POLITECNICO DI TORINO

Master Degree
in COMMUNICATIONS AND COMPUTER NETWORKS ENGINEERING

Master Degree Thesis

COMPUTATION OVERHEAD DUE TO NETWORKING IN VIRTUALIZED ENVIROMENTS

Supervisors
prof. Paolo GIACCONE
prof. Leonardo LINGUAGLOSSA

Candidate
Loredana LOGRUOSSO

Academic Year 2023-2024

Abstract

Virtualization has become an essential component of modern applications, providing flexibility and scalability at a manageable cost. This thesis examines network performance in virtualized Linux environments, with particular focus on the impact of CPU allocation on achievable throughput and the overhead introduced by system and virtualization layers. By analyzing different communication scenarios between hosts and virtual machines, the study explores how resource constraints affect network performance. The collected data are further examined through a targeted analysis of CPU behavior and workload distribution across both host and virtualized contexts.

The main objective is to quantify overhead in terms of CPU utilization and network performance, identifying conditions under which performance parameters remain within predictable ranges and revealing any deviations that might impact throughput. This analysis helps identify potential bottlenecks that arise under different configurations and serves as a foundation for developing advanced scheduling strategies in containerized and orchestrated environments. For applications that require guaranteed data transfer levels as part of specific Quality of Service (QoS) requirements, the findings provide guidance for scheduling policies that take into account CPU load, node-specific characteristics, and workload demands. In environments such as Kubernetes, this work can support the development of intelligent scheduling policies that optimize resource placement and allocation through informed decisions on CPU affinity, node selection, and replication strategies. In doing so, this thesis offers a systematic methodology and valuable insights for improving network performance in virtualized systems, contributing to effective resource management in modern, performance-driven network infrastructures.

Contents

Chapter 1

Introduction

The increasing reliance on virtualized environments for running networked applications presents unique challenges, particularly in understanding how CPU resource allocation impacts network performance. Virtualization offers scalability and flexibility, making it an essential component of cloud computing and containerized infrastructures. However, it introduces an inherent computational overhead that can degrade network throughput and efficiency, especially when processes compete for shared resources within the host system and across virtual machines (VMs).

This thesis investigates the interplay between CPU consumption and network throughput in virtualized Linux environments. It seeks to identify bottlenecks in the network stack and evaluate how these vary under different configurations. The research explores how virtualization impacts communication efficiency and offers insights into optimizing resource allocation to enhance performance.

The central issues addressed in this study are twofold:

- Assessing how CPU allocation influences network throughput in virtualized settings.
- Pinpointing specific areas of overhead within the network stack that negatively affect performance.

These challenges are particularly critical in scenarios demanding stringent Quality of Service (QoS) guarantees, where stable and predictable network performance is essential. Real-time applications rely on consistent throughput to maintain optimal performance and meet service-level requirements. By investigating the complexities inherent to virtualized environments, this thesis aims to provide insights for optimizing resource allocation and workload distribution, particularly in high-throughput use cases.

This research adopts an experimental approach, conducting detailed analyses across various configurations to uncover the relationship between CPU

cycles consumed by communication processes and the resulting network throughput. By directly correlating these factors, the study provides valuable insights into the efficiency of intra-server communication within virtualized systems. These findings aim to inform strategies for resource scheduling and workload distribution, particularly for optimizing high-throughput applications. Additionally, the research highlights opportunities to minimize overhead, enhance scalability, and improve the overall efficiency of virtualized network infrastructures, laying a foundation for future advancements in performance optimization under diverse workloads.

With this objective in mind, the study investigates how CPU usage is distributed across the processes involved in network communication. By analyzing the allocation of CPU cycles between transmission and reception tasks, it identifies critical points where resource contention occurs. Special attention is given to the function chains within the Linux network stack, where specific patterns and behaviors emerge based on system configurations. Both TCP and UDP protocols are examined to understand how different communication models impact performance. The research highlights how TCP's flow control mechanisms affect CPU resource consumption and throughput, while the stateless nature of UDP presents distinct challenges related to packet loss and data reliability. Parameters such as CPU limits, datagram sizes, and buffer configurations are varied to observe their impact on system behavior, enabling a comprehensive analysis of how processing resources influence network performance in virtualized environments.

Chapter 2

Fundamentals

To effectively analyze the virtualization overhead introduced by intra-server communication between virtual machines (VMs), it is essential to first understand the underlying technologies and system components that facilitate this communication. A foundational understanding of the following key areas is necessary:

- **Linux Network Stack**: Understanding intra-server communication between VMs requires a thorough look into the Linux network stack. This section will focus on how data packets are managed at the kernel level, traversing various levels of abstraction and network interfaces.
- **Virtualization and Hypervisor**: The concept of virtualization is introduce, with a specific focus on the role of the hypervisor. In this context, *QEMU* is explored, the open-source hypervisor used for system virtualization, describing its core functions and role in managing virtual resources.

2.1 Linux Network Stack

In the Linux networking stack, Figure 2.1, network functionality is divided into layers that closely correspond to the OSI model. However, Linux organizes these components specifically for kernel and user-space operations. One of the core design principles in Linux networking is the separation between kernel space and user space. Kernel space manages low-level networking operations, such as packet routing, network interfaces, and protocol processing. This structure enhances both security and performance, as user-space applications can access networking capabilities without interfering with critical system-level processes[1].

- **Applications:** User space is where applications run, such as web browsers, email clients, and networking tools (e.g. `ping`, `curl` or `iperf3`). These

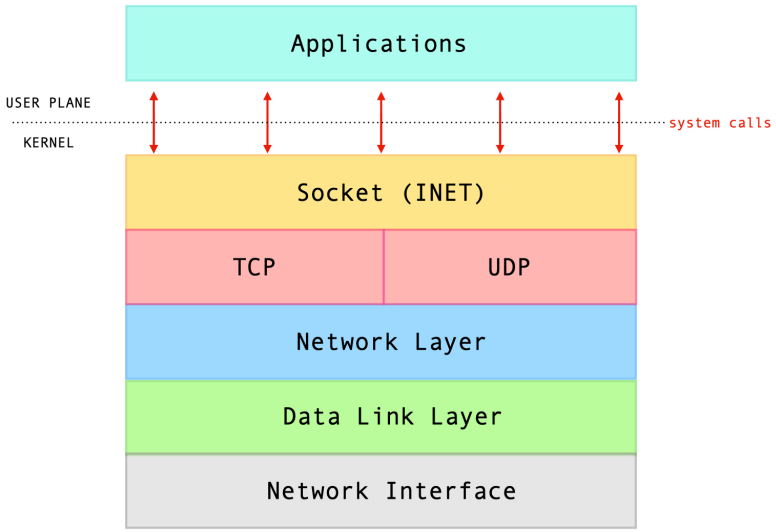

Figure 2.1: Representation of the Linux networking stack, highlighting the separation between user space and kernel space

applications interact with the kernel's networking components through system calls, using sockets as their interface to the network stack. Since applications operate in user space, any errors in these programs do not compromise the kernel, thereby ensuring system stability.

The kernel is responsible for handling core networking operations in the Linux stack, with a structure that aligns with traditional networking models. The layers in the kernel space include:

- **Socket (INET) Layer:** This is the first point of contact for user-space applications, allowing them to establish network connections. The socket interface provides an endpoint defined by an IP address and port, supporting both TCP and UDP connections.

- **Transport Layer (TCP/UDP):** The kernel processes data according to the protocol specified by the socket. TCP (Transmission Control Protocol) provides reliable, connection-oriented communication, while UDP (User Datagram Protocol) offers faster, connectionless data transfer. This layer ensures that data is organized and passed to the correct applications.

- **Network Layer (IP):** Within the Linux kernel lies the IP stack, responsible for all network layer functions, including IP protocols. This stack handles packet routing, determining the direction and destination of each packet based on routing tables and network rules.

- **Physical and Data Link Layer:** In Linux, the physical and data link layers are managed through network interfaces, both physical and virtual, which establish the connection between the system and the network. Devices such as Ethernet cards, Wi-Fi adapters, and the loopback interface (lo) are accessed through specific drivers that communicate directly with the hardware or virtual interface.

2.1.1 Socket Buffer in Linux

One of the core data structures used for managing network traffic within the kernel is the **socket buffer** (`sk_buff`). The `sk_buff` is employed to store packets temporarily as they pass through the various layers of the network stack, handling both packet data and metadata. This structure is fundamental to the operation of the kernel's networking components and is utilized throughout the transport, network, and interface layers .

The kernel manages incoming and outgoing network packets by associating them with `sk_buff` structures. These buffers hold the packet data and related metadata, such as pointers to the data, source, and destination information. Various network layers interact with the `sk_buff` to modify or forward packets efficiently as they traverse through the kernel. The `sk_buff` ensures efficient handling of packet data by allowing direct manipulation of pointers and enabling memory sharing between processes, which reduces the overhead of copying data. This modular approach contributes to both packet processing performance and scalability across the network stack [2].

2.2 The KVM-QEMU model

The KVM/QEMU model, shown in Figure 2.2, enables virtualization within Linux. KVM (Kernel-based Virtual Machine) provides virtual CPUs (vCPUs) to guest systems, mapped to the host's physical CPUs, using hardware-assisted virtualization for performance. While KVM executes guest code on these vCPUs, QEMU manages the virtual machine lifecycle, including process startup, memory allocation, I/O device emulation, and communication with KVM for process control [3].

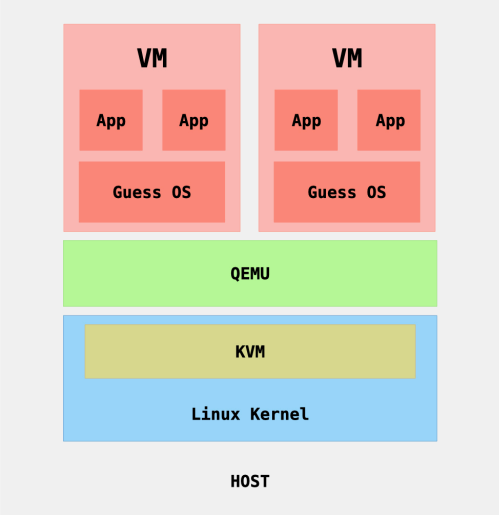

Figure 2.2: Overview of the KVM/QEMU virtualization model in Linux

2.2.1 NIC Emulation Architecture Inside Virtualized Systems

A virtual network setup within a virtualized environment is shown in Figure 2.3, where emulated network interface cards (NICs) are essential for enabling communication between virtual machines (VMs) and the host system. This setup relies on three primary components: the frontend managed by QEMU, the backend implemented via a TAP device on the host, and KVM (Kernel-based Virtual Machine), which ensures efficient execution of guest code and manages *vm exits* when necessary. In this configuration, QEMU acts as the frontend, emulating the virtual NIC and providing an interface that the guest operating system recognizes as a standard network interface. By simulating the NIC's registers and buffers, QEMU enables the VM to interact with the NIC as though it were a physical device. When the guest performs read or write operations on the emulated NIC's registers, it may trigger a *vm exits*, a context switch that transfers control to KVM on the host.

The backend is represented by a TAP device on the host, which serves as a bridge between the emulated NIC and the host's networking stack. The TAP device functions as a virtual network interface, facilitating the packet flow between the guest and the host's networking infrastructure. In this arrangement, the TAP driver creates a virtual bridge and a dedicated TAP interface for each VM, thereby enabling communication both externally and between VMs[4].

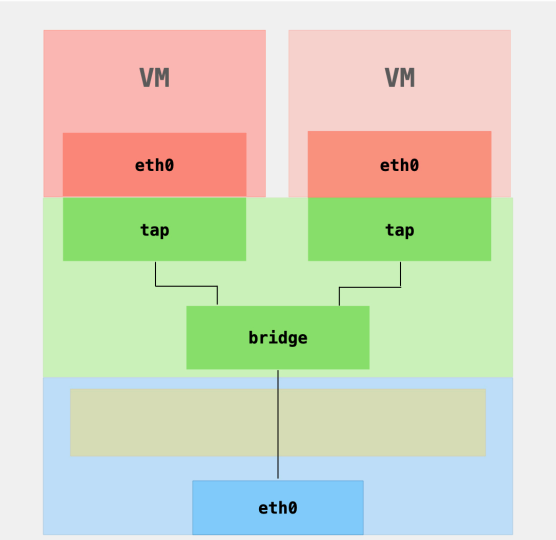

Figure 2.3: VM network connectivity via QEMU, TAP devices, and host bridge

KVM plays a crucial role in this setup by ensuring the efficient execution of guest code and managing VM exits. When the guest interacts with the NIC—for example, by writing data to a register—KVM detects the exit condition and calls on QEMU to handle the emulation in user space. Once QEMU completes the task, which may involve processing a transmission request, KVM resumes the guest's execution. This collaboration establishes an efficient emulation framework, where KVM handles low-level context switching and resource management, while QEMU performs higher-level emulation tasks [5].

2.2.2 Emulation of packets transmission

During packet transmission emulation via a virtual NIC, a structured sequence is triggered when a virtual machine (VM) sends data. The process begins with the VM preparing data for transmission, which triggers an event notifying the host of data ready to be sent. This event initiates a *vm exits*, transferring control to the KVM module on the host Figure 2.4. At this point, KVM invokes QEMU to manage the emulation in user space.

With control in user space, QEMU accesses the data in the VM's memory, translating the VM's memory addresses into host virtual addresses and packaging the data into network frames. These frames are then sent to the TAP backend, where the backend functions like a physical NIC, forwarding the packets to the host's networking stack.

The packets can then be routed either to another VM on the same host or

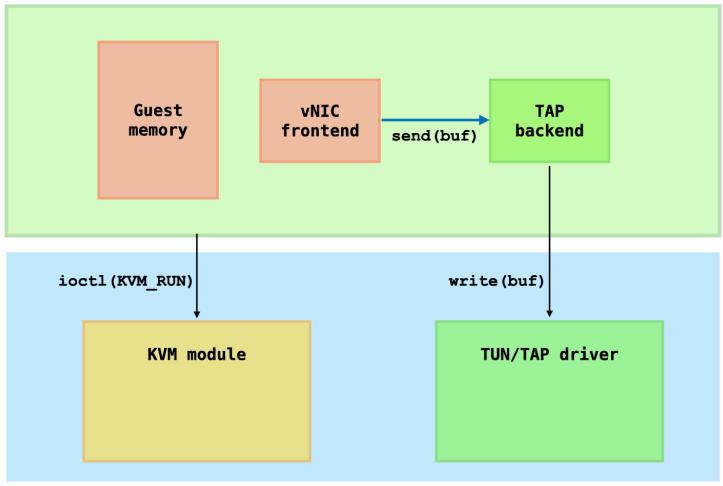

Figure 2.4: Transmission process: inter-working of QEMU and KVM

to an external network via a physical NIC, Figure 2.5.

After the transmission, QEMU signals the guest OS that the operation is complete by emulating an interrupt. This notification may cause another VM exit, allowing KVM to process the event and resume the guest's execution.

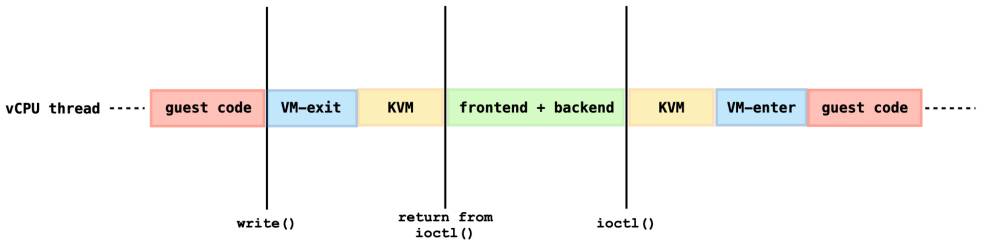

Figure 2.5: Trasmission Pipeline

2.2.3 Emulation of packets reception

In a virtualized network setup the process of receiving packets on an emulated NIC begins at the TAP interface on the host system. Incoming packets are buffered at this TAP interface until QEMU's I/O thread retrieves them for further processing, as shown in Figure 2.6. At this initial stage, KVM is not involved, since the I/O operations proceed asynchronously, independent of the guest VM's execution cycle.

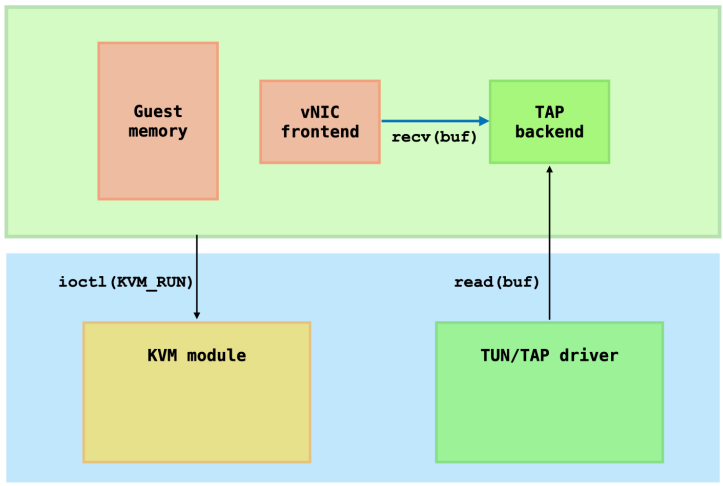

Figure 2.6: Reception process: inter-working of QEMU and KVM

Following the arrival of packets at the TAP interface, the TAP device reads the buffered data and alerts QEMU's I/O thread. QEMU then pulls this packet data from the TAP interface, processing it for compatibility with the guest's virtual NIC, as illustrated in Figure 2.6. This step includes translating the packet structure and filling the receive buffers within the guest VM's memory, making the data accessible to the guest OS.

Once processed, QEMU transfers the packet data into the designated buffers for the guest's NIC, marking it as ready for the guest OS to handle. At this point, QEMU issues a receive (RX) interrupt to notify the guest that new data has arrived. This RX interrupt triggers a VM exit, allowing KVM to manage the event and signal the guest OS, as detailed in Figure 2.7.

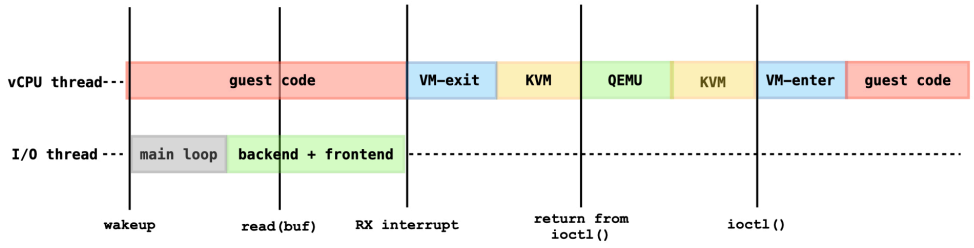

Figure 2.7: Reception Pipeline

2.2.4 Interrupt Management

The New API (NAPI) framework in Linux networking is designed to improve the efficiency of handling high volumes of network packets. During typical network operation, Network Interface Card (NIC) drivers use interrupt routines to respond to incoming packets by triggering an interrupt request (IRQ). However, excessive interrupts can cause system delays and degrade performance, as the CPU is forced to handle frequent context switches. To address this, NAPI reduces interrupt load by deferring heavy processing tasks to a kernel thread through the use of software interrupts (softirqs). When an interrupt occurs, the NIC driver initially disables further interrupts and instead schedules a poll callback via NAPI, which allows the kernel to process packets in batches, reducing the need for constant interrupts.

This polling mechanism, where the system temporarily switches from interrupt-driven to poll-driven processing, optimizes throughput and lowers the frequency of high-overhead interrupt handling. Specifically, each NAPI poll cycle is limited by a limited number of packets to process in a single run, which prevents any single NIC from monopolizing CPU resources. Once the processing budget is met, the NAPI thread releases control, allowing fair scheduling across multiple devices. If more packets are pending after the number is exhausted, the poll is rescheduled to avoid latency accumulation. As illustrated Figure 2.8, NAPI's adaptive switching between interrupt and polling modes balances processing efficiency with low receive latency, ensuring stable network performance even under heavy load [6].

