BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

NETWORK INTERFACE CARD PERFORMANCE TESTING

VÝKONNOSTNÍ TESTOVÁNÍ SÍŤOVÝCH KARET

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JOZEF KARABELLY**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

156328

Institut:	Department of Information Systems (DIFS)
Student:	**Karabelly Jozef, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Network interface card performance testing**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Study network performance testing methodology. Based on gained knowledge, build a test environment.
2. Create test scenarios to measure network performance for selected network interface cards and different processor types.
3. Test the selected devices, focus on the network throughput and processor utilization.
4. Evaulate the achieved results.

Literature:

- Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., & Moore, A. W. (2018, August). Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (pp. 327-341).
- Bradner, S., & McQuaid, J. , "Benchmarking Methodology for Network Interconnect Devices", RFC 2544, DOI 10.17487/RFC2544, March 1999, https://www.rfc-editor.org/info/rfc2544>.
- Seth, S., & Venkatesulu, M. A. (2008). TCP/IP Architecture, Design and Implementation in Linux. New York, NY, USA: Wiley.

Requirements for the semestral defence:
1., 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	30.10.2023

Abstract

This thesis explores the importance of NIC performance testing in network engineering, particularly for systems using the modern Linux kernel, due to rising network throughputs and multi-core processors expansion. It develops a scalable, adaptable test scenarios for NIC testing that handle the complexities of a rapidly evolving hardware and software landscape, aiming for stable, reproducible outcomes across different scenarios. The research includes analyzing Linux kernel's offloading features, using continuous integration tools for voluminous testing, and rigorously examining hardware setups. The test scenarios' effectiveness is validated through extensive testing on a specialized testbed, enhancing the understanding and optimization of NIC performance in complex Linux-based networks.

Abstrakt

Táto práca sa zaoberá významom testovania výkonnosti sieťových kariet (NIC) v inžinierstve sietí, najmä pre systémy používajúce moderné jadro Linuxu, v dôsledku rastúcej priepustnosti sietí a expanzie viacjadrových procesorov. Vyvíja škálovateľné a prispôsobiteľné testovacie scenáre pre testovanie NIC, ktoré zohľadňujú zložitosť rýchlo sa vyvíjajúceho hardvéru a softvéru a smerujú k stabilným, reprodukovateľným výsledkom v rôznych scenároch. Výskum zahŕňa analýzu akcelerácie sieťových mechanizmov jadra Linuxu, použitie nástrojov kontinuálnej integrácie pri objemnom testovaní a dôkladné preskúmanie hardvérových konfigurácií. Účinnosť testovacích scenárov je validovaná rozsiahlym testovaním na presne definovanom testovacom prostredí, čo zlepšuje pochopenie a optimalizáciu výkonnosti NIC v komplexných sieťových systémoch založených na Linuxe.

Keywords

testing, performane testing, network interface cards, continuous testing, Linux, hardware offloading

Klíčová slova

testování, výkonnostní testování, síťové karty, kontinuální testování, Linux, hardwarová akcelerace

Reference

KARABELLY, Jozef. *Network interface card performance testing.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Táto práca sa zaoberá kritickou úlohou testovania výkonu sieťových kariet (NIC) v inžinierstve a manažmente sietí. Zdôrazňuje komplexnosť a dôležitosť sieťového výkonu v rôznych aplikáciách. Optimálny sieťový výkon závisí na správnej konfigurácii viacerých komponentov vrátane CPU, PCI, prepojení soketov, sieťovej karty a konfigurácie jadra Linuxu. Táto práca predstavuje testovacie scenáre pre kontinuálne testovanie výkonu NIC na operačných systémoch založených na modernom jadre Linuxu. Potreba neustáleho testovania vzniká so zvyšovaním sieťovej priepustnosti na vyššie dátové rýchlosti a rozširovaním procesorov koncových systémov na viacjadrové. Tieto vývoje podnietili používanie techník ako hardvérová akcelerácia pomocou NIC a ladenie jadra Linuxu na optimalizáciu prenosu dát veľkej rýchlosti. Jadro Linuxu sa vyvíja do hĺbky, zlepšovaním schopnosti spracovania paketov a do šírky, podporou širšej škáli hardvéru. To má za následok rozsiahlu testovaciu maticu a komplexnosť manuálneho testovania. Okrem toho, rýchly a decentralizovaný charakter vývoja jadra Linuxu znamená vyšší objem opráv, ktoré vyžadujú testovanie na izoláciu potenciálnych problémov. Táto práca skúma funkcie odľahčenia jadra Linuxu, aby identifikovala efektívne konfigurácie a stanovila najlepšie postupy pre kontinuálne testovanie výkonu softvéru. Práca navrhuje škálovateľné testovacie scenáre pre rozsiahle testovanie, ktoré poskytujú konzistentné, reprodukovateľné výsledky v rôznych výkonnostných scenároch. Kapitola 2 sa podrobne venuje mechanizmom hardvérová akcelerácie siete v jadre Linuxu, čo je zásadný aspekt optimalizácie výkonu siete. Hardvérová akcelerácia odkazuje na prenos špecifických úloh spracovania siete z CPU na sieťovú kartu, čím sa zvyšuje efektivita. Následne, Kapitola 3 skúma integráciu nástrojov pre kontinuálnu integráciu (CI) v kontexte testovania výkonu NIC na systémoch založených na Linuxe. Hodnotí rôzne nástroje a platformy CI, posudzuje ich vhodnosť pre zvládanie komplexnosti a rozsahu testovania výkonu siete. Kapitola tiež pokrýva všeobecné techniky testovania softvéru, prispôsobuje tieto metodológie na riešenie jedinečných výziev, ktoré predstavuje testovanie výkonu NIC. Kapitola 4 poskytuje komplexný prehľad o hardvérove použitom pre testovanie výkonu NIC. Detailne opisuje špecifikácie a konfigurácie sieťových kariet vybraných pre experimenty, vysvetľuje odôvodnenie za každým výberom. Ďalej zdôrazňuje rôzne architektúry procesorov, na ktorých budú sieťové karty testované. Na základe predchádzajúcich kapitol, Kapitola 5 predstavuje systematický prístup k testovaniu výkonu NIC na operačných systémoch založených na jadre Linux. Tento prístup je navrhnutý tak, aby bol komplexný, pokrývajúc rôzne aspekty ako plánovanie testov, ich vykonávanie, zber dát a analýzu. Dôraz je kladený na automatizáciu, aby sa zabezpečili efektívne a opakovateľné testovacie procesy. Okrem toho predstavuje rad všeobecných testovacích scenárov na zjednodušenie efektívneho, opakovateľného testovacieho procesu, ktorý prináša konzistentné a porovnateľné výsledky. Dôležitým aspektom je škálovateľnosť, zabezpečujúca, že testovacie scenáre môžu byť implementované naprieč rôznymi sieťovými prostrediami a hardvérovými konfiguráciami. Rovnako dôležitá je prispôsobivosť, umožňujúca testovacím scenárom zostať aplikovateľnými v meniacom sa a vyvíjajúcom sa hardvérovom a softvérovom prostredí. Kapitola 6 ukazuje, že testovacie scenáre sú dostatočne flexibilné, aby zvládli širokú škálu sieťových konfigurácií, vrátane rôznych verzií IP (IPv4, IPv6), nastavení VLAN a transportných protokolov ako TCP alebo UDP. Budúca práca by mohla rozšíriť tieto scenáre o konfigurácie ako MPTCP alebo VXLAN. Štúdia ukazuje, že variácie v IP verziách a nastaveniach VLAN majú minimálny vplyv na sieťový výkon, pričom moderné procesory efektívne zvládajú až 100Gb obojsmerného toku dát. Vypnutie TCP Segmentation Offload (TSO) vedie k výraznému zníženiu lokálnej efektivity, medzi 60% a 70%, zatiaľ čo vplyvy na efektivitu vzdialeného zariadenia sú minimálne a jednotné na-

prieč rôznymi značkami NIC. Podobné dopady sú pozorované, keď sú vypnuté oba offloady výpočtu odoslaných a prijatých checksumov, hoci vzdialená efektivita klesá. Dôležitosť manažmentu architektúry NUMA na predchádzanie poklesu výkonu a význam nepretržitej integrácie pri reakcii na opravy jadra sú zdôraznené v tejto práci. Intel Sapphire Rapids vyniká ako najlepší výkonný hráč, pričom Intel E810 je označený ako najlepšia sieťová karta. Avšak výkon sa výrazne líši v závislosti od kombinácií procesorov a sieťových kariet, ovplyvňujúc celkovú efektivitu a optimalizáciu systému. Tieto poznatky sú kľúčové pre informované rozhodovania v oblasti inžinierstva a manažmentu sietí. Do budúcnosti by sa mohol klásť dôraz na začlenenie 400Gb kariet alebo na vývoj metriky pre priepustnosť na watt s porovnaním procesorov ARM s inými architektúrami, čo umožňuje porovnania, ktoré sú obzvlášť relevantné, keď veľkí poskytovatelia začínajú klásť dôraz na energetickú úspornosť [31].

Network interface card performance testing

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Matěj Grégr Ph.D. The supplementary information was provided by Ing. Ondrej Lichtner and Ing. Adam Okuliar. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jozef Karabelly
May 15, 2024

Acknowledgements

I would like to thank my supervisor Ing. Matěj Grégr Ph.D. for his valuable feedback and insights about the thesis. Furthermore, I thank Ing. Ondrej Lichtner for his constructive feedback and consultations about the theoretical aspects of Linux kernel. Lastly, I acknowledge Ing. Adam Okuliar for his consultation and insights about network performance testing.

Contents

1 Introduction **3**

2 Linux kernel network offloading **5**
2.1 Features of network interfaces . . . 5
2.2 Checksum offloads . . . 6
2.2.1 Transmit Checksum Offload . . . 6
2.2.2 Receive Checksum Offload . . . 7
2.2.3 Local Checksum Offload . . . 8
2.2.4 Remote Checksum Offload . . . 8
2.3 Segmentation offloads . . . 9
2.3.1 TCP Segmentation Offload . . . 9
2.3.2 UDP Fragmentation Offload . . . 10
2.3.3 Generic Segmentation Offload . . . 10
2.3.4 Generic Receive Offload . . . 11
2.3.5 Partial Generic Segmentation Offload . . . 11
2.4 Scaling in the Linux Networking Stack . . . 11
2.4.1 Receive Side Scaling . . . 12
2.4.2 Receive Packet Steering . . . 13
2.4.3 Receive Flow Steering . . . 14
2.4.4 Accelerated Receive Flow Steering . . . 15
2.4.5 Transmit Packet Steering . . . 16

3 Continuous integration network performance testing **18**
3.1 Continuous integration . . . 18
3.1.1 CircleCI . . . 19
3.1.2 Jenkins . . . 19
3.1.3 Travis CI . . . 20
3.2 Software testing . . . 20
3.3 Software benchmarking . . . 22
3.4 Performance testing . . . 23
3.5 Network performance testing . . . 24
3.5.1 nepta framework . . . 25
3.5.2 Catalog . . . 25

4 Testbed **27**
4.1 Nvidia ConnectX-6 Dx . . . 28
4.2 Intel Ethernet Controller E810 . . . 29
4.3 Broadcom BCM57508 . . . 30

5 Test scenario design **32**
5.1 Overview . . . 32
5.2 Test planning and test scenarios . . . 34
5.2.1 TCP scalability scenario . . . 37
5.2.2 Duplex TCP scenario . . . 37
5.2.3 Offloading impact scenarios . . . 38
5.2.4 UDP test scenario . . . 39
5.3 Execution and Implementation . . . 39

6 Test scenarios evaluation **43**
6.1 Baseline TCP scenarios . . . 43
6.2 Duplex test scenario . . . 48
6.3 NIC offloading . . . 51
6.4 UDP comparison . . . 57
6.5 NUMA impact . . . 59
6.6 Performance impact of mitigations . . . 63
6.7 Summary . . . 65

7 Conclusion **69**

Bibliography **71**

A Measurements **75**
A.1 Baseline measurements . . . 75
A.2 Duplex . . . 78
A.3 TSO offloading . . . 81
A.4 TX and RX offloading . . . 83
A.5 UDP . . . 85

Chapter 1

Introduction

This thesis delves into the crucial role of network interface card (NIC) performance testing in network engineering and management. It emphasizes the complexity and importance of network performance across various applications. Optimal network performance hinges on the appropriate configuration of multiple components, including the CPU, PCI, interconnect, network interface card, and Linux kernel configuration.

This thesis introduces test scenarios for ongoing NIC performance testing on operating systems based on the modern Linux kernel. The necessity for continuous testing arises from scaling up network throughput to higher data rates and expanding end-system processors to multiple cores. These developments have spurred the use of techniques like NIC offloads and Linux kernel tuning to optimize high-speed data transfer. The Linux kernel is evolving in-depth, enhancing its packet processing capabilities and breadth, supporting a wider range of hardware. This results in a vast testing matrix and the complexity of manual testing. Additionally, the rapid and decentralized nature of Linux kernel development means a higher volume of patches requiring testing to isolate potential issues. This thesis investigates the Linux kernel's offloading features to identify effective configurations and establish best practices for continuous software performance testing. The thesis proposes scalable test scenarios for extensive testing that yield consistent, reproducible results in various performance scenarios.

The thesis is organized as follows:

Chapter 2 delves into the specifics of network offloading mechanisms in the Linux kernel, an essential aspect of network performance optimization. Offloading refers to transferring specific network processing tasks from the CPU to the network interface card, enhancing efficiency.

After that, Chapter 3 explores the integration of continuous integration (CI) tools in the context of NIC performance testing on Linux-based systems. It evaluates various CI tools and platforms, assessing their suitability for handling the complexities and scale of network performance testing. The chapter also covers general software testing techniques, adapting these methodologies to address the unique challenges NIC performance testing poses.

