Performance on Virtualization Environments with Virtual TAP

A dissertation submitted for the Degree of Master of Computer Science

S C Senanayake

University of Colombo School of Computing

2024

Declaration

Name of the student: Supun Chandrajith Senanayake
Registration number:2020/MCS/085
Name of the Degree Programme: Master of Computer Science
Project/Thesis title:Performance on Virtualization Environments with Virtual TAP

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.

2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.

3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.

4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.

5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.

6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

Signature of the Student	**Date (DD/MM/YYYY)**
	23/09/2024

Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

	Supervisor 1	**Supervisor 2**	**Supervisor 3**
Name	Dr. C. I. Keppitiyagama	Mr. Tharindu Wijethilake	
Signature			
Date	23/09/2024	2024-09-23	

ACKNOWLEDGEMENTS

I extend my deepest gratitude to my supervisor, Dr. CI Keppitiyagama, for his invaluable guidance, support, and expertise throughout the course of this project. His patience and encouragement have been fundamental to my success, serving as the cornerstone of my journey through this research.

Equally, my sincere appreciation goes to my co-supervisor, Mr. TNB Wijethilake, whose insightful advice and steadfast support have been instrumental in navigating the complexities of my work. His contributions have not only enhanced the quality of my research but have also enriched my personal growth and understanding of the subject matter.

I am also profoundly thankful to Dr. Udayanga Wickramasinghe and Mr. Wathsala Vithanage from the UCSC Operating System Research Group. Their extensive knowledge and dedication to the field of operating systems have not only been a significant source of inspiration but have also greatly contributed to the depth and breadth of my research. Their willingness to share their expertise and insights has been invaluable, and for that, I am eternally grateful.

This project could not have reached its fruition without the collective wisdom, encouragement, and support of these distinguished individuals. Their belief in my abilities and their unwavering support have been pivotal in overcoming the challenges faced during this research. I am honored to have worked under their guidance and am deeply thankful for their contributions to both my academic and personal development.

I would finally like to express my profound gratitude to my parents and family, who have been my pillar of strength and support throughout all my endeavors. Their unwavering faith in me, unconditional love, and endless encouragement have been my constant source of motivation and resilience. It is with their support that I have been able to pursue my passions and navigate the challenges of my academic journey. Their sacrifices, understanding, and patience have not only fueled my determination but have also imbued me with the courage and confidence to strive for excellence in my research.

To my parents and family, I owe a debt of gratitude that can never be fully expressed in words. Your belief in me and your selfless support have made all the difference. Thank you for being my guiding light and for always standing by me. This achievement is not just mine but also yours

ABSTRACT

Network traffic monitoring is essential for maintaining the security and performance of software-defined networks (SDNs). It enables administrators to swiftly identify and address issues, optimize network performance, and protect sensitive data from cyber threats. With the increasing prevalence of virtual networks, the need for effective traffic monitoring has grown significantly. However, there is noticeable performance degradation (50% to 70%) when monitoring network traffic in virtual environments.

This study investigates the performance reduction associated with virtualized environments using virtual TAPs (Test Access Points) and proposes alternative methods to mitigate this issue. Initially, we utilized the Open vSwitch (OVS) virtual switch and VirtualBox VMs to construct the test network. However, due to inherent bottlenecks in VirtualBox, this approach was unsuccessful. We then built a virtual network using OVS and Docker containers, achieving a network speed between 30 Gbit/s to 35 Gbit/s, which provided a suitable bandwidth for further testing.

Our experiments examined the relationship between network traffic and mirror count, revealing no significant impact on network bandwidth. Subsequently, we analyzed network traffic with and without mirroring across various packet sizes, discovering a correlation between packet size and network performance. The most substantial performance drop, approximately 20%, was observed with 1024-byte packets, though this was notably less severe than findings reported by other researchers.

Using VTune Profiler and OS/Kernel Profile, we attempted to identify the root cause of the performance drop but were unsuccessful. Further investigation revealed that the physical NIC (Network Interface Card) was a bottleneck, as it struggled to handle mirrored traffic of 20–30 Gbit/s. High-capacity NICs are rare and expensive, prompting us to consider alternative solutions.

We propose a preprocessing tool integrated with an OVS-Docker container virtual network setup. This tool filters traffic by source and destination IP addresses and captures packet samples at specified intervals. Additionally, it allows users to input tshark commands directly, providing a flexible and efficient approach to network traffic monitoring.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

VM : Virtual machine

OVS : Open V Switch

VNI : Virtual Network Interface

VP: Virtual Port

eth0: Physical Network Interface of host 1

eth1: Physical Network Interface of host 2

SDN: software-defined networks

DPDK: Data Plane Development Kit

OVS: Open vSwitch

TAP: Test Access Point

SR-IOV:Single Root I/O Virtualization

LLC : Last Level Cache

GRE: Generic Routing Encapsulation

CPI: Cycles Per Instruction

CHAPTER 1

INTRODUCTION

1.1 Motivation

Network traffic monitoring is critical in software-defined networks (SDNs) because it allows network administrators to gain visibility into network activity and identify potential issues before they become major problems. Here are some specific reasons why network traffic monitoring is important in SDNs.

Security

- Detection of Malicious Activities:
    - Network traffic monitoring in SDNs helps in identifying suspicious activities, such as DDoS attacks, unauthorized data exfiltration, and intrusion attempts. By analyzing traffic patterns, SDNs can detect anomalies that deviate from normal behavior, enabling timely mitigation of potential threats.
- Policy Enforcement and Compliance:
    - SDNs can dynamically enforce security policies and compliance requirements by monitoring and controlling the traffic flowing through the network. This ensures that only authorized users and services have access to network resources, enhancing the overall security posture.
- Data Privacy and Integrity:
    - Monitoring allows for the inspection of data packets to ensure that sensitive information is encrypted and hasn't been tampered with during transmission. This is vital in protecting data privacy and maintaining integrity across the network.

Performance

- Traffic Analysis and Management:
    - By continuously monitoring network traffic, SDNs can analyze usage patterns, identify bottlenecks, and optimize resource allocation to improve network performance. This includes adjusting bandwidth allocations, rerouting traffic to avoid congested links, and scaling network resources dynamically based on demand.
- Quality of Service (QoS):
    - SDNs can prioritize traffic to ensure that critical applications receive the

necessary bandwidth and low-latency treatment. Network traffic monitoring is essential for implementing effective QoS policies that meet service level agreements (SLAs) and enhance user experience.

Troubleshooting

- Rapid Problem Identification:
    - Monitoring network traffic enables the quick identification of issues such as packet loss, high latency, or connectivity problems. This allows network administrators to promptly address and rectify issues before they impact end-users.
- Root Cause Analysis:
    - By providing detailed insights into network operations, traffic monitoring in SDNs helps in diagnosing the root causes of network issues. This can include hardware failures, misconfigurations, or external factors affecting network performance.
- Historical Data for Forensics and Planning:
    - Maintaining logs of network traffic enables historical analysis, which is useful for investigating security incidents and planning future network upgrades or expansions based on past usage trends and performance metrics.

In summary, network traffic monitoring is crucial for ensuring the security, performance of software-defined networks. It allows network administrators to quickly identify and address issues, optimize network performance, and keep sensitive data safe from cyber threats (Bird, 2023),(“What is Network Troubleshooting?,” 2023)

1.2 Statement of the problem

A virtualized Test Access Point (TAP) service provides monitoring capability in virtualized networks. However, virtual TAPs can contribute to performance degradation in a virtualized environment. (Wang et al., 2020) This research focuses on finding the root course to reduce performance on virtualization environments with Virtual TAPs and suggesting an alternative way to overcome the issue.

1.3 Research Aims and Objectives

1.3.1 Aim

Virtual TAPs are often implicated in causing notable performance degradation within these networks. This research endeavors to meticulously investigate the underlying reasons behind the performance issues. By doing so, it aims to propose an optimized virtual network configuration that not only reduces the performance reduction but also enhances overall efficiency. Furthermore, a significant part of this study is dedicated to developing a prototype to minimize mirror traffic volume. The prototype is achieved by implementing sophisticated preprocessing techniques, thereby addressing one of the critical challenges associated with virtual TAP services and ensuring a more seamless and efficient network monitoring process.

1.3.2 Objectives

- Do a proper literature review on current solutions that have for this performance issue.
- To simulate the issue, implement VMs in a virtualization environment with virtualized TAP and conduct deep investigations.
- Find the root course of the issue
- Design a solution
- Build prototype
- Evaluate the proposed network setup and prototype.

1.4 Research Questions

- How can we reduce the performance reduction of virtual networks when mirroring traffic?
- How to send mirrored data through a physical NIC without increasing the NIC's capacity?
- Why is there a performance reduction in virtual networks when mirroring traffic?

1.5 Scope

The scope of the project is to Implement two VM’s over a virtualization environment with virtualized TAP and identify the root course for performance degradation and design and implement a prototype. Following is considered as in scope for the proposed project.

- Do a proper literature review on current researchers that have to port mirroring and performance analysis.
- Study techniques and technologies (Data Plane Development Kit (DPDK), para-virtualization environment, Open V Switch (OVS), Single Root I/O

Virtualization (SR-IOV) that are used in those researches, finding strength and weakness of those technologies,and Gather sufficient knowledge to accomplish the desired task.

- Find the root cause of the issue.
- Design a solution.
- Build a prototype and measure performance using specialized tools.

1.6 Structure of the Thesis

- Chapter 2 of this report provides a detailed literature review of the current research regarding virtualized networks, virtualized TAP services, and their limitations.
- Chapter 3 describes the methodology of the proposed solution in detail.
- Chapter 4 describes the evolution plan of the proposed solution.

CHAPTER 2
LITERATURE REVIEW

2.1 Literature Review

Virtualization mainly has two types: Full Virtualization and Paravirtualization which are two techniques used in virtualization technology to create virtual machines (VMs) on physical servers

2.2 Full virtualization

Full virtualization is a technique that allows multiple operating systems to run simultaneously on a single physical server. Each operating system runs in its own virtual machine, which emulates the underlying hardware of the physical server, including the CPU, memory, storage, and network devices. The guest operating system running in the VM is not aware that it is running in a virtualized environment, as it has its own virtual hardware resources. The hypervisor, also known as the virtual machine monitor (VMM), manages the allocation of physical resources to virtual machines.

2.3 Paravirtualization

Paravirtualization is a technique in which the guest operating system is aware that it is running in a virtualized environment and has access to a special API (application programming interface) provided by the hypervisor. The guest operating system communicates with the hypervisor through this API to share resources and coordinate operations. This allows the guest operating system to achieve better performance than full virtualization, as it can directly access physical hardware resources rather than emulating them.

Table 2.1 : Full Virtualization Vs Paravirtualization

Features	Full Virtualization	Paravirtualization
Definition	It is the first generation of software solutions for server virtualization.	The interaction of the guest operating system with the hypervisor to improve performance and productivity is known as paravirtualization.
Security	It is less secure than paravirtualization.	It is more secure than full virtualization.
Performance	Its performance is slower than paravirtualization.	Its performance is higher than full virtualization.
Guest OS Modification	It supports all the Guest OS without any change.	The Guest OS has to be modified in paravirtualization, and only a few OS support it.
Guest OS hypervisor independent	It enables the Guest OS to run independently.	It enables the Guest OS to interact with the hypervisor.
Potable and Compatible	It is more portable and compatible.	It is less portable and compatible.
Isolation	It offers optimum isolation.	It offers less isolation.
Efficient	It is less efficient than paravirtualization.	It is more simplified than full virtualization.
Characteristic	It is software based.	It is cooperative virtualization.
Examples	It is used in Microsoft, VMware, and Parallels systems.	It is mainly used in VMware and Xen systems.

Table 2.1 explains and compares the main differences between Full Virtualization and Paravirtualization.(“Difference between Full Virtualization and Paravirtualization in Operating System - javatpoint,” 2023)

2.4 Docker

Docker is an open-source platform designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

In a way, Docker is a bit like a virtual machine. However, unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications to be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.The Docker platform is made up of several components:

Docker Engine: A lightweight and powerful open source containerization technology combined with a workflow for building and containerizing your applications.

Docker Hub: A cloud-based registry service which allows you to link to code repositories, build your images, test them, store manually pushed images, and link to Docker Cloud so you can deploy images to your hosts.

Docker Compose: A tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Docker simplifies and accelerates your workflow, while giving developers the freedom to innovate with their choice of tools, application stacks, and deployment environments for each project.("Docker Compose overview"; "Docker Engine overview," 100AD; "Overview of Docker Hub," 800; "What is Docker?," 2023)

2.5 Docker Container As Virtual Machines

Using Docker containers as lightweight virtual machines (VMs) is a popular practice for development, testing, and even production environments for certain use cases. While Docker containers are not VMs in the traditional sense they share the host OS kernel and do not require a hypervisor; they can be used to run isolated applications in environments that closely resemble VMs. ("Docker vs. VM (Virtual Machine)," 2023)

2.6 Data Plane Development Kit (DPDK)

DPDK stands for Data Plane Development Kit. It is a set of libraries and drivers that provide a high-performance framework for building packet processing applications. DPDK is designed to run on x86 processors and supports various operating systems such as Linux, FreeBSD, and Windows(currently a work in progress). DPDK provides a set of libraries and drivers for developing high-performance packet processing applications. These applications can be used in various use cases such as network function virtualization, cloud computing, and software-defined networking. DPDK provides a number of performance optimizations

that enable applications to achieve high packet processing rates with low latency.Some of the key features of DPDK include:

- Support for a wide range of NICs
- Support for multi-core processors
- Support for different packet processing modes such as polling and interrupt-driven
- Low-latency packet I/O
- Support for various network protocols
- Support for virtualization technologies such as SR-IOV and Virtio

DPDK is widely used in the networking industry and is supported by various hardware and software vendors. It is an open-source project that is maintained by the Linux Foundation. ("Overview — Data Plane Development Kit 24.03.0-rc1 documentation," 2023)

2.7 Open vSwitch (OVS)

Open vSwitch (OVS) is an open-source software switch designed to be used in virtualized environments such as data centers and cloud computing. It allows network administrators to create and manage complex virtual network topologies that can be used by virtual machines and containers.

OVS is compatible with many virtualization platforms, including KVM, Xen, and VMware. It is also supported by many cloud providers, including Amazon Web Services and Microsoft Azure.

Some of the key features of OVS include:

- Support for standard networking protocols such as OpenFlow, NetFlow, and sFlow. Support for virtual network interfaces, including VLANs and tunneling protocols such as GRE and VXLAN.
- Integration with orchestration frameworks such as OpenStack and Kubernetes. Fine-grained control over network traffic through the use of flow rules.
- Support Multiple tunneling protocols

("What Is Open vSwitch? — Open vSwitch 3.3.90 documentation," 2023)

2.8 Other Research related to the research.

In a paravirtualization system, inter-VM communication is expensive, and if used with a virtual tap, it can result in a 70% performance decrease. Using a hybrid approach that allows mirroring of VIRTIO port traffic to another VF (SR-IOV) via NIC hardware offloading, the performance decrease can be reduced to up to 50%. (Wang et al., 2020)

By implementing a new virtual Tap using DPDK, OVS with OpenFlow could achieve an 8 to 25 times throughput improvement. DPDK can accelerate the overall packet processing operations needed in vTAP, and an OpenFlow controller can provide a centralized and flexible way to apply and manage TAP policies in an SDN network. (Jeong et al., 2018)

Traffic mirroring is a configuration option for virtual switches. The most popular and widely used virtual switch alternative is Open vSwitch (OVS). Most current Linux distributions include OVS, which is open source. However, OVS performance, particularly throughput for smaller packets, is much lower than the line rate of the interface. To overcome this limitation, OVS was ported to the Data Plane Development Kit (DPDK), namely OVDK. The latter achieves an impressive line rate throughput across physical interfaces. (Shanmugalingam et al., 2016)

The internal complexity of VMs mainly affects the network throughput of SDN. If technologies like DPDK are used to bypass those internal complex layers, higher network throughput can be achieved. (Kourtis et al., 2015)

In summary, according to the research findings mentioned above, in an SDN, OVS with mirroring experiences significant performance reduction, and most attempts to find a solution involve combining OVS with DPDK. In this research, we are focusing on finding a solution with a virtual network that has OVS and Docker containers.

CHAPTER 3

METHODOLOGY

Figure 3.1 shows the methodology of this research, which has five steps.