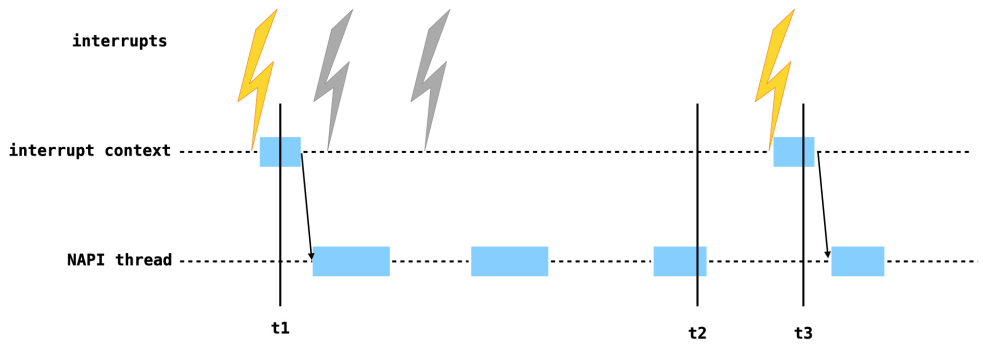

Figure 2.8: NAPI's interrupt and polling mechanisms

Chapter 3

System Design and Implementation

The design and implementation of a system model are presented, aimed at investigating CPU overhead in network communication, with a focus on understanding the relationship between CPU performance metrics and data transfer rates measured using `iperf3`. The study uses `iperf3` to assess network throughput and employs the Linux `perf` tool for a detailed analysis of CPU performance during data transfer.

The primary objective is to explore the correlation between CPU usage and network throughput by capturing key CPU metrics, such as **CPU cycles** and **CPU clock**. By analyzing how these metrics impact throughput under different network conditions, the approach uncovers insights into communication performance and highlights CPU inefficiencies that may hinder data transfer.

The system model and environment setup are also outlined, detailing the tools and methods used for analysis, including `iperf3` for network performance testing, and `perf stat` and `perf record` for monitoring CPU performance. These tools are utilized to measure CPU overhead and network throughput, providing a comprehensive understanding of their interrelationship.

3.1 System Model and Environment Setup

This section presents the system model and testing environment, detailing the hardware and software specifications of the primary machine used for experiments. Table 3.1 lists the key parameters of the test system, including the CPU model and core count, RAM, and the operating system version.

Parameter	**Specification**
CPU Model	Intel(R) Core(TM) i5-4278U CPU @ 2.60GHz
Number of CPU Cores	4
RAM	8 GB
Operating System	Ubuntu 24.04
Virtualization Tool	Multipass/QEMU
Kernel Version	6.8.0-48-generic
CPU Architecture	x86_64
Power Mode	Performance Mode

Table 3.1: System Specifications for Testing Environment

To simulate realistic network conditions, a virtualized environment is created and managed using **Multipass**, which leverages the **QEMU** driver. Multipass is a lightweight VM manager that allows for easy creation and management of Ubuntu virtual machines on various platforms, including Linux, macOS, and Windows. It is particularly useful for setting up isolated environments for testing and experimentation without requiring complex virtualization setups [7]. By using Multipass, the impact of virtualization overhead on CPU performance can be analyzed by comparing virtualized and non-virtualized environments.

3.1.1 Multipass Set-Up for Virtual Machines

To create a virtual machine (VM) with custom specifications using Multipass, the command 3.2 can be used.

```
multipass launch -n VM1 -c 1 -m 4G -d 10G
```

Listing 3.1: Command to launch a VM in Multipass

In the command 3.2, the `-c` parameter specifies the number of virtual CPUs allocated to the VM, the `-m` parameter defines the memory, and the `-d` parameter sets the disk size. If none options about the specific OS to install is specified, multipass automatically installs the latest Ubuntu image in the VM.

To modify the parameters of the VM after it has been created, the commands listed in 3.2 can be used.

```
$ multipass set local.VM1.cpus=2

$ multipass set local.VM1.memory=8G

$ multipass set local.VM1.disk=20G
```

Listing 3.2: Multipass commands usefull to modify the virtual machine set up

These commands allow adjustments to the number of CPU cores (`local.VM1.cpus`), memory allocation (`local.VM1.memory`), and disk size (`local.VM1.disk`), enabling customization of the virtual machine's configuration. Before applying any changes, the VM must be stopped. After the modifications are made, the VM can be restarted without data loss.

3.2 Iperf3 for Network Performance Testing

The `iperf3` tool is utilized in this study to measure network throughput across different test environments. `iperf3` is a popular open-source network testing tool designed to measure bandwidth between two endpoints over both TCP and UDP protocols. It operates as a client-server application, where one system runs the server process and the other runs the client process. The client initiates the communication. If the `-R` option is not specified, the client process transmits data while the server receives it. The tool supports a variety of options for controlling test behavior, including the ability to specify test duration, the number of parallel streams, and the desired data size [8].

Option	Description
`-s`, `--server`	Run `iperf3` in server mode
`-D`, `--daemon`	Run the server as a background daemon
`-p`	Specify the port to listen on (default is 5201)

Table 3.2: `iperf3` Server Options used in this thesis

Option	Description
`-c <host>`	Specify the server hostname or IP address
`-u`	Use UDP instead of TCP for testing
`-P n`	Run `n` parallel client streams
`-R`	Reverse the test direction (server sends to client)
`-l n[KMGT]`	Specify buffer length (default: 128KB for TCP)

Table 3.3: `iperf3` Client Options used in this thesis

3.3 Cpulimit as a CPU Usages Limiter

The `cpulimit` command-line tool is used to restrict the CPU usage of a specific process by sending `SIGSTOP` and `SIGCONT` signals to it, controlling the average CPU consumption over time [9]. This control can be useful in experimental setups involving `iperf3`, where limiting CPU usage on the trasmission side may be necessary to observe performance under constrained conditions.

The primary options for `cpulimit` used in these experiments is 3.3:

```
$ cpulimit -p <PID> -l <LIMIT>
```

Listing 3.3: Command to set CPU usage limit

where:

- `-p, --pid=N`: specifies the process ID of the target process.
- `-l, --limit=N`: sets the CPU usage limit as a percentage.

3.4 Performance Monitoring Tools

Often referred to as `perf_events`, `perf` is a powerful performance analysis tool in Linux. Accessed from the command line, it offers a variety of subcommands for statistical profiling of both kernel and userland code. It supports hardware performance counters, tracepoints, software performance counters, and dynamic probes, making it versatile for performance monitoring across the entire system [10]. Table 3.4 provides a summary of the commands used in the experiments to measure performance, enabling the correlation of bitrate with CPU usage.

Further in the details, the thesis focuses on `perf stat` and `perf record` (along with `perf report`), which are instrumental in assessing CPU performance. These tools capture essential CPU metrics, such as cycles, CPU frequency, and active CPU time during execution.

Command	Description
`perf stat`	Collects and displays performance counters, such as CPU cycles, instructions, and cache hits/misses.
`perf record`	Records events for later analysis, capturing event traces and statistics for post-execution review.
`perf report`	Analyzes recorded events, organizing results by process, function, or other criteria.

Table 3.4: Commonly Used `perf` Commands

3.4.1 `perf stat`

The `perf stat` command collects and displays performance counter data in real time, enabling immediate monitoring of specific events during application execution. Unlike `perf record`, `perf stat` updates event counters in real time, incrementing them each time a monitored event occurs in the target process. In these analysis, `perf stat` primarily monitors three events: 'cycles' and 'cpu-clock'.

- **cycles**: Reflects the total number of CPU cycles used by the CPU to execute the program.
- **cpu-clock**: Measures the total time (in milliseconds) that the CPU spends running the application.

3.4.2 `perf record` and `perf report`

The `perf record` command enables detailed event tracing by capturing performance data over a specified time period, generating a comprehensive log for post-execution analysis. Unlike `perf stat`, which provides instantaneous statistics, `perf record` samples events at regular intervals (default: 4 kHz), taking snapshots of the system state. If no specific event is selected, the default event `cycles` is used, offering a detailed view of application behavior over time. Due to its sampling nature, however, rapid or transient events might not be fully captured if they occur between sampling intervals, depending on the chosen frequency [11].

When the `call-graph` option is enabled, `perf record` also captures the call stack, allowing for a detailed reconstruction of the function call chain during the monitored process [12]. Once the recording is complete, the data is saved in the `perf.data` file, which can be analyzed using the `perf report` command. Through the call graph extension, it is possible to visualize the chain of functions traced during the execution and monitoring of an active process. Once the data is collected, `perf report` processes and organizes the recorded events, enabling a detailed analysis by function, process, or other criteria [13].

Chapter 4

Methodology and Objectives in Performance Analysis

Following the discussion of the theoretical concepts and tools, the methodology employed to achieve the performance analysis results is introduced. A detailed explanation of the experimental setup follows, including the tools and commands used to measure key performance indicators such as CPU usage, throughput, and network behavior. Particular emphasis is placed on the synchronization between machines involved in the experiments, which was critical for ensuring accurate and reliable data collection. Additionally, the chapter discusses the challenges and limitations encountered throughout the analysis, including the impact of system configurations and measurement precision that may have influenced the results. By addressing both the strengths and constraints of the approach, this section aims to provide a comprehensive understanding of the experimental process and its implications for the overall findings. Finally, this section explains the motivations behind this research and the objectives, which include identifying resource bottlenecks and understanding a possible virtualization setup to achieve a specific rate.

4.1 Data Collection and Analysis using `perf stat` and `perf record`

Using `perf stat` and `perf record` for performance analysis may initially appear redundant, as `perf record` alone could be considered sufficient. However, combining both tools provides a more comprehensive and accurate analysis due to differences in their data collection techniques.

`perf stat` is first employed to measure the CPU usage of both the `iperf3` client and server processes. By correlating the output of `iperf3`, which reports the communication bitrate, with the CPU usage statistics obtained from `perf stat`, it becomes possible to determine the CPU resources required to transmit a specific amount of data at a given rate. In contrast, `perf record` provides insight into how the processes operate by visualizing the hierarchy of function calls, showing how various functions interact to support the overall functionality of the process.

To clarify the rationale for using both `perf stat` and `perf record`, examples of monitoring an `iperf3` process receiving data through the loopback interface are presented in 4.1 and 4.2.

```
>> sudo perf stat -e cycles,cpu-clock -- iperf3 -c
   localhost -R

Performance counter stats for 'iperf3 -c localhost':

10,873,017,210   cycles                  #  2.814 GHz
3,864.46 msec    cpu-clock    #  0.772 CPUs utilized

5.004781651 seconds time elapsed
```

Listing 4.1: Output of the `perf stat` command

To clarify the rationale for using both `perf stat` and `perf record`, examples of monitoring an `iperf3` process receiving data through the loopback interface are presented 4.1, 4.2.

In this case, `perf stat` provides precise measurements related to *cycles* and *cpu-clock*, offering an overview of the total CPU usage of the process.

On the other hand, `perf report`, when sorted by the PID option, reveals the hierarchical structure of functions associated with each monitored process. In the example provided in 4.2, the `iperf3` process is attributed with 99.94% of the total recorded **cpu-clock events**. This percentage does not imply that the process occupies an entire CPU core; rather, it reflects that 99.94% of the sampled CPU cycles are associated with this process. Given that only this process is under analysis, such a high percentage is expected.

```
# Children      Self  Command  Shared Object       Symbol
# ........  ........  .......  .............  .......
#
    99.94%     0.00%  iperf3   [kernel.kallsyms]  [k]
       iperf3
    |
    ---entry_SYSCALL_64_after_hwframe
       do_syscall_64
       x64_sys_call
       |
       |--67.43%--__x64_sys_read
       |          ksys_read
       |          vfs_read
       |          sock_read_iter
       |          sock_recvmsg
       |          inet_recvmsg
       |          tcp_recvmsg
       |          tcp_recvmsg_locked
       |          |
       |          |+66.90%--skb_copy_datagram_iter
       |          |
       |           --0.53%--tcp_cleanup_rbuf
       |                     tcp_send_ack
       |                     __tcp_send_ack.part.0
       |                     __tcp_transmit_skb
       |                     ip_queue_xmit
       |                     __ip_queue_xmit
       |                     ip_local_out
       |                     ip_output
       |                     ip_finish_output
       |                     neigh_hh_output
       |                     __dev_queue_xmit
       |                     dev_hard_start_xmit
       |                     loopback_xmit
       |
        --32.51%-+__x64_sys_pselect6
```

Listing 4.2: Example output of the `perf report` command for the `iperf3` reception process using the TCP protocol

By examining the hierarchical breakdown of functions, it becomes possible to analyze the internal interactions within `iperf3`. Each function is represented with either an accumulated percentage, encompassing the overhead of the function and all its sub-functions, or a *self percentage %*, which reflects

the overhead attributable solely to the specific function. In this case, the *self* percentages are not displayed, as the sampling frequency was reduced to 1 kHz for illustrative purposes.

At lower sampling frequencies, functions with minimal CPU impact may not be fully captured, potentially leading to incomplete data on less resource-intensive operations. However, this does not compromise the overall analysis. The primary objective of using `perf stat` is to identify bottlenecks and throughput limitations in the traffic flow. Consequently, the omission of minor functions is inconsequential for detecting critical performance constraints.

It is important not to confuse the percentage reported by `perf record` with the *CPU clock percentage* shown by `perf stat`. In the examples provided by both tools, the same event was monitored in parallel. While `perf record` shows an overhead of 100%, `perf stat` reports a CPU clock percentage of 77%.

To estimate the CPU usage attributed to specific functions within the process, the *CPU clock percentage* from `perf stat` is used as a reference. This value is then scaled based on the overall CPU usage percentage for the monitored process. For instance, in 4.1, the CPU usage is reported as 0.7772 (77.72%). This percentage is then scaled by the value from `perf record`, which reports an overhead of 99.94%. The scaling factor is calculated as:

$$\text{Scaling Factor} = \frac{77.72}{99.94} \approx 0.77$$

This scaling factor is applied to the percentages of individual functions reported by `perf report`. For example, the function `skb_copy_datagram_iter` is reported as consuming 66.9% of the CPU usage in the `perf report` output. Applying the scaling factor:

$$\text{Adjusted Usage} = 66.9\% \times 0.77 \approx 52.02\%$$

By scaling the results, more consistent measurements across the two tools are obtained. This allows for a better understanding of the relative CPU consumption of each function in the context of the overall CPU usage reported by `perf stat`. The scaling ensures that a comparison between the CPU usage reported by `perf record` and the global values from `perf stat` is made, providing a more coherent and accurate analysis of the CPU usage at both the process and function levels.

4.2 Experiment Initialization and Data Transmission Setup

To conduct the performance experiments, two `iperf3` processes are employed on client and server sides.

Client Side In each test scenario, the client-side `iperf3` is initiated as 4.3. In this sequence, the client-side `iperf3` starts transmitting data, and its PID is saved immediately after initialization.

```
#START TRANSMIT DATA
>> iperf3 -c localhost -t 10 &

#SAVE IPERF3 PID
>> IPERF_CLIENT_PID=$!

#LIMIT CPU ON IPERF3 THROUGH ITS PID
>> cpulimit -l $cpu_limit -p $IPERF_CLIENT_PID &

#START PERF MEASUREMUENTS
>> sudo perf stat -p "$IPERF_CLIENT_PID" -e cpu-clock,
   cycles,context-switches -o "$PERF_CLIENT_FILE"
   --timeout 9500 &

>> PERF_STAT_CLIENT_PID=$!

>> sudo perf stat -p "$IPERF_SERVER_PID" -e cpu-clock,
   cycles,context-switches -o "$PERF_SERVER_FILE"
   --timeout 9500 &

>> PERF_STAT_SERVER_PID=$!

#WAIT UNTIL IPERF PROCESS IS OVER
>> wait "$IPERF_CLIENT_PID"

#IF THESE PROCESSES ARE STILL ACTIVE, TERMINATE THEM
>> kill -INT "$PERF_STAT_CLIENT_PID" || true
>> kill -INT "$PERF_STAT_SERVER_PID" || true
>> kill -TERM "$CPULIMIT_PID" || true
```

Listing 4.3: Command to set CPU limit through the iperf3 PID in a loopback network interface scenario

Following this, `cpulimit` is applied to restrict the client's CPU usage, and only after data transmission begins does the `perf` measurement process start.

The `perf stat` command starts after `cpulimit` to ensure that CPU measurements reflect the restricted state. To synchronize the processes, both `cpulimit` and `perf stat` are set to match the duration of `iperf3`. However, in practice, the duration of the `perf stat` measurements is slightly shorter than the data transmission duration. This decision is made to account for the inherent overhead between the execution of commands, as they are not executed in parallel at the exact same instant. Specifically, a timeout of 9500 ms for the `perf stat` measurements is chosen. This ensures that any idle time of processes, should they still be running after the transmission ends, is excluded from the measurements. By setting the measurement duration slightly shorter, the results avoid capturing any "dead time" or idle periods, minimizing noise in the data.

Once `iperf3` terminates, both `cpulimit` and `perf` (if `perf` is still alive) are stopped using the `kill` command.

Server Side The server runs in daemon mode, enabling it to manage incoming connections in the background. This setup allows the client mode to be activated independently for each experiment.

```
>> iperf3 -s -D -p 5201
```

Listing 4.4: wait until iperf3 stop its execution

From the server side, the 4.4 setup ensures that the server is always ready to accept data from the client, exposing its port for incoming connections. By running the server in the background, its process ID (PID) can be known in advance, allowing for the use of `perf` to monitor the server side even before the actual data transmission begins.

```
>> sudo lsof -i :5201

COMMAND PID   USER   TYPE  NODE   NAME
iperf3  832 iperf3   IPv6  TCP   *:5201 (LISTEN)
```

Listing 4.5: Checking the PID of the `iperf3` server

While the server side is idle, neither the cycles nor the CPU usage are reported by `perf` since there is no active execution. Indeed, after identifying the PID of the `iperf3` server (in this case, PID 832), `perf stat` can be used to track its resource consumption. However, since the server is in idle mode, the following output indicates that neither the cycles nor the `cpu-clock` events are counted:

```
>> sudo perf stat -e cycles,cpu-clock -p 832

Performance counter stats for process id '832':

   <not supported>      cycles
   <not counted> msec cpu-clock

    6.188331646 seconds time elapsed
```

Listing 4.6: Monitoring the idle server with `perf`

As shown 4.6, during the monitoring period of approximately 6.2 seconds, no cycles or CPU clock events are recorded because the process is not actively running.

4.3 Synchronization for the Experimental Environments

This thesis explores the performance of network communication by analyzing three distinct testing environments. The first involves loopback communication, which is tested directly on the host machine. The second scenario examines the loopback communication tested inside a virtual machine, and finally, the third setup focuses on communication between two separate VMs, each isolated from the other to provide a controlled and independent environment for testing. These configurations provide a comprehensive perspective on network performance across varying levels of abstraction and isolation.

4.3.1 Synchronization of Measurements in the Loopback Communication on the Host

For the loopback environment, as illustrated in Figure 4.1, the synchronization process is straightforward. Once the script that initiates communication and measurements is executed, the commands follow the order presented in the sreferenced cript 4.3. In this setup, no additional synchronization is required since all operations occur within the same system (host machine), ensuring that the measurements are effectively synchronized.