Chapter 4 provides a comprehensive overview of the hardware setup used for NIC performance testing. It details the specifications and configurations of NICs selected for the experiments, explaining the rationale behind each choice. Furthermore, it highlights various CPU architectures on which the NICs will be tested.

Based on the previous chapters, Chapter 5 presents a systematic approach for NIC performance testing on Linux kernel-based operating systems. The approach is designed to

be comprehensive, covering various aspects such as test planning, execution, data collection, and analysis. It emphasizes automation to ensure efficient and repeatable testing processes. Scalability is a crucial focus, allowing the test scenarios to be applied to various network environments and hardware configurations. Adaptability is also vital, enabling the test scenarios to remain relevant in evolving hardware and software landscapes. The chapter outlines strategies for ensuring the stability and reproducibility of test results, which are vital for reliable performance evaluation.

Next, Chapter 6 assesses the effectiveness of the proposed NIC performance testing approach and the performance of NICs under different test scenarios. This involves conducting a series of tests on the specified testbed, utilizing a sufficiently large sample size across diverse configurations and various test scenarios tailored to different workloads. The chapter summarises the findings from these tests, discusses their implications, and provides a detailed comparison of the results across the entire testbed.

Finally, Chapter 7 concludes the thesis by summarizing the key findings from the NIC performance tests, the effectiveness of the proposed approach, and test scenarios. It reflects on the implications of these findings for network engineering and management, particularly with Linux-based systems. The chapter discusses potential limitations and suggests areas for future research and improvement.

Chapter 2

Linux kernel network offloading

This thesis necessitates a basic familiarity with the Linux networking stack. To maintain focus and relevance to the main content of the thesis, extensive details and intermediate packet processing steps will be bypassed, concentrating instead on aspects crucial for understanding the subsequent sections. For those seeking a more in-depth exploration, a thorough description is available in [32]. However, it's important to note that the most current and authoritative resource on this subject is the source code, which is continuously updated.

2.1 Features of network interfaces

In the Linux networking subsystem, *feature flags* indicate network interfaces' capabilities and specific behaviours. Within the Linux kernel's networking subsystem context, these flags are known as *features*. These features are integral to the `net_device` structure, a key component in the Linux kernel that represents network interfaces. Network interface drivers typically set these flags, which can be altered (enabled or disabled) based on the hardware's capabilities and the desired network configuration.

The term *features* refers to specific hardware offloading or processing capabilities supported by the Network Interface Card (NIC). Examples of such capabilities include checksum offloading, TCP segmentation offloading (TSO), and scatter-gather I/O, among others. The kernel leverages these features to optimize how packets are handled for a particular interface. For instance, if a NIC supports checksum offloading, the kernel will forego computing the checksum via software, thus conserving CPU resources (as detailed in Section 2.2). Substantial enhancements can be achieved in network throughput and latency by utilising these network interface features.

Some standard features in the Linux kernel include `NETIF_F_HW_CSUM`, indicating the NIC's ability to compute packet checksums; `NETIF_F_TSO`, which allows the NIC to break down large data blocks into TCP segments; `NETIF_F_RXHASH`, enabling the NIC to distribute incoming network traffic across multiple CPU cores for improved load balancing; `NETIF_F_GSO`, which permits the NIC to offload the segmentation of large packets; and `NETIF_F_SG`, allowing the NIC to assemble data from various memory buffer locations into a single packet, thus reducing the number of I/O calls.

These features can be configured and examined using various Linux networking tools and commands, such as `ethtool`. This tool provides the functionality to enable or disable specific features on a network interface tailored to the network's requirements and the hard-

ware's capabilities. The current list of features can be viewed using the `--show-features` action in `ethtool`, and modifications to features that can be toggled from userspace are made using the `--features` action.

2.2 Checksum offloads

Checksum offloading is crucial in modern networking, particularly in the Linux kernel's networking stack. It involves transferring the responsibility of computing checksums from the CPU to the network interface card (NIC), significantly reducing CPU load and enhancing system throughput.

Originally, CPUs were responsible for all checksum calculations, a resource-intensive task, especially under high network traffic. The development of advanced NICs capable of handling these computations represented a paradigm shift, allowing for more efficient data processing and reduced latency.

Checksums are crucial for ensuring data integrity across networks. The NIC typically computes the checksum at the link layer, as it fully understands the protocol. This is known as the Frame Check Sequence for Ethernet, detailed at the end of the frame [4]. The NIC computes the checksum during data transmission and checks it upon reception; a nonzero value indicates a corrupted frame, which is then discarded.

At the network layer, IPv6 does not use a checksum [18], while IPv4 includes a checksum for its header, which is generally not computationally demanding due to the header's small size [2]. Many controllers can compute the IPv4 header checksum before transmission.

Transport layer protocols, like TCP [12], UDP [1], and DCCP [14], use a 16-bit one's complement checksum for the entire packet. Computing this checksum is resource-intensive, particularly due to large packet payloads and poor caching. NICs often provide checksum offloading for receiving and sending at this layer, enhancing efficiency.

The checksum verification is straightforward for received packets, independent of the specific protocol, provided the IP packet is not fragmented. For transmission, newer NICs can calculate checksums for any part of the packet and place them at any offset, allowing for broader protocol compatibility.

TCP and UDP also include a Pseudo Header in their checksum calculations, which may require the system software to pre-compute this part of the checksum [30].

In Linux, checksum offloading is managed through feature flags on a per-packet basis, accommodating scenarios like tunnel headers and VLAN tags. For an in-depth understanding, the Linux source file include/linux/skbuff.h can be referenced.

2.2.1 Transmit Checksum Offload

In the context of data transmission, especially when excluding Generic Segmentation Offload (GSO), see Section 2.3.3, the process becomes more intricate. Without GSO, basic checksum offloading [48] involves ensuring that packets have a valid checksum or that the subsequent software component, such as the network driver, can compute it. Specific feature flags indicate a driver's ability to calculate checksums. The flag `NETIF_F_HW_CSUM` signifies the driver's capability to compute any one's complement checksum as defined in the `sk_buff` structure fields. Other feature flags like `NETIF_F_FCOE_CRC` and `NETIF_F_SCTP_CRC` represent the ability to calculate the CRC for FCoE [26] and SCTP [37] packets, respectively. The flags `NETIF_F_IP_CSUM` and `NETIF_F_IPV6_CSUM`, though deprecated in favour

of NETIF_F_HW_CSUM, indicate the capability to compute the checksum for TCP/UDP packets over IPv4 and IPv6, respectively.

Notably, none of the feature flags account for the checksum of the IPv4 header. This omission is because computing the checksum for the relatively small 20-byte header is not resource-intensive, especially when the header is already constructed in software. An exception exists for IP packets generated through TCP Segmentation Offload (TSO), where the controller assembles the IP packets, necessitating special handling.

Earlier efforts focused on simpler cases of checksum computation, but these evolved into a more general capability for computing one's complement checksums. This development means the driver assesses whether the controller can compute the checksum. If the controller recognizes the headers, it is directed to compute the checksum; otherwise, the checksum calculation returns to the software. Drivers must also accommodate packets that are already checksummed or do not require checksum computation.

In scenarios involving generic checksum calculations, such as SCTP, the checksummed portion of the packet is defined as a suffix starting from the csum_start field in the sk_buff structure. The driver ensures the checksum is placed at the csum_offset. If the controller does not support generic checksum computation, the driver verifies the compatibility of these fields with the controller's capabilities.

Should there be doubts about the controller's checksum computing ability, the driver can employ helper functions like skb_csum_hwoffload_help(), skb_checksum_help(), or skb_crc32c_csum_help() to carry out the checksum calculation in software.

2.2.2 Receive Checksum Offload

In the driver, the feature flag NETIF_F_RXCSUM controls offloading received checksums [10]. However, the system's stack does not rely solely on the driver's functionality; it also consistently evaluates the ip_summed field. When a packet is received from the NIC, the driver cannot change the offload state, as it is too late. Thus, the driver must use the meta-information provided by the device to determine which checksums have been verified and then appropriately set the ip_summed field based on this information.

The ip_summed field can have several values:

- CHECKSUM_NONE: This indicates that the device did not perform a checksum on the packet, potentially due to a lack of capability. The checksum has not been verified, and the value of skb->csum is undefined.

- CHECKSUM_UNNECESSARY: Used when the hardware does not compute the entire checksum but verifies checksums for specific protocols (such as TCP, UDP, GRE, SCTP, FCOE). When checksums are verified, this state is set, indicating that no additional processing is required. However, skb->csum remains undefined.

- CHECKSUM_PARTIAL: This indicates a scenario where a checksum is offloaded to a device, often used in cases like GRO or remote checksum offload. It means checksums up to a certain point in the packet are verified, but those beyond the offloaded checksum are not. This state is important for understanding the extent of verification.

- CHECKSUM_COMPLETE: This state is used when a device provides the checksum for the entire packet, offering a more generalized approach where the device does not need to parse headers to compute the checksum. However, this state does not apply to SCTP and FCoE protocols.

Understanding these states is crucial for effectively managing checksum offloading and ensuring accurate packet processing in network systems.

2.2.3 Local Checksum Offload

Local Checksum Offloading (LCO) [41] is a technique in network processing designed to efficiently calculate the checksum of an encapsulated datagram, especially when the inner checksum is set for offloading. This method is particularly beneficial in scenarios involving encapsulated data packets, such as virtual networking or tunnelling protocols.

The fundamental principle of LCO is based on the properties of one's complement checksums. For these packets, a correctly calculated checksum means that the one's complement sum of the packet equals the complement of the sum of the pseudo-header. The checksum field effectively neutralizes the other sums due to its complementary nature. This principle applies to any case where an 'IP-style' one's complement checksum is employed.

In TX (Transmit) Checksum Offload, where the network interface card (NIC) computes the packet checksum, this principle is used for LCO. When the NIC calculates the checksum, the one's complement sum from a specified starting point (`csum_start`) to the packet's end equals the complement of the initial value in the checksum field. To compute the outer checksum without processing the entire payload, the summing can halt at `csum_start`, including the complement of the 16-bit word at the offset (`csum_start` + `csum_offset`).

The correct outer checksum emerges when the actual inner checksum is later computed (either by `skb_checksum_help()` or hardware) due to the inherent arithmetic properties. In the Linux stack, LCO is employed when constructing UDP headers for encapsulations like VXLAN[24] or GENEVE[16], as well as their IPv6 equivalents. It is also used for IPv4 GRE[23] headers but is not currently applied to IPv6 GRE headers, although it could be relevant.

2.2.4 Remote Checksum Offload

Remote Checksum Offloading (RCO)[17] is a method employed to circumvent the computation of the inner checksum in an encapsulated data packet, enabling the offloading of checksum calculation for the outer layer instead. This technique requires alterations to the encapsulation protocols, which must also be supported by the receiving end. Due to these prerequisites, RCO is not typically enabled by default. RCO proves particularly advantageous in the context of encapsulated network traffic, such as that utilizing UDP or GRE protocols.

When creating a packet at the transport layer (for example, a TCP or UDP segment), the system prepares it for checksum offloading. This preparation includes marking in the packet's metadata that the checksum calculation is designated for offloading. The packet's checksum field is either filled with a pseudo-header checksum or set to zero.

Subsequently, the encapsulation layer, which wraps the original packet with additional headers for routing or other functions, appends specific metadata to the packet. This metadata comprises the checksum start and offset values, instructing the NIC on the initiation point for checksum calculation and the placement of the computed checksum.

Once encapsulated and equipped with the necessary metadata, the packet is dispatched to the NIC. The NIC, capable of checksum offloading, performs the checksum computation for the outer header (such as the UDP checksum in UDP encapsulation), based on the metadata provided.

Upon computing and inserting the checksum, the NIC forwards the packet. The offloading process ensures that the remaining parts of the packet, particularly the inner header and payload, are not altered during transmission.

The implementation of RCO varies across encapsulation protocols, with many tunnel types providing control flags to enable it. For instance, in VXLAN, the flag `VXLAN_F_REMCSUM_TX` within the `vxlan_rdst` structure is utilized to activate RCO for transmissions to specific remote destinations.

2.3 Segmentation offloads

The Linux networking stack implements various segmentation offloads and incorporates software strategies that reduce the frequency of traversing the networking stack. These software approaches are crafted to integrate smoothly with hardware offloading techniques, thereby boosting overall efficiency.

To comprehend certain offloading techniques, it is crucial to understand the Network API (NAPI), a mechanism in the Linux kernel that minimizes the overhead caused by interrupts. When a network device receives a packet, it stores the packet in a DMA buffer in the host's memory, marks it, and then interrupts the host. Instead of processing the packet immediately during the interrupt, a NAPI-compatible driver schedules a softirq handler for polling and temporarily disables further interrupts. The driver then processes packets, including any received during this period, in a polling loop. This loop continues until no packets are left, after which the interrupt is re-enabled. This approach enhances the packet processing rate by preventing disruption from incoming packets.

The offloading techniques discussed below, often called Stateless Offloads, are designed to streamline networking operations. While some of these techniques, such as Large Receive Offload (LRO), may require maintaining state, the term „Stateless Offloads“ is a widely accepted and commonly used descriptor in the industry.

2.3.1 TCP Segmentation Offload

TCP functions as a stream-like conduit, creating the semblance of a continuous data stream over a network that, in reality, transmits discrete, bounded packets. Data inputted into this „stream“ is divided into segments, each assigned a sequential number for tracking purposes. This sequential numbering ensures that segments are neither lost nor delivered out of sequence at the application layer. Notably, TCP does not specify the size of these data segments, theoretically allowing data transmission to be as granular as individual bytes. This adaptability to various data sizes and network conditions is a key feature of TCP's design.