1. Simulate the problem.
2. Clarify the problem.
3. Deep study about the issue and design a solution.
4. Create a prototype of the suggested solution
5. Evaluation

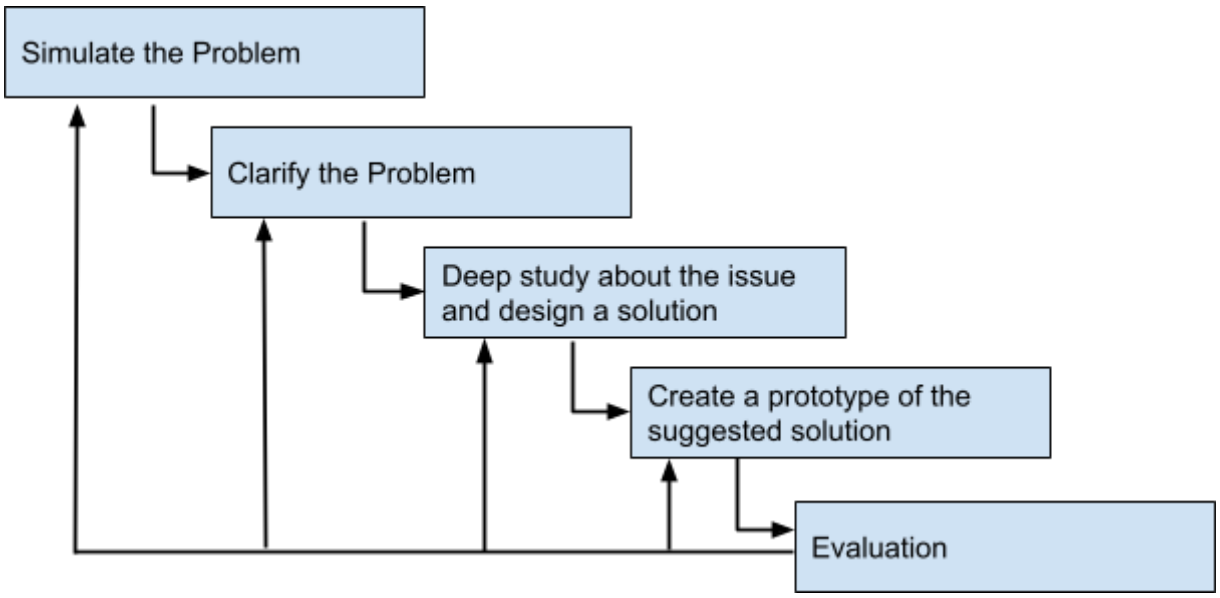

Figure 3.1 : Methodology

3.1 Simulate the Problem

In this research, it is necessary to simulate the virtual environment to study performance degradation in virtual networks when mirror traffic occurs. In this research, we have planned to use two different virtual environments for the experiments, such as Oracle VirtualBox and Docker. Figure 3.2 shows the proposed experimental setup for the research, and either

VirtualBox or Docker can be used as the virtualization platform. Each virtual machine will be connected using an OVS bridge.

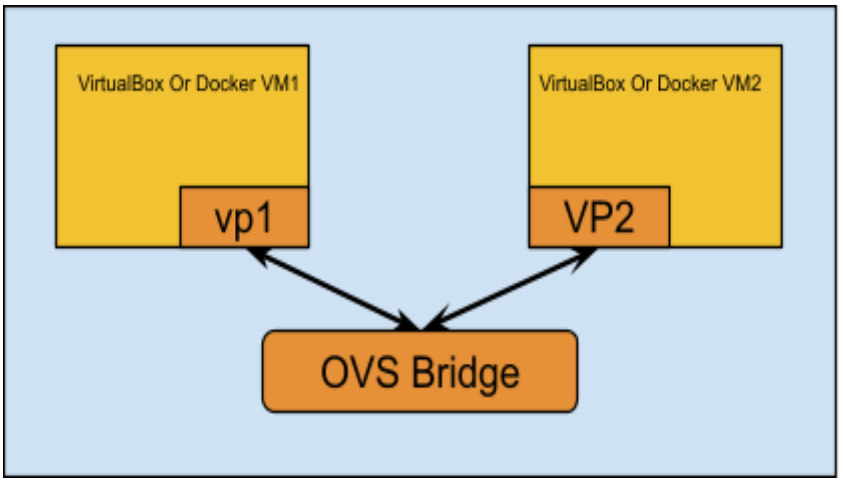

Figure 3.2 : OVS-VirtualBox and OVS-Docker Container network setup

In this experiment, we have planned to use Oracle VirtualBox version 7, Docker version 24.0.6, and OVS version 3.0.0. The operating system of the host machine will be Ubuntu 22.04.3 LTS. The virtual machines VM1 and VM2 will also be installed with Ubuntu 22.04.3 LTS. The Docker containers will use Alpine Linux v3.18.

3.2 Clarify the Problem

In the second stage, the primary focus will be on identifying whether there is an actual performance degradation when mirroring traffic in the virtual environment. We have planned several experiments to test the performance of the network with and without mirrors under multiple parameters. The parameters we planned to consider are as follows.

- Number of mirror counts.
- Data packet size.
- Multiple data transfer speeds.

In our experiments, we have planned to use the following tools to measure different aspects of the network connection.

iperf3

Version : 3.9

The tool was mainly used to generate traffic between VMs, and it was also used to monitor traffic. Using this tool, traffic was generated with different packet sizes and different bandwidths.

perf

Version 6.5.8

This is a performance analyzer tool used to analyze LLC details, L1 cache details, task clock, and cycle count.

Wireshark

Version 4.2.2

Wireshark is a free and open-source packet analyzer that is mainly used to check whether mirror ports are working. Also, the Wireshark tool is used to analyze the capability of physical NICs.

3.3 Deep Study about the issue and design a solution

This stage will be mainly focusing on identifying the root cause of the performance degradation when mirroring traffic and designing a solution. There will be a requirement of code modification in the OVS for implementing the proposed solution. The Hotspots Analysis in the VTune profiler will be used to find the functions in the OVS switch that consume the most CPU time. OS/Kernel profile analyses can be conducted using perf and the flame graph tool. Based on the findings, the best virtual network setup will be proposed, and a solution will address the physical network card bottleneck.

Following tools will be used in this stage of the research.

VTune profiler

Version 2023.2.0

The VTune Profiler is a performance analysis tool developed by Intel that helps optimize the performance of applications, systems, and kernel code. Hotspots Analysis, which is designed to identify the areas in your application where the most execution time is spent, was used. This is crucial for performance optimization because it allows developers to focus their efforts on optimizing the parts of the code that will yield the most significant performance improvements.

Perf

version 6.5.8

The tool is used to analyze the OS/Kernel profile. Perf is also used to record performance data system-wide. This data is saved in a file, which is not readily readable. The Flame Graph tool to convert this less readable file into a flame graph.

Flame Graphs

Git : https://github.com/brendangregg/FlameGraph/tree/master

This is an open source tool.The GitHub repository "FlameGraph" is a tool for visualizing stack traces. It turns profiled code into interactive, color-coded flame graphs that help in identifying performance bottlenecks and understanding system behavior. The repository

includes scripts for generating these graphs from various profiling tools like Linux perf, DTrace, and others

3.4 Create a prototype of the suggested solution

In this stage, the focus is on developing a functional prototype of the suggested solution and testing its functionality. A prototype will be developed using Java-based web service. This solution will have two main parts: frontend and backend. The frontend will be written using React.js, and the backend will use SpringBoot.

3.5 Evaluation

At this stage, the goal is to evaluate the proposed virtual network setup and suggested prototype.

Proposed virtual network Setup Evaluation

Mainly, the test focused on how performance drops when mirroring traffic. Traffic monitor with and without mirroring while changing various parameters, which are mentioned below.

- Number of mirror counts.
    - Used the test setup and iperf3 to generate traffic from VM1 to VM2. Additionally, the perf tool was used to find CPU usage statistics of the OVS switch.To collect statistics, iperf3 was also used.
- Data packet size.
    - In this test setup, the main difference from the previous is only using one mirror.iperf3 generates traffic from VM1 to VM2 and OVS mirror forward traffic from vp1 to mirror0-output.To collect statistics, iperf3 was also used.
- Network bandwidth
    - In this network test setup,using iperf3 generates different network traffic and monitors generated traffic and actual network data transfer speed.

Suggested prototype Evaluation

This prototype works as a tool to reduce mirror traffic sizeThis tool will be evaluated using Wireshark and custom tool output.

Finally, during the evaluation stage, when analyzing the results, certain network configurations and testing methods had to be reversed, rejected, or accepted depending on the situation. If reversal was necessary, it was required to revert to a previous stage, as indicated by an arrow starting from the evaluation stage and pointing back to preceding stages.

CHAPTER 4
EVALUATION AND RESULTS

4.1 Test Flow

The main focus of this section is to provide some basic ideas before delving into a detailed explanation of each test. It will outline the test setup and results at a high level, with Figure 4.1 illustrating how the test setup evolves from the start.

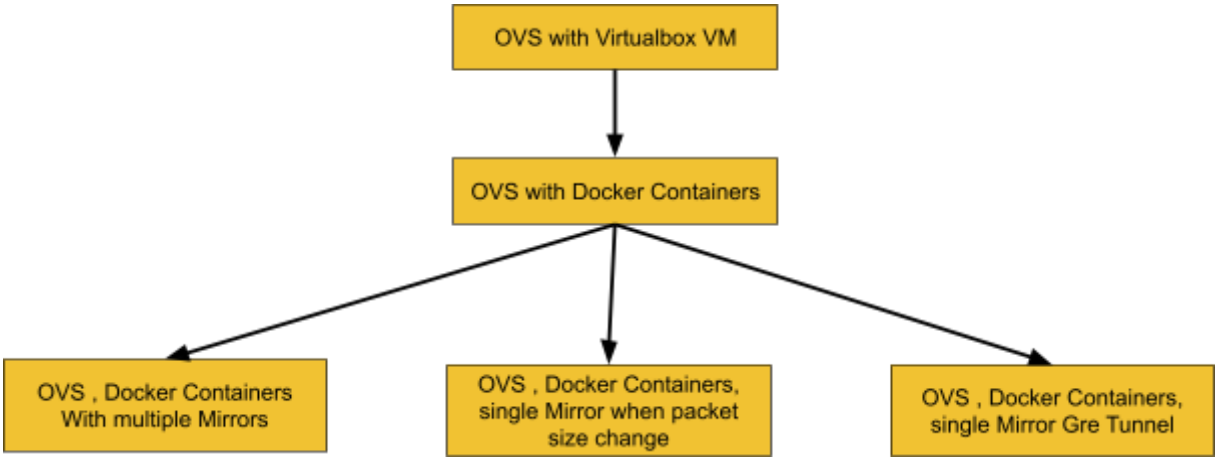

Figure 4.1: Test Flow

4.1.1 OVS with Virtualbox VM's

Since OVS is one of the latest technologies used to build virtual networks, it has been utilized along with VirtualBox to create virtual machines. The main intention was to simulate and study the research problem. However, with VirtualBox VMs, the expected results could not be achieved. With VirtualBox, the maximum network traffic achieved was 200 Kbits/s. With OVS mirroring, no change in network performance was observed. Virtualbox Virtual machine network traffic can be slower for several reasons. VMs emulate entire hardware systems, including the network stack, which can introduce additional overhead. Each VM operates with its own full instance of an operating system, leading to heavier resource usage and slower network performance. Due to vms bottleneck hiding the performance reduction, that happens due to mirrors. This occurs due to a bottleneck in VMs masking the performance decline caused by mirrors. Because of the mentioned issues the VirtualBox option as the virtual environment is not suitable for the experiment setup.

4.1.2 OVS with Docker Containers

Using Docker, the same OVS virtual network was utilized, and a maximum bandwidth of 33 Gbits/s was observed. However, even Docker has some issues when used as a VM. Docker

serves as a lightweight alternative to VMs because it employs operating system-level virtualization to package and isolate applications in "containers." This leads to efficient resource usage, as containers share the host system's kernel, rather than each virtual machine requiring a full operating system.

Since high network bandwidth was observed, the decision was made to continue with Docker containers instead of VMs. Consequently, the virtual network setup was finalized, involving OVS with Docker containers. Testing then continued in Three main ways, aiming to determine how network performance is affected by mirrors.

1. OVS, Docker Containers with Multiple Mirrors (Test One)
2. OVS, Docker Containers, single mirror when packet size changes (Test Two)
3. OVS, Docker Containers, Single Mirror Gre Tunnel (Test Three)

Those testing one by one explain according to the order.

4.1.3 Test One

To implement this configuration(Figure 4.3), simply include several mirror ports in the suggested network setup. The primary objective was to examine the correlation between network traffic and mirror count.In addition, the CPU utilization of OVS was measured while concurrently monitoring the mirror count.

In general, a rise in mirror ports results in elevated (Last Level Cache)LLC loads, LLC misses, and CPU branch activity. Monitoring results in additional costs or burdens. There are variations in network bandwidth across situations with and without monitoring, although the shift is just 2 Mbit/second. Consequently, as the correlation between the data rate and mirror count could not be determined, an alternative approach was employed.

4.1.4 Test Two

The next test was conducted in the same OVS Docker container setup (Figure 4.7), with one mirror, comparing different network bandwidths with and without a mirror. In this test, it was observed that the maximum network bandwidth with a mirror is around 4 Gbit/s less than without a mirror. For the first time, a considerable performance drop was observed.

After observing a performance drop, the next attempt was to determine whether there was a relationship with packet size. Therefore, using the same setup as previously, network performance was monitored with different data packet sizes. In this setup, the maximum performance drop observed with a packet size of 1024 Bytes was around a 20%(compared to without mirror traffic) drop.

Having observed a maximum 20% performance drop, the next focus was on identifying the root cause of that issue. Using Vtune Profiler to analyze the OVS switch process, flame graphs were generated with and without mirroring. However, despite some deviations, unfortunately, no evidence could be found to explain the performance degradation.

Since nothing useful could be observed, an attempt was made to take an OS/Kernel profile. For that kernel profile, flame graphs were also created with and without a mirror. Those two graphs showed some changes, but no evidence could be found to explain the performance degradation.

A maximum performance drop of 20% was found at the 1024-byte size, but the performance degradation for all other packet sizes was lower than this. However, due to the limited amount of time, further experiments to find the root cause of the 20% performance degradation were ceased.

4.1.5 Test Three

Since the mirror traffic bandwidth was around 30 Gbit/s, there were doubts about mirroring large traffic outside the host machine. Figure 4.16 shows the test setup. To test this, the Generic Routing Encapsulation (GRE) tunnel was used to forward traffic to another machine. However, since this physical NIC only supports 1 Gbit/s, we observed a bottleneck at the host machine NIC.

4.2 Tests

In this section, provide a detailed explanation for each test that was explained in the Test Flow.

4.2.1 OVS with Virtualbox VM's

According to the literature review there are many researchers mention a performance drop in virtual networks with virtual TAPs. In this research, I plan to start a test with an OVS virtual switch.

Initially made a virtual network with OVS with VM's that were made with virtual box.

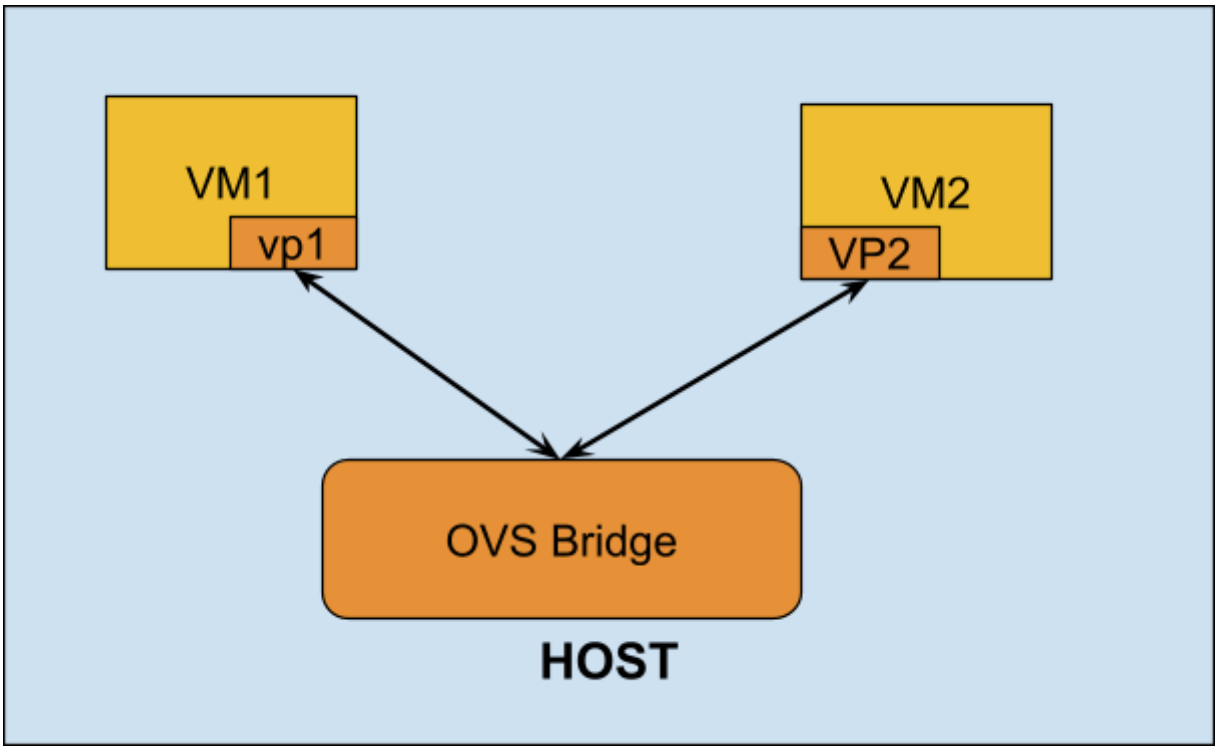

Figure 4.2 : OVS- VirtualBox

Figure 4.2 illustrates how the OVS virtual network connects two VirtualBox VMs. In this setup, the virtual network operates on a single host, with VM1 connected to VM2 via the OVS bridge. VP1 and VP2 are Virtual Network Interfaces (VNI's) that belong to VM1 and VM2, respectively. The following are the specs of the host computer, OVS bridge, and VMs.