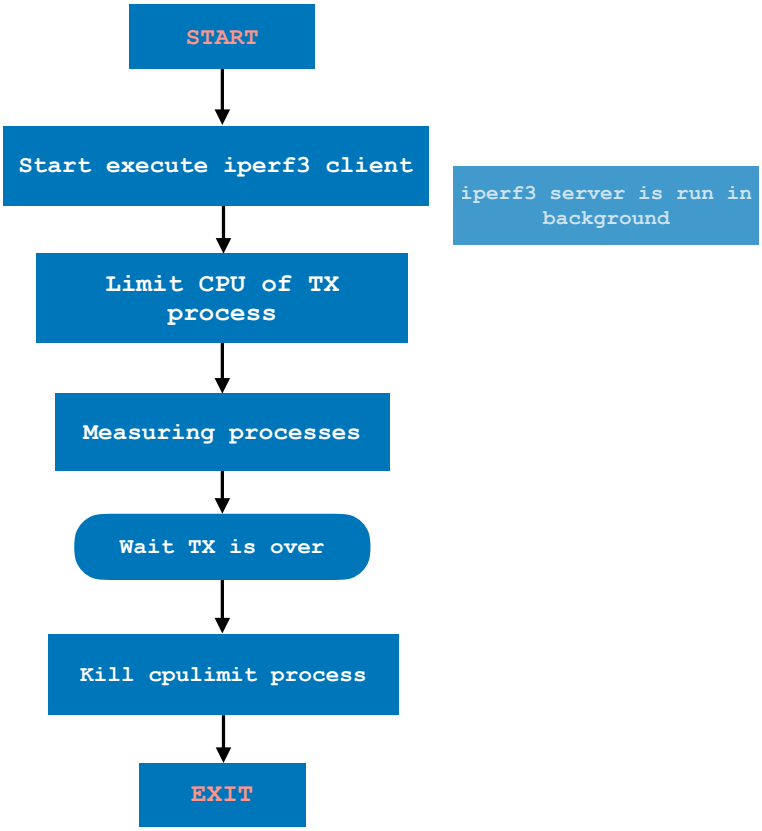

Figure 4.1: Flow of commands that are executed in sequence to ensure synchronized measurements during the communication process for loopback in host

4.4 Synchronization of Loopback Communication in Virtualized Environments

For the loopback scenario within a virtualized environment, synchronization between the host, which externally monitors the entire virtual machine process, and the virtual machine itself, where the loopback communication occurs via `iperf3`, is essential. As shown in Figure 4.2, the process begins on the host, which is responsible for initiating communication and conducting performance measurements. The host first enables the execution of the `iperf3` process inside the virtual machine using the command referenced in 4.7.

```
>> multipass exec VM -- ./start_script_iperf_TX &
>> VM_txProcee_ID=$!
```

Listing 4.7: example of starting multipass script able to start the communication with the host

This command runs the process in the background while saving the PID of the running program. Once initiated within the virtual machine, the host monitors it externally using `perf stat`. The host waits for the VM process to complete before concluding the entire tracking session. The execution flow inside the VM follows the same sequence as in the previous environment setup.

To ensure the measurements exclude idle time after the data transmission ends, all measurements are configured with a timeout of 9500 ms. Furthermore, the host script waits for the completion of the `iperf3` transmission process inside the VM by using the `wait $PID` command. The script only exits after the transmission process on the VM has fully terminated.

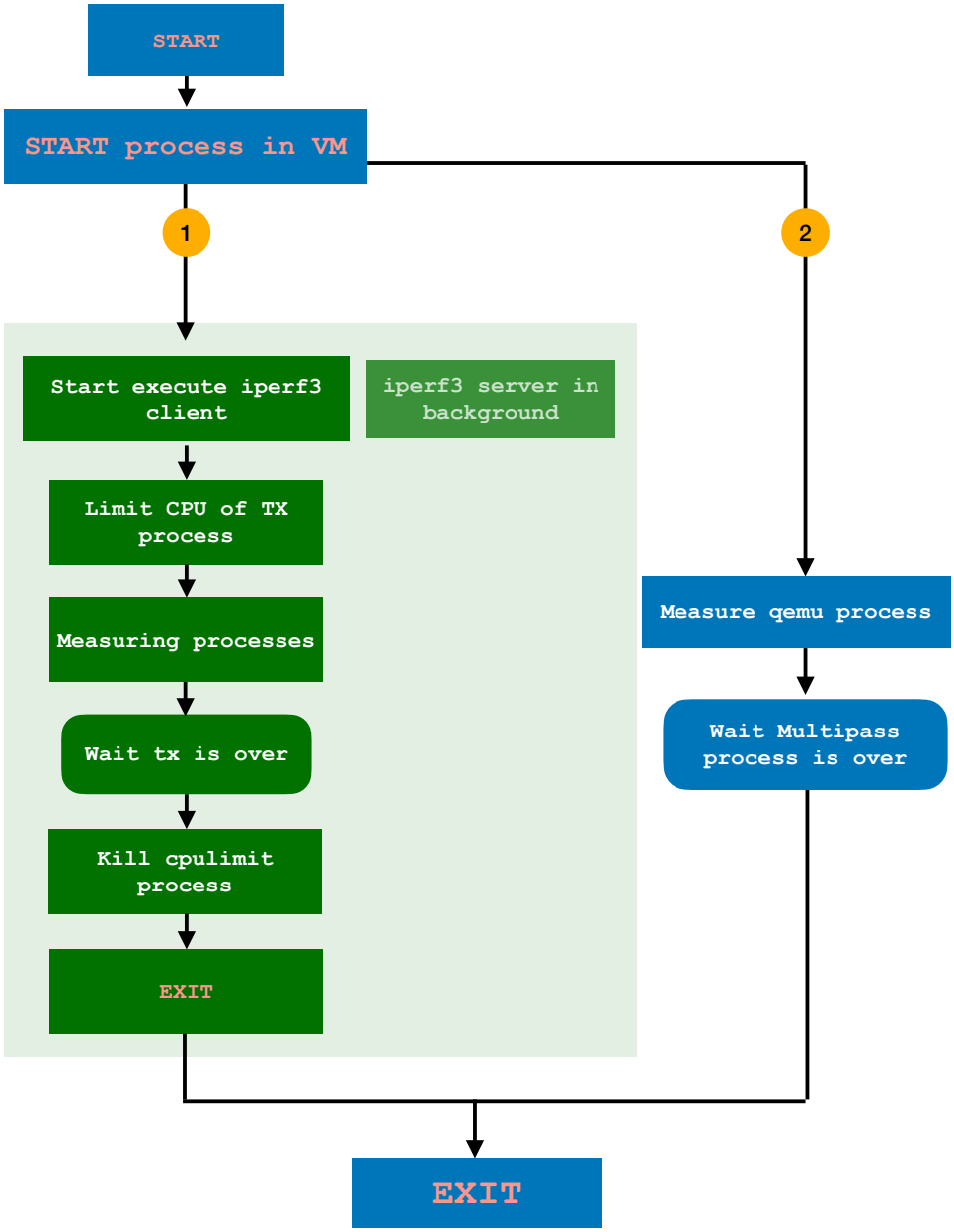

Figure 4.2: Flow of commands that are executed in sequence to ensure synchronized measurements during the communication process for loopback in virtualized environment

4.4.1 Synchronization of Communication Between Virtual Machines

For VM-to-VM communication,Figure 4.3, the host serves as a coordinator, even though it does not directly participate in the communication (acting neither as the client nor the server). The host ensures synchronization between the two VMs and monitors the `qemu` processes for both. The process begins with the host starting the transmission process on `VM1`, where the `iperf3` client is running. This step also involves applying `cpulimit` and initiating performance measurements. Next, the host initiates the measurement of the server-side process on `VM2`, which monitors the `iperf3` server daemon process. Both `VM1` and `VM2` run their respective operations with synchronized timeouts of 9500 ms. Each VM script exits automatically after completing its measurements. Simultaneously, the host measures the `qemu` processes for both VMs, allowing it to assess the virtualization overhead during the communication. The entire test concludes once the processes on both VMs terminate, ensuring all measurements are properly synchronized.

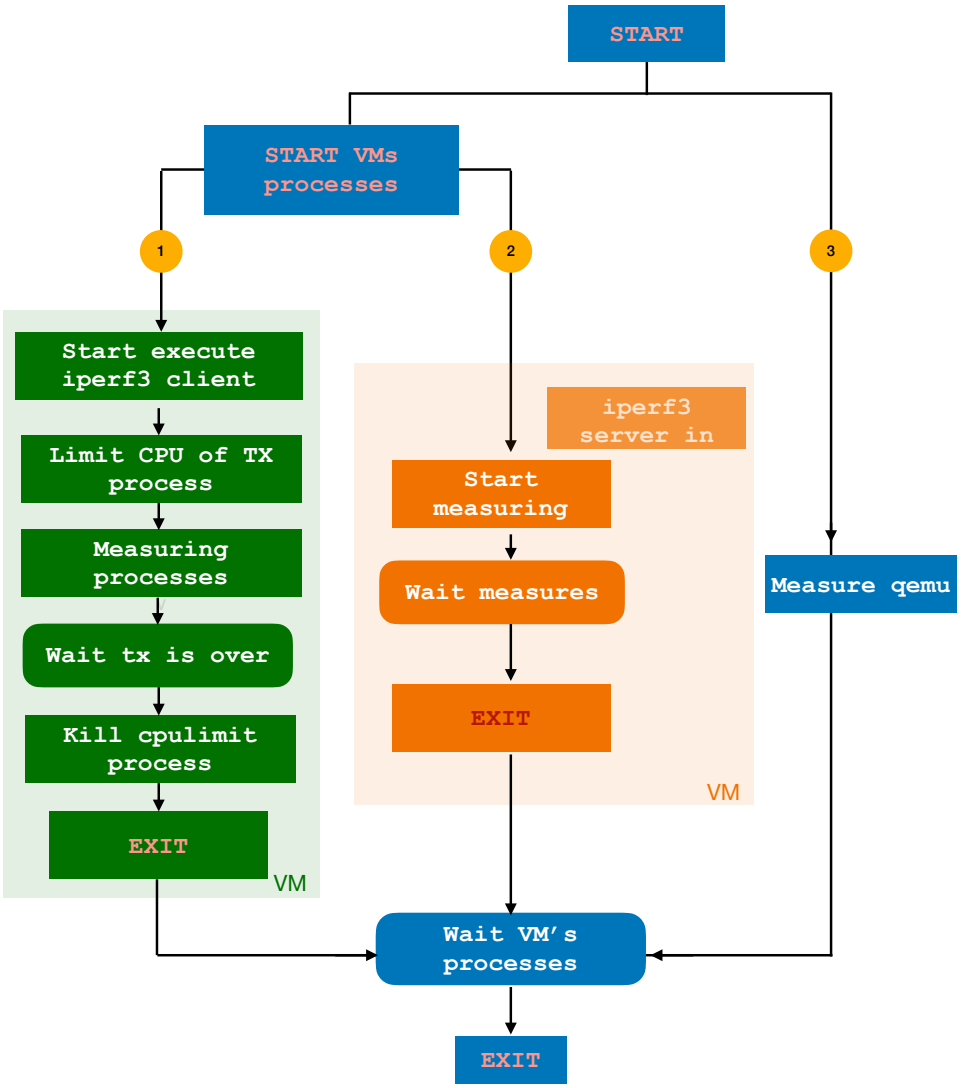

Figure 4.3: Flow of commands executed to ensure synchronized measurements during VM1-to-VM2 communication in a virtualized environment

4.5 Possible Noisy Measurements and Uncertainties

In conducting the experiments, the possibility of noisy measurements is acknowledged, arising from various factors, particularly since measurements are taken on both the host and the virtual machine (VM). Network conditions, resource contention, and background processes all introduce variability in the results. To mitigate the effects of noise, each experiment is repeated ten times, allowing for the calculation of an average of the measurements. Subsequently, a 95% confidence interval is computed to better assess the reliability of the results.

Additionally, it is important to consider the limitations imposed on CPU usage. The CPU limitation is applied after the execution of the `iperf3` command has commenced. This results in a transient phase where CPU consumption initially peaks before being constrained by the subsequent command. Given that each experiment lasts approximately 20 seconds, this transient behavior could skew the measurements if not adequately accounted for. The analysis ensures that this initial peak in CPU utilization is considered, focusing on steady-state performance once the CPU limitations are in effect.

Moreover, an additional limitation is the inability to use `perf record` for all experiments, which is required to calculate the average and confidence interval for the tracing to obtain the chain of functions of the processes. This is due to the `perf.data` log being in a format that can only be loaded via `perf report`, and once the log is converted into a text file, it becomes difficult to manage. The resulting format complicates the creation of a script to extract the data. Consequently, the values obtained from `perf report` are interpreted manually. Although this may not be considered an ideal engineering practice, this approach is adopted to perform an additional run of the experiments for post-tracing analysis of CPU usage. While this method is not entirely precise, it provides a general overview of CPU consumption and can offer insights for further research aimed at improving the feasibility of these data for more accurate analysis.

By systematically addressing these concerns, a comprehensive analysis of CPU overhead in network communication is presented. The consideration of measurement noise, transient states, and the limitations of performance tracing is crucial for interpreting the results accurately and drawing valid conclusions regarding the performance of different configurations and protocols.

4.6 Experimental Objectives

Building on the theoretical framework of networking in virtualized environments, this study investigates the utilization of the TAP interface to emulate network connectivity within virtualized environments, with QEMU and KVM serving as the hypervisors managing the virtual machines. Specifically, the research focuses on intra-server communication within Linux-based virtualized setups, examining interactions that occur entirely within the virtualized system, without involving physical network devices. The study explores the Linux network stack, particularly how virtualized networking interfaces, like TAP, integrate into the network communication processes.

The primary goal of this research is to quantitatively assess CPU utilization patterns in both native and virtualized environments, identifying how different virtualization configurations influence resource distribution across processes involved in network communication. Emphasis is placed on understanding the impact of network protocols, specifically TCP and UDP, on CPU overhead during intra-server communication. The role of packet size and its effect on CPU consumption is also examined, providing insights into how network traffic characteristics affect overall system performance.

Additionally, the study aims to estimate the minimum CPU resources required to achieve target throughput under varying network conditions, providing a basis for optimizing virtualized network performance. Through an analysis of CPU overhead, protocol efficiency, and resource consumption, the study aims to contribute to a deeper understanding of performance bottlenecks in virtualized networking, offering insights that can support the design and management of more efficient virtualized network environments.

Chapter 5

Analysis of Data Pathways and Network Stack Functions

The analysis focuses on constructing detailed schematics of the network stack across various test environments, tracing the path of data as it moves from the application layer through the socket layer, transport protocols (TCP/UDP), and the virtualization layers managed by QEMU and KVM, down to the TAP interface, and back to the application upon reception.

Using `perf report`, the study identifies and categorizes the key functions involved at each stage of transmission and reception, highlighting the interaction between user space and the kernel. This process emphasizes how virtualization impacts the flow of data through the network stack, with particular attention to how CPU cycles are distributed across different layers and functions. The definitions and details of these functions were sourced from the Linux Kernel documentation [14], the Linux Kernel Git Repository [15], and the Linux man pages [16], which provide in-depth explanations of kernel operations and system calls relevant to the study.

These schematics serve as a critical reference point for understanding how the network stack operates within each environment. By identifying potential bottlenecks and analyzing resource allocation, the diagrams provide a foundation for evaluating the performance constraints introduced by virtualization, offering insights into optimizing communication processes in intra-server scenarios.

5.1 Communication Over Loopback Interface

The loopback interface, used for communication between a client and server irrespective of the TCP or UDP protocol, is depicted in Figure 5.1. The diagram illustrates the fundamental setup where a client (transmitter) communicates with a server (receiver) via the loopback network interface, utilizing designated ports within the host system. Unlike traditional networking, which involves physical interfaces, communication over loopback bypasses external hardware, operating exclusively within the kernel's network stack.

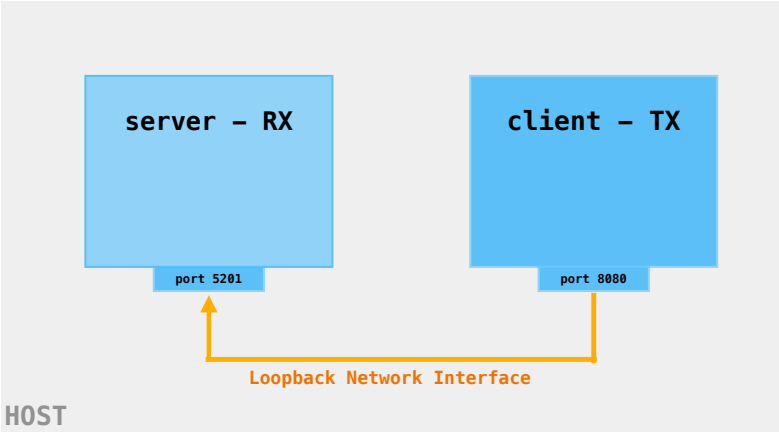

Figure 5.1: Client-server communication over the loopback interface.

This type of communication traverses the network stack of the host system. Although the implementation differs slightly between TCP and UDP protocols, they share many similarities in this context. The following subsections delve into the specific behavior of each protocol and their interaction with the network stack, accompanied by schematic representations and outputs from the `perf` tool. Note that detailed analysis of CPU cycle percentages for each function is deferred to the next chapter.

5.1.1 Communication Stack in TCP

TCP communication over the loopback interface provides a reliable, connection-oriented mechanism for transmitting and receiving data entirely within the host system. First, the process related to the transmission of data via `iperf` is explained, detailing how the data flows through the network stack, starting from the application layer and traversing through the TCP/IP layers until it reaches the loopback interface for transmission. Then, the reception process is analyzed, focusing on how the loopback interface processes the incoming data, including its interaction with the relevant network stack components.

Transmitter Process for TCP protocol

Through Figure 5.2[1], the following pipeline process is observed. The transmission begins at the application layer, where a user-space application like `iperf3` generates the data to be transmitted. Once created, this data is passed to the kernel via the `write()` system call, initiating a sequence of operations that include memory allocation, buffer management, and preparation for network transmission.

The data enters the transport layer through the `tcp_sendmsg_locked()` function, which acts as a critical point where multiple processes diverge. From this point, the kernel invokes `skb_do_copy_data_nocache()`, a function responsible for copying the data from user space into kernel-managed socket buffers (SKBs). This ensures the kernel gains exclusive control over the data during transmission, preventing any further modifications by the application. Concurrently, `sk_page_frag_refill()` manages the memory allocation for the SKB, guaranteeing there's sufficient space for the data as it traverses the network stack.

At this stage, the `tcp_push()` function is triggered, initiating the actual transmission process. This leads to a chain of calls, including `tcp_write_xmit()`, which prepares and pushes the data further down the stack. The data is subsequently passed to the IP layer through `ip_xmit`, which processes it before it is transmitted through the network interface.

Separately, the `release_sock` function ensures that the socket is properly released and its resources cleaned up. Other functions, such as `tcp_rcv_established`, `tcp_ack`, and `tcp_clean_rtx_queue`, are responsible for managing the reception and acknowledgment of packets. Meanwhile, `tcp_push_pending_frames` ensures that any pending data is pushed to the network.

[1]Note that the colors organizing the functions into blocks correspond to the colors representing the different network stacks, which are later shown in Figure 5.4

Regarding the loopback network interface, once the entire pipeline of the Linux network stack has been followed, the `dev_queue_xmit` function is invoked. Here, the loopback interface processes the packet, handling not only its transmission but also part of its reception. The loopback interface first processes the packet at the IP layer, then hands it over to the TCP layer, where it is treated as if it were an incoming packet. This dual role of the loopback interface highlights its unique position in managing both the transmission and reception within the local host.

```
iperf3
  |- Nwrite
  |- ksys_write
  |- vfs_write
  |- sock_write_iter
  |- inet_sendmsg
  |- tcp_sendmsg
      |
      |-tcp_sendmsg_locked
      |     - skb_do_copy_data_nocache
      |     - sk_page_frag_refill
      |     - tcp_push
      |       - tcp_write_xmit
      |           - tcp_transmit_skb
      |               - ip_xmit
      |                 - ip_output
      |                   - ip_finish_output
      |                     - neigh_hh_output
      |                       - __dev_queue_xmit
      |                         - __do_softirq - handle_softirqs
      |                           - net_rx_action
      |                            - __napi_poll
      |                             - netif_receive_skb
      |                              - ip_rcv
      |                               - ip_input
      |                                - ip_input_finish
      |                                 - ip_protocol_deliver_rcu
      |                                  - tcp_rcv
      |                                   - tcp_rcv_established
      |                                    - tcp_data_queue
      |                                     - tcp_data_ready
      |                                      - pollwake
      |- release_sock
                tcp_rcv_established
                tcp_ack
                tcp_clean_rtx_queue
                tcp_push_pending_frames
```

Figure 5.2: Function chain for the TCP transmitter process detected using `perf record`

Receiver Process for TCP Protocol

On the receiving side, as shown in Figure 5.3, the process begins when the kernel's network stack receives the looped-back packets, which are handled internally without involving external network interfaces. Once the loopback network interface completes the packet processing, it uses the `pollwake` function to wake up the receiver process, signaling that new packets are available. This triggers the scheduling process, invoking methods like `schedule()` and `sock_poll()` to begin the packet processing.

Figure 5.3: Function chain for the TCP transmitter process detected using `perf record`

At this point, the kernel performs integrity checks on the received packet to ensure that it matches the original data sent by the transmitter. The packet then enters the TCP layer, where the function `tcp_recvmsg()` is invoked to manage the reception. This function calls several sub-functions to ensure correct handling:

- `tcp_recvmsg_locked()`: This function handles the reception of data in a thread-safe manner by locking the socket buffer. It ensures that multiple threads can receive data concurrently without causing conflicts.
- `skb_copy_datagram_iter()`: This function copies the received data from the kernel buffer into user-space buffers, preparing the data for further processing by the receiving application.