Minimizing overhead by handling data in the largest chunks possible is efficient, as it reduces the frequency of data passing through the entire network stack. Software-wise, the network stack typically segments data as late as possible, facilitating batch processing. On the receiving end, the stack may merge segments from the same TCP stream before delivery, optimizing the process.

A Network Interface Card (NIC) might offer a feature that queues TCP packets larger than the network's Maximum Transmission Unit (MTU) and autonomously fragments these oversized packets into smaller, MTU-compliant segments for transmission. This capability, known as Large Send Offload (LSO), allows the network stack to operate beyond the constraints of the link MTU, dealing with larger data blocks.

The offloading process requires recognition up to the TCP header, which limits it to certain header combinations. Basic NICs support TCP over IP, while advanced ones can process TCP encapsulated within various tunnel types. Conversely, a Large Receive Offload (LRO) merges multiple TCP segments into a larger packet before it enters the OS's receive queue. This feature is optional to avoid protocol violations in bridging or routing scenarios. If LRO is inapplicable or needs deactivation (as in routing), Linux offers Generic Receive Offload (GRO) for packet coalescence, enabling larger data block handling.

Both LSO and LRO rely on checksum offloading for packet validation and creation. Managing multiple checksums, particularly in tunnelled protocols, is vital for these offloads. Additionally, TCP Segmentation Offload (TSO)[46] enhances performance in virtual environments by effectively increasing the virtual link MTU without compromising isolation.

LRO operates independently of the Linux networking stack. Once a driver receives a packet, LRO processing has already been completed. Thus, Linux's role is to provide a mechanism for activating LRO, achieved via the `NETIF_F_LRO` feature flag, which toggles LRO as needed.

Recognizing LRO's limited applicability, some NIC vendors have developed a stricter version of LRO that provides detailed metadata about original packet segmentation, enabling resegmentation later.

2.3.2 UDP Fragmentation Offload

UDP Fragmentation Offload (UFO)[49] is a relatively simpler technique compared to TCP segmentation offload, primarily because UDP datagrams have a maximum size limit of 64 KB. This size often surpasses typical network links' Maximum Transmission Unit (MTU). In contrast to TCP, UDP does not have an inherent concept of fragmentation and instead relies on IP fragmentation for payloads exceeding the MTU. Some network controllers can manage IP packet fragmentation directly on the chip. This process is less complex than TCP segmentation offload since it only updates the IPv4 header checksum and does not require modifications to the transport layer Protocol Data Unit (PDU). The IPv4 Identification (ID) field must remain constant across all fragments generated from a single IPv4 datagram.

However, the UFO feature has been deprecated in modern Linux kernels, which means these kernels no longer generate socket buffers (skbs) using UFO. Nevertheless, these kernels are still capable of processing UFO skbs received from devices like tuntap. Despite the deprecation of the UFO feature, the offloading for UDP-based tunnel protocols remains supported.

2.3.3 Generic Segmentation Offload

Generic Segmentation Offload (GSO)[40] is a software-only approach to manage situations where device drivers cannot perform certain hardware offloads. In GSO, packet segmentation occurs just before the packet is delivered to the driver. This approach streamlines the driver's task by eliminating the need for it to manage packet segmentation. The packets for segmentation may result from Generic Receive Offload (GRO) processing, or they might be directly generated from data sent through a socket.

In this process, the data in a `skbuff` is split into multiple resized `skbuffs`. Each of these is sized according to the Maximum Segment Size (MSS), which is defined in `skb_shinfo()->gso_size`. A crucial step before employing any hardware-based segmentation offload is to initiate a corresponding software offload using GSO. This preemptive

step is essential to avoid scenarios where a frame becomes untransmittable when rerouted between different devices due to incompatible segmentation specifications.

2.3.4 Generic Receive Offload

Generic Receive Offload (GRO)[39] is the counterpart to Generic Segmentation Offload (GSO). GRO functions by consolidating incoming packets into batches whenever possible. As packets are processed by the Network API (NAPI), they are organized into a `gro_list` composed of `sk_buff` structures. Each newly received packet is assessed against this list to determine if it can be amalgamated with existing ones. A distinctive feature of GRO is its adaptability: it is not restricted to specific protocol layers, thus allowing protocol handlers to make informed decisions about which aspects of the information can be compromised for the sake of merging. Packets that are candidates for merging typically exhibit similar header sequences, with only slight variations in certain fields. These packets remain in the `gro_list` until they are relayed to the upper layer for further processing.

Contrary to Large Receive Offload (LRO), GRO merges packets non-destructively. For successful combination by GRO, packets must belong to the same flow and share certain attributes, including timestamps. This meticulous merging process ensures that GRO-merged packets can be subsequently deconstructed into their original segments when needed. Consequently, GRO is also suitable for deployment in routers and bridges.

Network Interface Cards (NICs) implementing a strict version of LRO can offload GRO directly to the hardware. This is facilitated by utilising the `NETIF_F_GRO_HW` feature flag.

2.3.5 Partial Generic Segmentation Offload

Partial Generic Segmentation Offload[42], an amalgamation of features from TCP Segmentation Offload (TSO) and Generic Segmentation Offload (GSO), see Section 2.3.1 and Section 2.3.3, capitalizes on specific attributes of TCP and tunnelling protocols. This offload strategy focuses on updating only the innermost transport header and, if necessary, the outermost network header rather than modifying headers for each segment. This selective updating enables devices that do not support tunnel offloading or lack checksum capabilities to utilize segmentation benefits.

In this offload technique, all headers, except the inner transport header, are essentially set up for straightforward duplication. The only exception pertains to the outer IPv4 Identification (ID) field; in scenarios where this header does not include the Don't Fragment (DF) bit, device drivers must increment the ID field. The presence and utilization of this offload feature are denoted by the `NETIF_F_GSO_PARTIAL` flag.

2.4 Scaling in the Linux Networking Stack

Implementing multiple queues for communication between the host and the Network Interface Card (NIC) opens up various opportunities for performance improvement. The ideal count of these queues is subject to change based on the specific controller and its application. As will be explored, the benefits of employing multiple queues are not confined to multi-processor systems; they extend to a broader spectrum of use cases.

Additionally, the utilization of multiple receive queues is straightforward. The controller's primary responsibility in this setup is to allocate incoming network packets among

these queues. The particular distribution algorithm employed can facilitate the integration of various offloading techniques.

2.4.1 Receive Side Scaling

Modern Network Interface Cards (NICs) are equipped with multiple receive and transmit descriptor queues, a feature referred to as multi-queue. This capability allows a NIC to distribute incoming packets across various queues, thereby balancing the processing workload across multiple CPUs. This distribution utilizes a filtering mechanism that assigns packets to different logical flows, with each flow directed to a distinct receive queue for processing by separate CPUs. This technique, commonly known as Receive-side Scaling (RSS)[45], aims to enhance performance uniformly.

RSS typically uses a hash function on network or transport layer headers, such as a 4-tuple hash of IP addresses and TCP ports. A standard hardware implementation involves a 128-entry indirection table, where each entry represents a queue number. A hash function, often a Toeplitz hash, determines the packet's receive queue. A Toeplitz hash is generated based on a predefined Toeplitz matrix, characterized by constant descending diagonals from left to right. When applied to data like packet headers in networking, the Toeplitz hash consistently distributes network traffic across multiple paths or resources.

More advanced NICs provide greater flexibility, enabling packets to be steered to queues based on programmable filters. For example, packets heading to a web server's TCP port 80 can be routed to a specific queue. These „n-tuple“ filters can be configured using tools like `ethtool`, specifically with the `--config-ntuple` option.

Drivers for NICs supporting multi-queue often include a kernel module parameter for setting the number of hardware queues. For instance, in the bnx2x driver, this parameter is `num_queues`. A typical RSS setup might involve one receive queue per CPU, assuming the device supports enough queues. If not, allocating at least one queue per memory domain (where a memory domain encompasses CPUs sharing a specific memory level) is common.

The RSS device's indirection table, which maps a queue based on a masked hash, is usually configured by the driver during initialization. By default, queues are distributed evenly, but the table can be accessed and modified using `ethtool` commands, `--show-rxfh-indir` and `--set-rxfh-indir`, for runtime adjustments like varying queue weights.

Each receive queue in a NIC corresponds to a unique IRQ (Interrupt Request). The NIC triggers the associated IRQ to notify the corresponding CPU when new packets arrive in a queue. PCIe devices use message-signaled interrupts (MSI-X) for this purpose, directing each interrupt to a specific CPU. The current queue-to-IRQ mappings can be viewed in `/proc/interrupts`. Typically, any CPU may handle an IRQ, but distributing these interrupts across CPUs is beneficial, especially considering the significant packet processing involved in interrupt handling. IRQ affinity can be manually adjusted, though systems with the `irqbalance` daemon, which optimizes IRQ assignments dynamically, might override manual settings.

RSS should be enabled when reducing latency is critical or processing receive interrupts becomes a bottleneck. Distributing the load across CPUs reduces queue length. Ideally, allocate as many queues as there are CPUs for low-latency networking, or the maximum number supported by the NIC if it's lower. For achieving high throughput, the most effective setting usually involves the minimum number of receive queues needed to prevent CPU saturation and queue overflow. This is because more queues can lead to more interrupts and increased workload with default interrupt coalescing settings.

2.4.2 Receive Packet Steering

Receive Packet Steering (RPS)[44] acts as a software-based counterpart to RSS (see Section 2.4.1). Functioning later in the data processing path, RPS is responsible for selecting the CPU for higher-level protocol processing post-interrupt. It does this by placing the incoming packet into the chosen CPU's backlog queue and then signalling that CPU for processing. Due to its software-level operation, RPS offers certain advantages over RSS. It is compatible with any NIC, allows easy integration of software filters for new protocols, and does not increase the hardware device interrupt rate.

RPS is activated during the latter part of the receive interrupt handling. When a driver sends a packet up the network stack using `netif_rx()` or `netif_receive_skb()`, RPS kicks in. At this juncture, the `get_rps_cpu()` function is called to determine the appropriate CPU queue for processing the packet.

The initial phase of CPU selection in RPS involves computing a flow hash based on the packet's addresses or ports, utilizing either a 2-tuple or 4-tuple hash, depending on the protocol. This hash is a consistent identifier for the packet's flow and is either provided by the hardware or computed within the stack. If the hardware is capable, it supplies the hash in the packet's receive descriptor, often the same hash used in RSS. The hash is stored in `skb->hash` and is used within the stack to represent the packet's flow.

Each hardware receive queue is associated with a list of CPUs, and RPS may direct packets to these CPUs for processing. The process involves calculating an index from the flow hash modulo the size of the CPU list, thereby selecting a CPU for packet processing. The packet is then added to that CPU's backlog queue. After completing the bottom half routine, Inter-Processor Interrupts (IPIs) are sent to CPUs with packets in their backlog queues, triggering backlog processing and further network stack processing.

RPS is available in kernels compiled with the `CONFIG_RPS` kconfig symbol, typically enabled by default in SMP systems. However, RPS must be explicitly configured to be active. The specific CPUs that RPS can route traffic to can be set individually for each receive queue via a corresponding `sysfs` file entry, implementing a bitmap of CPUs. RPS is disabled when set to zero; in such cases, packets are processed on the interrupting CPU.

In a single-queue device, a common RPS setup involves setting `rps_cpus` to the CPUs within the same memory domain as the interrupt-handling CPU. If NUMA concerns are negligible, it might encompass all system CPUs. However, at high interrupt rates, excluding the interrupting CPU from this map is advised to reduce its workload.

For multi-queue systems with RSS configured to align each hardware receive queue with a specific CPU, RPS may be unnecessary. Yet, if there are fewer hardware queues than CPUs, RPS can be beneficial. Configuring `rps_cpus` for each queue to include CPUs in the same memory domain as the queue's interrupting CPU can be advantageous.

RPS effectively distributes network traffic processing across multiple CPUs, avoiding packet reordering issues. However, this method can lead to CPU workload imbalances when different data flows have varying packet rates, especially if one flow heavily dominates. This is particularly noticeable in server environments with multiple active connections and could indicate configuration issues or Denial of Service attacks with spoofed source addresses.

The Flow Limit feature in RPS addresses this imbalance during high CPU usage by favouring smaller data flows. It activates when the CPU's incoming packet queue exceeds half its maximum length (`net.core.netdev_max_backlog`). The system tracks the number of packets per flow over the latest 256 packets. If a flow exceeds a set proportion (default 50%) when a new packet arrives, the new packet is discarded, while packets from other

flows are discarded only when the queue reaches maximum length. Below the critical threshold, no packets are dropped, maintaining connectivity even for large flows. Flow Limit is beneficial in systems with many concurrent connections, where a single connection occupying 50% of a CPU's capacity indicates an issue. In such cases, enabling Flow Limit on all CPUs handling network rx interrupts is advisable.

2.4.3 Receive Flow Steering

Receiver Packet Steering (RPS) steers packets based on their hash values, typically resulting in an evenly distributed network load. However, RPS does not consider the specific location of the application that is processing the packets. This limitation is addressed by Receive Flow Steering (RFS)[43], which enhances data cache efficiency by directing the kernel's packet processing to the CPU running the relevant application thread. RFS uses the same methods as RPS for queuing packets in a different CPU's backlog and signalling that CPU to initiate processing.

In RFS, the packet's hash is used as a reference in a flow lookup table rather than for direct forwarding. This table maps network flows to specific CPUs handling them. As mentioned in Section 2.4.2, the flow hash determines the table index. Each table entry records the CPU that most recently processed the flow. If an entry doesn't correspond to a valid CPU, the packet is handled using standard RPS methods. It's common for multiple entries in this table to refer to the same CPU, especially when numerous flows and limited CPUs are involved, leading to a single application thread managing multiple flows.