- Host machine configuration
    - Ram : 16 Gb
    - Processor : {Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz,Core Count: 4}
    - Ubuntu 22.04 Operating System
- OVS bridge
    - OVS Version : 3.0.0
    - Inside OVS, create a mirror.
- VM1(virtual machine)

    - This is created using virtualbox.
    - Ubuntu 22.04 Operating System
- VM2 (virtual machine)
    - This is created using virtualbox.
    - Ubuntu 22.04 Operating System

Using iperf3, traffic was generated from vm1 to vm2, but it did not exceed 200 Kbits/s. Didn't observe any performance difference with mirror and without mirror.

Doing various configuration changes (bellow mentions) tried to increase the network speed.

- Checked and adjusted the VMs' network adapter settings.
- Allocated more resources to the VMs.
- Reviewed the host system's network load and reduced unnecessary consumption.
- Temporarily disabled antivirus/firewall software to test if it was causing the slowdown.
- Updated VirtualBox and the guest OS to the latest versions.

However, all attempts were unsuccessful. There can be various reasons for this low network bandwidth.

Virtual machines run on top of a hypervisor (like VMware ESXi, Microsoft Hyper-V, or Oracle VirtualBox), which emulates physical hardware. Each VM includes a full copy of an operating system, the application, necessary binaries, and libraries, which consumes more resources. They typically take a longer time to boot up because they need to load the entire operating system and the virtualized hardware. The additional layer of the hypervisor in VMs can lead to performance degradation. (Strydom, 2024; "Why is My Virtual Machine so Slow (5 Tips to Speed up)," 2022)

So, the likely reason for not observing performance degradation with the mirror is that the VM bottleneck is greater than the performance drop. Then we have to think of other solutions that can be used instead of virtualbox VM's. After doing some research Docker was chosen.

Docker, primarily known for containerization, allows applications to run in isolated environments, ensuring consistency across different systems. Although not a virtual machine (VM) in the traditional sense, Docker provides a lightweight alternative to VMs by enabling applications to share the same operating system (OS) kernel, reducing overhead. Unlike VMs, which emulate entire hardware stacks, Docker containers interact directly with the host OS, offering faster startup times and lower resource usage. This makes Docker an efficient tool for

developers looking to deploy applications seamlessly across different environments, without the need for full-fledged VMs.

Then, the VirtualBox VMs were changed to Docker containers on the same network setup. With the new setup tested, the network bandwidth observed was around 33 Gbits/s, which was a huge success. It was decided to go with this OVS-Docker container network setup. Then, testing continued with this proposed network setup.

4.2.2 Test One

In this setup, just add multiple mirror ports to the proposed network setup. The main intention was to check the relationship between network traffic and mirror count.

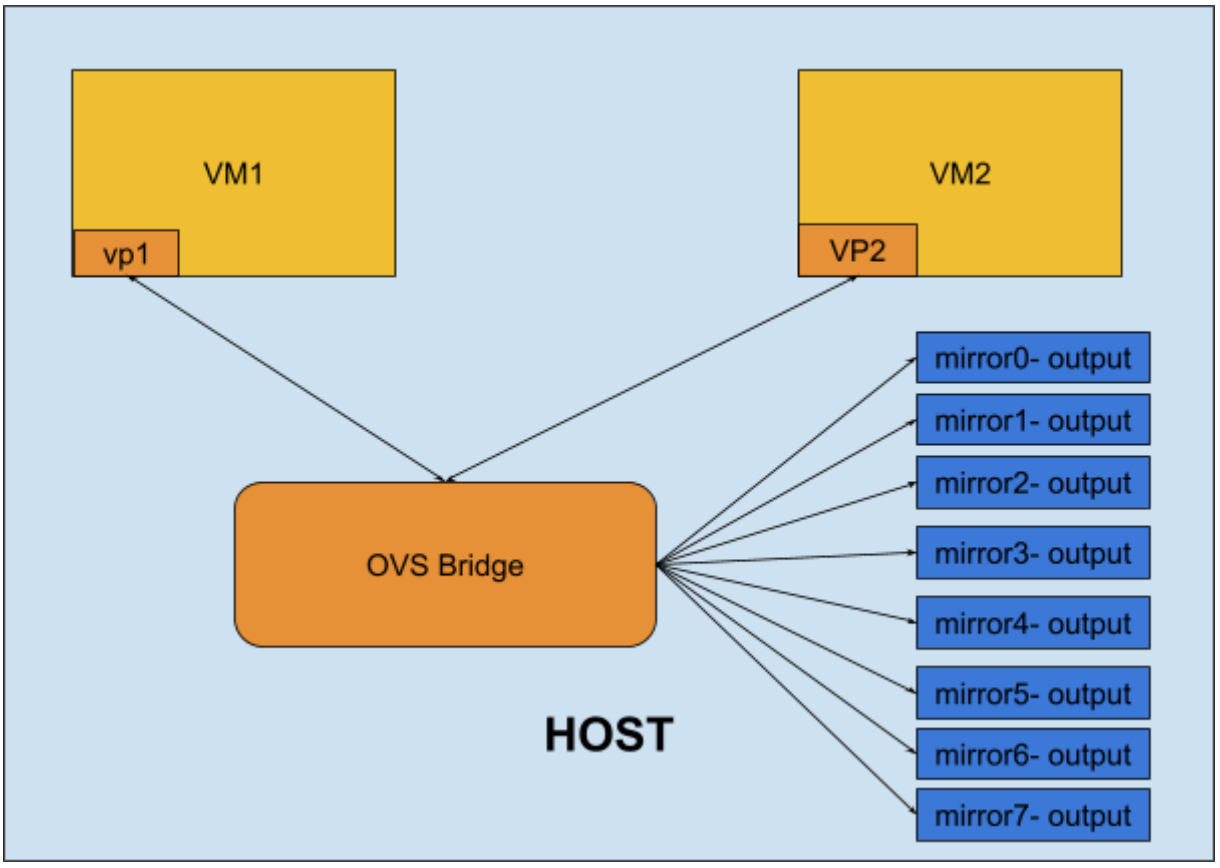

Figure 4.3 : Test 1 Setup

Figure 4.3 illustrates the test setup: build a virtual network that runs on a single host. VM1 is connected to VM2 over the OVS bridge. VP1 and VP2 are VNIs that belong to VM1 and VM2, respectively. All the traffic coming from the VP1 port is mirrored to mirror ports. The following are the specs of the host computer, OVS bridge, and VMs.

- Host machine configuration
  - Ram : 16 Gb
  - Processor : {Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz,Core Count: 4}
  - Only allow to use 2 and 3 cores to host machine (“isolcpus” cmd use)

    - Ubuntu 22.04 Operating System
- OVS bridge
    - OVS Version : 3.0.0
    - Run on core 0 (“taskset -c 0”cmd use)
    - Vp1 port all traffic (ingress and egress) mirror to mirror-output port
    - Inside OVS create a mirror
- VM1(Docker Container)
    - This is created using a Docker container.
    - Vm1 run on core 0 (“taskset -c 0”cmd use)
    - Alpine Linux v3.18 Operating System
- VM2 (Docker Container)
    - This is created using a Docker container.
    - Vm2 run on core 1 (“taskset -c 1”cmd use)
    - Alpine Linux v3.18 Operating System

Initially, create a virtual network using OVS, and then create ports vp1 and vp2. Next, create Docker containers vm1 and vm2 and connect them with the virtual network that was created earlier. Set proper IP addresses for each, vp1 and vp2, and then create a number of mirrors as required for each test. Generate traffic from vm1 to vm2 using iperf3.

Test case design

In this test, the data rate and CPU usage was measured with the number of mirror ports. The test was initiated with 8 mirrors (VP1 port mirrored traffic to mirror0-output, mirror1-output, mirror2-output, ..., mirror7-output). To achieve this, OVS mirrors were created for each mirror-output port because an OVS mirror only allows one output port.

Traffic Generation

Use iperf3 , traffic generated from vm1 to vm2. To Capcher traffic also use iperf3

Mirror traffic

Using OVS mirror mirrored the vp1 port traffic to(mirror0-output, mirror1-output ,mirror2-output, mirror3-output, mirror4-output, mirror5-output, mirror6-output, mirror7-output) Ports

Monitoring

Following cmd used to monitor each mirror port and used above cmd to save mirrored data. The purpose of this command is to add listeners to each mirror port.

"timeout 120s ifstat -n -t -b -i mirror0-output -i mirror1-output -i mirror2-output -i mirror3-output -i mirror4-output -i mirror5-output -i mirror6-output -i mirror7-output 1 >> testdata.csv "

Following cmd used to generate traffic from VM1 to VM2. This can also be used to get transfer data size and transfer data rate.this cmd run on VM1

"**iperf3 -c 192.168.1.22 -l 64 -t 120 -p 5201**"

Following cmd is used to measure cpu usage parametros of OVS switch.This cmd runs on the host machine. Those data are in Table 4.1.

"**perf stat -e LLC-prefetches,LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores,task-clock,context-switches,cpu-migrations,page-faults,cycles,instructions,branches,branch-misses -p 6251 sleep 140**"In Test 1, 9 separate test scenarios are used.

- Test 1: With 8 Mirror Ports. With monitoring of each mirror port.
- Test 2: With 8 Mirror Ports. Without monitoring each mirror port.
- Test 3: With 4 Mirror Ports. With monitoring of each mirror port.
- Test 4: With 4 Mirror Ports. Without monitoring of each mirror port.
- Test 5: With 2 Mirror Ports. With monitoring of each mirror port.
- Test 6: With 2 Mirror Ports. Without monitoring of each mirror port.
- Test 7: With 1 Mirror Ports. With monitoring of each mirror port.
- Test 8: With 1 Mirror Ports. Without monitoring of each mirror port.
- Test 9: With 0 Mirror Ports.

Results for Tests 1 to 9 are in Table 4.1 (The results came from the above command in each test.).

Table 4.1 : Test Setup 1 Result

Mirror Ports	8		4		2		1		0
Monitoring	Yes	No	Yes	No	Yes	No	Yes	No	N/A
Test No	1	2	3	4	5	6	7	8	9
Interval sec	120	120	120	120	120	120	120	120	120
Transfer GBytes	3.25	3.27	3.26	3.27	3.27	3.28	3.28	3.28	3.28
Bitrate Mbits/sec	233	234	234	234	234	234	235	234	235
Retr	0	0	0	0	0	0	0	0	0
PPS	455078.125	457031.25	457031.25	457031.25	457031.25	457031.25	458984.375	458984.375	458984.375
LLC-loads	1,939,796	1,872,257	2,001,652	1,834,107	1,988,224	1,959,110	2,036,004	1,933,427	1,892,003
LLC-load-misses	1,381,918	1,330,585	1,274,091	1,154,136	1,302,411	1,235,982	1,440,283	1,206,000	1,192,630
LLC-stores	170,210	186,115	184,771	178,626	158,013	128,018	154,785	153,901	160,988
LLC-store-misses	128,334	114,262	116,616	114,478	109,680	94,086	106,727	107,043	111,587
L1-dcache-loads	57,319,649	58,816,894	58,938,595	63,163,986	57,937,050	59,812,751	48,526,816	55,079,482	57,563,139
L1-dcache-load-misses	3,701,280	3,494,224	3,919,786	4,141,517	3,861,730	4,163,733	3,211,951	3,729,892	3,956,830
L1-dcache-stores	36,990,272	33,931,431	37,824,447	39,706,528	35,910,048	39,274,603	29,428,585	35,719,819	37,438,082
Task Clock (msec)	133.17	123.02	120.74	130.92	119.34	144.36	122.61	116.95	120.14
Context Switches	1,168	1,196	1,397	1,356	1,199	1,246	1,135	1,221	1,221
Cycles	337,225,517	297,320,917	346,294,664	337,222,176	320,840,017	334,099,218	267,877,872	346,202,619	346,202,619
Instructions	219,947,617	204,036,891	217,314,221	204,993,905	206,492,553	221,036,018	170,008,618	219,699,289	219,699,289
Branches	47,487,476	45,086,447	47,705,629	44,892,828	45,558,194	48,712,986	36,218,650	43,690,917	45,752,198
Branch Misses	1,284,138	1,241,560	1,383,192	1,286,884	1,274,958	1,268,057	1,059,110	1,254,779	1,324,651
CPI	1.53321	1.45719	1.59352	1.64504	1.55376	1.51151	1.57567	1.5758	1.5758

Table 4.1 lists many parameters, but the most important ones are cycles, CPI, and Bit Rate. To observe how these parameters change with test numbers, refer to Graphs 4.4, 4.5, and 4.6.

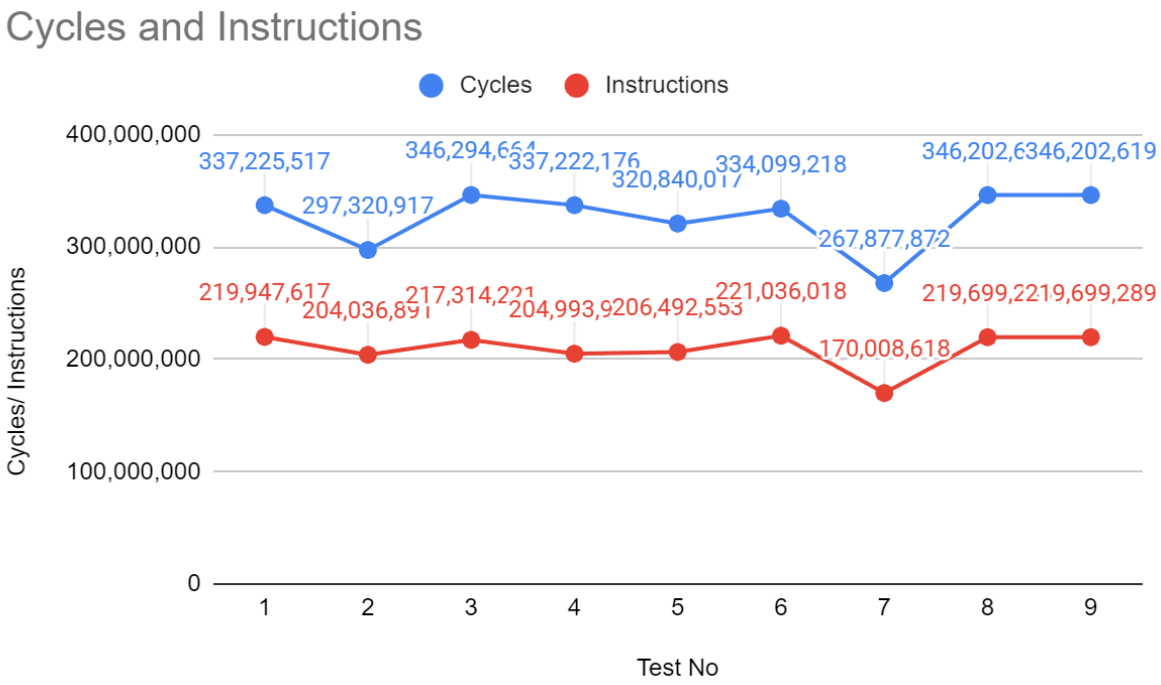

Figure 4.4 : OVS Switch Cycles and Instructions in each Test Case

The number of cycles for the OVS switch in each test case is shown in Figure 4.4, and a summary is explained as follows.