Flow control is managed by the TCP protocol's acknowledgment mechanism. The function `tcp_send_ack()` sends an acknowledgment (ACK) back to the transmitter, indicating that the data has been successfully received. This acknowledgment process follows the same flow as the transmission process.

To ensure the data is received in the correct order, the TCP layer performs reassembly of incoming packets. The function `tcp_do_rcv()` handles the reassembly of fragmented packets, while `tcp_cleanup_rbuf()` ensures that any leftover buffers are cleared after processing.

After the data has been successfully reassembled and validated, it is copied from the kernel buffers to user space. The function `skb_copy_datagram_iter()` reads the data from the socket and transfers it to the application's user-space buffer. The application, such as `iperf3`, can now access and process the data. Simultaneously, system calls like `sys_read()` and `ksys_read()` provide the interface for the application to request the data, while `vfs_read()` interacts with the virtual file system to complete the read operation.

Integrated View of the Communication Stack in TCP

The entire TCP communication flow over the loopback interface, from transmission to reception, is illustrated in Figure 5.4. This diagram provides an integrated view of how data moves through the network stack, emphasizing critical components such as memory management, data segmentation, and flow control. The transmission process begins with the allocation and queuing of data for transmission. The loopback interface, operating as an internal network device, reroutes the data back to the kernel for immediate processing. Polling mechanisms and softirqs ensure that the packets are promptly delivered to the receiving application, maintaining low latency and high throughput. On the receiver side, data is validated, reassembled, and copied back to user space, where it becomes accessible to the application.

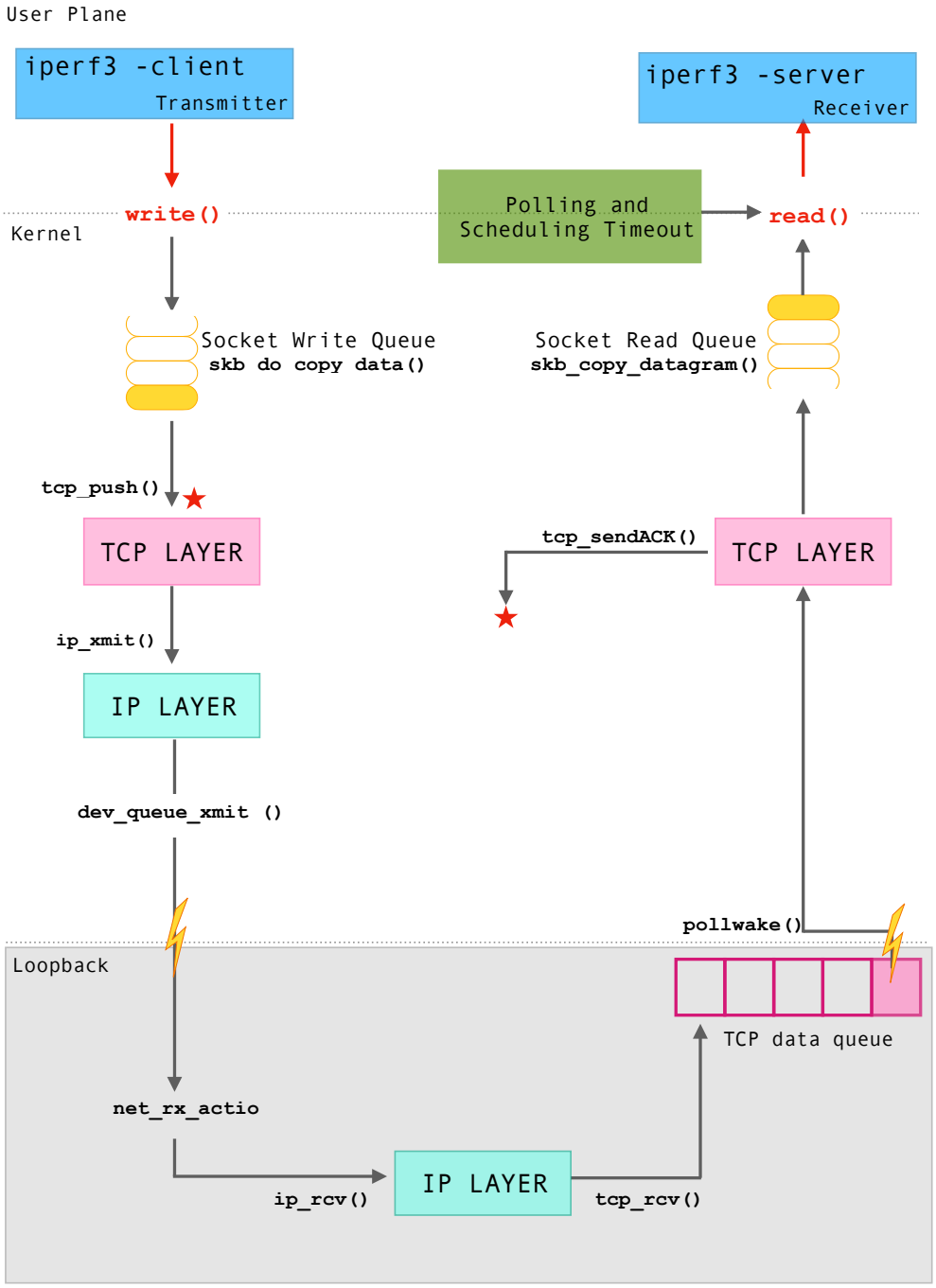

Figure 5.4: Schematic of TCP communication over the loopback interface, showing TX, loopback, RX processes, and memory copying operations

5.1.2 Communication Stack in UDP

UDP communication on the loopback interface bypasses some of the complexities of connection-oriented protocols like TCP. Unlike TCP, which handles reliable data transfer through flow control, congestion control, and acknowledgment, UDP is a connectionless protocol that does not guarantee reliability. This characteristic of UDP leads to a more streamlined and less resource-intensive communication process, which is ideal for scenarios that require low overhead.

Transmitter Process for UDP

The UDP transmission process, as shown in Figure 5.5, is more straightforward compared to TCP due to the connectionless nature of UDP, which does not require the same reliability mechanisms or flow control. The process begins when the user-space application writes data to a socket, triggering the `ksys_write()` and `vfs_write()` system calls, which prepare the data for transmission. These functions pass the data to `sock_write_iter()` and subsequently to `inet_sendmsg()`. Unlike TCP, UDP does not involve connection establishment or flow control mechanisms. This simplification results in a more direct path to `udp_sendmsg()`, which prepares the UDP packet for transmission. The data is placed into a socket buffer (skb) via `ip_make_skb()`, bypassing the more complex packet segmentation and ordering required in TCP.

Once the data is encapsulated in the skb, it is forwarded for IP processing via `udp_send_skb()`. The IP layer then processes the packet with `ip_output()` and queues it for transmission using `dev_queue_xmit()`. The absence of connection tracking and acknowledgment handling results in a less resource-intensive and lower-latency transmission process in UDP. When the data reaches the loopback interface, it follows a similar path to the one in TCP, but with UDP-specific handling. The loopback interface reroutes the data back to the kernel, where it is immediately processed.

UDP Reception Process

The UDP reception process for loopback communication, as shown in Figure 5.6, begins once the packet arrives at the loopback interface. Since UDP is a connectionless protocol, it doesn't involve complex handshakes or state management like TCP. The incoming packet is passed to the IP layer through `ip_rcv()`, which, in the loopback case, directly processes the packet without the need for network transmission steps. The packet is then handled by the UDP layer via `udp_rcv()` and placed in the socket buffer (skb). From there, the data is made available to the receiving user-space application. The kernel directly delivers the data using `udp_recvmsg()` and `sock_read_iter()`,

```
iperf3
- Nwrite
- ksys_write
- vfs_write
- sock_write_iter
- inet_sendmsg
- udp_sendmsg
  | - ip_make_skb
  |    - ip_append_data
  |      - skb_do_copy_data_nocache
  |       - sk_page_frag_refill
  | - udp_send_skb
      - ip_send_skb
        - ip_output
          - ip_finish_output
            - neigh_hh_output
              - __dev_queue_xmit
                - __do_softirq - handle_softirqs
                  - net_rx_action
                    - __napi_poll
                      - netif_receive_skb
                        - ip_rcv
                          - ip_input
                            - ip_input_finish
                              - ip_protocol_deliver_rcu
                                - udp_rcv
                                  - udp_rcv_established
                                    - udp_data_queue
                                     - udp_data_ready
                                       - pollwake
```

Figure 5.5: Function chain for the UDP transmitter process detected using `perf record`

which are responsible for reading the data and transferring it to the user space. Functions like `skb_copy_datagram_iter()` and `skb_recv_udp()` assist in copying the packet's contents from kernel buffers into user-space memory.

```
iperf3
  - Nread
  |- ksys_read
  |- vfs_read
  |- sock_read_iter
  |- inet_recvmsg
  |- udp_recvmsg
  |     |
  |      -- skb_copy_datagram_iter
  |     -- skb_recv_udp()
  |
  |- sys_pselect6
      - do_select
        - schedule
        - sock_poll
```

Figure 5.6: Function chain for the UDP receiver process detected using `perf record`.

Integrated View of the Communication Stack in UDP

An analogous representation of the entire UDP communication flow over the loopback interface is illustrated in Figure 5.7. Similar to TCP, the UDP communication process consists of the transmission, loopback, and reception stages. However, a key difference lies in the absence of acknowledgment mechanisms for received packets in UDP, as it is a connectionless and stateless protocol.

In the UDP transmission process, once the application writes data to the socket, the data is processed and forwarded through the relevant kernel functions, eventually reaching the IP layer and transmitted via the loopback interface. On the reception side, the UDP packet is handled by the IP layer, passed to the UDP layer, and directly placed into the socket buffer without the need for acknowledgment or flow control, as seen in TCP.

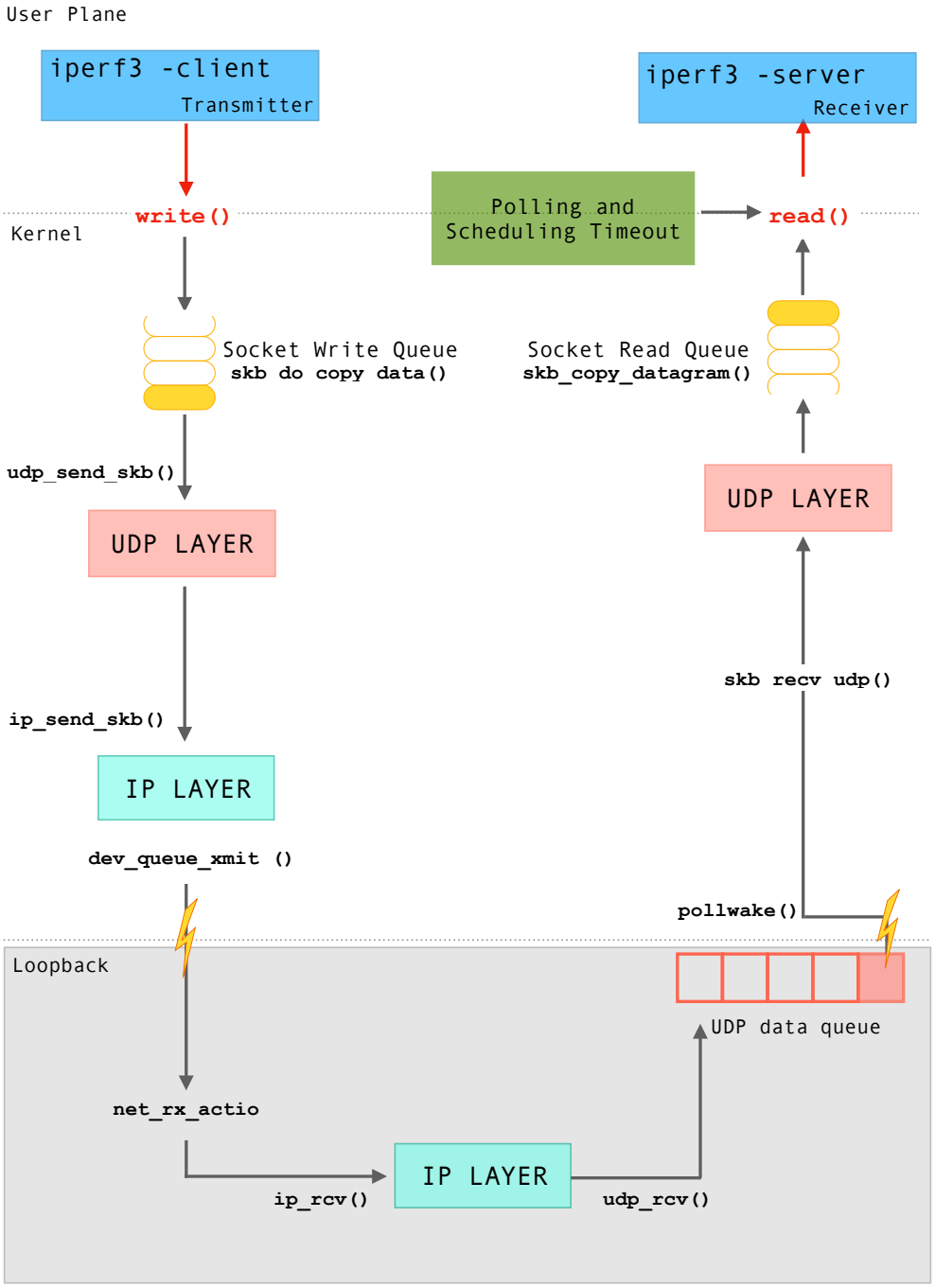

Figure 5.7: Schematic of UDP communication over the loopback interface, showing TX, loopback, RX processes, and memory copying operation

5.2 Communication Flow Over Loopback in a Virtual Machine

The loopback interface, used for communication between a client and server within the same virtual machines is depicted in Figure 5.8.

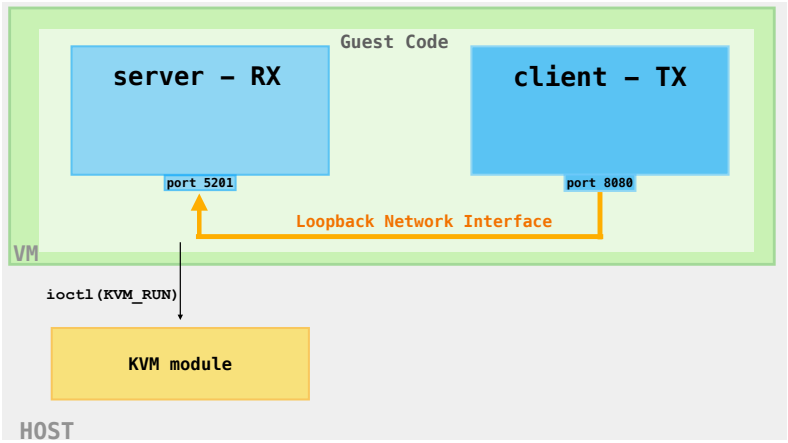

Figure 5.8: Client-server communication over the loopback interface inside a VM

From a network stack perspective, there are no significant differences between loopback communication in a host system and within a virtual machine. The schematics presented in the previous sections, both TCP and UDP, remain valid for this scenario. However, from an external perspective (i.e., from the host point of view), analyzing the communication flow using tools such as `perf` or `perf record` reveals only the global process associated with the VM. This limitation arises because the host can monitor the hypervisor but cannot directly access the internal operations of the guest code without additional tracing tools.

In a virtualized environment, the Kernel-based Virtual Machine (KVM) hypervisor plays a fundamental role in managing the execution of guest virtual machines (VMs). When performing loopback communication within a VM, as depicted in Figure 5.8, the entire network stack is processed within the guest space—from the application layer down to the virtual network interface. This ensures that loopback traffic remains confined within the VM, without directly involving the host's network stack.

Figure 5.8 illustrates a function chain captured through a `perf report` during an `iperf3` loopback test running entirely within the VM. The figure highlights system calls such as `ioctl()` from the guest kernel, which

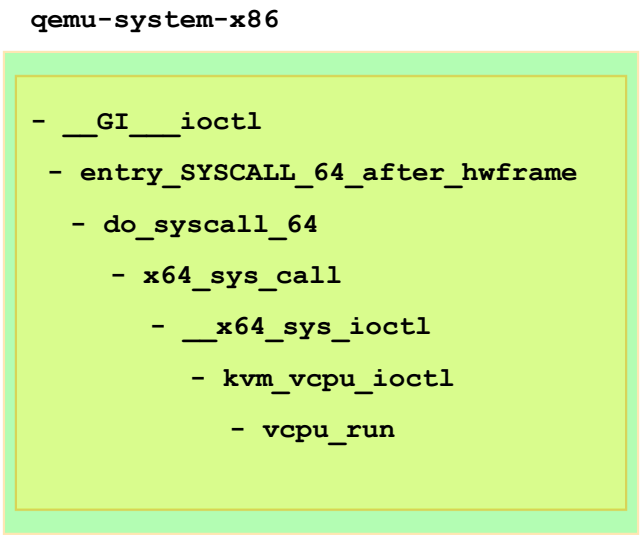

Figure 5.9: monitoring VM process from host during iperf3 run

lead to the invocation of KVM-specific functions like `kvm_vcpu_ioctl()` and `vcpu_run()`. These functions are responsible for managing the communication between the guest and the KVM hypervisor, ensuring that the guest virtual CPU (vCPU) executes its workload correctly. However, they provide limited insight into the specific execution details of the processes running inside the VM.

For processes that execute entirely within the guest, the host can only monitor the overall CPU consumption of the VM, represented by the global `qemu` process. This limits the visibility of the host to a high-level comparison between the CPU usage of the specific process inside the VM and the total CPU consumption by the VM to execute that process. Beyond this, the host lacks granular information about the internal execution of isolated processes within the VM.

5.3 Communication Flow Between Two Isolated Virtual Machines on the Same Host

The communication between two isolated virtual machines (VMs) on the same host, irrespective of whether the TCP or UDP protocol is used, is depicted in Figure 5.15. In this setup, the client residing in VM1 communicates with the server located in VM2. Unlike loopback communication, where the traffic remains confined within a single virtual environment, this scenario requires the data to traverse multiple virtual network components.

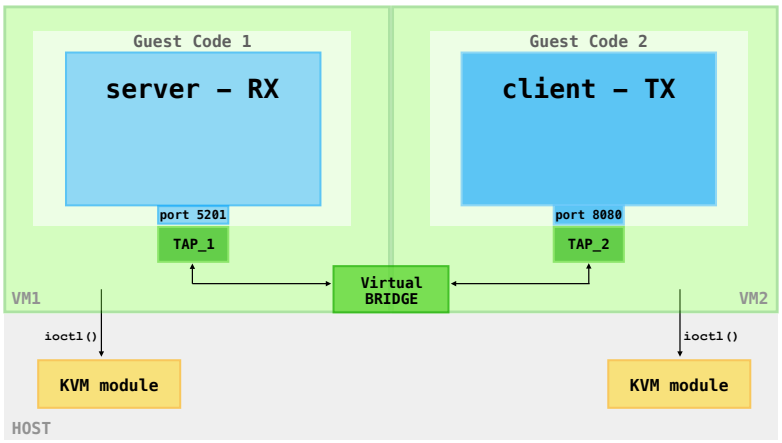

Figure 5.10: Communication flow between two isolated virtual machines on the same host.

The process begins when the client in VM1 sends data through its virtual network interface, which is connected to a TAP (Tun/Tap) interface. The TAP interface (TAP_1) acts as a virtual Ethernet device, responsible for forwarding the outgoing packets from the VM to the host system [17]. Once the packets reach the host, they are passed through a virtual bridge, which functions as a software-based network switch, connecting multiple TAP interfaces [18]. The virtual bridge forwards the packets to the receiving TAP interface (TAP_2) associated with VM2. TAP_2, in turn, delivers the incoming packets to the network stack of the receiving virtual machine. At this point, the TAP interface notifies the guest operating system within VM2 of the arrival of new packets, allowing them to be copied into the guest's memory space and processed by the server application.

This multi-step process introduces additional overhead compared to loopback communication, as the packets must transition between the virtual network interfaces, the bridge, and the TAP devices, requiring context switches

and memory operations at each stage.

5.4 Communication Flow Between Two Virtual Machines

This section provides a detailed breakdown of the communication flow between two isolated virtual machines (VMs) on the same host, which occurs over a virtual network. The process is divided into two main stages, each of which is further subdivided into two sections.