The `rps_sock_flow_table` is a global flow table that tracks the optimal CPU for various network flows, specifically the CPU actively managing the flow in user space. This table's values, which are indices of CPUs, are updated when functions like `recvmsg` and `sendmsg` are called during the execution of `inet_recvmsg()`, `inet_sendmsg()`, and `tcp_splice_read()`.

When a thread is moved to a new CPU while having pending receive packets on the original CPU, there's a risk of receiving packets in the wrong order. RFS counters this using an additional flow table, `rps_dev_flow_table`, unique to each hardware receive queue. This table contains two pieces of information for each flow: a CPU index and a count. The CPU index shows the current CPU where packets for the flow are queued for kernel processing. Ideally, both kernel and user space processing would occur on the same CPU. However, mismatches may occur if the scheduler moves a user-space thread to a new CPU while kernel packets are still queued on the previous CPU.

The `rps_dev_flow_table` records the backlog length of the current CPU when a packet from a particular flow was last added to the queue. A tail counter is derived by adding the queue's length to the head counter to locate the last item in the queue. Essentially, the `rps_dev_flow[i]` counter tracks the latest item added from flow `i` to the queue of the CPU currently assigned to handle flow `i`. Assigning flows to CPUs is based on hashing, and multiple flows can be directed to the same CPU queue.

A specific method is used for selecting the CPU for packet processing, as determined by the `get_rps_cpu()` function. This involves comparing the `rps_sock_flow` table, indicating the preferred CPU for a flow, with the `rps_dev_flow` table of the queue where the packet was received. If the preferred CPU matches the one currently handling the flow, the packet is added to that CPU's backlog. If not, the CPU is updated to match the preferred one under certain conditions:

- The queue head counter of the current CPU is greater than or equal to the tail counter value in rps_dev_flow[i].
- The current CPU is not set (indicated by being greater than or equal to nr_cpu_ids).
- The current CPU is offline.

After these checks, the packet is sent to the updated CPU. These rules aim to ensure that a flow is transferred to a new CPU only if there are no pending packets on the old CPU, preventing out-of-order packet arrival.

RFS is enabled if the CONFIG_RPS kconfig symbol is activated (default for SMP systems). However, it remains inactive until explicitly configured. The recommended number of flow entries is based on the expected number of active connections at any given time, usually much lower than the total number of open connections. Experience suggests setting rps_sock_flow_entries to 32768 for moderately loaded servers.

For a single-queue device, rps_flow_cnt is typically set to match rps_sock_flow_entries. For multi-queue devices, rps_flow_cnt for each queue is usually set to rps_sock_flow_entries divided by the total number of queues (N). For instance, with 32768 rps_sock_flow_entries and 16 receive queues, each queue's rps_flow_cnt would likely be 2048.

2.4.4 Accelerated Receive Flow Steering

Accelerated Receive Flow Steering (Accelerated RFS)[38] is akin to RFS (see Section 2.4.3), much like how Receive-Side Scaling (see Section 2.4.1) relates to Receive Packet Steering (see Section 2.4.2). It is a hardware-augmented method of load balancing that leverages soft state to route network flows based on the location of the application thread handling each flow's packets. Accelerated RFS is anticipated to surpass the performance of RFS by directly sending packets to a CPU that is either hosting the consuming application thread or is closely located within the cache hierarchy to the CPU of that thread.

To enable Accelerated RFS, the networking stack uses the driver's ndo_rx_flow_steer function to specify the preferred hardware queue for packets of a particular flow. This function is invoked automatically by the network stack whenever there is a change in a flow entry in the rps_dev_flow_table. The driver then employs a device-specific method to configure the NIC to route packets to the designated queue.

Each flow's choice of hardware queue hinges on the CPU information recorded in the rps_dev_flow_table. The network stack maintains a map that associates CPUs with hardware queues, which is constantly updated by the NIC driver. This map is a counterpart to the IRQ affinity table, viewable in /proc/interrupts. To construct this map, drivers can use functions from the cpu_rmap (CPU affinity reverse map) available in the kernel library. The map is organized so that for each CPU, the linked queue is processed by the closest CPU in terms of cache proximity.

The availability of Accelerated RFS is contingent upon the kernel being compiled with the CONFIG_RFS_ACCEL option and the support of the NIC and its driver for this feature. Moreover, it requires the activation of ntuple filtering via ethtool. The mapping of CPUs to their respective queues is deduced from the IRQ affinities set by the driver for each receive queue, thus negating the need for additional configuration.

Accelerated RFS is recommended for scenarios where RFS is desired, and the NIC is capable of hardware acceleration. It provides an effective method to optimize network packet routing in alignment with application thread processing.

2.4.5 Transmit Packet Steering

Transmit Packet Steering (XPS)[47] is a method employed in devices with multiple queues to intelligently select the appropriate transmit queue for sending a packet. XPS achieves this by maintaining two mappings: one linking each CPU to one or more hardware transmit queues and another associating each receive queue with specific hardware transmit queues.

XPS, using the CPUs map, is designed to allocate queues to a group of selected CPUs, ensuring that the completion of transmissions for these queues is handled by a CPU within this group. This strategy offers two key benefits: it significantly reduces competition for access to the device queue lock, as fewer CPUs compete for the same queue (this contention can be eliminated if each CPU has a dedicated transmit queue), and it decreases the rate of cache misses during transmit completion, especially for data cache lines that contain the `sk_buff` structures.

XPS, utilizing a receive queues map, chooses a transmit queue based on the configuration of a receive queue(s) map, as determined by the administrator. While it is possible to map multiple receive queues to multiple transmit queues, the most common scenario involves a one-to-one mapping. This setup facilitates transmitting and receiving packets on the same queue pairings, which is particularly advantageous for multi-threaded workloads using busy polling, especially when associating a specific CPU with a particular application thread is challenging. In such configurations, application threads, which are not fixed to specific CPUs, handle packets received on individual queues, with the number of the receive queue stored in the connection's socket.

Transmitting packets through the same transmit queue corresponding to the receive queue offers benefits such as minimizing CPU overhead. The work related to transmit completion is confined to the same queue association polling by the application, thereby avoiding the need to trigger an interrupt on another CPU. Additionally, when the application processes the packets during busy polling, it can handle transmit completion in the same thread context, reducing latency.

XPS is configured for each transmit queue by creating a bitmap that identifies which CPUs or receive queues are allowed to use that particular queue for transmission. Each network device computes and updates a reverse map linking CPUs to transmit or receive queues to transmit queues. When a flow's first packet is ready for transmission, the function `get_xps_queue()` is called to select a queue. This function matches the receive queue ID associated with the socket connection to entries in the receive-to-transmit queue lookup table, or it uses the ID of the currently active CPU to find a corresponding queue in the CPU-to-queue lookup table. If multiple queues match, one is chosen based on an index calculated from the flow's hash.

The transmit queue selected for a specific flow, such as a TCP connection, is stored in the flow's corresponding socket structure. This chosen queue is then consistently used for all subsequent packets in that flow to avoid the risk of packets arriving out of order. This strategy also spreads the cost of invoking the `get_xps_queues()` function across all packets within the flow. To maintain order and avoid out-of-order packets, the queue assigned to a flow can only be changed if the `skb->ooo_okay` flag is set for a packet within that flow, indicating that the flow has no pending packets and allowing the transmit queue to be changed without causing out-of-order delivery.

The responsibility for correctly setting the `ooo_okay` flag lies with the transport layer. For instance, TCP sets this flag when it has received acknowledgements for all the data in

a connection, indicating it's safe to switch the transmit queue without disrupting packet order.

XPS becomes available when the kconfig symbol `CONFIG_XPS` is activated, the default setting for SMP (Symmetric Multi-Processing) systems. If incorporated into the system, XPS's availability and specific configuration during device initialization depend on the driver. The mapping between CPUs/receive-queues and transmit queues can be viewed and set up through `sysfs`.

When dealing with a network device with only one transmission queue, configuring XPS has no impact because there's no alternative queue. However, it's ideal to configure XPS in systems with multiple queues so that each CPU corresponds to a specific queue. In scenarios where the number of queues matches the number of CPUs, creating a one-to-one mapping is possible, with each queue exclusively paired with a single CPU, thereby eliminating contention. Conversely, if the system has more CPUs than queues, the most effective strategy is likely to assign CPUs that share a cache with the CPU handling transmit completions (or transmit interrupts) for a particular queue to share that queue.

When choosing a transmit queue based on receive queue(s), XPS requires explicit configuration to map these receive-queue(s) to specific transmit queue(s). If the user-defined mapping for receive queues is not applicable or relevant, then the transmit queue selection will default to being based on the existing map of CPUs.

Chapter 3

Continuous integration network performance testing

The chapter delves into the intricacies and importance of implementing continuous integration (CI) in the agile software development landscape. It underscores the role of CI in streamlining the software development process, emphasizing its effectiveness in integrating regular updates, automating crucial tasks, and enhancing cost efficiency. The chapter also outlines the broader testing context within the Software Development Life Cycle (SDLC), discussing various testing stages, methodologies, and the importance of creating realistic testing scenarios. Moreover, it touches upon software benchmarking, an essential process in quality engineering. It concludes with an in-depth look at the performance testing of Network Interface Cards (NICs) and the Linux kernel network stack, highlighting their pivotal roles in network engineering and management.

3.1 Continuous integration

The widespread adoption of agile methodologies in software companies has led to a growing interest in Continuous Integration (CI) systems. These systems enhance the pace of software development and improve cost efficiency by enabling more frequent integration of software updates. This efficiency results from the automation of tasks like building, testing, and reporting test outcomes, which are crucial for identifying and resolving issues swiftly, thereby reducing development costs [53].

CI, an essential practice before software deployment and delivery, involves automated software construction and testing [22]. It supports expanding engineering teams' size and output capacity, allowing developers to work simultaneously and independently on different features. This approach enables quick and independent integration of these features into the final product. The subsequent sections detail notable open-source CI servers.

In CI environments, regression testing is key for a rapid and cost-effective method of validating and deploying new software updates. This improves fault detection and software quality [52]. The importance of regression testing arises from the need for immediate test feedback in CI, where test cycles are short. These cycles, subject to a time budget, vary in duration and include selecting relevant tests, executing them, and relaying the results to developers.

3.1.1 CircleCI

CircleCI is a purpose-built CI/CD (Continuous Integration/Continuous Deployment) platform designed to deliver speed and reliability in software development processes [8]. It supports both cloud-based and private infrastructure environments. The tool is tailored for DevOps workflows, providing a range of features to automate and enhance software development and deployment activities. CircleCI stands out for its efficiency in handling builds across multiple environments and its robust support for various programming languages and machine types.

CircleCI offers the flexibility to build on any machine type in any language. It allows teams to configure CPU and memory for optimal performance and increased speed. Additionally, workflows can be defined and orchestrated for building, testing, and deploying applications with complete control.

Built by DevOps professionals, CircleCI focuses on improving the entire development process. It includes insights for tracking status and monitoring duration, SSH debugging for quick problem resolution, advanced caching to speed up builds, and a VS Code extension for interacting with pipelines.

Moreover, it supports building with GitHub, GitLab, or Bitbucket and offers thousands of existing integrations or the option to create custom ones. It utilizes pre-built integrations (orbs) for services like Slack and AWS and provides webhooks and a full-featured RESTful API for additional customizable integrations.

The platform ensures the pipeline's security, supports authentication via OpenID Connect, and meets the rigorous standards of FedRAMP. It also ensures clean execution environments for jobs and allows the creation of policies for organizational compliance and standardization.

CircleCI offers cloud-based hosting with setup, security, and maintenance support, including using self-hosted runners. It also allows for installation on private servers, providing flexibility in hosting continuous integration.

Also, it is known for its fast performance, with features like smart test-splitting, optimized build speeds, and native docker layer caching. It can work with any machine type and language, offering seamless scaling.

3.1.2 Jenkins

Jenkins is an open-source automation server crucial in software development, especially in continuous integration (CI) and continuous delivery (CD). It stands as one of the most widely used automation tools globally. Written in Java, Jenkins is highly valued for its flexibility and extensibility, essential for automating different stages of software development [8].

Jenkins automates various phases of the software delivery process, integrating stages like build, test, and deployment seamlessly and efficiently. This integration enhances the efficiency and reliability of the development process, contributing to the stability of the final product. Jenkins utilizes user-defined „pipelines" to customize and control the CI/CD processes tailored to specific project requirements.

A key strength of Jenkins is its extensive plugin ecosystem. With over a thousand plugins available, Jenkins can integrate with nearly every CI/CD toolchain, making it highly adaptable to various environments and development scenarios. Its plugin architecture also supports custom plugin development to fulfil unique needs.

As a free and open-source tool, Jenkins is accessible to developers and teams of various sizes. Its platform-independent nature adds to its adaptability. Jenkins offers versatility in job creation, with both freestyle and pipeline options. However, its user interface is often outdated and less intuitive, particularly for CI/CD novices. It can be complex and challenging to configure and maintain. Additionally, Jenkins may experience performance issues, especially when overloaded with numerous plugins on a local server.

3.1.3 Travis CI

Travis CI is a well-regarded continuous integration service, notable for its effective integration with cloud-based repositories, especially GitHub. It automates the building and testing of software projects, playing a key role in continuous integration (CI) and delivery (CD) processes. A notable feature of Travis CI is its user-friendliness and quick setup, appealing to developers and teams of various sizes [8].

A prominent feature of Travis CI is its Build Matrix, which facilitates the parallel execution of different build stages, thereby improving testing efficiency. Each build operates in a separate virtual machine, ensuring a pristine and stable environment. This aspect is particularly advantageous for projects needing specific configurations or dependencies. Travis CI's robust integration with GitHub and compatibility with a broad spectrum of third-party tools, such as Coveralls and BrowserStack, enhance its adaptability to various development scenarios.