- There's significant variation in the Cycles count, ranging from 267,877,872 to 346,294,664.
- The lowest Cycles count is observed in Test 7 (with 1 Mirror Port and Monitoring enabled), and the highest is in Test 3 (with 4 Mirror Ports and Monitoring enabled).
- This suggests that the number of Cycles might be influenced by the combination of Mirror Ports and Monitoring status, but not in a straightforward linear relationship.

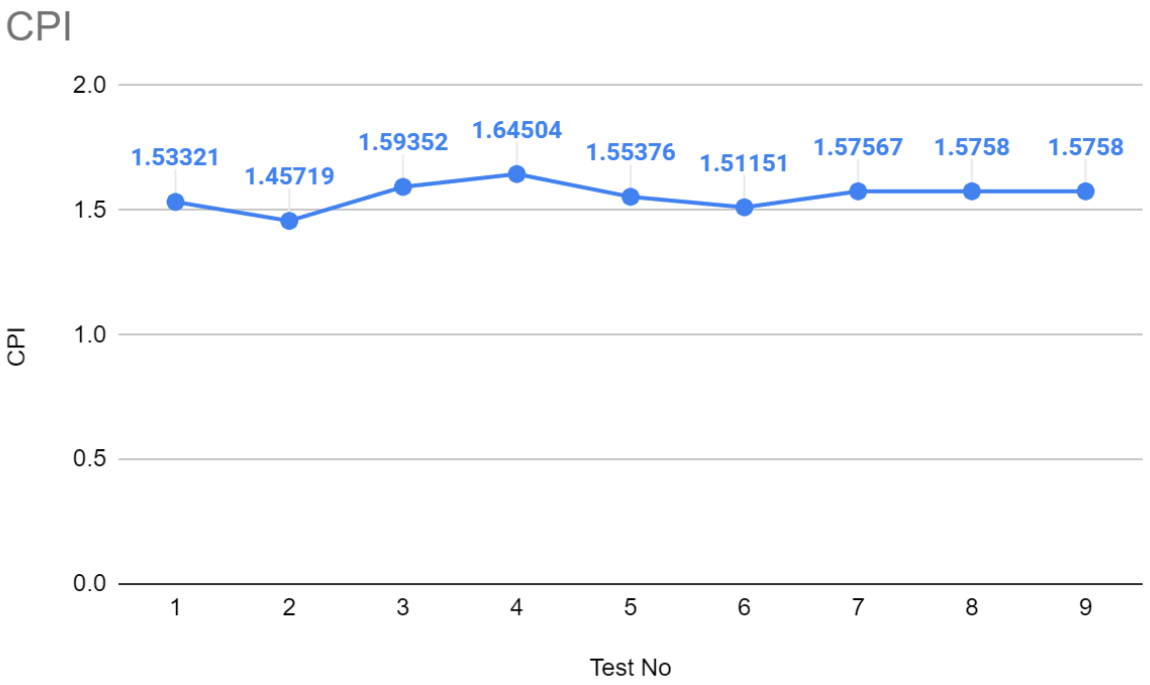

Figure 4.5 :OVS Switch CPI in each Test Case

The Cycles Per Instruction (CPI) count for the OVS switch in each test case is shown in Figure 4.5, and a summary is explained as follows.

- The CPI values range from approximately 1.457 to 1.645. A lower CPI suggests more efficient CPU usage.
- The lowest CPI (most efficient CPU usage) is observed in Test 2 (8 mirror ports, no monitoring), while the highest CPI (less efficient CPU usage) is in Test 4 (4 mirror ports, no monitoring).

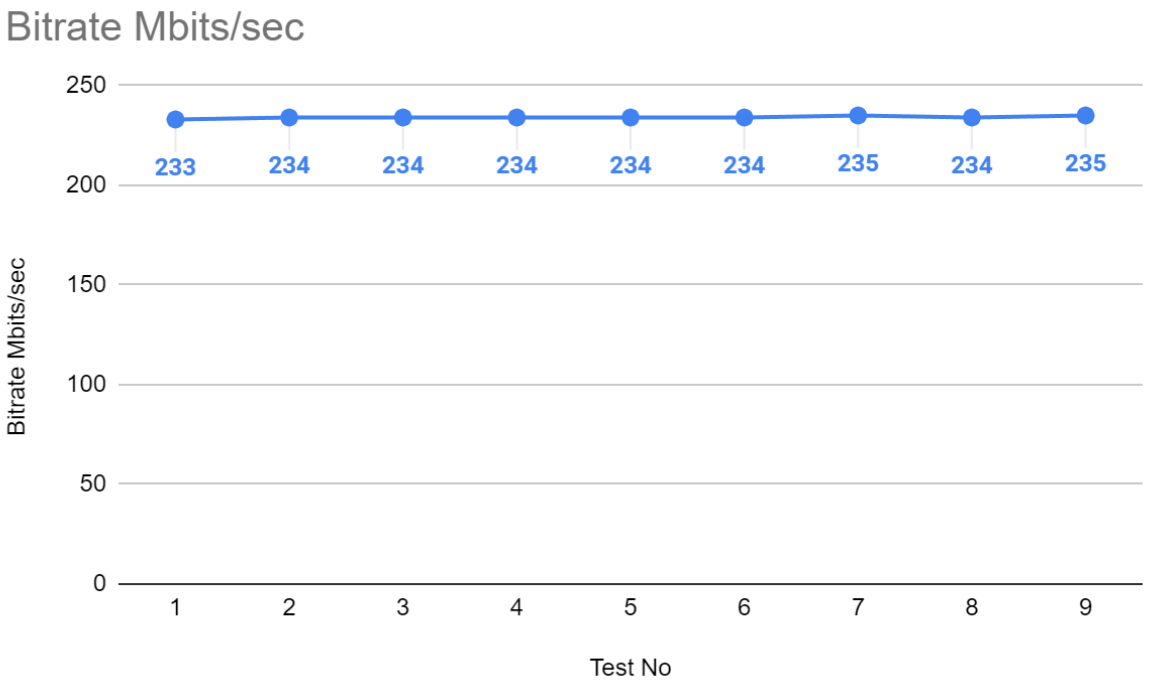

Figure 4.6 : Bitrate of VM1 to VM2 in each test case

The data rate of the network for each test case is shown in Figure 4.6, and a summary is explained as follows.

- The Bitrate is mostly consistent, hovering around 234 Mbits/sec, with only Tests 1 and 7 showing slight variations (233 and 235 Mbits/sec, respectively).
- This consistency in Bitrate across different Mirror Port settings and Monitoring conditions may indicate stability or a lack of direct influence from these factors on the Bitrate.

Test One summary

Generally, an increase in mirror ports leads to higher LLC loads, LLC misses, and CPU branch activity. Monitoring incurs some overhead. There are network bandwidth differences between scenarios with and without monitoring, but it is a 2 Mbit/second change. Therefore, since the relationship between the data rate and mirror count could not be found, another method was adopted.

4.2.3 Test Two

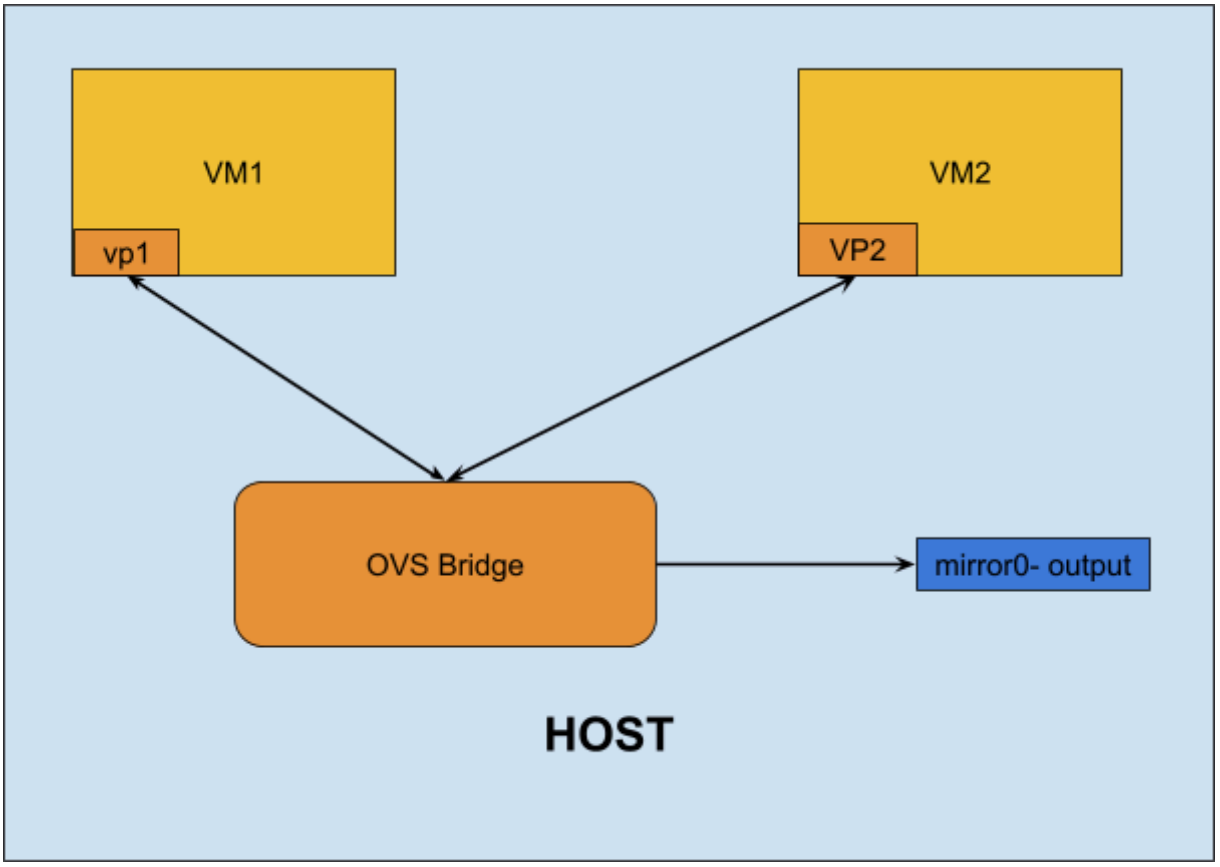

Figure 4.7 : Test 2 Setup

Figure 4.7 shows the Test Setup. Since the previous test couldn't observe any result, it was decided to go with a single mirror. To determine whether mirroring can affect network bandwidth, testing was extended to a few more aspects.

In the same test setup, the actual data transfer speed was measured over the data sending speed. The data sending speed could be controlled using the iperf3 traffic generator, and the actual data transfer speed was obtained from the result set that returned after the iperf3 run. As mentioned before, data was transferred from VM 1 to VM 2. Using the following types of Iperf3 cmds to control network traffic: “iperf3 -c 192.168.1.22 -b 1G”

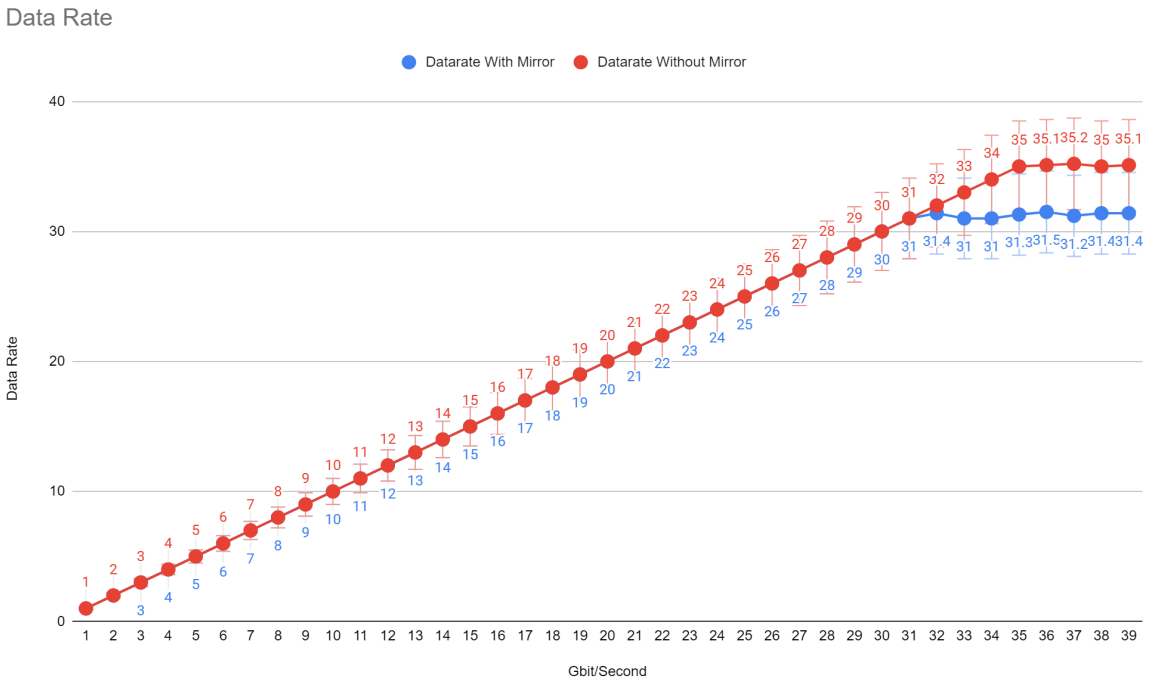

Figure 4.8 : With Mirror and Without Mirror Network Speed over Data Rate

Figure 4.8 shows actual network traffic variation over configured traffic. The description of the test result observed is provided as follows.

- The data rate ranges from 1 Gbit/second to 39 Gbit/second.
- The network traffic became saturated with mirrored traffic, unlike before without a mirror.
- From this, it can be said that the mirror has some effect on network traffic.

Since performance discrepancies were observed, an investigation into the issue was attempted. Therefore, an attempt was made to collect network speed and PPS data over different packet sizes, with and without a mirror. To generate traffic in different packet sizes, use the command “iperf3 -c 192.168.1.22 -l 100”