The first stage focuses on the **guest code** running inside the VMs, specifically the client and server processes of `iperf` on the two VMs. The second stage delves into the **network interface processes**, examining the interaction between the TAP interface and the virtual bridge. The first section studies the **TAP interface** of the transmitting VM, which forwards the outgoing packets to the virtual bridge. The virtual bridge is responsible for routing the packets toward the receiving VM. The second section focuses on the **network interface** of the receiving VM. The TAP interface on the receiving side retrieves the forwarded packets from the virtual bridge and delivers them to the guest operating system for further processing. This structured approach highlights both the application-level and network-level mechanisms involved in inter-VM communication on a single host.

5.4.1 Guest Code of the VMs

Guest Code of the Transmission Side

The data flow illustrated in Figure 5.11 progresses through several kernel functions, starting from `Nwrite` and continuing until the packets reach the TAP interface. This sequence includes function calls such as `ksys_write`, `vfs_write`, `sock_write_iter`, and `inet_sendmsg`.

Once the data reaches the network stack, it is prepared for transmission. Depending on the protocol in use, the function `udp_sendmsg` or `tcp_sendmsg` encapsulates the data into UDP or TCP packets. These packets are then transmitted through the virtual network interface (TAP) and forwarded by the virtual bridge to the receiving VM.

A comparison between this flow and the loopback communication flow reveals that the same kernel functions are involved. This is because both flows traverse the same Linux network stack. However, unlike the loopback scenario, where the transmitting process also handles part of the reverse path of the network stack, in this case, the process is limited to forwarding the packets to the TAP interface, with no involvement in handling the return path.

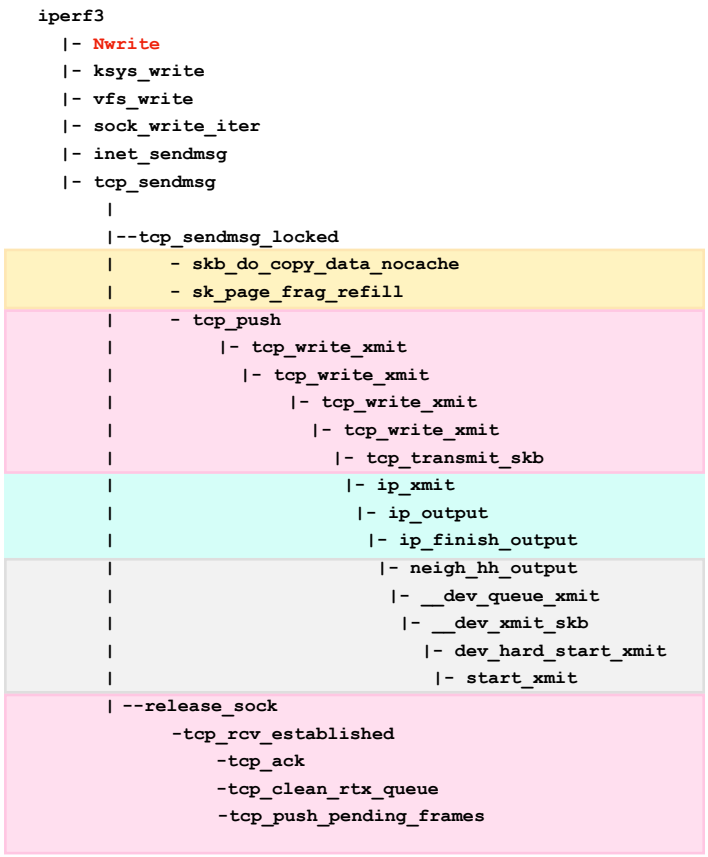

Figure 5.11: Function chain for data transmission in the guest VM

Guest Code of the Reception Side

In the `iperf3` process that receives data from outside the VM, it can be observed that the function pipeline is essentially the same as in the loopback process, except for the part related to `net_rx_action` and below. In this case, `net_rx_action` is responsible for handling the processing of incoming packets on the virtual network when a network interrupt is triggered. The function `handle_softirqs` manages software interrupts (SoftIRQ), which are

triggered to allow more efficient packet processing by deferring certain operations to later stages. Once the packet is received by the virtual network, it is processed by the network stack in reverse order. After the reception is confirmed (through the send acknowledgment), the packet is prepared to be passed on to the application layer.

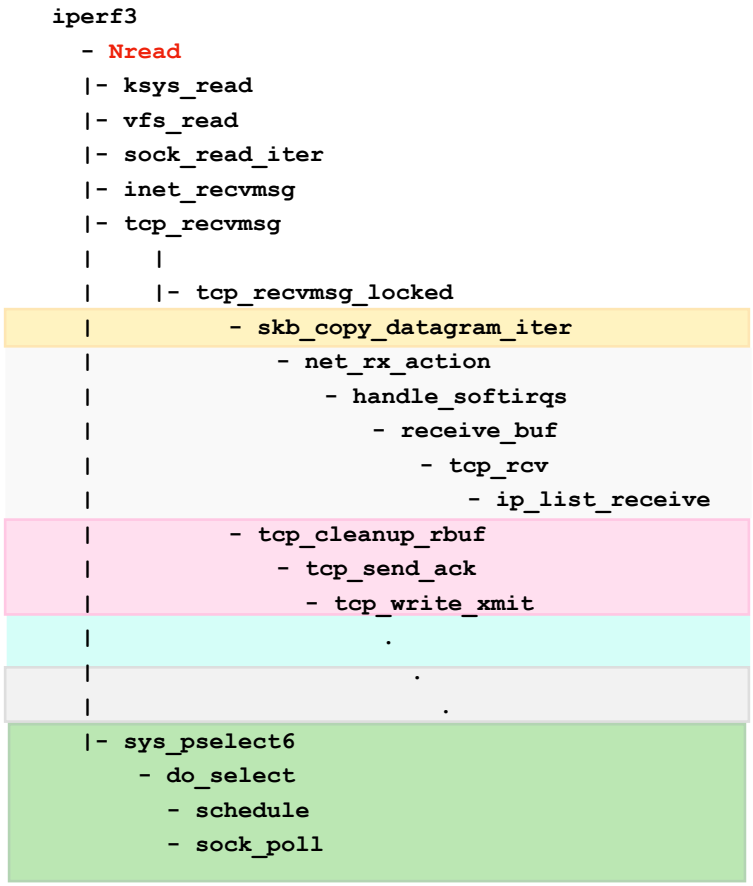

Figure 5.12: Function chain for data reception in the guest VM.

5.4.2 Transmission and Reception Using TAP Interfaces and Virtual Bridge

In this section, both the transmission and reception processes of TAP interfaces and the virtual bridge are examined. TAP interfaces in Linux are associated with a queuing discipline, or **qdisc** (Queue Discipline), which is a kernel mechanism responsible for managing the packet queues of the device [19]. The qdisc controls the flow of packets, temporarily storing them before they are processed for transmission or reception. The qdisc allows the kernel to manage how data is transmitted and received through the TAP interface, ensuring that packets are handled in an orderly manner. It acts as a buffer, holding packets in the queue until the network device is ready to process them.

Transmission Process Using TAP Interfaces

The transmission process of data in the guest VM, as depicted in Figure 5.15, begins when the client application writes data to the socket. The data is passed through a series of functions, starting with `do_writev` and `vfs_writev`, which initiate the write operation. The data is then forwarded to the TAP interface using the `tun_chr_write_iter` function, which transfers the data to the kernel's buffer. Here, the function `tun_get_user` handles copying the data from the user-space application into the kernel buffer.

After the data is copied into the buffer, the kernel function `skb_copy_datagram_iter` moves the data into the TAP interface's qdisc. At this stage, the data is temporarily stored in the qdisc before being forwarded to the virtual bridge for further processing. The virtual bridge, responsible for routing the data between the VMs, processes the packet and determines its destination. The forwarding process involves functions such as `br_handle_frame`, `br_pre_routing`, and `br_forward`, which manage the flow of the packet through the network stack. The data is then sent to the appropriate destination VM, and the process concludes with `dev_queue_xmit`, which transmits the packet.

The figure Figure 5.15 illustrates the entire process, from the moment the data enters the TAP interface to when it is forwarded through the virtual bridge. This flow ensures that data is efficiently transmitted from the guest VM to the virtual network and ultimately to the destination VM.

```
qemu-system-x86

- do_writev
- vfs_writev
- tun_chr_write_iter
    - tun_get_user
       - skb_copy_datagram_iter
         - tun_rx_batched
           - br_handle_frame
             - br_pre_routing
               - br_pre_routing_finish
                 - br_handle_frame_finish
                   - br_forward
                     - br_forward_finish
                       - br_post_routing
                         - br_dev_queue_push_xmit
                           - dev_queue_xmit
```

Figure 5.13: Function chain for data transmission from the guest VM through the TAP interface and virtual bridge

5.4.3 Reception Process Using TAP Interfaces

On the receiving side, the process begins when the virtual bridge forwards the incoming packets to the receiving VM's TAP interface. In contrast to the loopback communication, where the data is directly processed within the guest's network stack, the reception process in this case involves additional stages. The data is copied into the kernel's buffer via the function `skb_copy_datagram_iter`, making it ready for further processing by the

guest VM.

The data is then handed off to the guest operating system, where it is processed by the appropriate network stack functions. The functions `tun_chr_read_iter` and `tun_do_read` are responsible for copying the data from the TAP interface into the guest's memory. This additional layer of processing, while ensuring the efficient handling of packets in a virtualized environment, introduces some overhead in terms of CPU usage.

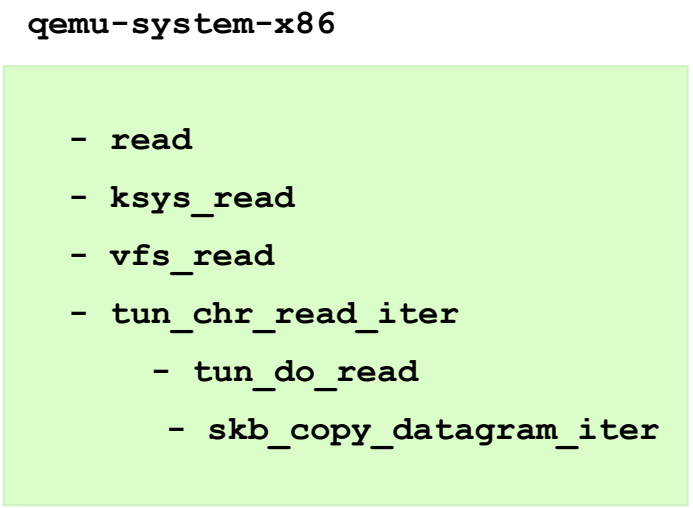

Figure 5.14: Function chain for data reception in the guest VM from the virtual bridge.

This reception process, while similar to the one in a non-virtualized environment, requires additional steps due to the involvement of the virtual network interface and the management of packet forwarding. Despite these complexities, the process remains efficient and is crucial for ensuring smooth communication between VMs in a virtualized environment

5.4.4 Integrated View of the Communication Stack including Virtual Interfaces

After analyzing the function chains involved in both the transmission and reception processes, the final diagram can now be composed to illustrate the communication flow between the two isolated virtual machines. As shown in Figure 5.15, this diagram provides a comprehensive visualization of how data is transmitted from one VM to another, passing through the various components of the virtual network stack.

The diagram integrates the different stages of the process, including the involvement of the TAP interfaces, the virtual bridge, and the kernel's network stack. It encapsulates the entire flow of data between the transmitting VM and the receiving VM, showing how the packets are forwarded, processed, and eventually received by the destination application.

By capturing the key interactions between the guest code, TAP interfaces, and the virtual bridge, the diagram serves as a visual summary of the communication process described in the previous sections. It highlights the complexity of the data flow, especially in a virtualized environment where network packets traverse multiple virtual network components before reaching their final destination.

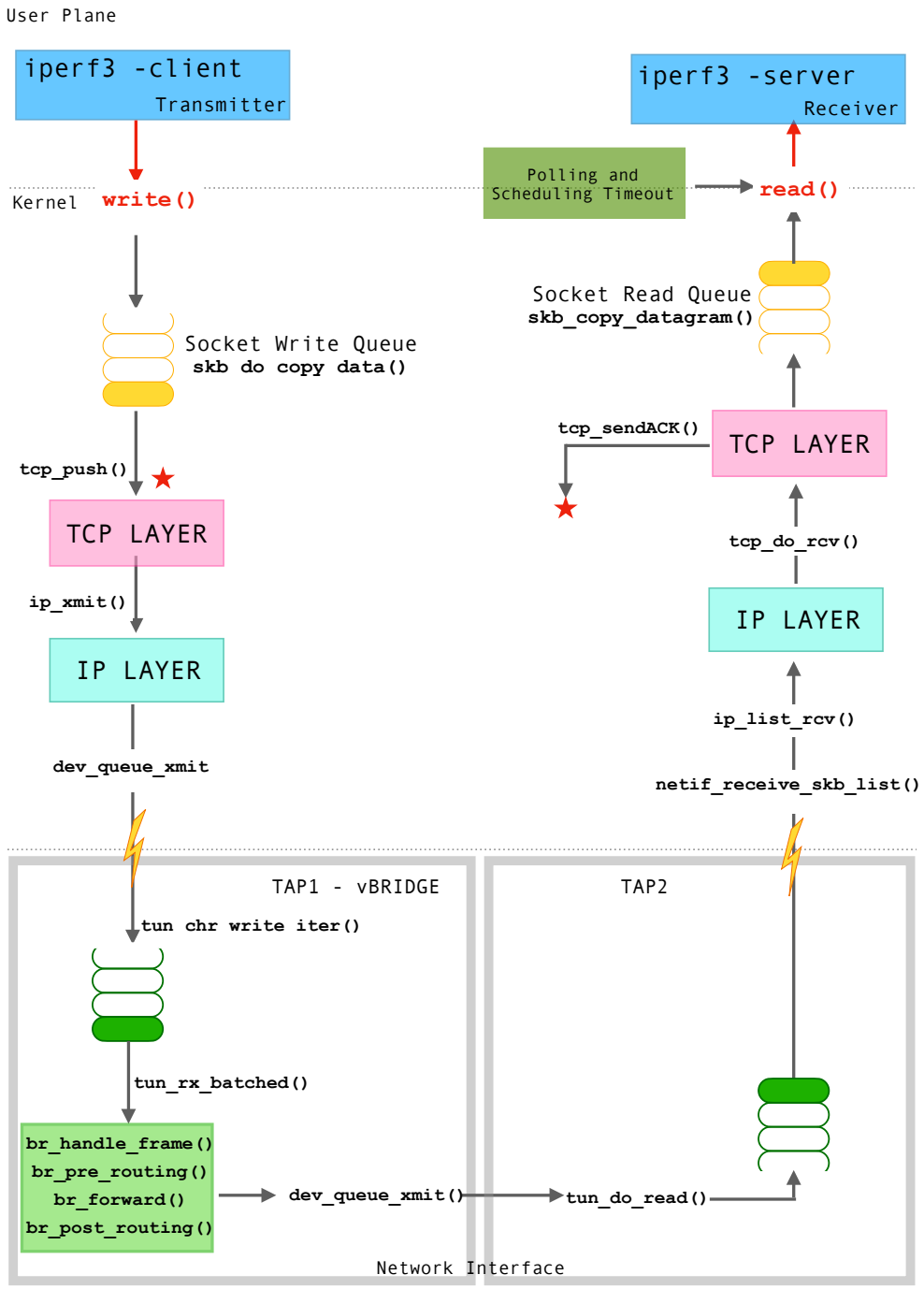

Figure 5.15: Client-Server communication between two isolated VMs

Chapter 6

Experimental - Analysis of TCP/UDP Performance

An analysis of TCP and UDP traffic is conducted across two primary testing environments, each focusing on different types of network interfaces. The first environment examines the **loopback interface**, which facilitates internal communication within a single system. This is further divided into two scenarios: one evaluates loopback communication directly on the host system, without any virtualization, while the other explores the performance of loopback communication within a virtual machine (VM). The second environment investigates **emulated network interfaces** in a virtualized context, where communication takes place between two isolated VMs hosted on the same physical machine. Both environments leverage the *iperf3* tool to assess performance metrics such as bitrate and CPU utilization, offering insights into the efficiency and potential overhead associated with different network configurations.

6.1 TCP Performance Analysis

6.1.1 Loopback Interface Performance

The loopback network interface serves as the baseline for performance evaluation due to its ability to provide a controlled environment with minimal overhead. It represents the most direct and low-latency communication path within a system, capturing the raw performance potential of the network stack. This approach ensures that results reflect the maximum achievable bitrate and CPU utilization under optimal conditions, before introducing additional complexity such as virtualization network interfaces. This streamlined communication allows for highly predictable and consistent performance metrics, making it ideal as a reference point. By establishing the loopback

benchmark, the analysis gains a clear upper limit for network stack performance in terms of achievable bitrate and CPU consumption. This baseline provides a comparative foundation to evaluate the cost of additional overhead introduced in more complex scenarios, such as communication involving virtual machines or inter-host connections.

To investigate the relationship between CPU utilization and achieved bitrate, `iperf3` provides the `-P` option, which enables the use of parallel streams. This option creates multiple client threads, each independently transmitting data to the same server port. For these experiments, P is varied from 1 to 2. The reason for limiting to two parallel processes is that, with $P = 2$, there are two threads for transmitting and two additional threads receiving, resulting in a total of four processes. Considering that the computer used for these tests has 4 CPU cores, this number is chosen as an optimal configuration. Increasing the number of parallel threads beyond 2 is found to introduce additional overhead due to CPU migration and context switching, which compromised the validity of the results. As the number of parallel streams increases, total CPU consumption scales proportionally. Each client thread adds approximately 100% CPU load, resulting in a total CPU capacity offered (`cpumax offered`) that can be expressed as:

$$\texttt{cpumax offered} = P \times 100$$

where P represents the number of parallel streams. For instance, when $P = 1$, the client operates as a single-threaded process with a maximum CPU usage of 100%, while at $P = 2$, the client utilizes up to 200% across two threads.

The achieved bitrate for each configuration depends on the number of parallel streams and the corresponding CPU allocation. By varying P, this experiment examines how increasing the offered CPU capacity impacts the overall transmission rate, providing insights into CPU utilization efficiency and network stack performance under different loads.

Loopback Communication in the Host System

The analysis focuses on the performance of the loopback interface on the host system by varying the CPU usage allocated to the transmitter and the number of parallel processes. The results are shown in Figure 6.1. In the left column of the figure, each plot shows the measured CPU consumption as a function of the CPU offered, controlled using `cpulimit` on the transmitting process. It should be noted that the CPU utilization in this analysis reflects the actual runtime CPU consumption on the client side, rather than the nominal CPU quota imposed. This distinction emphasizes that the observed bitrate scaling is primarily influenced by the client's effective CPU resources

rather than by the theoretical CPU limit. In the right column, the plots display the resulting bitrate as a function of the CPU usage for both the server and client processes. The experiments are organized in rows, with each row representing a different value of `-P`, ranging from 1 to 2. This allows for a clear illustration of the impact of parallelism on both CPU utilization and achievable bitrate.

In each plot representing the bitrate, a theoretical line illustrates the expected relationship between the bitrate and CPU utilization of the transmitting process. The slope of each line indicates how efficiently the bitrate increases relative to CPU utilization. The configuration with $P = 1$ demonstrates the steepest slope, reaching its maximum achievable bitrate more effectively. However, when comparing configurations, it is essential to consider both the slope and the total CPU resources allocated for transmission. Table 6.1 summarizes the line functions, CPU utilization ranges, and maximum bitrates for each configuration, offering a clear overview of the performance characteristics across different settings. The slope of the line function represents the transmission rate per unit of CPU usage. The maximum bitrate is derived by multiplying the slope by the maximum CPU usage for each configuration. It is observed that configurations with higher CPU utilization ranges achieve higher maximum bitrates, albeit with increased CPU resource demands.