Beyond its primary functions, Travis CI boasts a comprehensive API, further expanding its utility in automating and customizing CI/CD workflows. The API enables various operations, including reading data and managing builds, which assists in integrating with other tools or bespoke workflows.

Despite these advantages, Travis CI presents some challenges. The platform may not be as straightforward for those new to CI tools. Configurations, mainly through the `.travis.yml` file can become complex for advanced setups. Furthermore, its reliance on cloud-based repositories may restrict its applicability to private or self-hosted version control systems projects. Nevertheless, Travis CI remains accessible and valuable for automating and optimizing software development processes, thanks to its open-source nature and extensive documentation.

3.2 Software testing

Testing is the process of evaluating whether a specific system fulfils its defined requirements. This process involves validation and verification to determine if the system meets user-defined needs. As a result, testing identifies discrepancies between actual and expected outcomes. In software, testing aims to identify bugs, errors, or unmet requirements in a system or software. Thus, it is an investigative activity informing stakeholders about the product's quality.

Software testing can be viewed as an activity based on assessing risks. During the testing process, it is crucial for testers to know how to condense a vast number of tests into a manageable set. They must also make informed decisions about which risks are critical to test and which are not [29].

Testing in software development is structured into various levels and steps, with different individuals responsible for testing at each level. Unit, Integration, and System Testing are

the three fundamental stages of software testing. Each stage is conducted by software developers or quality assurance engineers, also called software testers.

The testing stages mentioned are integral to the Software Development Life Cycle (SDLC). In software development, it's crucial to divide the process into modules, each assigned to different teams or individuals. Unit Testing is the first step, where developers test each module to ensure it meets expectations. The next phase is Integration Testing, combining independently developed modules, often revealing integration errors. The final stage is System Testing, which evaluates the entire software comprehensively. Software testing also ensures that integrated units do not negatively impact other modules. However, testing large or complex systems can be time-consuming, as testing every combination and scenario becomes more challenging, highlighting the need for more efficient and optimized software testing processes [20].

The testing cycle comprises several stages, starting with Test Planning and culminating in Test Result Analysis. Test Planning is the initial stage where the overall plan for the testing activities is formulated. This is followed by Test Development, where specific test cases are created for testing. The subsequent phase is Test Execution, involving the actual running of these test cases and reporting any bugs found in the Test Reporting phase. The final stage, Test Result Analysis, involves analyzing defects, typically by the developer or in collaboration with the client, to decide what needs fixing, enhancement, or modification [13].

The initiation of the testing process begins with the creation of test cases. These test cases are crafted using diverse techniques to ensure effective and precise testing. The primary techniques employed include Black Box Testing, White Box Testing, and Grey Box Testing [21].

White Box Testing is a comprehensive method that evaluates software's functionality and internal structure. It requires programming expertise for designing test cases, known as clear or glass box testing. Applicable across various levels, including unit, integration, and system testing, this method is also a form of Security Testing, ensuring data protection and maintaining functionality. White Box Testing scrutinizes all independent module paths, logical decisions, loop boundaries, and internal data structures. However, its complexity stems from the need for programming skills in testing.

Black Box Testing is a method that focuses on testing the application's functionality without delving into the details of its implementation. This technique is applicable at every testing level within the Software Development Life Cycle (SDLC). It conducts tests to cover all application functionalities, checking if they meet the user's initial requirements. This approach effectively identifies incorrect functionalities by evaluating them across minimum, maximum, and base case values. Renowned for its simplicity, Black Box Testing is a widely used testing technique globally.

Grey Box Testing merges the strengths of both White Box and Black Box Testing techniques. This approach enhanced testing effectiveness, allowing testers to be mindful of the application's internal structure. By understanding the inner workings, testers can evaluate the application's functionality more effectively, considering its interior architecture.

In the Software Testing Life Cycle's (STLC) initial phase, the Quality Assurance team reviews the software requirements, gaining a thorough understanding of the core requisites for the testing process. Should any discrepancies arise, coordination with the development team is essential to clarify and resolve these issues. The subsequent phase, test planning, is pivotal as it outlines the entire testing strategy. This phase focuses on creating a comprehensive test plan, serving as a fundamental guide for the testing process. A Test Plan

is an essential document primarily focused on the functional testing of an application. The testing process cannot proceed without this critical document [34].

In the test designing phase of the STLC, the creation of test cases takes place, and the test planning activities conclude. The QA team either manually writes suitable test cases or generates automated ones. Each test case details a set of test inputs or data, specific execution conditions, and anticipated results. The chosen test data should elicit expected outcomes and include deliberately flawed data to identify errors during testing. This approach is typically employed to determine under what conditions the application fails to function correctly.

The Test Execution Phase involves the test cases according to the previously prepared test plan. The test is cleared or passed if a functionality successfully passes this phase without any reported bugs. Conversely, every failed test case is linked to the identified bug or error. The output of this activity is creating a defect or bug report.

Test Reporting involves communicating the outcomes obtained from executing the test cases. This phase also includes reporting bugs, which are then conveyed to the development team for resolution.

The significant improvement in the testing procedure has guided it towards Test Automation. This involves continuous integration software to perform tests and compare actual outcomes with anticipated results. Test Automation is efficient in terms of time, as it reduces the need for time-consuming manual testing.

Test Automation is applied during both the implementation and testing phases. It supersedes manual testing by minimizing its necessity and revealing errors and deficiencies that manual methods might not detect.

Regression Testing, a key testing type, is time-consuming when performed manually. It primarily checks if software or applications function correctly after bug fixes, as error ratios can increase post-fixation. Automated test suites are compiled into a regression test suite to reduce time spent on regression testing. Additionally, Test Automation aids in early problem detection, thereby saving substantial costs and effort in later modification stages [33].

Furthermore, employing Test Metrics is critically important for significantly improving the testing process's effectiveness. These metrics are key indicators of efficiency, correctness, and analytical evaluation. Additionally, they assist in pinpointing areas needing enhancement and guide the necessary actions to address these shortcomings.

A significant challenge in the current testing process is aligning the testing approach with the developed application. Not all testing methods are suitable for every application. For instance, testing network protocol software versus an e-commerce application involves different complexities in test cases. This highlights the crucial role of human involvement in the testing process beyond mere dependence on existing test cases.

3.3 Software benchmarking

Benchmarking is a systematic approach in quality engineering and management, focusing on a comparative process to identify best practices for specific issues. Initially used to improve business processes, benchmarking involves examining industry standards, comparing them with an organization's practices, and formulating an action plan with clear objectives to enhance process quality and efficiency. Its relevance has expanded to scientific research and software engineering, driven by finding superior methodologies or algorithms for scientific

challenges and aiding various phases of software development, such as automated code testing [35].

Software benchmarking quantifies performance aspects like response time, throughput, and resource utilization. These metrics are crucial for evaluating software efficiency and effectiveness. The process typically involves comparing the software's performance with established standards or best practices, which may come from industry benchmarks, competitor analysis, or previous software versions. A key aspect of benchmarking is uncovering superior techniques or strategies employed by other software systems, often through research of competitors or industry leaders.

Unlike monitoring, benchmarking addresses specific questions and is often used to compare different versions, configurations, or deployments of a system in a non-production environment. This evaluation uses selected metrics during distinct moments across multiple test runs, with post-test analysis conducted offline. It can involve micro benchmarks, focusing on very detailed features, such as the performance of a single method. Benchmarking typically requires setting up the system on multiple machines, including dependent systems, and using a separate machine for measurement.

3.4 Performance testing

Developing an efficient network infrastructure critically depends on two main aspects: functionality and performance. 'Functionality' refers to what the network kernel allows regarding data transmission and processing capabilities, including data throughput, packet routing, and protocol support. On the other hand, 'performance' pertains to the kernel's efficiency and reliability in handling data, aiming for optimal transmission speeds and minimal packet loss, even in high network traffic or with limited hardware capabilities.

Pre-deployment kernel network performance testing is vital to preempt network failures due to performance issues. However, many teams face challenges due to the absence of specialized performance testing methodologies, leading to potential availability, reliability, and scalability issues in real-world network implementations.

Performance testing is critical for all network infrastructures, especially in sectors where network reliability and efficiency are paramount, such as telecommunications, internet service providers, cloud computing, and large-scale data centres.

Performance Testing, often called Load Testing, evaluates a system's responsiveness and stability under a simulated real-world workload. It involves using tools to replicate actual user behaviour to identify system limitations, aiming to assess scalability, availability, and overall system performance from both hardware and software perspectives. Key metrics monitored and analyzed include CPU and memory utilization, cache effectiveness, data integrity across various storage mediums, power usage, and network bandwidth.

Successful performance testing relies on several factors, ensuring the testing process identifies bottlenecks and provides actionable optimization insights. Creating realistic testing scenarios is paramount, simulating a user environment resembling operating conditions. This approach is essential for uncovering real-world performance issues that may not be evident in controlled testing settings [11].

A thorough analysis of both hardware and software components is also crucial. Performance testing should encompass the entire infrastructure, including network components, databases, and servers, to determine the source of performance issues, whether in application code, database queries, network setup, or a combination thereof. This is particularly evident in distributed network environments like the cloud [28].

Moreover, performance testing requires continuous monitoring and iterative improvements. It involves an ongoing cycle of testing, analyzing, optimizing, and retesting, ensuring sustained system optimization [6]. Effective data collection and analysis are also fundamental, as performance testing generates vast data volumes that must be efficiently analyzed to derive meaningful conclusions.

The benefits of thorough performance testing are vast, enhancing system reliability, user experience, and cost-effectiveness in the development lifecycle. By identifying and addressing performance issues early, costs and time spent on post-deployment fixes are significantly reduced. Performance testing also informs scalability planning and ensures high availability and business continuity in high-traffic situations [27].

In summary, performance testing is invaluable for enhancing user experience, system reliability, cost-effectiveness, scalability, and availability. It's an essential part of software development and network management, contributing significantly to the success and sustainability of technological systems.

3.5 Network performance testing

Performance testing of Network Interface Cards (NICs) and the Linux kernel network stack is a vital component of network engineering and management, reflecting the intricacy and significance of network performance in various applications.

NICs are the crucial link between a computer system and its network, handling data transmission. Therefore, testing NICs' performance is key to determining their efficiency in managing data flow. This includes evaluating throughput, latency, and packet processing capabilities, which ensure NICs can handle expected network loads, particularly in high-demand environments. Optimizing NICs for better data handling and reduced latency is necessary [36].

The Linux kernel network stack, responsible for routing, packet processing, and protocol management, is integral to effective network communication. Testing its performance is essential to ensure these functions are performed efficiently. This type of testing identifies bottlenecks in the stack, aiding optimizations to enhance data flow and processing, as discussed in Chapter 2.

Performance testing of NICs and the Linux kernel network stack extends beyond speed and efficiency enhancements. In network security, this testing is crucial for identifying the performance impacts of vulnerability mitigations.

The role of NICs and the Linux kernel network stack as core components of network infrastructure underscores the importance of their performance testing. This testing ensures optimal performance in terms of speed and efficiency and contributes significantly to maintaining network security. Insights from this testing are invaluable for improving and securing network systems.

Performance testing of NICs and the Linux kernel network stack is a specialized field within performance testing. This specialization stems from their unique characteristics and roles in network infrastructures.

NIC performance testing focuses on network-specific parameters like throughput, latency, packet loss, and error rates, requiring specialized tools and methodologies to simulate network traffic and monitor these metrics [50]. In contrast, general performance testing often revolves around user experience metrics such as load times or response speeds.

Performance testing of the Linux kernel network stack necessitates a deep understanding of low-level network operations and interactions. It involves assessing the effectiveness of the

stack in managing network traffic, processing protocols, and handling network congestion, often requiring kernel tuning and monitoring.

The implications and applications of this testing also differ. While general performance testing might aim at user satisfaction and application efficiency, testing NICs and the Linux kernel network stack informs network infrastructure design and management decisions. These tests are pivotal for network architecture and scalability planning.

Therefore, performance testing of NICs and the Linux kernel network stack is distinct due to its focus on network-specific metrics and operations. This specialization requires dedicated tools and an in-depth understanding of network protocols and kernel operations and has broader implications for network architecture and management. The complexity and critical nature of these components in network infrastructures necessitate a focused and nuanced approach to performance testing.

3.5.1 nepta framework

`nepta`[1] framework is a network performance testing automation tool. It provides easy extensibility, customization and relatively quick and simple test configuration. `nepta` contains APIs for direct interaction with the underlying operating system and various commands, such as work with network or filesystem. From the perspective of the testing cycle from Section 3.2, this framework takes care of the representation of test cases/scenarios and the whole Test Execution Phase.

Furthermore, the framework implements multiple workflow strategies for test environment preparation and configuration, environment synchronization between hosts and test result reporting. These strategies differ in network performance testing compared to other software performance testing.

Moreover, `nepta` is independent of the underlying performance testing tool, so users can extend it with their performance testing tools. However, it comes by default with support for `iperf3` and `netperf` tools.

Thanks to its tool abstractions, the framework allows for the programmatic creation of test scenarios from the test host configuration of networking and system packages, configuration synchronization between the specified testing host to the configuration of provided or custom test scenarios and their order of execution.

3.5.2 Catalog

Catalog is an internally developed solution for performance test result storage and test reporting. This tool provides functionality from the later parts of the testing cycle from Section 3.2, specifically test result monitoring, regression testing and bug reporting.

To take advantage of the test automation process, *Catalog* exposes REST API for performance test result storage, querying and searching.