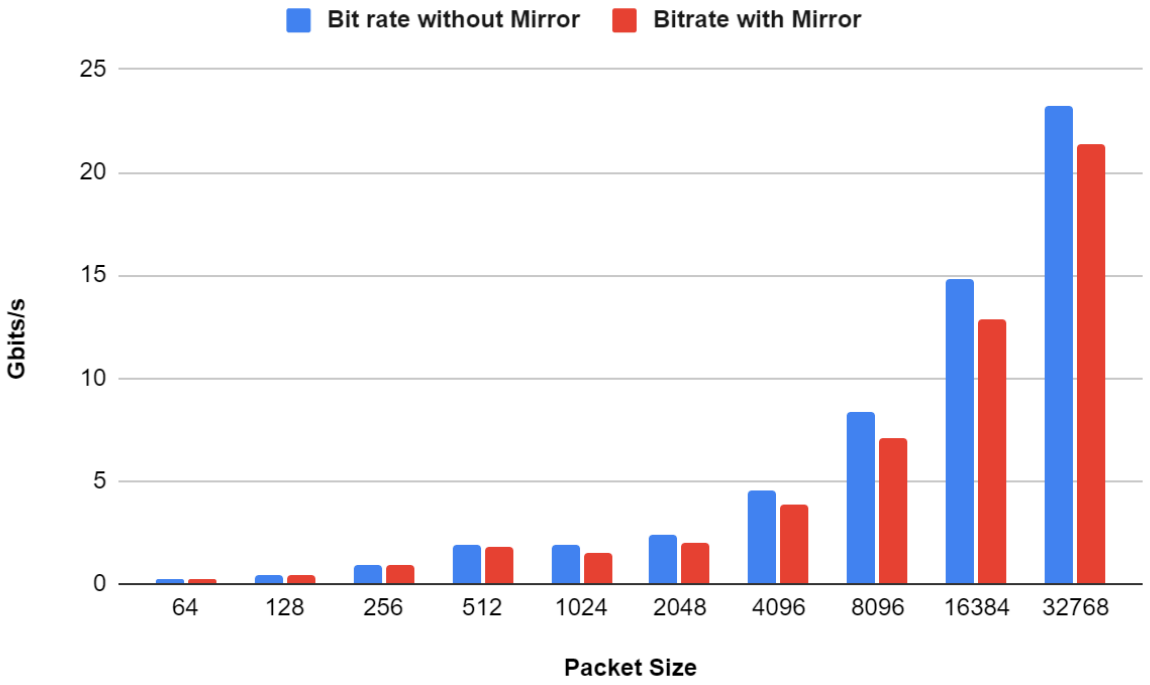

Figure 4.9 : With Mirror and Without Mirror Network Speed over Packet Size

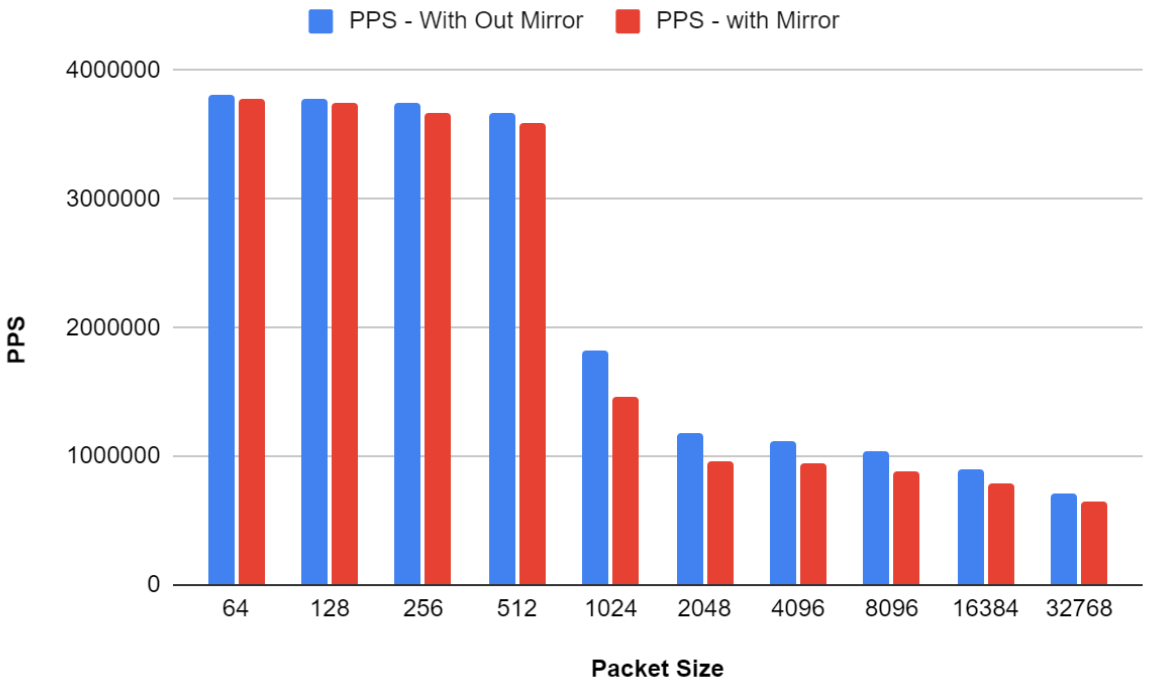

Figure 4.10 : With Mirror and Without Mirror PPS over Packet Size

Figures 4.9 and 4.10 show network speed in Gbits/s and PPS over with and without mirror. It is clear that there are some performance drops due to mirrors, so it would be better to determine which packet size experiences the highest performance drop.

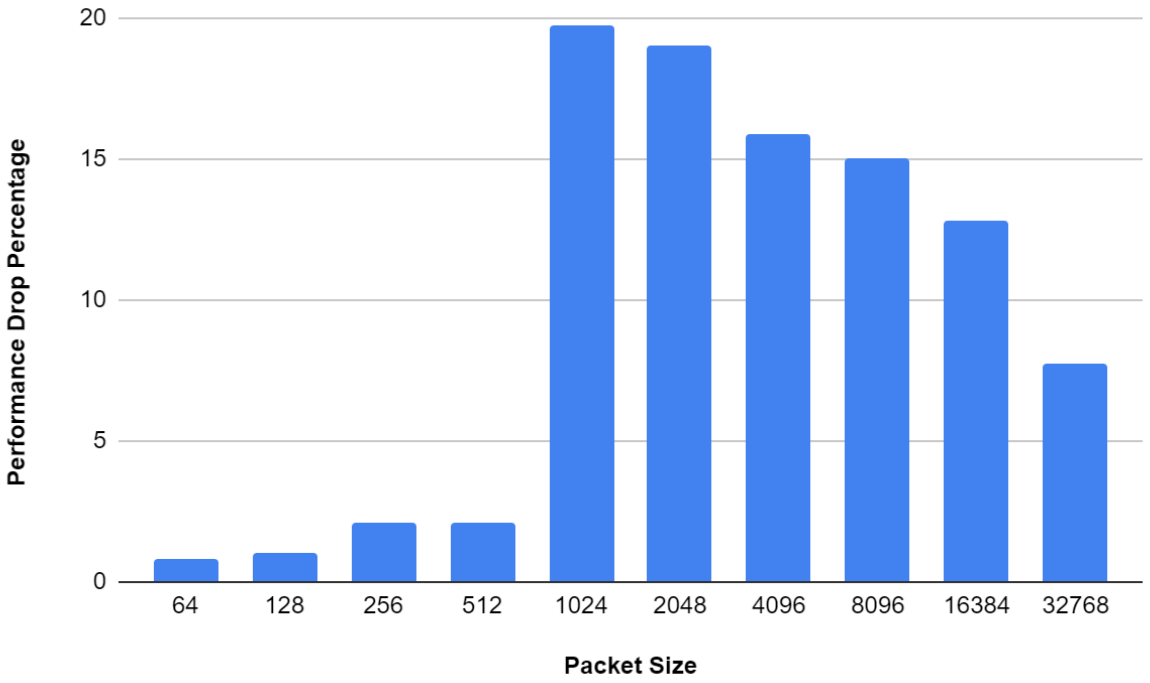

Figure 4.11 : Performance Drop Over Packet size

Figure 4.11 shows the performance drop percentage compared to without a mirror in each packet size. The description of Figure 4.11 is as follows.

- In the range of 1 to 4 MPPS, when the packet size increases, the MPPS count decreases.
- When packet size increases, data rate increases.
- When the packet size is 1024 bytes, the maximum performance drop is observed, and it is less than 20%.

Since a maximum 20% performance drop was observed, the focus was on finding the root cause of this performance drop. Perhaps the reason for this is the OVS switch or OVS source code. Anyway, VTune Profiler was used to analyze the OVS switch.

Since the maximum performance drop recorded was with a 1024-byte packet size, OVS performance was checked both with and without mirroring for 1024-byte packet size, using the VTune Profiler.

Sample collection time 100 S.

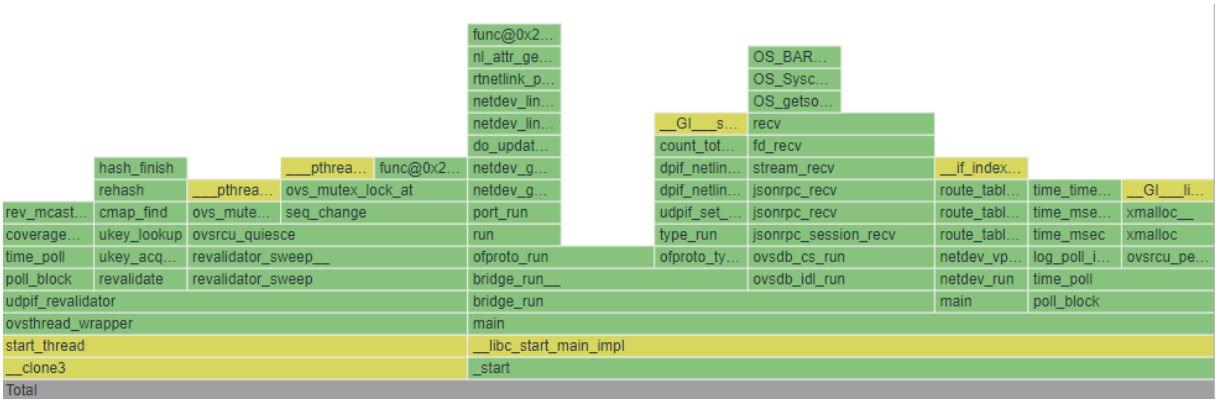

Figure 4.12 : Without Mirror Flame graphs of ovs-vswitchd

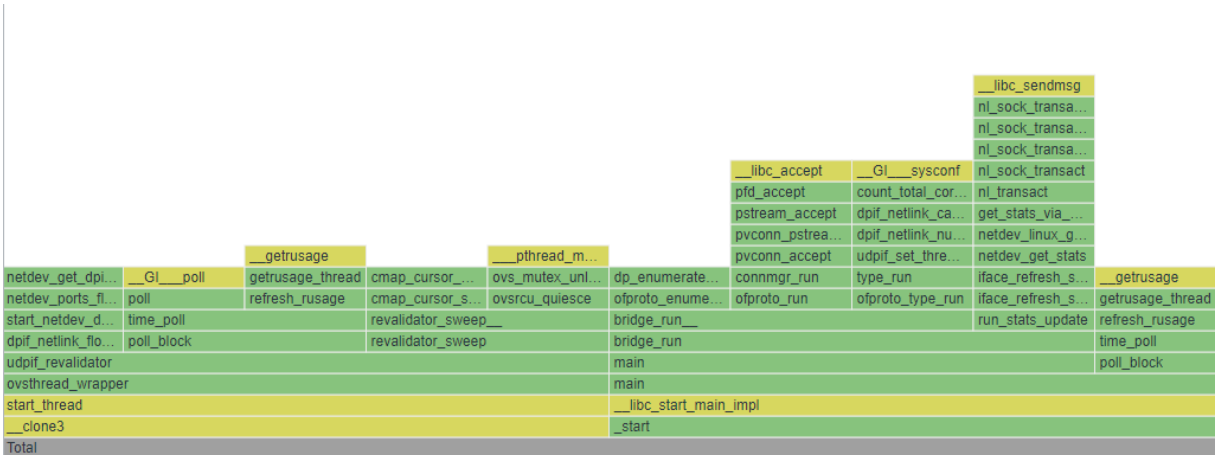

Figure 4.13 : With Mirror Flame graphs of ovs-vswitchd

Compared to with mirror (Figure 4.13), the _start function takes a lot of CPU time without mirror (Figure 4.12), but high CPU usage wasn't observed in ovs-vswitchd. Because of the mirror CPU time that is used in the OVS switch (0.13 seconds) without the mirror (0.1 seconds), the OVS switch does not use much CPU power. So these two graphs did not show any direct relation to performance degradation.

Low CPU utilization is a good sign for network performance; however, in this case, there is a difference in performance patterns between the two scenarios. Since this flame graph didn't show kernel symbols, the decision was made to look at the OS/kernel profile as well. Following cmd is used to capture OS/kernel profile data samples.

cmd : perf record -F 99 -a -o ./perf.my.data -g -- sleep 65

Using the aforementioned command, it monitors the system's performance during a 65-second sleep period. Data was captured on the host machine. This can be useful for capturing a

snapshot of system activity and performance issues that occur in a relatively idle state or to compare with more active system states during other operations.

Performance data was collected using the above command on a host machine, both with and without mirroring.

The aforementioned command output was a perf.my.data file, which is somewhat unreadable, so it has to be converted to a readable form. “FlameGraph” is a tool that can convert it to. SVG flamegraph from

OS/kernel profile is analyzed by applying the command with and without mirror to the host machine, then using the FlameGraph tool. data file converted to the SVG flamegraph, as shown in Figures 4.14 and 4.15.