	Line Function	***CPU$_C$ (%)***	***CPU$_S$ (%)***	***Max Bitrate (Gbps)***
P=1	$y = 0.28 \frac{\text{Gbps}}{\%\text{CPU}_\text{C}} \cdot x$	0 : 91	0 : 68	26
P=2	$y = 0.19 \frac{\text{Gbps}}{\%\text{CPU}_\text{C}} \cdot x$	0 : 188	0 : 138	35.5

Table 6.1: CPU utilization and performance metrics for each parallel stream configuration in a loopback host environment

An important observation is that the relationship between the number of parallel processes and the achieved bitrate is non-linear. For example, if $P = 1$ achieves a bitrate of 26 Gbps with a CPU usage of 91%, the expected CPU usage for $P = 2$ to maintain the same rate would theoretically double to $91 \times 2 = 182\%$, and the expected bitrate would be 26 Gbps $\times\ 2 = 52$ Gbps. However, this linear scaling does not occur.

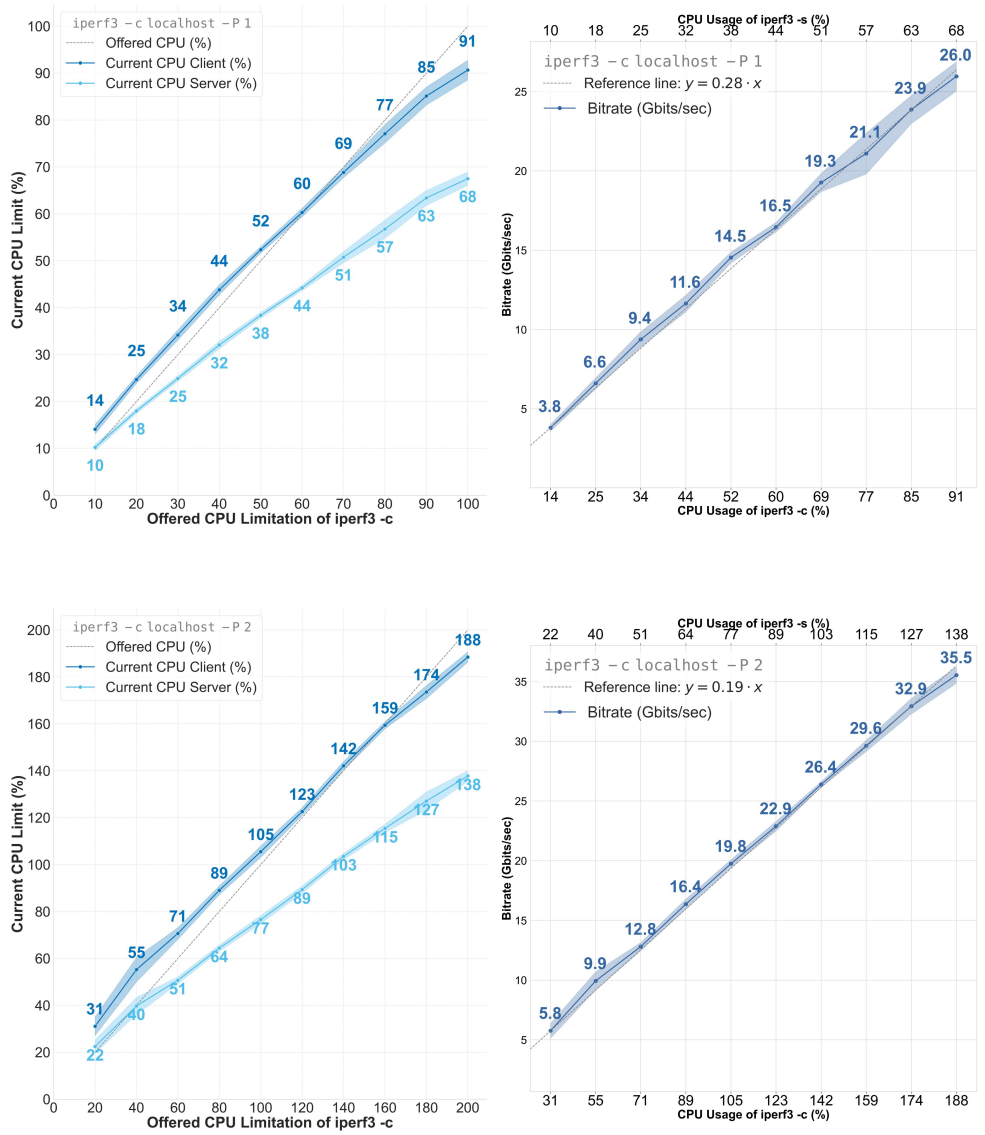

Figure 6.1: Comparison of CPU consumption and achieved bitrate for different parallel stream configurations (`-P` values from 1 to 2) in a loopback host environment

To investigate the reason behind the non-linear scaling of the bitrate, the CPU usage of each parallel process is analyzed using `perf record`[1]. In this chapter, instead of analyzing each function in the chain, the focus will be on the points where the function call sequence converges into a single path. At these junctures, the CPU usage represents the total usage of all the functions that follow in the chain.

Transmission Side ($P = 1$):

- `tcp_sendmsg`: 87.3%
  - `tcp_sendmsg_locked`: 72.90%
    - `skb_do_copy_data`: 40.9%
    - `tcp_push_one`: 25.8%
  - `release_sock`: 13.8%
    - `tcp_ack`: 3.9%
    - `tcp_push_pending_frames`: 8.8%

Reception Side ($P = 1$):

- `pselect6`: 15%
- `tcp_recvmsg`: 42.70%
  - `tcp_recvmsg_locked`: 41.3%
    - `skb_copy_datagram_iter`: 27.6%
    - `tcp_cleanup_rbuf`: 11.85%
      - `tcp_send_ACK`: 11.65%

Transmission Side ($P = 2$):

- `tcp_sendmsg`: 94%
  - `tcp_sendmsg_locked`: 91.12%
    - `skb_do_copy_data`: 39%
    - `tcp_push_one`: 28.56%
  - `release_sock`: 16.95%
    - `tcp_ack`: 5%
    - `tcp_push_pending_frames`: 9.7%

[1]It is important to note that the following analysis refers to a single run, which, while reflecting the maximum CPU usage shown in the plots, does not represent a general case

Reception Side ($P = 2$):

- `pselect6`: 17%
- `tcp_recvmsg`: 48.10%
  - `tcp_recvmsg_locked`: 45.52%
    - `skb_copy_datagram_iter`: 23.0%
    - `tcp_cleanup_rbuf`: 19.5%
      - `tcp_send_ACK`: 19.16%

The function chain analysis (see Section 5.2) reveals that, for $P = 2$, the processes limiting the rate are mainly the server-side functions that handle the received data. An increase in CPU utilization is observed in the `tcp_send_ACK` function, which indicates a rise in the ACK transmission rate. An increase in ACK transmissions reduces the overall throughput, as the sender must wait for the acknowledgment before sending additional packets, thereby limiting the transmission rate. As `tcp_send_ACK` increases, the functions responsible for copying data from the kernel to the application, such as `skb_copy_datagram_iter`, decrease because fewer data bytes are being received. On the client side, the increase in CPU usage for functions related to socket release, `release_sock`, suggests that more ACKs are being received, which also correlates with the increased function `tcp_push_pending_frames`. This indicates that more packets are being retransmitted, further limiting the overall throughput. Therefore, while the expected increase in parallelism (from $P = 1$ to $P = 2$) should theoretically double the throughput, the congestion from the ACKs and the higher processing cost due to the retransmission-related functions result in inefficient scaling. In addition, it is observed that CPU consumption associated with `tcp_send_ack` increases, along with the functions directly linked to it, such as `release_sock`, `tcp_push`, and related operations.

Loopback Communication in a Virtual Environment

To ensure comparability with the host loopback experiment, a VM with 4 vCPUs, matching the host's physical CPU count, is created. This configuration allows for parallel process testing with P values ranging from 1 to 2, mirroring the host experiment. By varying P, the experiment investigates how increased parallelism impacts achievable bitrate and CPU utilization in a virtualized environment, accounting for the overhead introduced by QEMU.

Figure 6.4 illustrates the performance of the loopback scenario within the VM. In the left column, the plots show the CPU utilization not only for the client and server processes but also for the QEMU process managing the virtual machine. The right column depicts the resulting bitrate as a function of CPU consumption across both the client and server. This comparison highlights the impact of parallelism and virtualization overhead on overall performance.

Table 6.2 provides a detailed view of performance metrics, now including the overhead introduced by QEMU virtualization. This overhead is calculated as the percentage of CPU usage attributed to virtualization, defined as the difference between total CPU usage and effective CPU usage for the virtualized workload.

	Line Function	***CPU$_C$ (%)***	***CPU$_S$ (%)***	***CPU$_Q$ (%)***	***Bitrate$_{MAX}$***
P=1	$y = 0.27\frac{\text{Gbps}}{\%\text{CPU}_\text{C}} \cdot x$	0 : 91	0 : 74	0 : 179	25.3 Gbps
P=2	$y = 0.17\frac{\text{Gbps}}{\%\text{CPU}_\text{C}} \cdot x$	0 : 185	0 : 153	0 : 356	31.9 Gbps

Table 6.2: CPU utilization and performance metrics for each parallel stream configuration in a loopback virtual environment

Before delving into the results from a `perf report` run, it is essential to highlight the discrepancy in CPU utilization between the `qemu` process and the combined CPU usage of the client and server within the virtual machine (VM). In the configuration with a single parallel process ($P = 1$), the total CPU utilization reaches 179%. Out of this, 74% is consumed by the client, while the server utilizes 91%, leaving an approximate overhead of 14% attributed to the `qemu` process. This overhead reflects the computational resources required by the hypervisor to manage and facilitate the VM's operations. As the number of parallel processes increases to $P = 2$, the total CPU utilization rises significantly to 356%. In this scenario, 185% of the CPU is allocated to the client and 153% to the server, resulting in an increased overhead of approximately 18% for the `qemu` process. This increase underscores the additional computational burden imposed by the hypervisor, which must manage the coordination and resource allocation of the virtualized environment, even when both client and server processes operate solely within the

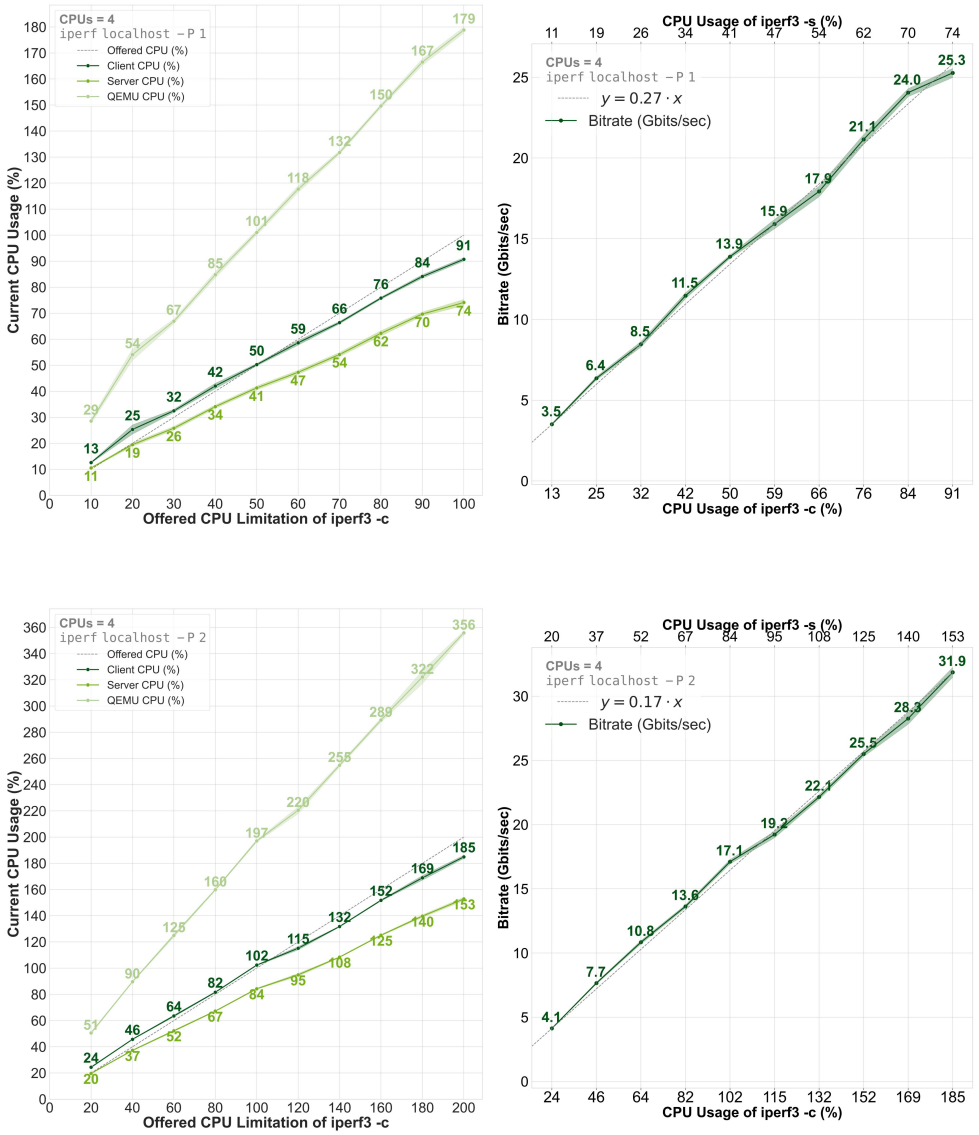

Figure 6.2: Comparison of CPU consumption and achieved bitrate for different parallel stream configurations (`-P` values from 1 to 2) in a loopback virtualized environment

guest system.

Despite the transmission process operating within a similar CPU range to the host loopback experiment (see Section 6.1), the observed bitrate exhibits minimal variation:

- **P = 1):** The bitrate decreases slightly from 26 Gbps (host loopback) to 25.3 Gbps.
- **P = 2):** The total bitrate decreases by approximately 4 Gbps compared to the host scenario, highlighting the impact of virtualization.

Additionally, while the client process maintains a comparable CPU range across both scenarios, the server process shows increased CPU consumption, particularly for the parallel configuration ($P = 2$).

To understand these results, a single `perf report` run is analyzed for each configuration, focusing on key functions within the guest code.
Transmission Side ($P = 1$):

- `tcp_sendmsg`: 86.72%
  - `tcp_sendmsg_locked`: 72.90%
    - `skb_do_copy_data`: 51.14%
    - `tcp_push_one`: 17.32%
  - `release_sock`: 11.7%
    - `tcp_ack`: 5.0%
    - `tcp_push_pending_frames`: 5.3%

Reception Side ($P = 1$):

- `pselect6`: 14.1%
- `tcp_recvmsg`: 56.0%
  - `tcp_recvmsg_locked`: 54.3%
    - `skb_copy_datagram_iter`: 41.0%
    - `tcp_cleanup_rbuf`: 9.2%
      - `tcp_send_ACK`: 9.0%

Transmission Side ($P = 2$):

- `tcp_sendmsg`: 89.9%
  - `tcp_sendmsg_locked`: 76.64%
    - `skb_do_copy_data`: 46.7%
    - `tcp_push_one`: 23.2%
  - `release_sock`: 11.7%
    - `tcp_ack`: 5.0%
    - `tcp_push_pending_frames`: 6.0%

Reception Side ($P = 2$):

- `pselect6`: 17.8%
- `tcp_recvmsg`: 51.5%
  - `tcp_recvmsg_locked`: 49.42%
    - `skb_copy_datagram_iter`: 32.0%
    - `tcp_cleanup_rbuf`: 10.63%
      - `tcp_send_ACK`: 10.5%

The CPU distribution in this experiment differs from that observed in the previous scenario. The percentage associated with `tcp_send_ack` is lower, indicating that the receiver is not significantly limiting the transmission by sending excessive acknowledgments (ACKs). Instead, the performance appears to be more constrained by CPU availability on the transmitting side. This is evident from the lower percentage allocated to data push operations compared to the previous experiment, while a higher percentage is spent on copying data. When comparing the configurations with $P = 1$ and $P = 2$, a similar pattern emerges. The `tcp_send_ack` percentage for $P = 2$ is higher than that for $P = 1$, suggesting that increased acknowledgment traffic in the parallel process configuration could contribute to limiting the achievable bitrate.

6.1.2 Network Emulation Between Two Isolated Virtual Machines

The interaction between two isolated and identical virtual machines (VMs) is examined, each configured with 2 vCPUs. This configuration is determined by the limitations of the host system, which is equipped with only 4 physical CPU cores. As a result, allocating 4 vCPUs to both VMs simultaneously is not feasible. Instead, two identical VMs, each with 2 vCPUs, are deployed to maintain a balanced and isolated environment. Given the available vCPUs, the experiment is conducted using a single transmission process ($P = 1$) for both the transmitter and receiver. Previous experiments have shown that when the number of parallel processes equals the number of available CPU cores, optimal performance is achieved when the system has at least twice the number of cores relative to the parallel processes. This ensures sufficient resources for both transmission and reception. With each VM limited to 2 vCPUs, running parallel processes equal to the number of cores would lead to resource contention and increased context switching. Therefore, a single-process configuration is preferred to maximize CPU efficiency and minimize virtualization overhead.

In this scenario, the impact of varying CPU resources on the transmitting process is analyzed, similar to the previous experiments, by assessing how the bitrate changes with different CPU limits, as shown in Table 6.3. When the CPU is varied from 0% to 100%, a key difference compared to previous tests is observed: with 100% CPU allocation (unlimited), the transmitter process consumes approximately 90% of the available CPU, while in this case, the maximum CPU usage for the transmitting process is limited to 38%. The receiver process, however, shows a higher maximum CPU usage of 80%. This indicates that the receiver process, rather than the transmitter, becomes the more CPU-intensive process under these conditions.

The maximum bitrate achieved in this experiment is approximately 8 Gbps, which is significantly lower than the 25.3 Gbps observed in the loopback experiment with a single process. Additionally, the slope, which was 0.27 in the loopback test, is now 0.24, indicating that more CPU cycles are required to transmit the same amount of data.

	CPU_{VM} (%)	CPU_{QEMU} (%)
VM1 Transmitter	12 : 38	38 : 129
VM2 Receiver	20 : 80	41 : 135

Table 6.3: CPU utilization for both VMs

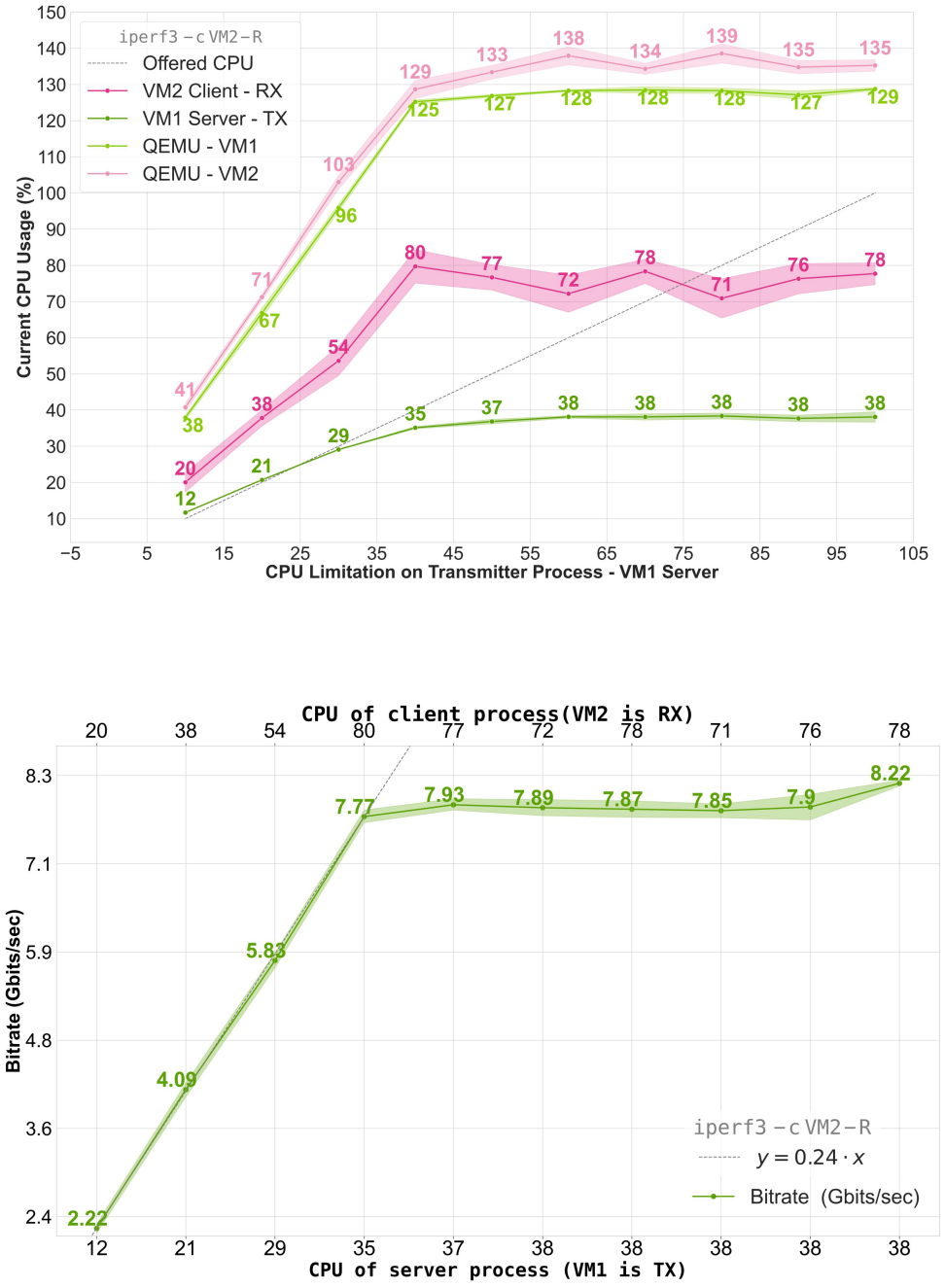

Figure 6.3: Comparison of CPU utilization and achievable bitrate between two isolated VMs during communication

A critical factor in this observation is the role of the virtualized network interface, which is not directly handled by the `iperf3` process but is instead managed within the CPU consumption of the QEMU process. Unlike in the loopback tests, where the loopback interface was part of the transmission process, here, the network interface is handled by QEMU, which emulates the hardware devices, including the virtual CPU and network interface. As discussed in the previous chapter, the QEMU process for the transmitting VM includes both the TAP interface and the virtual bridge, thus accounting for the additional CPU usage for packet forwarding. The receiving VM's QEMU process similarly includes the TAP interface, responsible for receiving the packets.