The design of *Catalog* is tailored to meet the specific needs of performance testing, ensuring data integrity and efficiently retrieving test results. Specifically, the representation of the testing process parts like test suite execution, multiple test scenario runs with various settings and the ability to compare them.

Moreover, each execution of the tests can be complemented by metadata describing the configuration and test setup for future easier searching and comparing.

[1] https://github.com/rh-nepta/nepta-core

PerfQE Catalog

Search

	Timestamp	RHEL	Packages	Hosts	Test Scenario	UUID	Role
+	14 Jan 2024 21:55	RHEL-9.4.0-20231230.12	kernel-5.14.0-407.el9.x86_64		Standard	87dd7522-4434-4bc7-8133-afdd727e1cb6	Baseline Target
+	14 Jan 2024 03:27	RHEL-9.4.0-20231230.12	kernel-5.14.0-406.el9.x86_64		Standard	a5d3bba1-36c9-4d22-9f88-5170b46a0062	Baseline Target
+	13 Jan 2024 19:20	RHEL-9.4.0-20231230.12	kernel-6.7.0-68.eln134.x86_64		Standard	9b3b3a1e-7a8f-4cad-af16-5a31521f1cd7	Baseline Target
+	13 Jan 2024 08:58	RHEL-9.4.0-20231230.12	kernel-6.7.0-68.eln134.x86_64		Standard	038c5f83-4c61-4db3-a134-c8169f3961ca	Baseline Target
+	12 Jan 2024 17:27	RHEL-9.4.0-20231230.12	kernel-6.7.0-68.eln134.x86_64		Standard	9e6b9a24-a896-4ebc-b618-5a9ecf93dcbe	Baseline Target
+	12 Jan 2024 10:49	RHEL-9.4.0-20231230.12	kernel-5.14.0-402.el9.x86_64		Standard	6df5e564-03d4-42d6-9dd5-130e4a0e75ae	Baseline Target
+	12 Jan 2024 02:57	RHEL-9.4.0-20231230.12	kernel-5.14.0-407.el9.x86_64		Standard	7da66017-a9ae-4386-b621-be32265f1573	Baseline

Select All

HWInventory SWSetup Timeline

Figure 3.1: The example of the web interface of *Catalog* service.

Furthermore, *Catalog* is equipped with a user interface (see Figure 3.1), a feature that significantly enhances its utility. This interface allows for manual analysis and comparison of historical test data. The ability to manually review and analyze data is indispensable for developers and quality engineers, as it will enable exploring data beyond predefined analytical models or raw data reports.

Chapter 4

Testbed

This chapter describes the industry's leading Network Interface Cards (NICs): the Nvidia ConnectX-6 Dx, the Intel Ethernet Controller E810, and the Broadcom BCM57508. Each of these NICs stands at the forefront of modern networking technology, equipped with unique features and capabilities to meet the diverse needs of today's network infrastructures.

The Nvidia ConnectX-6 Dx exemplifies the combination of high bandwidth and advanced offloading features, aligning well with the demands of modern data centres, especially in supporting AI and machine learning workloads. In contrast, the Intel Ethernet Controller E810 is known for its versatility and adaptability, making it an ideal choice for various applications, from virtualization to cloud computing. Its sophisticated offloading features are key to enhancing network performance, marking it as a vital component in contemporary network design.

The Broadcom BCM57508 stands out as a strong competitor, offering features specifically designed for high-density environments and optimized for energy efficiency. This NIC represents the industry's shift towards more sustainable networking solutions that do not sacrifice performance.

For the evaluation in this thesis, these NICs will be tested on machines equipped with AMD CPUs from the Genoa and Milan generations, Intel CPUs from the Skylake, Ice Lake, and Sapphire Rapids generations, and ARM Altra, specifically see Table 4.2.

	CPU Family	No. sockets	No. of cores
#1	AMD Milan	1	32
#2	AMD Genoa	1	24
#3	Intel Skylake	2	12
#4	Intel Icelake	2	16
#5	Intel Sapphire Rapids	2	24
#6	ARM Altra	1	80

Table 4.1: Testbed CPU vendors and generations with number of cores in each socket.

	CPU Family	NICs
#1	Milan	ConnectX-6 / BCM57508
#2	Genoa	ConnectX-6 / Intel E810
#3	Skylake	BCM57508
#4	Icelake	ConnectX-6 / Intel E810
#5	Sapphire Rapids	ConnectX-6 / Intel E810
#6	ARM Altra	ConnectX-6 / Intel E810

Table 4.2: Testbed CPU and NIC testing matrix.

4.1 Nvidia ConnectX-6 Dx

NVIDIA's ConnectX-6 cards are distinguished by their advanced offloading capabilities [25], greatly enhancing network and data processing efficiencies. These smart NICs (Network Interface Cards) offload various tasks traditionally managed by the CPU, including network functions such as encryption, decryption, and packet processing. This offloading reduces the CPU's workload, freeing it up for other critical operations. This feature is particularly valuable in data-intensive settings like data centres, where optimizing processing and network speeds is essential. The ConnectX-6 cards' offloading abilities thus play a significant role in improving overall system efficiency and performance.

The card has state-of-the-art stateless offloading capabilities that boost network performance and efficiency. It supports stateless TCP/UDP/IP offloading, which handles network protocol processing directly on the card, thereby reducing CPU overhead. Large Send Offload (LSO) and Large Receive Offload (LRO) allow for more efficient management of large data packets, further reducing CPU strain. Additionally, ConnectX-6 provides checksum offload for improved error-checking and correction.

Enhanced features like Receive Side Scaling (RSS) and Transmit Side Scaling (TSS) optimize network traffic distribution across multiple CPU cores, ensuring balanced loads and increased throughput. RSS is also effective for encapsulated packets, maintaining performance in intricate networking environments. The cards also support network customization through VLAN and MPLS tag insertion and stripping and receive flow steering, which channels incoming traffic to specific cores or processes for optimized network resource utilization.

Moreover, the ConnectX-6 card supports Single Root IO Virtualization (SR-IOV) and VirtIO Acceleration. SR-IOV facilitates efficient I/O virtualization by allowing multiple virtual functions (VFs) to share a single physical hardware resource, thus reducing overhead in virtualized environments. It supports up to 1000 VFs per port, allowing extensive virtualization capabilities to share network resources among numerous virtual machines efficiently. The card also supports up to 8 Physical Functions (PFs), the primary interface for controlling VFs, ensuring robust resource management in virtualized settings.

The card also handles overlay network protocols like VXLAN, NVGRE, and Geneve through Encapsulation/Decapsulation. This feature is crucial for modern data centres, as it involves adding or removing network headers as data moves through network layers.

Additional features include the Data Plane Development Kit (DPDK) for Kernel Bypass Applications, which enables direct access to network data by applications and bypasses the kernel's network stack for faster packet processing. Open vSwitch (OVS) Offload using ASAP 2 offloads OVS processing to the network card, enhancing software-defined network

performance. Flexible Match-Action Flow Tables allow customized network traffic handling with programmable rules, increasing network flexibility and control. Lastly, the header rewrite supports hardware offloading of the NAT router, which improves network address translation efficiency by offloading it to the hardware.

4.2 Intel Ethernet Controller E810

The Intel Ethernet Controller E810 [19] is engineered to enhance network performance and efficiency across various applications significantly. It achieves this through a combination of advanced offloading capabilities, encompassing both generic and stateless offloads. The controller's standout features include:

- Receive-Side Scaling (see Section 2.4.1) efficiently distributes network traffic across different processor cores, enhancing processing efficiency and reducing bottlenecks.
- Data Center Bridging (DCB) feature improves network reliability and efficiency, especially critical in data centre environments.
- Customizable Queue Selection allows for queue configuration based on Source Address (SA), VLAN pairs, or individual SAs or VLANs. This customization enables optimal performance by allocating tasks more efficiently.
- Remote Direct Memory Access (RDMA) with supporting up to 32 Virtual Functions (VFs), the E810's RDMA capabilities reduce CPU load and accelerate data transfers between servers. Additionally, the controller can manage many Virtual Machines (VMs) and VFs, supporting up to 768 VMs and 256 VFs per device, making it ideal for high-density server virtualization.

In network virtualization and RDMA, the E810 stands out for its ability to offload stateless tasks, even for tunnelled packets – a critical feature for network virtualization. It supports iWARP and RoCE v2 protocols for RDMA, handling up to 256K Queue Pairs (QPs). However, when configured for more than 4-port operations, there are certain limitations in its RDMA support. It's important to note that the E810 does not support the Userspace Direct Access (UDA) feature, which is confined to iWARP connection setup and error handling within the kernel.

Designed for low-latency and high-throughput environments, the E810 can support up to 8x10GbE connections, particularly in the E810-CAM2 model. It integrates an optimized transmission scheduler to regulate traffic flow and prevent network congestion. The controller also supports IEEE 1588, ensuring precision time measurement essential for synchronized timing across network devices. Its enhanced support for the Data Plane Development Kit (DPDK) bolsters its capabilities in Network Functions Virtualization, advanced packet forwarding, and efficient packet processing.

For virtualization and Quality of Service (QoS), the E810 offers extensive capabilities in Network Functions Virtualization (NFV) and Network Virtualization Overlays (NVO), which are crucial for modern data centres and cloud computing. The Dynamic Device Personalization (DDP) feature provides a programmable pipeline adaptable to various network protocols and standards. It also supports multiple network virtualization overlay protocols and vSwitch Assist for improved virtual switch operations. The E810's QoS features, including Priority-based Flow Control, Enhanced Transmission Selection, and Differentiated

Services Code Point, facilitate effective traffic management and prioritization. Additionally, its support for Single Root I/O Virtualization (SR-IOV) with extensive queue pair and virtual station interface capabilities, along with programmable Virtual Ethernet Bridging (VEB) and Access Control Lists (ACLs), underlines its suitability for complex network environments.

4.3 Broadcom BCM57508

The BCM5750x series [9], comprising Ethernet controllers, is engineered to meet the specific needs of cloud and enterprise data centres. These controllers are highly efficient in handling big data, machine learning, web services, storage, and database applications. The Thor NIC family, part of this series, offers flexible single, dual, and quad-port configurations, providing up to 200/100 Gb/s aggregate throughput, setting a benchmark for power efficiency in the industry.

This section explores the stateless offload capabilities of these Ethernet controllers, demonstrating their efficiency in handling such tasks.

The controllers include Checksum Offloading for IP, TCP, and UDP, configured by host software to compute checksums as per RFC standards ([2], [3], and [1]). They identify the start of IP, UDP datagrams, and TCP segments in a frame, which varies depending on frame tagging (VLAN or LLC/SNAP encapsulation). After identification, checksums are calculated for the entire datagram and inserted into the protocol header, covering all frame types, including those with IP datagram options and TCP segments.

The UDP Fragmentation Offload (UFO), particularly in the Linux environment, allows large UDP/IP datagrams to be segmented into smaller packets, reducing CPU load in UDP applications.

For TCP, the controllers offer TCP Segmentation Offload and Large Send Offload (LSO), also known as Large Segment Offload. This feature enables large TCP messages to be divided into several TCP/IP packets, easing CPU strain in TCP applications.

Generic Receive Offload (GRO) and Large Receive Offload (LRO) provide hardware acceleration for receiving TCP data. These features, supported by Transparent Packet Aggregation (TPA), lower CPU load and boost throughput for TCP applications by aggregating TCP streams based on source and destination IPs and ports. GRO is preferred in TPA for its ability to maintain packet boundaries, which is essential for routing applications using LSO for transmission.

Header and Data Split functionality allows TCP/IP packets to be received with separate header and payload data buffers, improving TCP/IP performance. This feature, available in Windows and Linux, facilitates efficient header caching and advanced operations like page flipping and zero-copy by the host TCP/IP stack.

The controller's VLAN Tag Insertion and Removal supports IEEE 802.1Q-compliant VLAN tags in transmitted and received frames, including stripping VLAN tags upon reception.

Packet Steering and Receive Side Scaling (RSS) balance packet processing across multiple processors while ensuring in-order data delivery. Packets are processed on different CPUs/cores in parallel, using a Toeplitz algorithm for 4-tuple match and an indirection table for stream-to-CPU mapping. Symmetric RSS ensures consistent mapping of packets from a given TCP or UDP flow to the same receive queue.

Accelerated Receive Flow Steering (aRFS, or RFS) directs packets to queues based on CPU locality, reducing memory access latency and enhancing performance. aRFS steers flows based on n-tuple filters or the RSS hash, taking precedence over RSS.

Lastly, Data Center Bridging (DCB) includes protocols like DCBX, LLDP, ETS, and PFC designed for data centre environments. Broadcom Ethernet NIC controllers support these protocols, focusing on priority flow control.

Chapter 5

Test scenario design

Integrating Network Interface Cards (NICs) with the Linux kernel is critical for ensuring efficient network communication in various computing systems. Traditional performance testing methods for NICs often lack automation, have scalability issues, and cannot adapt to the rapidly evolving Linux kernel. This chapter introduces a new methodology designed specifically for continuous integration (CI) performance testing of NICs on the Linux kernel, emphasizing automated testing processes, scalability, and adaptability.

5.1 Overview

The proposed methodology for Network Interface Card (NIC) performance testing in the context of the Linux kernel is designed to address the unique challenges inherent in this environment (see Section 3.5). It focuses on automation, scalability, adaptability, comprehensive metrics, and integration with existing workflows to enhance the efficiency and accuracy of NIC testing. Based on the knowledge about general software testing from Section 3.2 is the proposed testing process architecture from Figure 5.1.