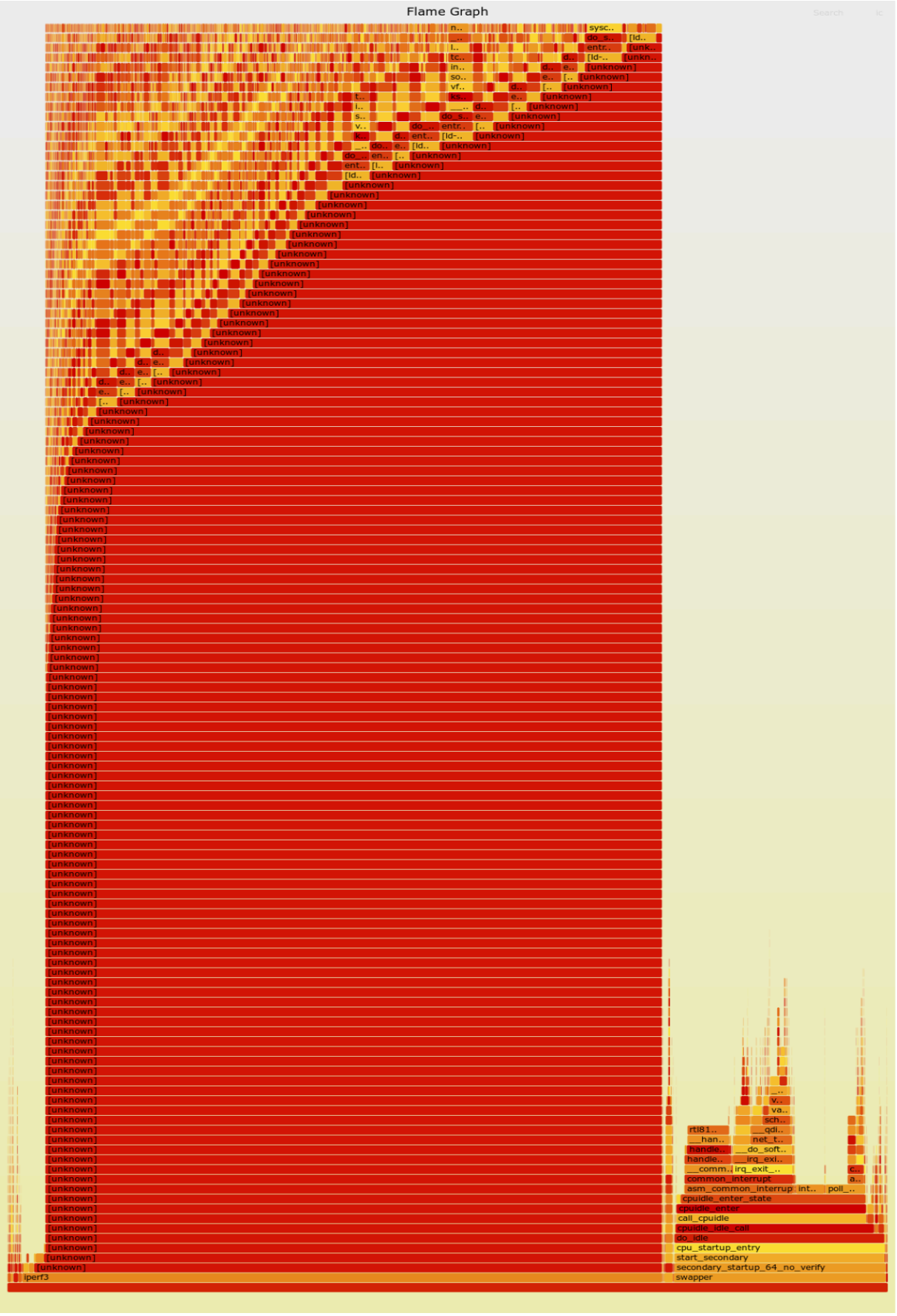

Figure 4.14 : With Mirror Kernel profiler FlameGraph

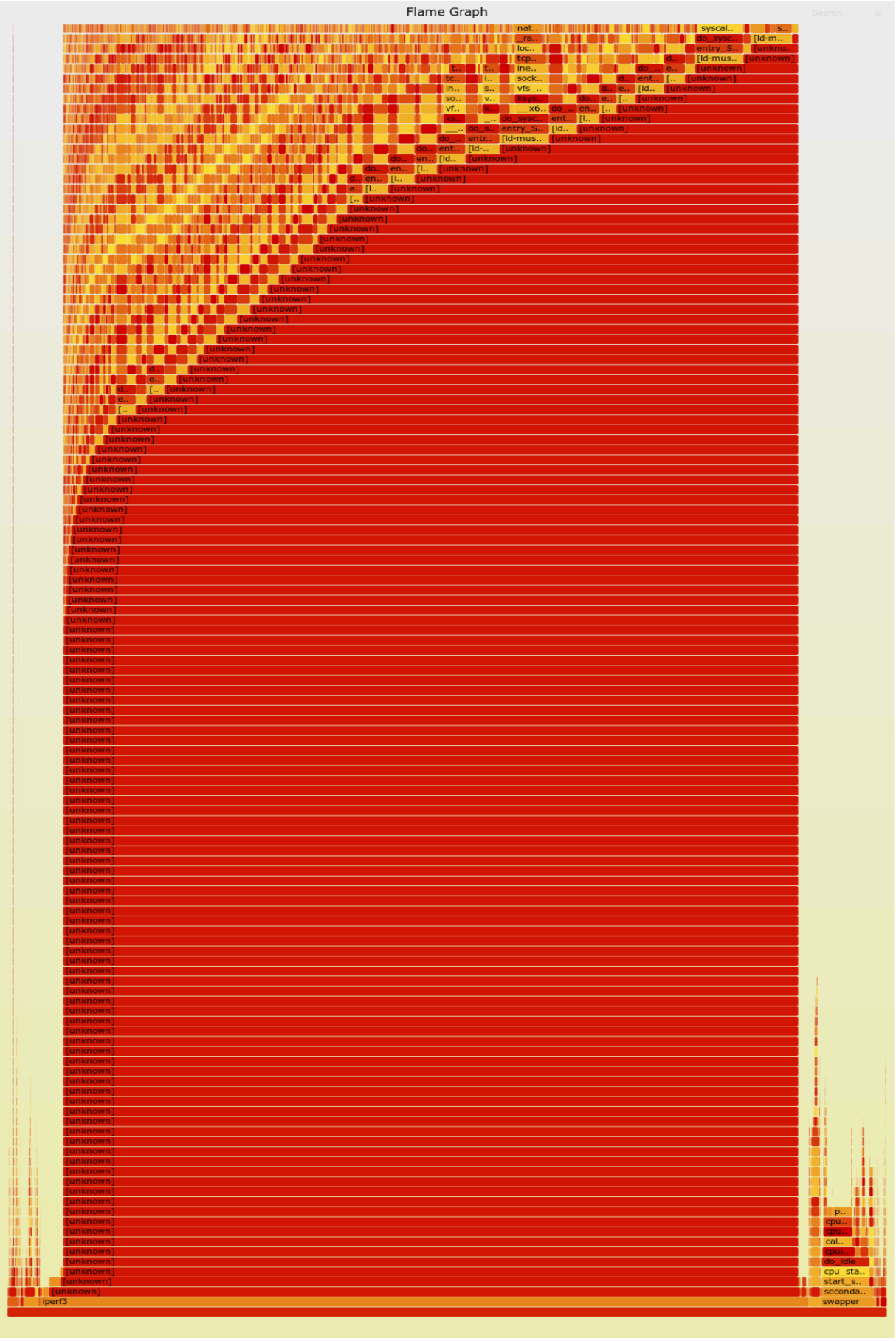

Figure 4.15 : Without Mirror Kernel profiler FlameGraph

The explanation of Figures Figure 4.14 and Figure 4.15's observations is provided as follows.

- I saw clearly swapper function used more cpu time with mirror
- with mirror swapper function usage :24.1 %
- without mirror swapper function usage : 5.99%
- iperf3 is the most significant CPU consumer.

The Linux kernel's'swapper' process, sometimes referred to as process 0, is an essential component. When the system has no runnable processes, the scheduler executes the idle process. The primary function of the swapper is to control CPU idle time rather than, as its name may imply, "swap" processes in and out of memory. Therefore, we could not find a direct correlation between the swapper function and the observed performance decline. I made the decision to cease looking into the core reason for the 20% maximum performance loss because of the time constraints on my research assignment.

Test 2 summary

I observed a 20% drop in Atmos performance at a packet size of 1024 bytes, which is a far smaller performance drop compared to what the research papers explained in the literature review. Therefore, the test OVS Docker container network setup can be considered a good network setup. However, I have tried to find the root cause of the 20% performance drop. Despite continuing testing after a certain level.

4.2.4 Test Three

Since the mirror traffic bandwidth was around 30 Gbit/s, there were doubts about mirroring traffic outside the host machine. To test this, the GRE tunnel was used to forward traffic to another machine.

GRE Tunnel

A GRE (Generic Routing Encapsulation) tunnel is a protocol used to encapsulate a wide variety of network layer protocols inside virtual point-to-point links over an Internet Protocol network. Developed by Cisco Systems, GRE allows for the encapsulation of packets from a wide range of network protocols, making it versatile for various kinds of data transport.(“What is GRE tunneling?,” 2023)

Figure 4.16 shows the setup. We built a virtual network that runs on host 1. VM1 is connected to VM2 over the OVS bridge. VP1 and VP2 are VNIs that belong to VM1 and VM2,

respectively.

All the traffic came from the VP1 port mirror to the ETH0 port. Using the GRE tunnel, forward all the traffic to host 2 ETH1. In this setup, we forward mirror traffic to another physical host. This is the main difference between test 2 and test 3. The following are the specs of the host computers, OVS bridge, and VMs.

- Host 1 machine configuration
    - Ram : 16 Gb
    - Processor : {Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz,Core Count: 4}
    - Only allow to use 2 and 3 cores to host machine (“isolcpus” cmd use)
    - Ubuntu 22.04 Operating System
- OVS bridge
    - OVS Version : 3.0.0
    - Run on core 0 (“taskset -c 0”cmd use)
    - Vp1 port all traffic (ingress and egress) mirror to mirror-output port
    - Inside OVS create a mirror vp1 mirrored to eth0
    - Create Gre Tunnel
- VM1(Docker Container)
    - This is created using a Docker container.
    - Vm1 run on core 0 (“taskset -c 0”cmd use)
    - Alpine Linux v3.18 Operating System
- VM2 (Docker Container)
    - This is created using a Docker container.
    - Vm2 run on core 1 (“taskset -c 1”cmd use)
    - Alpine Linux v3.18 Operating System
- Host 2 machine configuration
    - Ram : 16 Gb
    - Processor : {AMD Ryzen 5 4600H with Radeon Graphics 3.00 GHz}
    - Windows Operating System
- Monitor
    - Wireshark used to capture packets
- Network Interface Card
    - Host 1
        - Link Speed : 1Gbs

- Host 2
  - Link Speed : 1Gbs

Initially, create a virtual network using OVS. Then, create ports vp1 and vp2. Next, create Docker containers named vm1 and vm2, and connect them to the previously created virtual network. Assign appropriate IP addresses to each port, vp1 and vp2. After that, create a mirror and use a GRE tunnel with its destination set to the Eth1 NIC IP address. Finally, generate traffic from VM1 to VM2 using iperf3.

The command used to create a GRE tunnel

"ovs-vsctl add-port br0 gre0 -- set interface gre0 type=gre options:remote_ip=<IP of eth0 on host2> "

The command utilizes the Open vSwitch command-line utility, ovs-vsctl, to configure a Generic Routing Encapsulation (GRE) tunnel interface. It starts by adding a port named gre0 to the bridge br0. Following this, it specifies that the interface gre0 should be configured as a GRE tunnel interface using the set command. Additionally, it sets the remote IP address for the GRE tunnel to be the IP address of the eth0 interface on host2. This is achieved through the option options:remote_ip=<IP of eth0 on host2>. The double dash (--) separates the options for ovs-vsctl from those for the set command. This command is typically used in networking configurations to establish GRE tunnels between network devices.

Mirror traffic

Using an OVS mirror mirrored the vp1 port traffic to eth0 Port and created a Gre tunnel to Host 2

Monitoring

Using wireshark monitor the traffic

Traffic Generation

"iperf3 -c 192.168.1.22 -b 100M -t 60" CMD used to generate traffic and also used to find data rate transfer data size. also run this command on VM1. Using "**-b 100M**" can change the traffic.

Test case design

Data is transferred from VM1 to VM2 using iperf3 on host 1, and the mirrored data is measured on host 2 using Wireshark at different bandwidths (data can be sent using different bandwidths with iperf3). For test results, check Table 4.2.

Following, Figure 4.16 shows the high-level diagram of Test Setup 3.

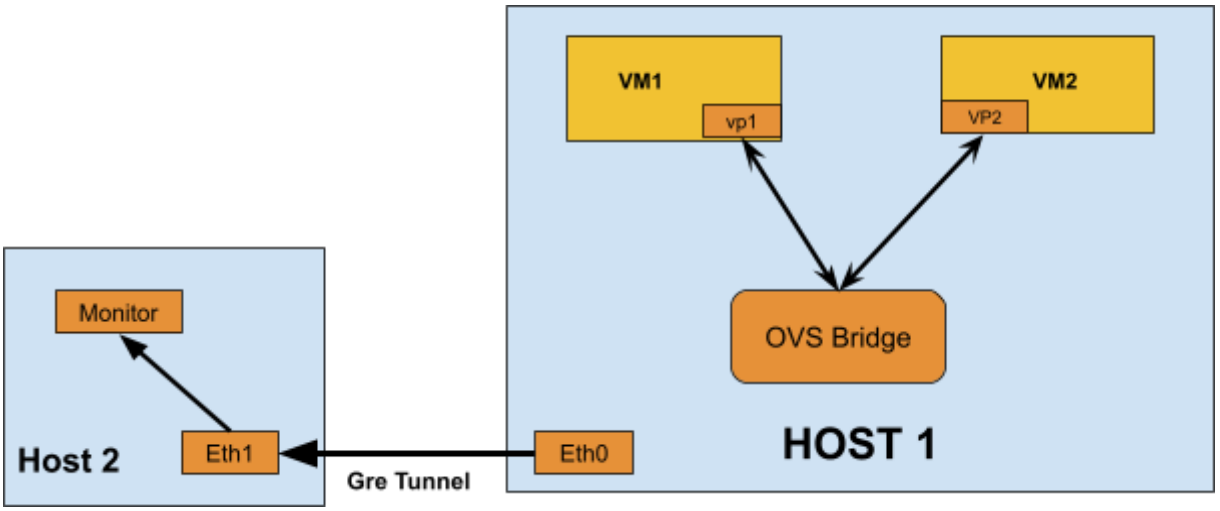

Figure 4.16 : Test Setup 3

Test No	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Transfer GBytes	0.715	1.4	2.1	2.79	3.49	4.19	4.89	5.59	6.29	6.98	7.68	8.38	9.08	9.78	10.5
Bitrate Mbits/Second	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
Wireshark Reserved Data GBytes	0.783	1.566	2.337	2.921	3.922	4.704	5.317	7.131	6.777	7.359	6.802	6.678	7.325	6.722	7.098

Table 4.2 : Test Setup 2 Results

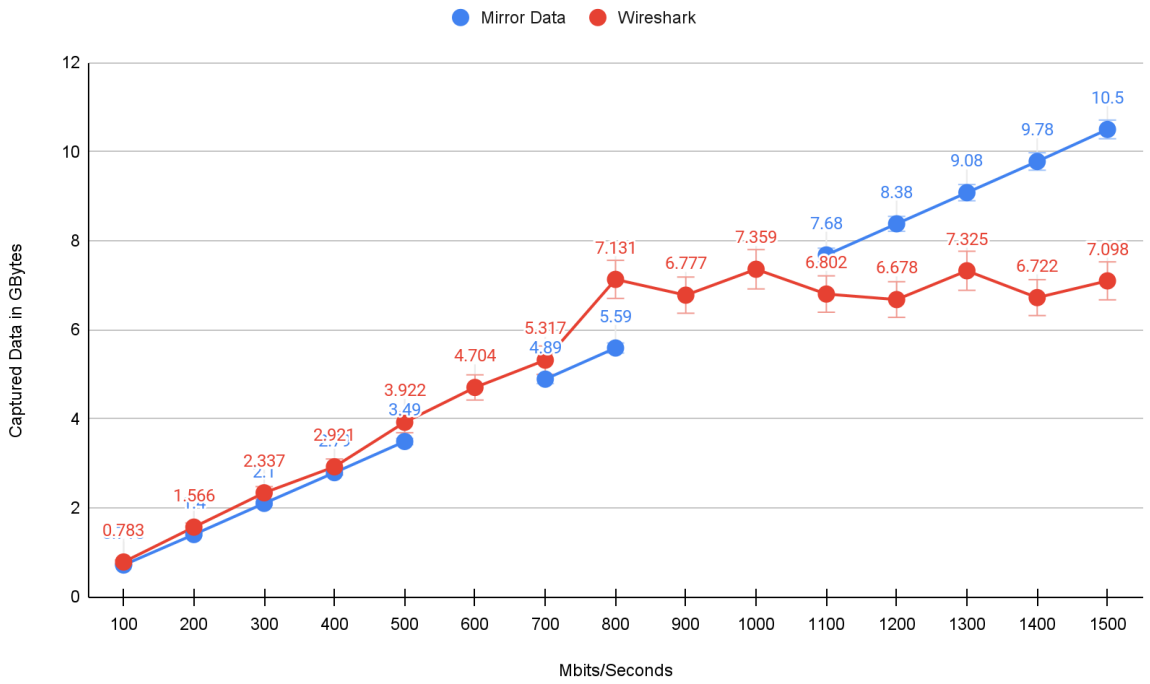

Figure 4.17 : Mirror Data vs Wireshark Data in each test case

Table 4.2 dataset capture using wireshark and iperf3 was also used to generate graphs in Figure 4.17 using the table data. Transfer GByte capture using an iperf3 report that runs on

VM1 Wireshark Reserved Data (Capture Data) gets from Wireshark that runs on Host 2. The result of Test Three is shown in Figure 4.17. The Wireshark Reserved Data also increases with the Mirror Data, but the trend is not as linear or smooth as the Mirror Data. when reaching the maximum data rate of the network interface card, 1000 Mbits/second.

Summary

When the data rate approaches 1000 Mbits/second, it reaches the maximum capacity of the network card of the hosts. Therefore, the bottleneck is the network interface card. If someone wants to monitor mirrored network traffic from outside the host machine, they have to consider preprocessing.

Overall Summary

In Test 1, an increase in the mirror count leads to higher LLC loads, LLC misses, and CPU branch activity. Additionally, a network performance drop of 2 Mbit/s (0.85%) is observed. However, since no considerable performance drop was observed with an increase in the mirror count, there appears to be no relationship between mirror count and network performance.

From Test 2, I observed a maximum performance drop of 20%, which is significantly smaller compared to the performance drops discussed in the research papers I have read. This test used Docker containers instead of virtual machines and recorded a maximum network performance of around 30 Gbit/s.

From Test 3, the network was able to communicate at a speed of 30 Gbit/s with a mirror, even with a performance drop of less than 20%. But NIC that supports only 1 Gbit/s, for our test setup, the main bottleneck is the NIC.

4.3 Proposed Solution

Even with a noticeable 20% performance drop in OVS with Docker containers' virtual network setup, which still maintains higher network bandwidth even after the performance drop, a 20% decrease is favorable. This is because, according to most research papers, they observed a 50% to 70% performance degradation. Therefore, I propose OVS with a Docker container network setup as the best network setup.

4.3.1 Propose Network Setup

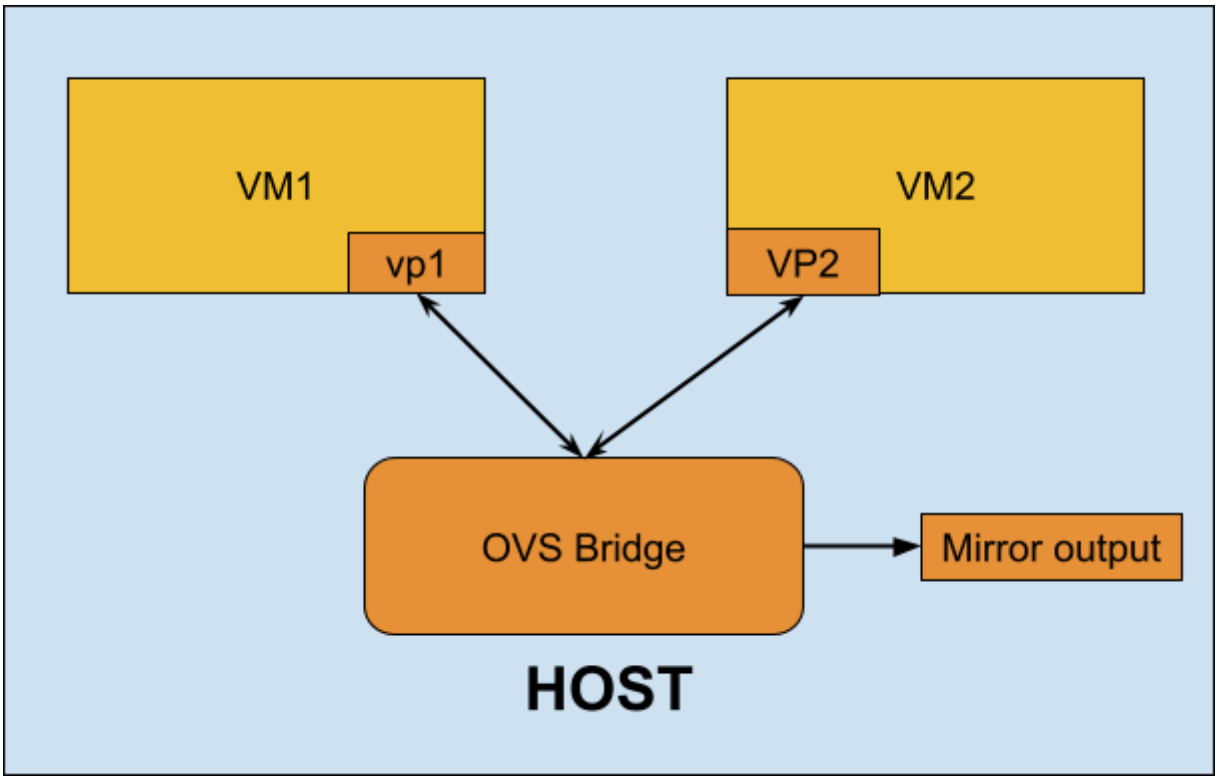

Figure 4.18 : Network Setup

Figure 4.18 above illustrates how the OVS virtual network connects two Docker containers. In this setup, the virtual network operates on a single host, with VM1 connected to VM2 via the OVS bridge. VP1 and VP2 are VNIs that belong to VM1 and VM2, respectively. All the traffic came from the VP1 port mirror to the mirror-output port.