To conclude, the distribution of CPU utilization between the processes is analyzed, highlighting the different contributions of the transmitting and receiving processes and the influence of virtualization overhead on overall CPU consumption. From the host's perspective, two active processes are involved in the communication, one for VM1 (the transmitting process) and one for VM2 (the receiving process). The analysis begins by examining the guest code processes within each virtual machine—specifically, the `iperf` client process in VM1 and the `iperf` server process in VM2. To provide a broader perspective beyond the guest code, the analysis then focuses on the corresponding `qemu` processes for both virtual machines, which manage the underlying virtualized resources.

Transmission Side - Guest Code VM1:

- `tcp_sendmsg`: 34.8%
  - `tcp_sendmsg_locked`: 32%
    - `skb_do_copy_data`: 29.1%
    - `tcp_push_one`: 1.76%
  - `release_sock`: 2.16%
    - `tcp_ack`: 0.8%
    - `tcp_push_pending_frames`: 1.9%

Reception Side - Guest Code VM2:

- `pselect6`: 7%
- `tcp_recvmsg`: 63.2%
  - `tcp_recvmsg_locked`: 59.22%
    - `skb_copy_datagram_iter`: 52%
    - `tcp_cleanup_rbuf`: 4.49%
      - `tcp_send_ACK`: 4.0%

Transmission Side - Host Code - QEMU VM1:

- `ppoll`: 8.28%
- `write`: 69.53%
  - `tun_get_user`: 64.7%
    - `skb_copy_datagram_iter`: 46.01%
    - `tun_rx_batched`: 17.34%
- `read`: 1.38%
- `ioctl`: 49.87%

Reception Side - Host Code - QEMU VM2 :

- `ppoll`: 10.25%
- `read`: 31.05%
  - `tun_do_read`: 26.7%
    - `skb_copy_datagram_iter`: 20%
    - `consume_skb`: 5.8%
- `write`: 5.4%
- `ioctl`: 87.9%

The transmission process consumes minimal CPU resources, with the majority of the load concentrated on the function `skb_copy_datagram_iter`. As for the physical transmission, it is mainly limited to sending packets to the network interface. Then the traffic is sent to the network interface with a consumption of (1.76%). On the receiver side, the `iperf` process consumes more CPU resources, with approximately 52% of the CPU used for data copying, out of the total 78% consumed by the process.

From an external perspective, for the transmitting virtual machine VM, polling is involved (similar to the receiver process). However, since data exits the VM, it must first be copied into the qdisc associated with the TAP interface. In this case, the process spends 46.5% of the CPU on copying data and only 17.24% on transmission, virtual bridge included. The remaining CPU usage is associated with the overall VM operation and the necessary functions for its proper functioning. On the receiver side, CPU consumption is lower compared to transmission. The VM spends 20% of CPU time copying data from the network buffer to the application buffer using the function `skb_copy_datagram_iter`. The remaining 5.8% of CPU time is dedicated to processing the packets with the `consume_skb` function, which removes the packets from the buffer and marks them as processed, ensuring memory is

properly freed and preventing memory leaks. Since this analysis involves TCP communication, the receiver also sends ACKs. In both processes, the `qemu` process includes minor read and write operations related to the transmission and reception functions. For the transmitting `qemu` process, there are minimal read functions, whereas the receiving `qemu` process handles more read operations.

In conclusion, it is observed that the `iperf` processes do not reach the maximum CPU consumption, as seen in the loopback experiments, due to the backend virtualization layer. This additional overhead is caused by the emulation of network interfaces and memory, which adds significant CPU consumption.Unlike in the loopback experiments, where the primary overhead comes from data copying during both transmission and reception, in the virtualized environment, an additional overhead arises from the backend memory emulation. This is due to the TAP interface, which includes a queuing discipline (qdisc). The qdisc acts as an intermediary memory space where data is copied before being transmitted or received, adding further CPU consumption as part of the network emulation process. This increased overhead is inherent to the virtualization process, as it involves emulating physical network interface functions, such as Ethernet interfaces.

6.2 UDP Performance Analysis

To study UDP performance, the experiments focus on how the bitrate varies with datagram size. UDP, being a connectionless protocol, handles data in units called datagrams, which can vary in size. A UDP datagram consists of both the payload and the headers. Therefore, analyzing UDP performance requires considering the entire datagram, rather than just smaller packet fragments. The size of a UDP datagram ranges from a minimum of 10 bytes to a maximum of 65,535 bytes, with 8 bytes allocated for the UDP header and up to 65,527 bytes for data. However, due to constraints imposed by the IPv4 protocol, the actual maximum data length is 65,507 bytes (65,535 bytes minus the 8-byte UDP header and 20-byte IP header)[20].

While the previous section focused on analyzing the relationship between CPU allocation and maximum bitrate, this section examines how changes in the datagram size affect UDP transmission. Additionally, the data loss percentage is calculated using the following formula:

$$\text{loss\_percentage} = \left(\frac{\text{tx\_bytes} - \text{rx\_bytes}}{\text{tx\_bytes}} \right) \times 100$$

The experimental setup remains the same as in the previous section, with a shift in focus to UDP traffic and datagram loss behavior. The goal is to analyze how the transmission rate changes with varying datagram sizes and determine the percentage of datagrams lost under different conditions.

6.2.1 Loopback Interface Performance

As in the TCP performance analysis, the loopback interface is tested to evaluate the behavior of the network in terms of data transmission and packet loss. The experiments are conducted in both environments, the host system and a virtualized environment.

Loopback Communication in the Host System

The performance of the loopback interface within the host system is analyzed. In Figure 6.4, two plots are presented. The first plot illustrates the variation in CPU consumption for both the server and client processes of `iperf3`. The second plot shows the transmitted (TX) and received (RX) bitrates, highlighting the percentage of data loss as a function of the UDP datagram size.

As the datagram size increases, the overall CPU consumption remains relatively constant. This behavior occurs because both the `iperf3` client and server processes attempt to maximize the achievable bitrate with different datagram sizes. However, while the total CPU utilization remains stable, the

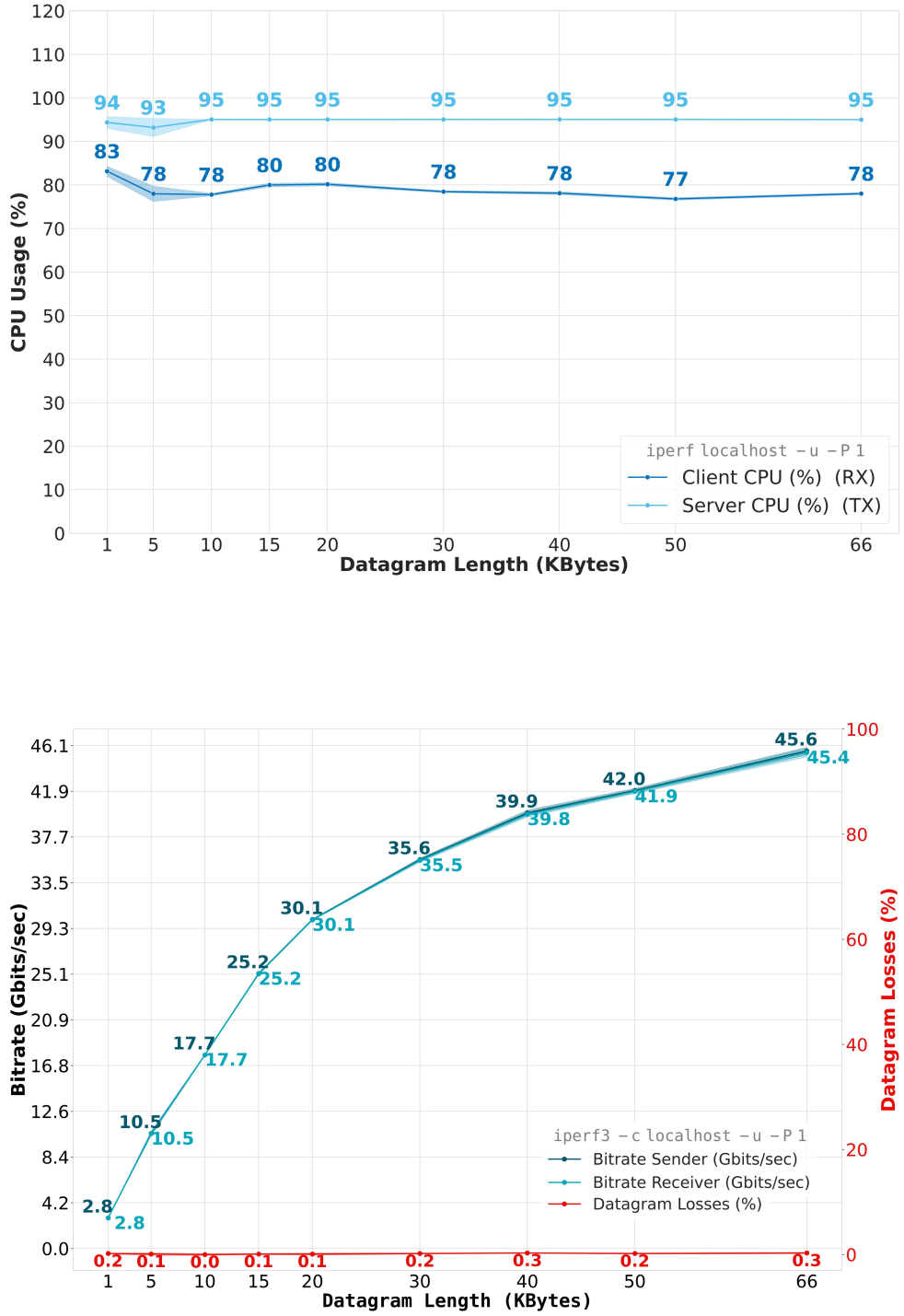

Figure 6.4: CPU utilization, transmitted and received bitrates, and packet loss for varying datagram sizes in a loopback host environment using UDP

internal parameters and function calls of each process vary significantly with changes in the datagram size.

Although UDP lacks flow control and reliability mechanisms, increasing the number of parallel processes does not result in a linear increase in bitrate, despite a significant rise in CPU consumption. The analysis of the main functions involved in transmission and reception reveals a similar CPU load distribution across single and parallel process configurations, indicating that the key functions behave consistently regardless of parallelism. However, the increase in context switches with parallel processes introduces substantial overhead, limiting the expected throughput gains.

To study CPU distribution as a function of UDP datagram size in single-process configurations, two scenarios were analyzed: transmission with a small datagram of 1 KB and a maximum-size datagram of 65.5 KB.

Transmission Side ($P = 1$ - $l = 1000Bytes$):

- `udp_sendmsg`: 64.32%
  - `ip_make_skb`: 18.42%
  - `udp_send_skb`: 41.55%

Reception Side ($P = 1$ - $l = 1000Bytes$):

- `pselect6`: 41.42%
- `udp_recvmsg`: 8.44%
  - `skb_copy_datagram_iter`: 2.32%
  - `skb_consume_udp`: 4.12%

Transmission Side ($P = 1$ - $l = 65500Bytes$):

- `udp_sendmsg`: 86.83%
  - `ip_make_skb`: 68.78%
  - `udp_send_skb`: 16.88%

Reception Side ($P = 1$ - $l = 65500Bytes$):

- `pselect6`: 23.89%
- `udp_recvmsg`: 45.7%
  - `skb_copy_datagram_iter`: 37.28%
  - `skb_consume_udp`: 5.9%

A comparison between the CPU distribution for datagram lengths of $l = 65500$ Bytes and $l = 1000$ Bytes shows a significant shift in resource allocation. With $l = 65500$ Bytes, the transmission process primarily consumes CPU in `ip_make_skb` (68.78%), indicating that a substantial amount of resources is dedicated to buffer creation and packet preparation. In contrast, with $l = 1000$ Bytes, CPU consumption in `ip_make_skb` drops to 18.42%, while `udp_send_skb` increases to 41.55%. This shift suggests that the transmission process becomes more packet-bound, as a higher number of packets is required to transmit the same amount of data. Consequently, the system allocates more resources to handle the increased packet rate, resulting in higher packet processing overhead per unit of data.

On the reception side, similar trends are observed. For $l = 65500$ Bytes, `skb_copy_datagram_iter` accounts for 37.28% of CPU consumption, reflecting the effort needed to copy large chunks of data from kernel space to user space. With $l = 1000$ Bytes, this percentage decreases to 2.32%, as smaller payloads reduce the required copy operations. However, the polling mechanism (`pselect6`) increases from 23.89% to 41.42%, indicating that frequent polling is necessary to manage the higher packet rate, introducing additional overhead.

Overall, the reduced datagram size shifts the bottleneck from data copying and buffer management to packet handling and polling. The system's resources are increasingly consumed by the need to manage a larger number of smaller packets, rather than the efficient handling of fewer large packets. This behavior underscores the trade-off between datagram size and CPU efficiency in UDP-based transmissions, particularly in scenarios where minimizing packet loss is crucial.

Loopback Communication in a Virtualized Environment

The loopback interface using the UDP protocol within a virtualized system follows the behavior shown in Figure 6.5.

A comparison between the two experiments, one conducted in the host system and the other in a virtual machine, reveals that the packet loss percentage remains relatively constant, with both experiments showing a maximum loss of 0.3% while achieving a bitrate higher than 40 Gbps. Regarding the achieved bitrates, for datagram sizes ranging from 1 KB to 50 KB, the virtualized environment consistently shows bitrates approximately 1 Gbps lower than those in the host environment. However, when transmitting datagrams close to the maximum size, the bitrate difference increases to about 3.5 Gbps. In the host-based experiment, the maximum bitrate reaches 45.6 Gbps with an average packet loss of 0.3%, whereas in the virtualized environment, the maximum bitrate is 42.1 Gbps with a slightly lower packet loss of 0.2%.

Another significant difference is observed in the CPU usage of the `iperf`

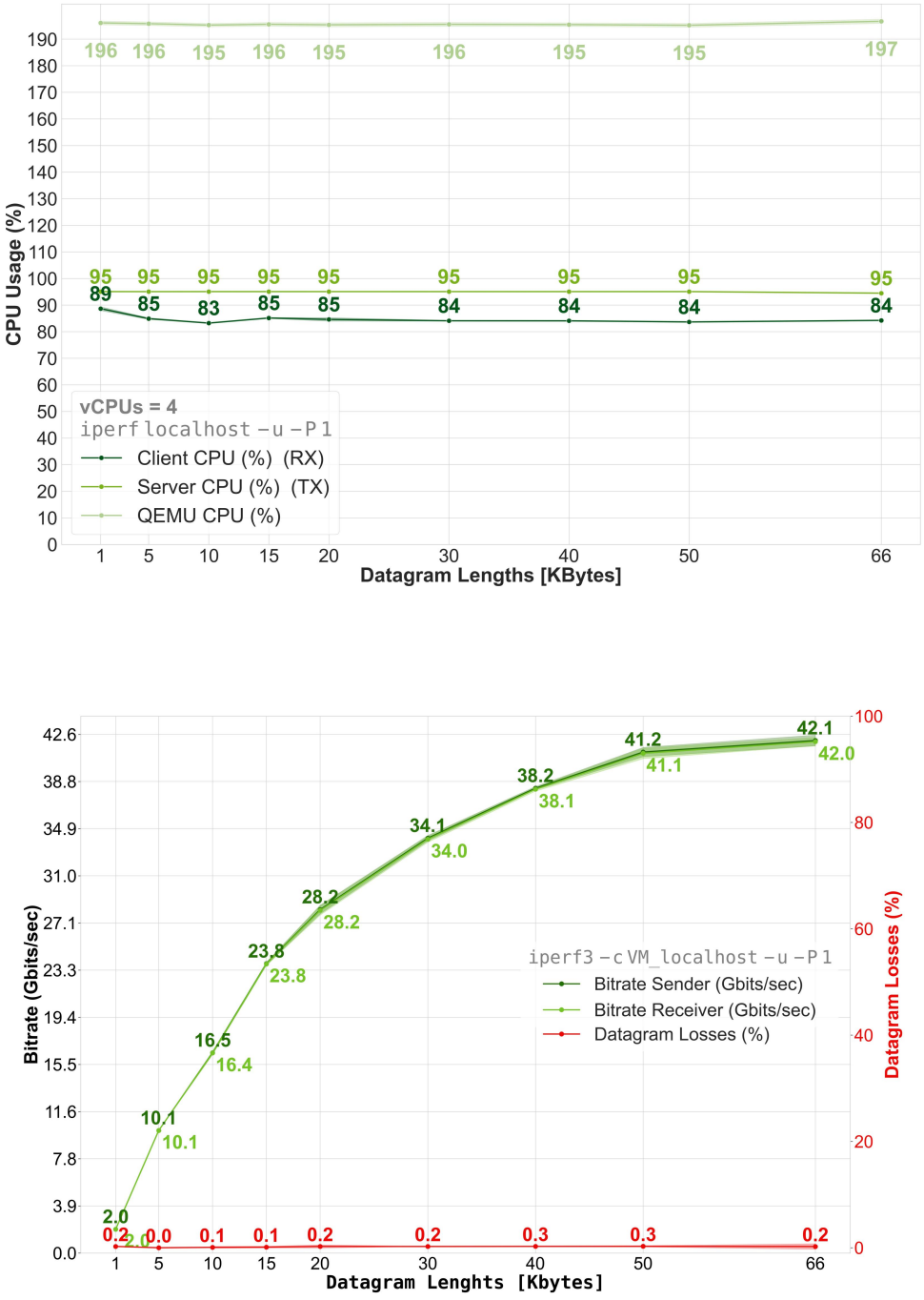

Figure 6.5: CPU utilization, transmitted and received bitrates, and packet loss for varying datagram sizes in a loopback virtualized environment using UDP

process on the receiving side. In the host environment, the CPU usage is around 78%, whereas in the virtualized environment, it averages around 85%. Meanwhile, both transmission processes consistently utilize the maximum CPU, operating at 95%. When analyzing the VM's performance from the host's perspective, the CPU usage of the `qemu` process remains constant, regardless of variations in bitrate and datagram size. The overhead remains almost constant at 18%.

A more detailed analysis of the CPU distribution is conducted to identify potential bottlenecks, focusing on both the minimum and maximum tested datagram sizes. To investigate the significant drop in bitrate and understand why the maximum achievable rate is higher than in the previous test, the results of `perf record` are analyzed for datagram sizes of 1 kB, 50 kB, and 65.5 kB.