The cornerstone of this methodology is the development of an automated test suite. This suite is tailored to integrate seamlessly with the Linux kernel's continuous integration (CI) pipeline, minimizing manual intervention and accelerating testing. The Jenkins continuous integration tool (see Section 3.1.2) facilitates this automation. This tool can automatically trigger performance tests upon new kernel updates or NIC driver submissions, ensuring immediate and consistent testing. The automation also extends to the setup and teardown of test environments and the collection and initial analysis of test results. Jenkins could also be utilised for automated setup and teardown of the test environments. However, the Beaker[1] tool is used since it's better suited for this task.

A scalable testing infrastructure is essential to simulate various network environments and loads. This methodology will be evaluated on multiple servers equipped with NICs and CPUs mentioned in Chapter 4, to create a sufficient hardware matrix for testing. These servers are always in pairs with the exact hardware specifications connected back to back.

Adapting to the continuously evolving Linux kernel is critical to this methodology. AI/ML algorithms could be implemented to analyze past test results and predict potential impact areas in new kernel updates, allowing for proactive adjustments to the test suite. However, that is not the focus of this thesis, and continuous monitoring of kernel updates and development trends will suffice to keep the tests up-to-date.

[1] https://beaker-project.org/

A set of comprehensive and relevant performance metrics is established to evaluate NIC performance thoroughly. These metrics include throughput, local machine CPU utilisation, and remote machine CPU utilisation. Furthermore, there will be compound metrics of local machine throughput per core and remote machine throughput per core, referred to as local/remote efficiency later on, calculated as follows:

$$\text{efficiency} = \frac{\text{throughput}}{\text{number of utilised cores}}, \tag{5.1}$$

where local efficiency represents throughput divided by the number of utilised cores from the sender machine in a test and remote efficiency represents the same calculation with values from the receiver machine, resulting in the unit being throughput per core.

Throughput metrics will be measured using `iperf` and CPU metrics by `mpstat`. These simple tools are supplemented by `nepta` tool (see Section 3.5.1) for more specific and unique testing scenarios proposed in this thesis. The methodology emphasizes the importance of consistently and reliably measuring these metrics across different testing scenarios to ensure the results are actionable and comparable.

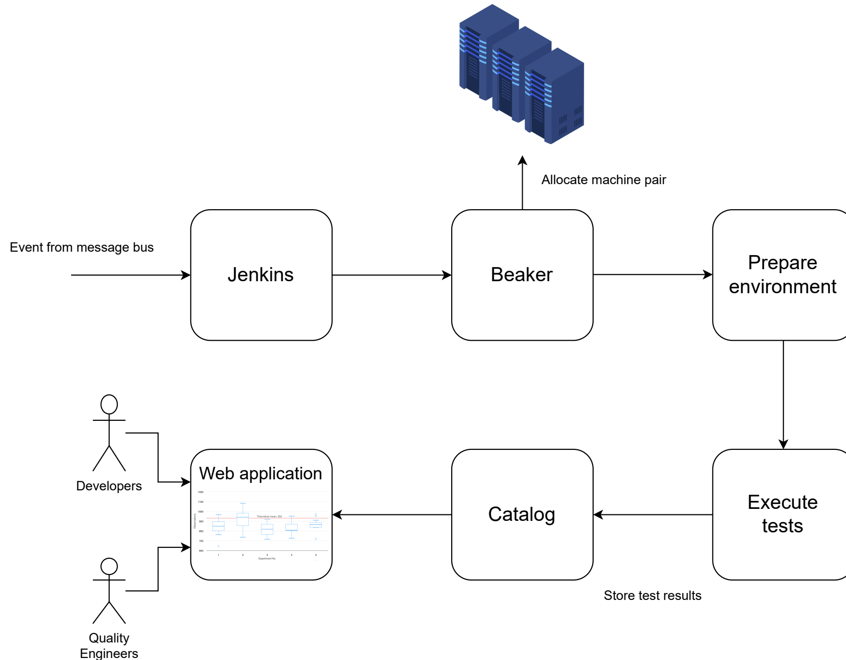


Figure 5.1: Architecture of continuous network performance testing pipeline.

An integral part of this methodology is establishing a feedback loop where test results are analyzed in-depth. This analysis informs the improvement of NIC hardware and the refinement of the Linux kernel. Continuous improvement is a key focus, with stakeholders, including developers, testers, and hardware engineers, reviewing results and deciding on subsequent actions. This collaborative approach ensures that improvements are comprehensive and align with the overall objectives of kernel development and NIC performance optimization.

As mentioned in Section 3.2, the whole testing process should account for test result storage and test reporting. This could be achieved by choosing a shelf solution. However, this thesis will utilize an internally developed solution named *Catalog*.

The design of *Catalog* and the functionality along with a user interface are explained in more depth in Section 3.5.2.

5.2 Test planning and test scenarios

As mentioned in Section 3.2, Test Plan creation is part of the initial testing phase with a focus on understanding the core requisites for the testing process, in this case, testing of NICs specifically from the testbed. First, it's essential to review the progress made over the past decade to understand the test scenarios in this section.

In the last decade, processor cores have reached a limit in clock speed, with no anticipated increases in CPU clock frequencies. Meanwhile, data rates in optical fibre networks have consistently risen, overcoming physical limitations such as scattering, absorption, and dispersion through improved optics and precision equipment. Despite these advancements at the physical layer, the system software's capacity for protocol processing remains a bottleneck. Consequently, efficient protocol processing and comprehensive system-level optimization are essential to enhance network throughput at the application layer.

TCP, known for its reliability and connection-oriented nature, ensures the sequential delivery of data from sender to receiver, concentrating most of the protocol processing tasks on the end system. Implementing TCP's functionalities requires a significant level of complexity, and as an end-to-end protocol, these functionalities are embedded within the end system. Consequently, enhancements to current TCP implementations typically fall into two categories. Firstly, offloads aim to integrate TCP functions with the lower layers of the protocol stack, such as hardware, firmware, or drivers. This integration is pursued to boost efficiency at the transport layer. Secondly, tuning parameters are designed to add more complexity to the upper layers, including software, system operations, and systems management.

All test scenarios in this thesis concentrate on affinity within tuning parameters. Affinity, or core binding, essentially involves deciding which resources are utilized on which processor in a networked multiprocessor system. The Linux network's New API for Networks (NAPI), see Section 2.3, facilitates two distinct operational contexts for the Network Interface Controller (NIC). The first is the interrupt context, typically implemented with coalescing. Here, the NIC signals the processor after receiving a specific number of packets. The NIC dispatches these packets to the processor using Direct Memory Access (DMA). Subsequently, the NIC driver and the operating system (OS) kernel continue processing protocol until the data is prepared for the application. The second context involves polling, where the kernel regularly checks the NIC for any incoming network data. When such data is found, the kernel processes it according to the network and transport layer protocols, delivering it to the application via the sockets API.

There are two kinds of affinity to consider. First is Flow affinity, which decides which core is designated to handle the network flow processing. Second, Application affinity identifies the core responsible for running the application process to receive the network data. Flow affinity will be configured using the `tuna` tool. On the other hand, Application affinity is established directly through `iperf` command argument `--affinity`, which handles this functionality.

As end-system processor architectures have shifted towards scaling out with multiple cores instead of scaling up in clock speed, system designers have encountered distinctive challenges in leveraging this parallelism for network-related processing. Various related yet fundamentally distinct methods have been developed to address these challenges.

Receive-side scaling (RSS), see Section 2.4.1, is a technology integral to NIC drivers, enabling multiqueue-capable NICs to harness the multiprocessing prowess of multicore end-systems. Specifically, RSS facilitates scheduling the interrupt service routine (ISR) on a designated core. This ensures that the same core which receives the data is also responsible for processing the interrupt, effectively synchronizing flow and interrupt affinity across most modern operating systems. NICs may incorporate RSS compatibility directly in hardware or through the driver implementation.

Parallel to RSS, Receive Packet Steering (RPS), see Section 2.4.2, is another significant kernel-level technology. RPS allows for the strategic selection of the CPU core to undertake protocol processing for incoming packets in a receiving end-system. This technology is instrumental in enhancing network throughput and balancing load across multiple CPU cores.

Building upon the foundations of RPS, Receive Flow Steering (RFS), see Section 2.4.3, adds an additional layer of control. RFS extends the capabilities of RPS by ensuring that the flow and application processing occur on the same logical core. This alignment is crucial for optimizing CPU cache utilization and reducing context-switching overheads, enhancing overall system performance in network-intensive applications.

Since all NICs from the testbed are capable of 100 Gbps+ and one core is no longer sufficient to fill this line rate, the test scenarios should rely upon the above-mentioned scaling optimizations as much as possible. Furthermore, it is desired to introduce compound metrics mentioned in Section 5.1, like throughput per core or throughput per watt, to capture better the efficiency of all the NIC offloads as well as prepare this methodology to keep up with the development of computing in the future.

Furthermore, as the number of cores in commodity multicore systems increases, the time required for resources to be accessed between these cores is no longer consistent. This change is due to the impracticality of maintaining uniform propagation speeds across core-to-core interconnects. This disparity primarily impacted memory access, classifying these systems as Non-Uniform Memory Access (NUMA) architectures. However, in contemporary high-speed computing environments, this non-uniformity extends beyond memory access, significantly influencing the movement of network and I/O data, given that the throughput of modern I/O and network devices is now comparable to that of memory.

Choosing cores that share the lowest level of cache hierarchy for network processing tasks is widely recommended. This approach, as discussed in various studies [15] [7], suggests that when a specific core, say core 1, is tasked with the protocol or interrupt processing, another core that shares the lowest level of cache with core 1 should be assigned to run the corresponding user-level application. By adopting this strategy, the number of context switches can be minimized, cache performance can be enhanced, and as a result, overall system throughput is likely to improve.

The new methodology aims to produce reproducible and well-comparable performance results, so the Linux `irqbalance` daemon will be disabled. Due to its utilization of round-robin scheduling for distributing the interrupt processing load across cores, this method can lead to suboptimal outcomes. As a result, a more strategic approach is necessary, one that allows for a more informed and controlled selection of cores specifically for interrupt processing tasks.

The design of Intel's processors features a direct connection between each CPU socket and its own PCI-Express bus. This architecture implies that specific PCI-Express slots are physically tied to a particular socket, limiting the flexibility in how these slots can communicate with different CPU sockets. Intel introduced the Ultra Path Interconnect

(UPI), a low-level inter-socket communication technology to address this limitation. UPI enables faster and more efficient data transfer between CPU sockets, enhancing overall system performance in multi-socket configurations.

In contrast, AMD's approach offers a different solution, particularly with their newer technologies. AMD employs its Infinity Fabric interconnect, which is a key feature in its EPYC processor lines. Infinity Fabric provides high bandwidth and low latency communication within the processor for core-to-core communication and between multiple CPUs in a server. This technology allows AMD processors to efficiently share resources and coordinate tasks across multiple sockets, effectively overcoming the limitations seen in earlier interconnect designs.

Lastly, protocols have been specifically tailored for the swift and reliable transfer of large data volumes within closed networks, such as those within or between data centres. Recently, the focus of these protocols has shifted towards facilitating direct data movement from the memory of one system to another. A notable modern instance of this technology is Remote Direct Memory Access (RDMA), initially paired with its original physical layer, InfiniBand. Typically, these protocols necessitate meticulously managed networks due to their specific addressing and routing requirements, leading to their predominant use in intra-datacenter traffic scenarios.

Taking into account the comprehensive details provided in this section and to evaluate the performance as well as the overall efficiency of transfers across various affinitization scenarios, all the metrics outlined in Section 5.1 will be measured in the following three test scenarios:

- **BestNode** represents a test scenario where the Flow affinity and Application affinity are pinned to the cores from the same socket on the CPU to which is NIC connected. For example, if the CPU has 18 cores, the Flow affinity might be pinned to cores 14-17, and the Application affinity might be pinned across 0-13 cores. This scenario aims to test various offloading features and the sharing of the lowest level of cache hierarchy, as mentioned above in this section.

- **NeighbourNode** is similar to the previous test scenario; one difference is that the Flow affinity and Application affinity are pinned to the cores from the different sockets on the CPU. For example, the CPU has 36 cores with two sockets. The Flow affinity might be pinned to a range of 0-7 cores, and the Application affinity might be spread across 18-24 cores. The test scenario aims to test performance when the application runs on a socket to which the NIC is not directly connected.

- **Unpinned** test scenario is the closest scenario to the real-world scenario where the system administrator only uses NIC without further fine-tuning the system. The Flow affinity is pinned on the cores of the socket to which the NIC is connected. Otherwise, the Application affinity is not pinned and is left to the system to decide.

The test scenarios demonstrate that Flow affinity is consistently assigned to the cores on the socket connected to the NIC. Moreover, Application affinity is deliberately set to different cores to avoid any overlap with Flow affinity. This approach is adopted to maximize efficiency and minimize any interference in the throughput metric that could occur if Flow and Application affinity coincide on the same core. Such a collision typically results in suboptimal performance outcomes and increased noise, which can potentially obscure genuine performance issues that may arise later.

5.2.1 TCP scalability scenario

In the unidirectional TCP test scenario designed for network performance evaluation, two machines are configured with network interface cards (NICs) connected directly (back-to-back configuration). This setup allows for controlled testing of TCP traffic flow in a single-side manner, meaning that data can flow in both directions but only in one direction. This scenario provides valuable insights into how the systems handle TCP traffic under constrained communication conditions. The packet size in this test scenario is 128KB.