- Host machine configuration
  - Ram : 16 Gb
  - Processor : {Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz,Core Count: 4}
  - Operating System Ubuntu 22.04.3 LTS
- OVS bridge
  - OVS Version : 3.0.0
  - Vp1 port all traffic (ingress and egress) mirror to mirror-output port
  - Inside OVS create a mirror
- VM1(Docker container)
  - This is created using a Docker container.
  - Alpine Linux v3.18 Operating System
- VM2 (Docker container)

- This is created using a Docker container.
- Alpine Linux v3.18 Operating System

4.3.2 Custom Tool

As previously explained, even mirroring traffic to route it outside the host is problematic due to NIC bottlenecks. The function of this custom tool is to perform some preprocessing and somehow reduce the filtered traffic.

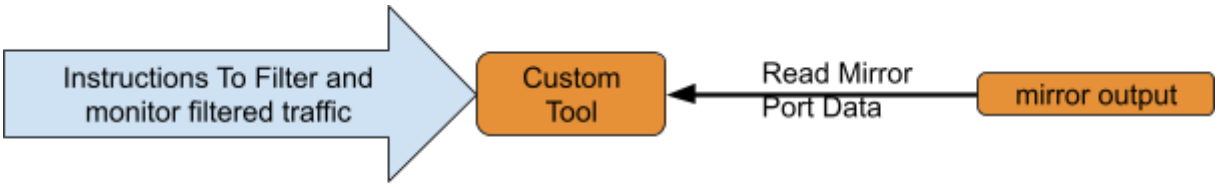

Figure 4.19 : Proposed Solution

Figure 4.19 explains a high-level diagram of the custom tool. From mirror output, take all the traffic from the mirror port, then filter the data using the given instructions, and the user of the tool can see filtered traffic through the UI.

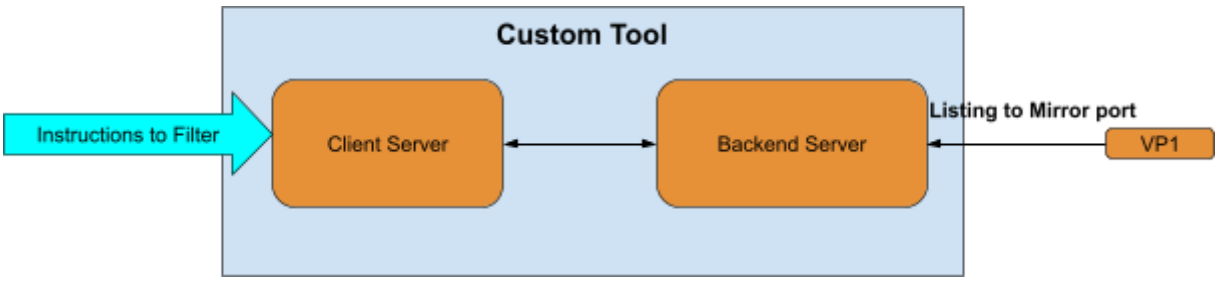

Figure 4.20 : Proposed Solution details view

Figure 4.20 provides a detailed view of the custom tool, which is built as a web service with both a backend and frontend. It is hosted on a machine that has a virtual network it monitors. The backend servers handle all processing tasks, including reading network data from any given port and filtering it. Users can view filtered data live by accessing the client server. For packet processing, I used the Tshark tool inside the backend server.

This is designed as a web service so that when a user provides instructions and conducts filtered traffic monitoring, both can be done from anywhere. This web service should be deployed on the host machine.

4.3.3 What is Tshark

TShark is a network protocol analyzer that enables capturing and displaying the contents of network packets. It is the command-line or terminal-based version of Wireshark, one of the most popular and powerful tools used for network analysis and troubleshooting. TShark is

part of the Wireshark suite and offers much of the same functionality but is designed for use in a non-graphical environment. (“tshark,” 2023)

Features and uses of TShark include:

Capturing Network Packets: TShark can capture live packet data from a network interface in real-time or read packets from a previously saved capture file.

Filtering: It allows users to apply capture and display filters to only show packets that match specific criteria. This is useful for isolating relevant data from a large set of network traffic.

Protocol Analysis: TShark can analyze packets for hundreds of protocols and can decode protocol information, making it easier to inspect the details of network communications.

Exporting Data: Users can export the output of TShark to various formats for further analysis, including text files, CSV, XML, or even the pcap format used by other network analysis tools.

Automation and Scripting: Because it is a command-line tool, TShark is well-suited for automation through scripts or batch processing, making it a powerful tool for network diagnostics, security analysis, and automated monitoring tasks.

TShark is used by network administrators, security professionals, developers, and anyone who needs to analyze network traffic or diagnose network issues. Its command-line interface makes it particularly useful for headless environments, remote analysis, or integration into automated processes.

4.3.4 Main Functions Proposed Custom Tool Support

Two primary functionalities were originally supported in the prototype.

1. Filter network traffic by source IP and destination IP.
2. Capture samples of data packets given a time interval.

Since only the two functions mentioned above were provided, a new feature was added that enables users to enter custom Tshark commands via the UI, allowing them to capture packets and view them live.

The proposed solution is a web server that has a frontend and a backend.

Frontend

Developed using React 18.2.

Main Functions

- Get the user cmd and send it to the back-end server.
- Display captured data packets.

Backend Server

The server is written using spring boot. Spring boot version 3.2.2

Main Functions

- Communicate with the client server.
- Communicate with the Tshark tool.

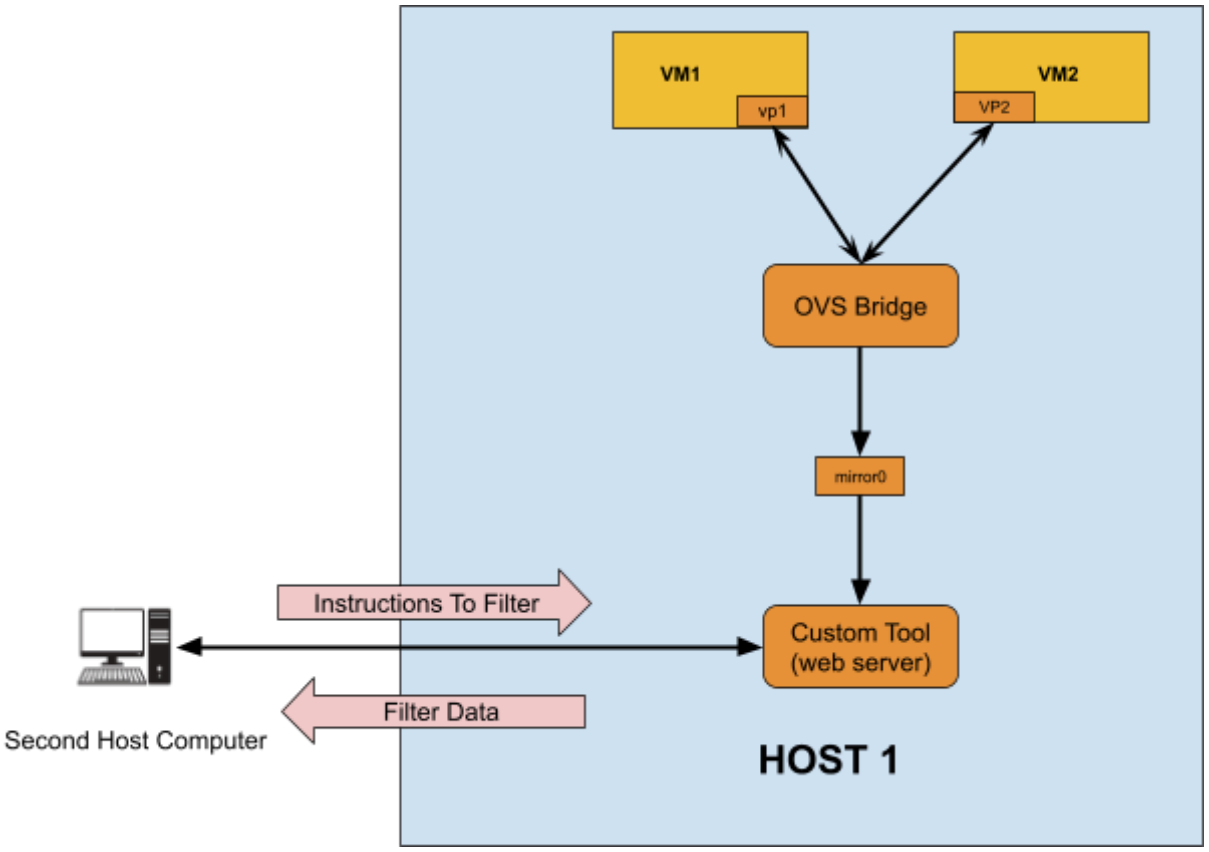

Figure 4.21 : Proposed Solution After Deployment

Figure 4.21 explains in detail how the work proposes a solution. Once it is deployed on.and how a user accesses the custom tool from a different machine.

4.3.5 Custom Tool UIs

The figures below display the UI for each main function of the custom tool.

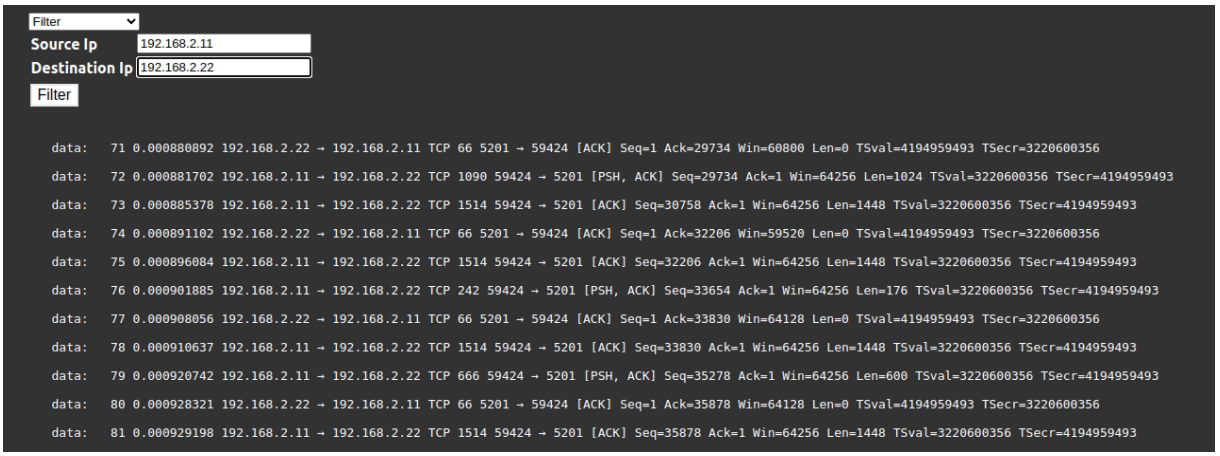

Figure 4.22 : Custom Tool UI : Filter

Figure 4.22 explains a customer UI for filtering traffic by source IP and destination IP.

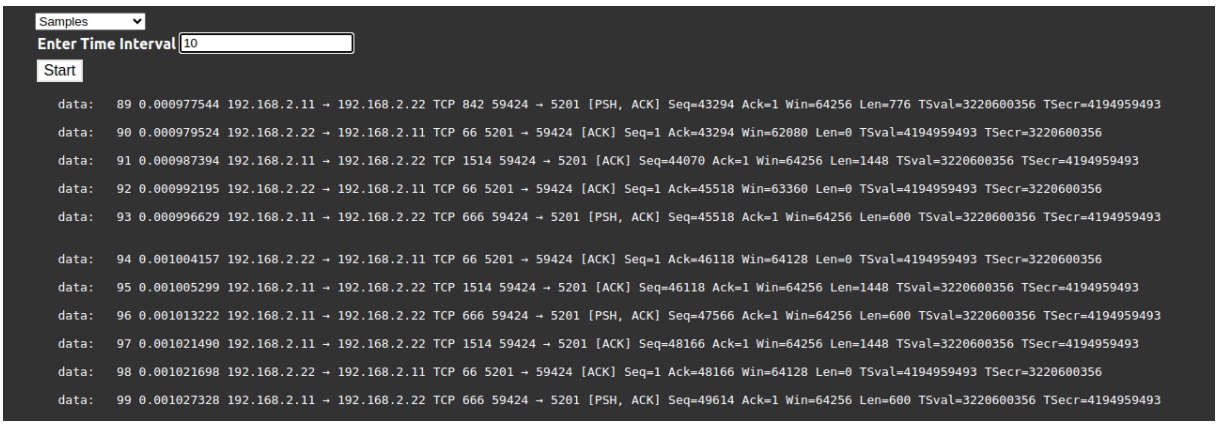

Figure 4.23 : Custom Tool UI : Capturing Samples

Figure 4.23 explains a customer UI for capturing network packet samples every 10 seconds.

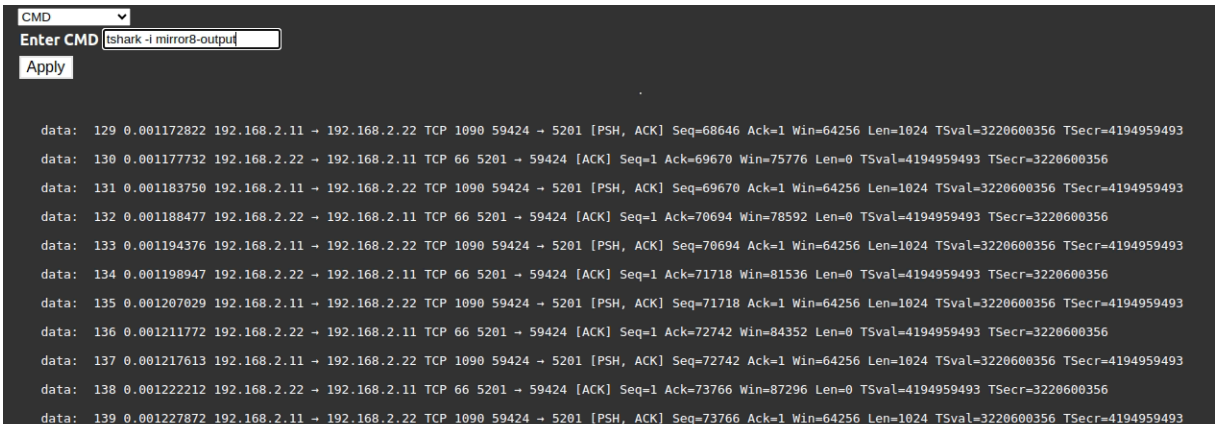

Figure 4.24 : Custom Tool UI : Capturing packet when given cmd

Figure 4.24 explains and displays a customer UI designed for capturing network packets using a custom tshark command.

4.3.6 Testing Custom Tool Functionality

In this section, we discuss how to evaluate custom tool functions.

1. Filter traffic from the SRC IP and the DST IP.
2. Capture sample data for the given interval.

4.3.6.1 Filter Traffic From Src ip and Dest Ip Function

In this test, monitor the filter function. Previously, we only tested with two Docker containers. In this test, we added one more Docker container, but the network setup is as explained in Test 2, so I am not going to repeat the same configuration process.