Transmission Side ($P = 1$ - $l = 1000$Bytes):

- `udp_sendmsg`: 62.65%
  - `ip_make_skb`: 20.01%
  - `udp_send_skb`: 38.12%

Reception Side ($P = 1$ - $l = 1000$Bytes):

- `pselect6`: 42.8%
- `udp_recvmsg`: 16.81%
  - `skb_copy_datagram_iter`: 7.02%
  - `skb_consume_udp`: 4.56%

Transmission Side ($P = 1$ - $l = 65500$Bytes):

- `udp_sendmsg`: 85.84%
  - `ip_make_skb`: 70.63% +2
  - `udp_send_skb`: 14% -2/3

Reception Side ($P = 1$ - $l = 65500$Bytes):

- `pselect6`: 21.08%
- `udp_recvmsg`: 51.95% 45.7
  - `skb_copy_datagram_iter`: 42.4%
  - `skb_consume_udp`: 8%

The analysis of the two samples with datagram sizes of 1 kB and 65.5 kB reveals that the primary transmission function consumes a similar percentage of CPU in both cases, with each process utilizing approximately 95% of the available CPU. However, when examining the critical functions involved—specifically, those related to packet preparation and memory management as well as the actual transmission—a noticeable difference emerges. In the virtualized loopback environment, the CPU time spent on transmission-related functions is lower, indicating a reduced number of packets being transmitted. Conversely, functions such as `ip_make_skb`, responsible for packet preparation, show increased CPU utilization.

A similar trend is observed in the reception process. While polling functions consume a comparable amount of CPU in both scenarios, functions responsible for copying data from the kernel to the application, such as `skb_copy_datagram_iter`, exhibit higher utilization in the virtualized environment. Despite this, the CPU remains unsaturated, allowing the system to process nearly all incoming packets, with a maximum datagram loss rate of only 0.2

Therefore, while the `iperf` transmission process consistently operates at 95% CPU utilization, a larger portion of this time is allocated to data preparation rather than actual transmission. This shift contributes to the slightly lower throughput observed in the virtualized loopback environment compared to the host-based loopback experiment.

6.2.2 Network Emulation Between Isolated Virtual Machines

The final experiment involves communication between two isolated virtual machines (VMs), each equipped with 2 virtual CPUs. **??** consists of two subplots: the first focuses on the CPU utilization of the `iperf3` processes within the VMs (guest code) and the corresponding `qemu` processes on the host, while the second examines the network performance as a function of UDP datagram size. The transmission process is configured to maximize throughput using the `iperf3 -c IP -u -b 0` command. In the upper subplot the CPU usage of both the `iperf3` processes and the `qemu` processes is analyzed. The lower subplot investigates the achieved bitrate and the percentage of lost datagrams as the datagram size varies. As observed in the TCP-based analysis, the network virtualization overhead in a virtualized environment is not directly attributed to the CPU usage of the `iperf3` transmission and reception processes. Instead, it is encapsulated within the background operations of the `qemu` processes responsible for VM management.

From the plot, it can be observed that for small datagram sizes, such as 1 kB, the percentage of lost packets is significantly high, reaching approximately 28%.

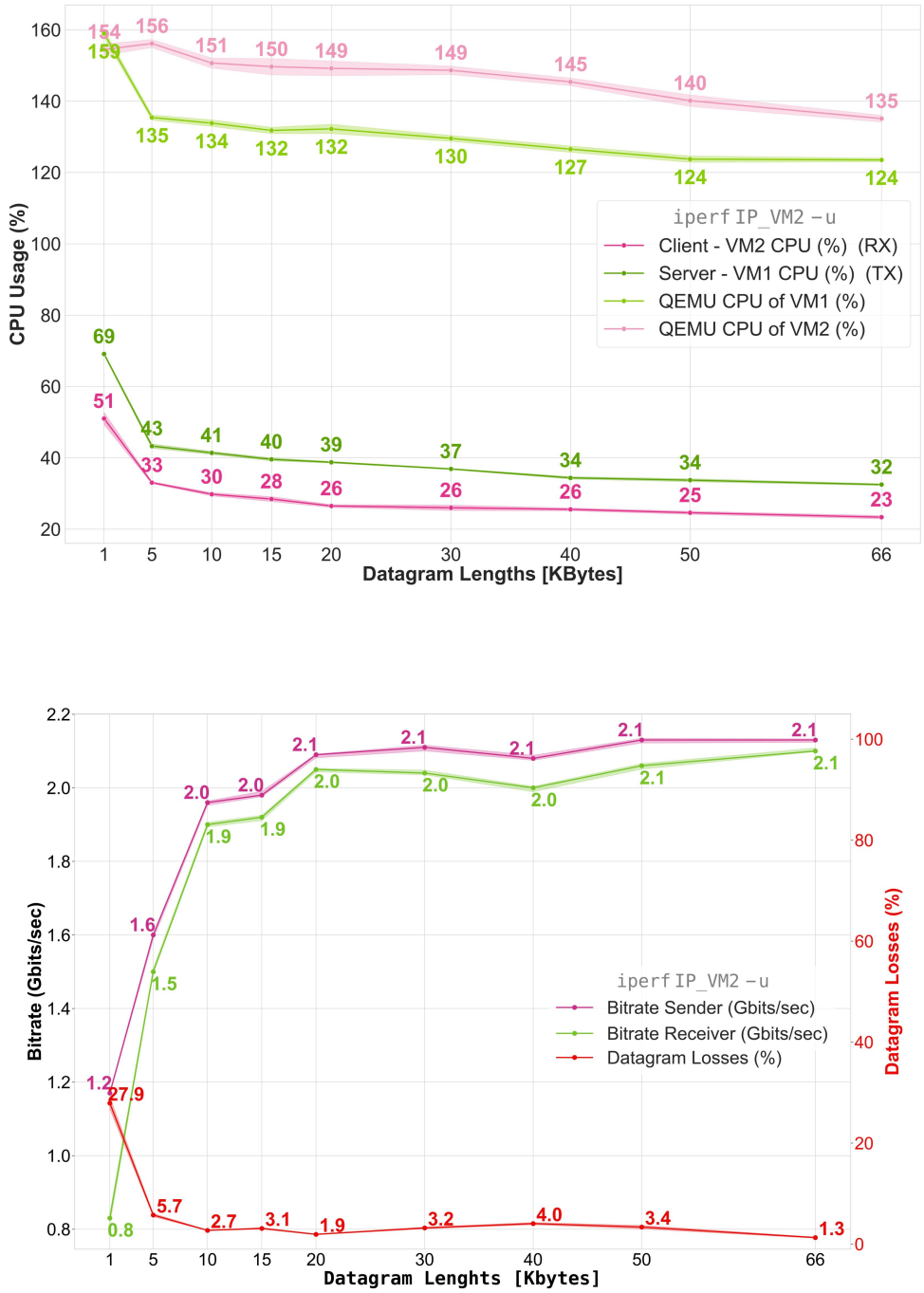

Figure 6.6: CPU utilization (upper plot) and bitrate with packet loss (lower plot) during UDP communication between two isolated VMs.

For the 20-second experiment with a 1 kB datagram and a 1.2 Gbps rate, the total transmitted data is:

$$\text{Total\_Data} = 1.2\,\text{Gbps} \times 20\,\text{s} = 24\,\text{Gbit}$$

The number of transmitted datagrams is:

$$\text{Num\_Datagrams} = \frac{\text{Total\_Data}}{1\,\text{kB} \times 8} = 3.0 \times 10^6$$

With a 28% loss rate, the number of lost datagrams is:

$$\text{Lost\_Datagrams} = \text{Num\_Datagrams} \times 0.28 = 840{,}000$$

The lost data is:

$$\text{Lost\_Data} = \text{Lost\_Datagrams} \times 1\,\text{kB} \times 8 = 6.72\,\text{Gbit}$$

The effective receiver bitrate is:

$$\text{Recv\_Rate} = \frac{\text{Total\_Data} - \text{Lost\_Data}}{20\,\text{s}} = 0.86\,\text{Gbps}$$

As the datagram size increases, the packet loss rate decreases. For a datagram length of 20 kB, the transmission bitrate saturates, and further increasing the datagram size leads to a minimal packet loss rate. For datagram sizes ranging from 10 kB to the maximum size of 65507 bytes, the packet loss rate oscillates between 1.3% and 4.0%.

In relation to CPU utilization, the transmitting `iperf3` process exhibits higher CPU usage for smaller datagram sizes, gradually decreasing as the datagram size increases and stabilizing around 25%. The receiving `iperf3` process, while consistently consuming more CPU than the transmitter, follows a similar trend, stabilizing at around 34%. Notably, once the CPU consumption stabilizes for both processes (from a datagram length of approximately 20 kB), the difference in CPU usage between the transmitting and receiving processes remains around 9%.

Regarding the `qemu` processes, both the transmitting and receiving VMs show a similar trend. Initially, for a datagram length of 1 kB, the CPU consumption of the transmitting VM exceeds that of the receiving VM. However, as the datagram length increases, the transmitting VM consistently consumes less CPU than the receiving VM. Overall, as the datagram length increases, the CPU consumption of both the `iperf3` and `qemu` processes decreases.

To better understand this behavior, samples were analyzed for datagram lengths of 1 kB and 65.5 kB.

Transmission Side - Guest Code VM1 -($l = 1000$**Bytes)**:

- `write`: 65.26%
  - `udp_sendmsg`: 46.39%
    - `ip_make_skb`: 17.11%
    - `udp_send_skb`: 20.7%
    - `ip_route_output_flow`: 8.43%

Reception Side - Guest Code VM1 -($l = 1000$**Bytes)**:

- `pselect6`: 24.97%
- `read`: 22.0%
  - `udp_recvmsg`: 8.45%
    - `skb_copy_datagram_iter`: 1.91%
    - `skb_consume_udp`: 2.84%

Transmission Side - Host Code - QEMU VM1 - ($l = 1000$**Bytes)**:

- `ppoll`: 1.51%
- `write`: 76%
  - `tun_get_user`: 60.04%
    - `skb_copy_datagram_iter`: 5.62%
    - `tun_rx_batched`: 51%
- `ioctl`: 81.5%

Reception Side - Host Code - QEMU VM2 - ($l = 1000$**Bytes)**:

- `ppoll`: 24%
- `read`: 34.47%
  - `tun_do_read`: 11.41%
    - `skb_copy_datagram_iter`: 6.27%
    - `consume_skb`: 3.01%
- `ioctl`: 95.56%

Transmission Side - Guest Code VM1 -($l = 65507$**Bytes)**:

- `write`: 31.63%
  - `udp_sendmsg`: 30.38%
    - `ip_make_skb`: 14.67%
    - `udp_send_skb`: 15.18%
    - `ip_route_output_flow`: 0.47%

Reception Side - Guest Code VM1 -($l = 65507$**Bytes)**:

- `pselect6`: 11.06%
- `read`: 10.08%
  - `udp_recvmsg`: 9.53%
    - `skb_copy_datagram_iter`: 5.66%
    - `skb_consume_udp`: 3.24%

Transmission Side - Host Code - QEMU VM1 - ($l = 65507$**Bytes)**:

- `ppoll`: 3.72%
- `write`:71.53%
  - `tun_get_user`: 62.56%
    - `skb_copy_datagram_iter`: 14.7%
    - `tun_rx_batched`: 43.6%
- `ioctl`: 48.75%

Reception Side - Host Code - QEMU VM2 - ($l = 65507$**Bytes)**:

- `ppoll`: 8.25%
- `read`: 41.83%
  - `tun_do_read`: 15.76%
    - `skb_copy_datagram_iter`:8.54%
    - `consume_skb`: 5.13%
- `ioctl`: 84.92%

The impact of datagram size on system performance has been evaluated by analyzing the CPU usage and function behavior associated with datagram lengths of 1 kB and 65.5 kB in a virtualized environment. Significant differences are observed in both transmission and reception operations depending on the datagram size, influencing CPU utilization across various functions

on the guest and host systems. For 1 kB datagrams during transmission, the `write` system call on the guest side accounts for 65.26% of total CPU usage, primarily due to the high frequency of invocations required to transmit numerous small packets. Within the `write` call, `udp_sendmsg` consumes 46.39%, as it handles the construction and queuing of UDP packets for transmission. This function further delegates tasks to `ip_make_skb` (17.11%), which builds the socket buffer (SKB), `udp_send_skb` (20.7%), which sends the SKB to the IP layer, and `ip_route_output_flow` (8.43%), which performs route determination for each outgoing packet. The elevated CPU consumption of `ip_route_output_flow` highlights the routing overhead associated with handling a large number of small packets.

On the reception side, the `pselect6` system call is responsible for polling and scheduling incoming packets and contributes 24.97% to CPU usage. The `read` function, which manages data retrieval from the SKB, accounts for 22.0%. Within the `read` operation, `udp_recvmsg` consumes 8.45%, while lower-level functions such as `skb_copy_datagram_iter` and `skb_consume_udp` contribute 1.91% and 2.84%, respectively, reflecting the frequent copying and consumption of SKBs when processing small packets.

In contrast, for 65.5 kB datagrams, the CPU utilization shifts significantly. The `write` system call on the guest side reduces its contribution to 31.63%, as fewer large packets reduce the number of system calls required. The `udp_sendmsg` function accounts for 30.38%, with `ip_make_skb` and `udp_send_skb` consuming 14.67% and 15.18%, respectively. The routing overhead of `ip_route_output_flow` decreases to 0.47%, demonstrating the efficiency gain associated with transmitting larger datagrams. On the reception side, `pselect6` and `read` exhibit reduced CPU usage at 11.06% and 10.08%, respectively, while `udp_recvmsg` accounts for 9.53%. The functions `skb_copy_datagram_iter` and `skb_consume_udp` show increased percentages of 5.66% and 3.24%, respectively, as the processing cost scales with the size of each datagram.

From the host system perspective, the `ioctl` system call consistently exhibits high CPU usage for both datagram sizes due to its role in managing memory, virtualization, and data transmission between the virtual machine and the external network. For 1 kB datagrams, `tun_get_user`, which reads data from the tap interface, accounts for 60.04% of CPU usage. This increases to 62.56% for 65.5 kB datagrams, highlighting the higher data volume per packet despite fewer packets being processed. Functions such as `skb_copy_datagram_iter` and `tun_rx_batched`, responsible for copying and batching data, show higher CPU usage when handling larger datagrams. Conversely, `read` and related operations on the host system experience higher load with 1 kB datagrams due to the increased frequency of packet processing. The `ppoll` function, which manages the scheduling of incoming packets, exhibits higher CPU usage for small datagrams because of the frequent

polling operations required for each individual packet. Larger datagrams, on the other hand, benefit from reduced polling frequency, leading to a more efficient processing pipeline.

In conclusion, the analysis demonstrates that small datagrams incur a higher CPU overhead due to the increased frequency of system calls, routing decisions, and data copying operations. Larger datagrams, while reducing the frequency of these operations, introduce higher processing costs per packet, particularly in memory management and data handling functions. However, in the case of a datagram size of 65.5 kB, no compromise is necessary, as this size achieves the optimal balance between transmitted data and the number of packets lost. The 65.5 kB datagrams offer the most efficient outcome in terms of CPU utilization and throughput, indicating that this configuration provides a superior performance-to-loss ratio, making it a preferable choice in environments prioritizing data integrity and transmission efficiency.

Chapter 7

Conclusion

The analysis conducted in this study highlights how the overhead introduced by virtualization can significantly impact the transmission of bitrate. In particular, local traffic within the same virtual machine (VM), when using the loopback interface with both TCP and UDP protocols, remains comparable to loopback traffic within a non-virtualized local host. The minor reduction in bitrate is primarily attributed to the increased CPU utilization required by virtualization processes, especially for memory operations such as copying data between the kernel and application layers.

However, when considering communication scenarios that better reflect real-world systems, virtualized environments exhibit entirely different behaviors. In loopback experiments, the bottleneck is often caused by the `iperf3` processes themselves, which saturate the allocated CPU resources. In contrast, in inter-VM or intra-server communication, the limiting factor shifts to backend processes that emulate the network interface. Despite occurring within the same physical server, data is copied multiple times: once from the guest's application layer to the kernel space on the transmission side, then to the TAP interface emulating a queuing discipline (qdisc). On the receiving side, the TAP interface performs another data copy to emulate the qdisc, followed by a final copy from the kernel to the application layer of the receiving guest. These additional memory operations introduce significant overhead, primarily in CPU usage, due to the continuous transition between the VM and host environments. The hypervisor, specifically KVM in this case, is responsible for managing these transitions, handling the execution of processes that require exiting the VM, performing context switching, and ensuring memory isolation between the guest and host systems. This results in considerable resource consumption and can severely limit network performance.

When comparing the two protocols, TCP and UDP, the impact of virtualization becomes even more evident. While TCP can be leveraged in non-virtualized environments for applications requiring low latency, its performance in virtualized systems is notably superior compared to UDP. The experiments show a significant reduction in bitrate for UDP, with throughput dropping from approximately 8 Gbps for TCP to around 2 Gbps for UDP, representing a 75% decrease. This disparity is primarily due to the additional CPU consumption caused by the virtualization layers and the memory-intensive operations discussed earlier.

In conclusion, while this research has provided valuable insights, it has been conducted on a system with relatively limited resources. Future studies should aim to validate these findings on more performant hardware to better understand the scaling behavior and potential bottlenecks. However, the primary objective of this study is not solely to assess the absolute performance metrics, as these may vary across different hardware configurations. Instead, it aims to propose a methodology for analyzing intra-server traffic behavior, offering benchmark values and guiding principles for resource allocation in scenarios where applications require a lower bound of CPU and network resources to achieve specific throughput guarantees. Such an approach could serve as a foundation for optimizing application deployment strategies in virtualized environments, ensuring efficient resource utilization and maintaining performance stability under varying workloads.

Bibliography

[1] G. Whittaker, "Exploring linux network protocols for better packet processing," 2024. https://www.linuxjournal.com/content/exploring-linux-network-protocols-better-packet-processing.

[2] A. Stephan and L. Wüstrich, "The path of a packet through the linux kernel," *Technical University of Munich, Chair of Network Architectures and Services, School of Computation, Information and Technology*, 2024.

[3] L. Rizzo, "Netmap: A novel framework for fast packet i/o," *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[4] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, pp. 36–47, 2012.

[5] V. Maffione, "Virtio networking: A case study of i/o paravirtualization," tech. rep., 2016.

[6] A. A. L. Tariro Mukute, Joyce Mwangama, "Linux networking performance profiling towards network function virtualisation improvements," *Proceedings of the 23rd Annual Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, 2021.

[7] Canonical Ltd., "Multipass documentation: Lightweight ubuntu vms for developers." https://multipass.run/docs.

[8] iPerf Development Team, "iperf documentation: A tool for network bandwidth measurement." https://iperf.fr/iperf-doc.php.

[9] "cpulimit: A tool for limiting cpu usage of processes - man page." https://manpages.ubuntu.com/manpages/trusty/man1/cpulimit.1.html.

[10] Kernel.org, "perf_stat documentation." https://perf.wiki.kernel.org/index.php/Tutorial#Counting_events:_perf_stat.

[11] A. R. Ghods, "A study of linux perf and slab allocation sub-systems," Master's thesis, University of Waterloo, 2016.

[12] L. M. Pages, "Perf record command: Detailed event tracing, options, and sampling frequency for performance data collection." https://man7.org/linux/man-pages/man1/perf-record.1.html.

[13] Kernel.org, "Perf record and analyzing results with perf report." https://perf.wiki.kernel.org/index.php/Tutorial#Profiling:_perf_record_and_perf_report.

[14] The Linux Foundation, "Linux kernel documentation." https://www.kernel.org/doc/html/latest/.

[15] Linus Torvalds, "Linux kernel git repository." https://github.com/torvalds/linux.

[16] Man7.org, "Linux man pages." https://man7.org/linux/man-pages/index.html.

[17] Kernel.org, "Kernel documentation on tun/tap interfaces: Configuring and using virtual network interfaces." https://www.kernel.org/doc/html/latest/networking/tuntap.html.

[18] L. Foundation, "Linux bridge documentation: Setting up and managing network bridges in linux." https://wiki.linuxfoundation.org/networking/bridge.

[19] L. Foundation, "Queue disciplines in linux: Understanding traffic control and network packet queuing." https://www.linuxfoundation.org/blog/understanding-qdisc-queue-discipline-in-linux/.

[20] Wikipedia, "User datagram protocol (udp): A connectionless transport layer protocol used for fast and lightweight data transmission in networking."