction to EBPF." *LWN.Net*, Eklektix, Inc., 2 Dec. 2017, https://lwn.net/Articles/740157/.

Galov, Nick. "111+ Mind-Boggling Linux Statistics and Facts for 2021 - Linux Rocks!" *Hosting Tribunal*, 16 Jan. 2021, https://hostingtribunal.com/blog/linux-statistics/.

Ghosh, Bamdeb. *Send and Receive UDP Packets via Python*. https://linuxhint.com/send_receive_udp_python/.

Gregg, Brendan. "Linux EBPF Tracing Tools." *BrendanGregg.Com*, 2021, https://www.brendangregg.com/ebpf.html.

Ingham, Kenneth, and Stephanie Forrest. *A History and Survey of Network Firewalls*. The University of New Mexico, Feb. 2012, https://www.cs.unm.edu/~treport/tr/02-12/firewall.pdf.

Keshavarz, Amir. "Absolute Beginner's Guide to BCC, XDP, and EBPF." *DEV Community*, 8 Aug. 2021, https://dev.to/satrobit/absolute-beginner-s-guide-to-bcc-xdp-and-ebpf-47oi.

Kurose, James F., and Keith W. Ross. *Computer Networking: A Top-down Approach*. Eighth edition, Pearson, 2021.

MIT Laboratory for Computer Science. *MIT Exokernel Operating System*. Parallel and Distributed Operating Systems Group, 5 Mar. 1998, https://pdos.csail.mit.edu/archive/exo/.

Monaghan, Maura. "Website Load Time Statistics: Why Speed Matters in 2022." *Website Builder Expert*, 29 July 2020, https://www.websitebuilderexpert.com/building-websites/website-load-time-statistics/.

Monnet, Quentin. "EBPF - EBPF Updates #4: In-Memory Loads Detection, Debugging QUIC, Local CI Runs, MTU Checks, but No Pancakes." *EBPF.Io*, 23 Feb. 2021, https://ebpf.io/blog/ebpf-updates-2021-02/.

National Conference on Recent Developments in Computing and its Applications, Alam, M. Afshar, et al., editors. *Proceedings of National Conference on Recent Developments in Computing and Its Applications, August 12-13, 2009*. I.K. International Pub. House, 2009.

Šabić, Nedim. "EBPF and XDP for Processing Packets at Bare-Metal Speed." *Sematext*, 3 June 2019, https://sematext.com/blog/ebpf-and-xdp-for-processing-packets-at-bare-metal-speed/.

Salzman, Peter Jay, et al. *The Linux Kernel Module Programming Guide ; Kernel 2.6*. Nachdr, SoHo Books, 2009.

Schulist, Jay, et al. *Linux Socket Filtering Aka Berkeley Packet Filter (BPF)*. https://www.kernel.org/doc/html/latest/networking/filter.html.

Slay, Jill. *IP and UDP Packet Headers*. ResearchGate, Jan. 2011, https://www.researchgate.net/figure/IP-and-UDP-packet-headers_fig1_221352750.

"User Datagram Protocol (UDP)." *Imperva Learning Center*, Imperva, https://www.imperva.com/learn/ddos/udp-user-datagram-protocol/.

Wang, Jiong. *Demystify EBPF JIT Compiler*. Netronome, 11 Sept. 2018, https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf.

"What Is EBPF? An Introduction and Deep Dive into the EBPF Technology." *EBPF.Io*, https://ebpf.io/what-is-ebpf/.

Zakharov, Ilja Sergeevich, et al. "Interaction of Linux Kernel Modules With Their Environment." *ResearchGate*, ResearchGate, May 2015, https://www.researchgate.net/figure/Interaction-of-Linux-kernel-modules-with-their-environment_fig1_299373758.

ACADEMIC VITA

Matthew S. Sickler

linkedin.com/in/matthewsickler

EDUCATION University Park, PA

The Pennsylvania State University | Schreyer Honors College Anticipated May 2022

Bachelor of Science in Computer Science

Minors: Cybersecurity, Mathematics

Awards: Paul Morrow Endowed Scholarship, President's Freshman Award

WORK EXPERIENCE

Capital One – McLean, VA Jun – Aug 2021

Software Engineering Intern

- Rewrote critical sections of an internal assessment tool which will save agile leads **500+ hours** per year across the company
- Mastered **Angular** and **Java Springboot** in two weeks to develop effectively in a collaborative full-stack environment
- Developed a new dynamic director-level dashboard with custom Angular **HTML5/CSS3** components displaying aggregated data and comparison graphs
- Improved site loading times by **40%** through improved back-end API calls and better caching across pages
- Upgraded sprint work items with improvements outside of original scope while maintaining sprint deadlines
- Leveraged internal training on top of existing work to further personal knowledge base and secure coding practices

WebstaurantStore.com – Lancaster, PA May – Aug 2020

Software Engineering Intern

- Rewrote a Release Manager internal application that tracks sprint work item completion and company-wide app releases in **C# .NET Core**
- Developed a **RESTful** API for the Release Manager application
- Designed and implemented the company's first cross-platform, **Javascript Azure DevOps** extension to send app release information automatically to Release Manager
- Shifted over a fourth of all company applications' **CI/CD** processes to Azure DevOps and optimized processes using **YAML** logic insertion and templates, reducing build agent loads
- Communicated effectively with developers, SREs, and database engineers to ensure new deployments methods are functional and properly understood by everyone on the project team

Penn State University – University Park, PA May 2019 – Jan 2021

Learning Assistant/Grader - Discrete Mathematics

- Led new TAs and instructors in both presenting and selecting course material, utilizing my expertise with course concepts
- Prepared and delivered lectures on logical deduction and rigorous proof

RELEVANT EXPERIENCE

LionCloud Device Driver – University Park, PA Mar – Apr 2020

- Developed a device driver in **C** that enabled communication between a device simulator and file system for a systems programming class

CodePSU – University Park, PA Mar 2021

- Placed 2nd at Penn State's largest competitive coding competition on a team of two

KEY SKILLS

Languages: Typescript, Java, Python, Javascript, SQL, HTML5, CSS3, YAML, C, C#

Frameworks: Angular, Springboot, Django, .NET Core

Tools: Azure DevOps, Jenkins, EC2, ECS, RDS, Git, Agile Development