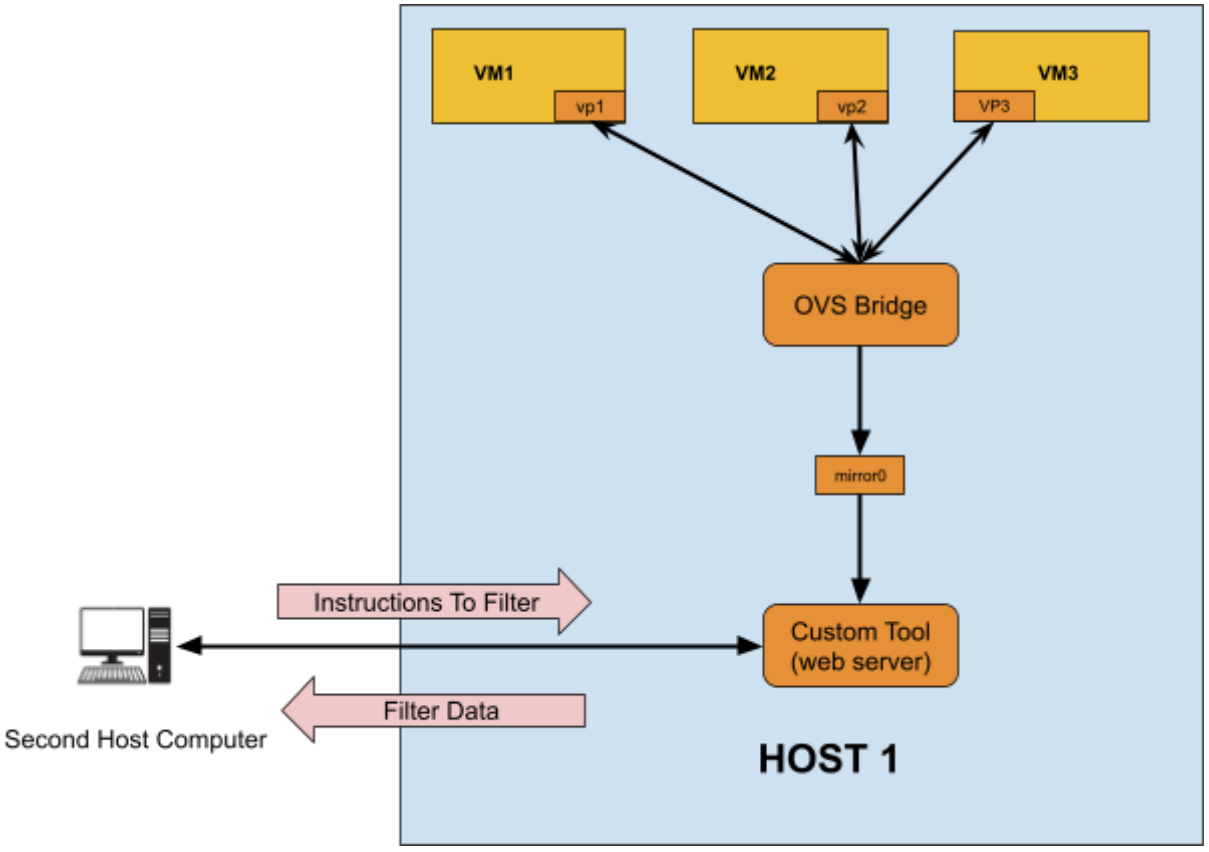

Figure 4.25 : Test Setup for Custom Tool

Figure 4.25 shows the test setup for filtering traffic from the Src IP and the Dest IP tests. Traffic was generated using iperf3 from VM1 to VM2 at a rate of 1 Gbit/s and we also created another traffic stream from VM3 to VM1 at the same rate of 1 Gbit/s. All VMs were created using Docker containers, similar to Test Two. In this setup, VP1's port is mirrored to another port called Mirror0 output. Assume you only need to monitor traffic from VM3 to VM1 (for this test).

Now, there are 2 traffic flows through VP1:

1. VM1 to VM2

2. VM3 to VM1

Since we only want traffic from VM3 to VM1, it's not necessary to capture the entire traffic. Using the tool, create a pcap file and then, using wireshark, analyze the captured packet. The user also see live filter packers on his UI, as explained in Figure 4.22.

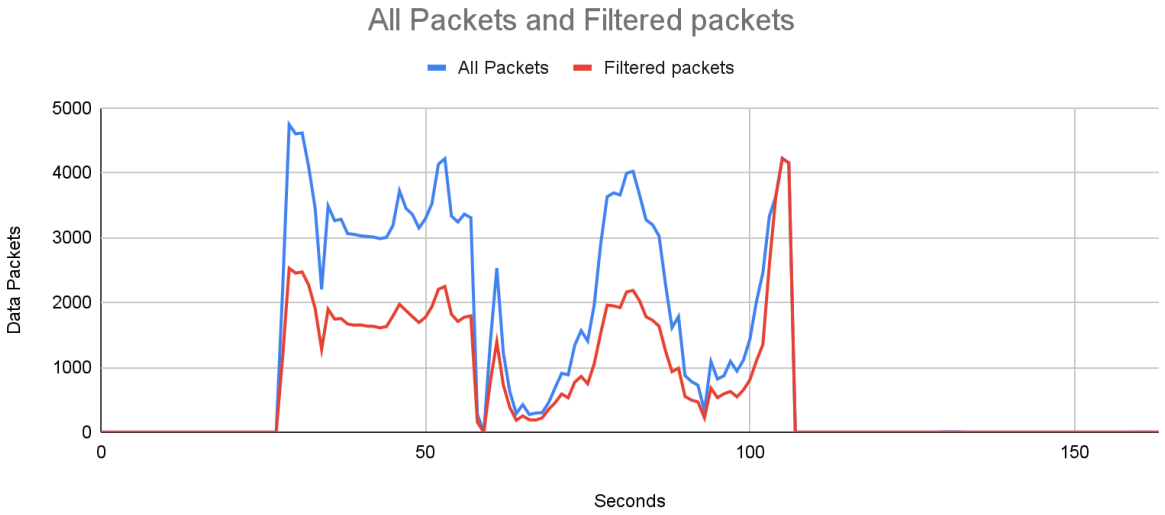

Figure 4.26 : Packet Filter by IP

Red Color : vm3 to vm1 traffic
Blue Color : vm3 to vm1 and vm1 to vm3 traffic

In Figure 4.26 The red color shows low traffic clearly. That is the traffic going out of the tool.

Since two 1 Gbit/s traffic streams are created over VP1, the total network traffic amounts to 2 Gbit/s. If only VM3 to VM1 traffic is required, there's no need to forward all traffic; it is an unnecessary resource allocation. Utilizing filtered packets (via a networking tool) can cut down 50%.

In this test, the main focus is to assess the tool's capability to filter traffic from the source IP and destination IP. According to this test setup, the custom tool can reduce 50% of the unnecessary traffic.

Since the Tshark tool is used inside the custom tool, there is a huge scope, if necessary, to extend the custom tool's functionality.

4.3.6.2 Capture Sample data Function (Every 10 seconds for this test, and this can change based on user requirements.)

The test setup used in Figure 4.21 was utilized to evaluate this functionality. There was

network traffic (1 Gbit/s for 30 seconds) from VM1 to VM2, and the traffic from the vp1 port was mirrored to the mirror0 port. The traffic from mirror0 was then captured by a custom tool for filtering. Additionally, Wireshark was used to determine the total number of packets sent through mirror0.

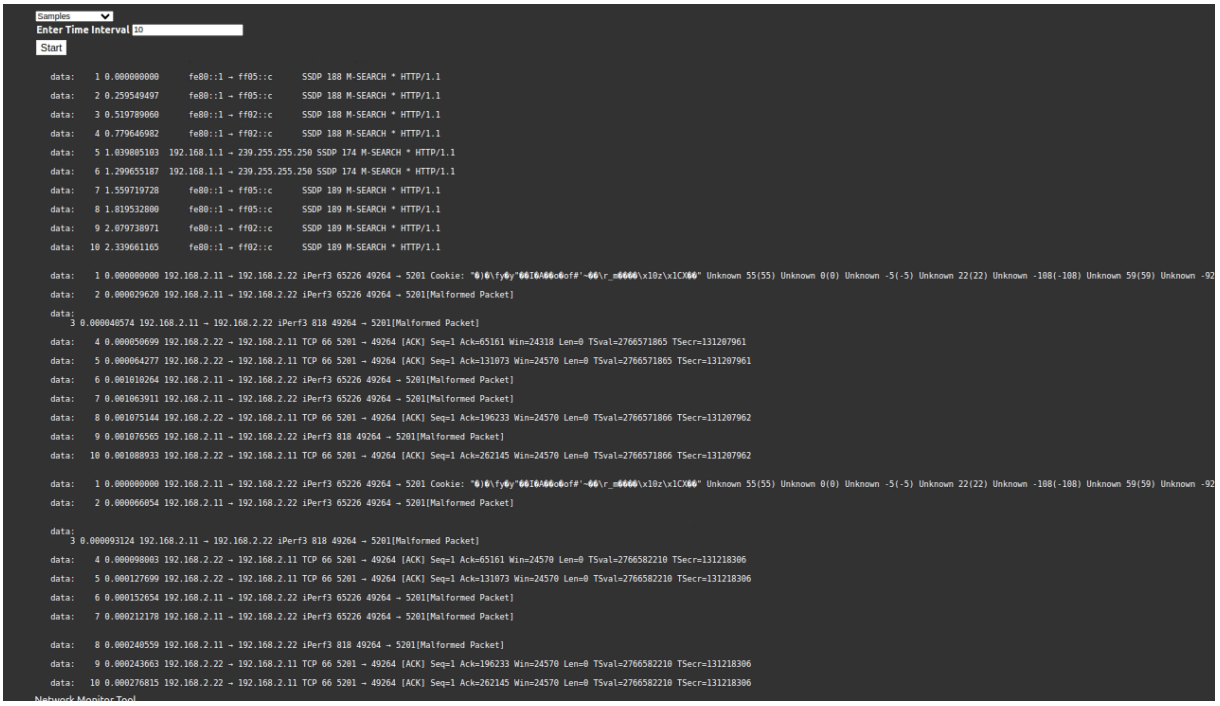

Figure 4.27 : Custom tool user perspective.

Every 10 seconds. 10 packets of samples are gathered and sent to the user. In that 30-second period, 143,739 packets reportedly passed through mirror0, but the custom tool only managed to capture 30 of them, representing a reduction of more than 99%. This rate can change according to user input. Figure 4.27 shows how the custom tools display captured packets.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This chapter presents the conclusions of the research project, reflections on the research problem, auxiliary findings, and directions for future work.

5.1 Reflection on the Research Questions

How can we reduce the performance reduction of virtual networks when mirroring traffic?

Even with a noticeable 20% performance drop in OVS with Docker containers' virtual network setup, it still maintains higher network bandwidth compared to other setups. This 20% decrease is favorable because, according to most research papers, a 50% to 70% performance degradation is typically observed (explained in section 2.8 Literature Review). Therefore, I propose using OVS with a Docker container network setup to reduce the performance reduction of virtual networks when mirroring traffic.

How to send mirrored data through a physical NIC without increasing the NIC's capacity?

Due to the higher network bandwidth of mirror traffic (20–30 Gbits/s), a normal NIC finds it hard to manage the traffic. Finding a NIC that supports that much traffic is difficult and very expensive. Therefore, I proposed a preprocessing tool that can further reduce the traffic according to the user's requirements.

Why is there a performance reduction in virtual networks when mirroring traffic?

Under Test 2, we tried to find the root cause. We noticed a performance reduction of up to 20%, which is far less than what has been observed in research papers. We attempted to identify the root cause of this maximum 20% performance drop using Vtune Profiler, the perf tool, and flame graph tools. However, due to time limitations, we couldn't find the root cause.

5.2 Significance of the Research

This research project evaluates the answers to the research problem defined in the research proposal, and during that process, the following contributions were made.

As a result of this research, two outcomes emerged: network setup and a prototype preprocessing tool.

Proposed an efficient network setup

OVS-Docker container network setup is the proposed network setup. According to the test results, this network setup has a maximum performance drop of only 20%. In the literature review, it was discussed that using a virtual TAP in a network with OVS can lead to performance degradation by up to 70%.

Prototype of a solution

The prototype is a custom tool designed to filter network traffic, allowing the monitoring of network traffic without forwarding the entire traffic outside of the machine. Even with the proposed network setup, which incurs a maximum performance drop, there is still network traffic exceeding 23 Gbit/s. Without tools like this, it is harder to monitor traffic from outside the host machine.

5.3 Future Work

We noticed a performance reduction of up to 20%, which is far less than what has been observed in research

papers. An attempt was made to identify the root cause of this maximum 20% performance drop using Vtune Profiler, the perf tool, and flame graph tools. However, due to time limitations, further investigation must be considered as future work, as discussed in the literature review. Most researchers have attempted to address this issue by combining OVS with DPDK. According to the results of this research, there is a chance to further reduce the performance drop if OVS, DPDK, and Docker containers are used together; therefore, this is also considered future work.

REFERENCES

Bird, J., 2023. Software Defined Networking (SDNs) - Benefits, Challenges & Applications. Deliv. Soc. URL https://deliveredsocial.com/software-defined-networking-sdns-benefits-challenges-applications/ (accessed 3.2.24).

Difference between Full Virtualization and Paravirtualization in Operating System - javatpoint [WWW Document], 2023. . www.javatpoint.com. URL https://www.javatpoint.com/full-virtualization-vs-paravirtualization-in-operating-system (accessed 3.3.24).

Docker Compose overview [WWW Document],. . Docker Doc. URL https://docs.docker.com/compose/ (accessed 3.3.24).

Docker Engine overview [WWW Document], 100AD. . Docker Doc. URL https://docs.docker.com/engine/ (accessed 3.3.24).

Docker vs. VM (Virtual Machine): Key Differences You Need to Know | Simplilearn [WWW Document], 2023. . Simplilearn.com. URL https://www.simplilearn.com/tutorials/docker-tutorial/docker-vs-virtual-machine (accessed 3.3.24).

Jeong, S., Lee, D., Li, J., Hong, J.W.-K., 2018. OpenFlow-based virtual TAP using open vSwitch and DPDK, in: NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium. Presented at the NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, pp. 1–9. https://doi.org/10.1109/NOMS.2018.8406232

Kourtis, M.-A., Xilouris, G., Riccobene, V., McGrath, M.J., Petralia, G., Koumaras, H., Gardikis, G., Liberal, F., 2015. Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration, in: 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN). Presented at the 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), pp. 74–78. https://doi.org/10.1109/NFV-SDN.2015.7387409

Overview — Data Plane Development Kit 24.03.0-rc1 documentation [WWW Document], 2023. URL https://doc.dpdk.org/guides/prog_guide/overview.html (accessed 3.3.24).

Overview of Docker Hub [WWW Document], 800. . Docker Doc. URL https://docs.docker.com/docker-hub/ (accessed 3.3.24).

Shanmugalingam, S., Ksentini, A., Bertin, P., 2016. DPDK Open vSwitch performance validation with mirroring feature, in: 2016 23rd International Conference on Telecommunications (ICT). Presented at the 2016 23rd International Conference on Telecommunications (ICT), pp. 1–6. https://doi.org/10.1109/ICT.2016.7500387

Strydom, M., 2024. Why Are Virtual Machines So Slow? - Computer Info Bits [WWW Document]. URL https://computerinfobits.com/why-are-virtual-machines-so-slow/ (accessed 3.7.24).

tshark [WWW Document], 2023. URL https://www.wireshark.org/docs/man-pages/tshark.html (accessed 3.6.24).

Wang, L.-M., Miskell, T., Fu, P., Liang, C., Verplanke, E., 2020. OVS-DPDK Port Mirroring via NIC Offloading, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. Presented at the NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, pp. 1–2. https://doi.org/10.1109/NOMS47738.2020.9110293

What is Docker? | IBM [WWW Document], 2023. URL https://www.ibm.com/topics/docker (accessed 3.3.24).

What is GRE tunneling? | How GRE protocol works [WWW Document], 2023. . Cloudflare. URL https://www.cloudflare.com/learning/network-layer/what-is-gre-tunneling/ (accessed 3.6.24).
What is Network Troubleshooting? - The Ultimate Survival Guide [WWW Document], 2023. . Obkio. URL https://obkio.com/blog/network-performance-monitoring-network-troubleshooting/ (accessed 3.2.24).
What Is Open vSwitch? — Open vSwitch 3.3.90 documentation [WWW Document], 2023. URL https://docs.openvswitch.org/en/latest/intro/what-is-ovs/#overview (accessed 3.3.24).
Why is My Virtual Machine so Slow (5 Tips to Speed up), 2022. URL https://www.softwarehow.com/virtual-machine-slow/ (accessed 3.7.24).