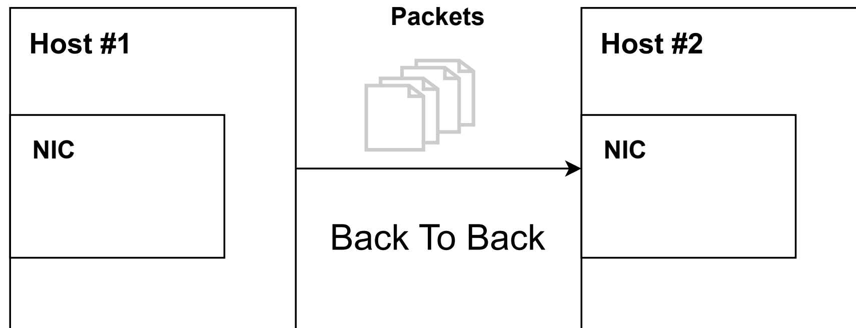


Figure 5.2: Diagram of back-to-back connected machine pair where packets are flowing only in one direction from server #1 to server #2.

The test uses IPv4 and IPv6 configurations to determine the impact of different IP protocols on the performance. Each configuration is tested with and without a Virtual Local Area Network (VLAN). VLANs are used to segment network traffic logically, offering an additional layer to test the network's ability to handle traffic with varying priorities and complexities. This segmentation is crucial for assessing performance differentials that may arise due to the overhead introduced by communication over VLAN.

5.2.2 Duplex TCP scenario

In the duplex TCP test scenario, the setup involves two machines configured with network interface cards (NICs) connected directly in a back-to-back configuration, similar to the previous unidirectional TCP test scenario. However, unlike the single-side scenario, where data flow is unidirectional, the duplex scenario allows simultaneous bidirectional data transfer. This means data can flow in both directions simultaneously, making it a more dynamic and representative test of network performance under typical communication conditions.

This duplex scenario enhances understanding of how network systems manage concurrent data streams, which is crucial for applications requiring real-time data exchange. It tests the network interface cards' ability to handle multiple data packets simultaneously, assessing throughput and efficiency in a more stringent and realistic network environment.

As in the previous section, this test uses IPv4 and IPv6 configurations to determine the impact of different IP protocols on performance. Each configuration is again tested with and without a Virtual Local Area Network (VLAN) to evaluate how VLANs affect the bidirectional communication dynamics. The use of VLANs in the duplex test scenario is particularly pertinent for understanding how network segmentation and the associated overhead influence the simultaneous handling of incoming and outgoing traffic.

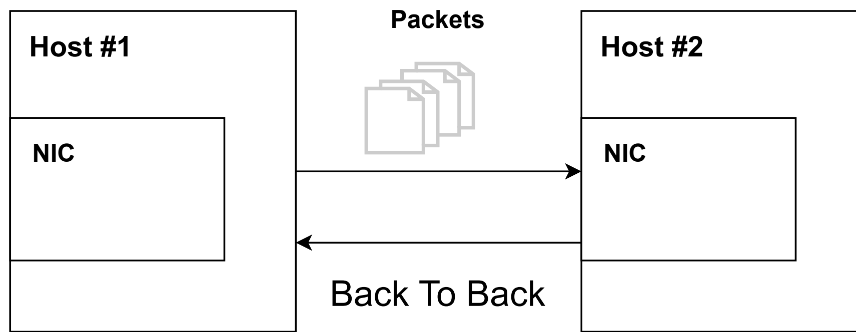


Figure 5.3: Diagram of back-to-back connected machine pair with duplex communication, where packets are flowing simultaneously in both directions between server #1 and server #2.

The duplex TCP test scenario thus extends the insights gained from the single-side TCP test by introducing the complexities and challenges of managing bidirectional network traffic, offering a comprehensive view of network interface card performance under typical operational conditions.

5.2.3 Offloading impact scenarios

Building upon the test scenarios outlined previously, a specific area of interest involves examining various offloads, particularly the TCP Segmentation Offload (TSO), a critical feature in network performance optimization. TSO allows a network interface card (NIC) to divide larger data packets into smaller segments, a task typically handled by the host processor, for more detail see Section 2.3.1. This offload is crucial as it can significantly reduce the CPU overhead of transmitting large packets over a network, enhancing overall system performance.

The test scenario for evaluating the impact of TSO will follow the same setup as detailed in the previous Section 5.2.1. By disabling TSO, we can assess the performance implications and gauge the efficiency of the NIC's implementation of this offload. This method provides a clear comparison between the NIC's performance with and without the offload, offering insights into how much processing work the NIC can offload from the CPU and the effect on network throughput and CPU load.

Other essential offloads like Transmit and Receive Checksum offloads will also be tested. These functions are integral to network operations, where the NIC handles the computation of checksums for outgoing and incoming packets, respectively (see Section 2.2.1 and Section 2.2.2). Typically, these tasks consume processor resources, so offloading them to the NIC can improve performance by reducing CPU utilization. The performance impact of disabling these checksum offloads will be evaluated to understand their role in network efficiency further.

The results from disabling these offloads—TSO, Transmit Checksum, and Receive Checksum—will provide valuable data on the network interface cards' effectiveness in handling what the CPU generally considers routine tasks. This analysis will help quantify the performance benefits these offloads bring to network operations and can guide system architects and network engineers in optimizing their configurations for maximum efficiency. Such in-

sights are crucial for environments demanding high network performance and minimal CPU involvement, ensuring optimal system responsiveness and stability.

5.2.4 UDP test scenario

Building upon the duplex TCP test scenario, the bidirectional UDP test scenario is designed to evaluate the maximum possible workload that can be handled by network interface cards (NICs) under intense network conditions. This setup also involves two machines with NICs connected directly in a back-to-back configuration, similar to the duplex TCP arrangement previously described. However, the key difference in this scenario is that the UDP protocol is used instead of TCP, catering specifically to applications that require high-speed data transfer without the overhead of connection management and error correction that TCP involves.

In the bidirectional UDP test scenario, both machines simultaneously send and receive UDP packets, creating a highly dynamic and stressful environment to measure how well the NICs can handle extreme levels of data packets without TCP's typical checks and balances. This test is crucial for understanding the raw throughput capabilities of the NICs, as UDP does not guarantee packet delivery, placing a greater emphasis on the network's ability to handle large volumes of data efficiently.

Similar to the duplex TCP scenario, this UDP test employs IPv4 and IPv6 configurations to determine the impact of different IP protocols on performance. Each configuration is tested with and without Virtual Local Area Network (VLAN) tagging. Including VLANs is particularly relevant for assessing how network segmentation and its overhead affect the NICs' performance under maximum UDP traffic load.

This bidirectional UDP test scenario rigorously assesses NIC performance, pushing the boundaries of data handling capabilities in a network. It extends the duplex TCP insights by introducing the challenges of managing high-speed, bidirectional UDP traffic, essential for applications such as live streaming, gaming, and other real-time services that rely on rapid data exchange. This scenario offers a comprehensive view of how network systems perform under high throughput demands, which is crucial for optimizing network infrastructure in high-demand environments.

5.3 Execution and Implementation

All the test scenarios are implemented using `nepta` framework (see Section 3.5.1) and can be run manually, but this thesis focuses on continuous performance testing, so let's go over the event path in the proposed architecture from Section 5.1

The whole process starts with triggering an execution by an event in Jenkins, which creates specific Beaker jobs on desired machine pairs. Next, it installs the preferred Linux distribution with the wanted kernel version and launches the preparation of the environment using the `nepta` framework. `nepta` then prepares the whole environment on local and remote end systems and launches `iperf` servers on the remote end system.

During the test execution phase, the testing framework systematically initiates all outlined test scenarios, deploying a varying number of `iperf` streams that range from 1 to N. This range allows for a comprehensive examination of network performance under different levels of load and concurrency. Each stream is executed repeatedly – precisely, 5 times – to enhance the accuracy and reliability of the test results. This repetition is crucial for

mitigating anomalies and ensuring the data reflects consistent performance patterns rather than one-off occurrences or outliers.

The size of the messages in each stream varies according to the specific test scenario being executed. For TCP test scenarios, the message size is set at 128KB, whereas for UDP test scenarios, it is significantly smaller, at 64 bytes.

Furthermore, the length of each `iperf` test is fixed at 30 seconds. This duration is significant for several reasons. Firstly, it is sufficiently long to ensure that system caches and network queues are fully utilized during the test, which is essential for accurately assessing the performance under typical operating conditions. Cache lines and queues are vital in overall system performance, affecting data transfer rates, latency, and CPU efficiency. A test duration of 30 seconds ensures that these components reach a steady state, thus providing a more accurate reflection of the system's capabilities.

Additionally, a 30-second test window allows for observing performance over a long period to factor in transient states and temporary fluctuations in network throughput or system resource utilization. This can include the initial ramp-up period, where connections are established and data flow begins, and any potential stabilization phase, where performance metrics level off.

After the test executions, a critical data aggregation and storage phase commences. All performance results generated from the tests are systematically collected and stored in a specially designed datastore, aptly named *Catalog* (see Section 3.5.2). This data store plays a crucial role in the overall testing process, as it serves as a centralized repository for all test data, ensuring that the information is organized, secure, and readily accessible for further analysis.

The configuration capabilities within the `nepta` framework have been enhanced, specifically by extending its configuration class to include additional attributes for interrupt settings and more refined CPU pinning strategies. These enhancements aim to improve the control and efficiency of test environment setups, particularly with how hardware resources are allocated and managed during network tests.

```
class UBenchPath(Path):
    cpu_pinning: Sequence[Sequence[Any]]

    def __init__(self, mine_ip, their_ip, tags, cpu_pinning, irq_settings):
        super().__init__(mine_ip, their_ip, tags, cpu_pinning)
        self.irq_settings = irq_settings
```

Listing 5.1: Example of the extended Path configuration from nepta framework

Instances of this extended class are pivotal in configuring the test scenarios effectively. They facilitate the specification of which Ethernet interfaces to use, along with corresponding IP and VLAN configurations. Additionally, they carry informative tags, which later aid in the analysis phase. The enhancements for CPU pinning and IRQ settings are particularly crucial as they directly influence application affinity and flow affinity, optimizing the allocation of computing resources to specific tasks or data flows.

An illustrative example of how these configurations are employed can be seen in the setup for a BestNode configuration, detailed in Section 5.2, using an IPv4 setup on Intel Sapphire Rapids processors. This example demonstrates the application of multiple streams, showcasing the versatility of the CPU pinning and IRQ settings in handling varying network load levels.

```
UP(
        host1.intf.eth.ice_0.v4_conf[0],
        host2.intf.eth.ice_0.v4_conf[0],
        [tag.ipv4, tag.ice, tag.speed_100g],
        [
            [(0, 0)], # one stream
            [(0, 0), (1, 1)], # two streams
            [(i, i) for i in range(0, 4)], # four streams
            [(i, i) for i in range(0, 8)], # eight streams
            [(i, i) for i in range(0, 14)], # fourteen streams
            [(i, i) for i in range(0, 28)], # twentyeight streams
        ],
        [
            ('ice', '1-27'),
            ('ice', '2-27'),
            ('ice', '4-27'),
            ('ice', '8-27'),
            ('ice', '14-27'),
            ('ice', '0-27'),
        ],
    )
```

Listing 5.2: Example of BestNode configuration on Intel Sapphire Rapids.

In this thesis, the proposed approach and the execution flow outlined in this section are encapsulated within a generic class called `UBenchGeneric`. This class is foundational for implementing and customizing various test scenarios as specified in Section 5.2. For instance, specialized extensions of this class, such as `UBenchBestNode`, `UBenchNeighbourBidir`, and `UBenchBestBidirUDP`, are designed to cater to specific testing needs. `UBenchBestNode` is tailored for optimal node configurations, `UBenchNeighbourBidir` is developed for NeighbourNode configurations in a duplex communication scenario, and `UBenchBestBidirUDP` is structured around UDP test scenarios, ensuring that each test scenario is thoroughly addressed with suitable configurations.

Furthermore, the `nepta` framework underpins these configurations and incorporates sophisticated capabilities for managing and toggling network offloads. At the initialization of the `EthernetInterface` class within this framework, offloads can be turned on or off depending on the test requirements. This feature is crucial for experiments that require a clean baseline to assess the impact of specific network functions on overall performance.

The configuration setup in the `nepta` framework is organized in a hierarchical tree structure, simplifying the management of complex settings and enabling precise adjustments across various layers of the network stack. An example of how such configurations can be manipulated is demonstrated in the following Python snippet:

```
def add_checksum_off_offloads_ethernet(conf):
    for eth in conf.get_subset(m_type=EthernetInterface):
        eth.offloads['rx'] = 'off'
        eth.offloads['tx'] = 'off'

        clone_and_modify(
            'NewBench',
```

```
        'NewBenchChksOff',
        add_checksum_off_offloads_ethernet
    )
```

Listing 5.3: Example of disabling transmit and receive offloads

In this example, a function named `add_checksum_off_offloads_ethernet` is defined to disable both transmit ('tx') and receive ('rx') offloads on Ethernet interfaces within a given configuration. This function iterates over a subset of configuration elements that match the type `EthernetInterface`, setting the offload settings for each interface to 'off'. The changes are then applied through a cloning process, creating a modified version of a test scenario, which is useful for comparisons against the original setup. This approach enhances the flexibility and adaptability of the testing framework and ensures that each test scenario can be precisely configured to reflect the most accurate and relevant results.

Chapter 6

Test scenarios evaluation

This chapter reports the proposed configuration results from Chapter 5 in various testing scenarios. Table 4.2 lists combinations of CPU and NIC details of the systems used to evaluate the approach and produce results presented in this chapter. The discussed results are obtained by comparing several generations of processors with various cards on many test scenarios built upon the proposed approach. The chapter presents throughput results and proposed compound metric efficiency from Equation 5.1, where local efficiency represents throughput divided by a number of utilised cores from the sender machine in a test, and remote efficiency represents the same calculation with values from the receiver machine.

6.1 Baseline TCP scenarios

As the first set of results is presented a baseline measurement of Nvidia ConnectX-6 and Intel E810 cards on the fourth generation of AMD Epyc processor using BestNode test scenario with IPv4 configuration from Section 5.2.1.

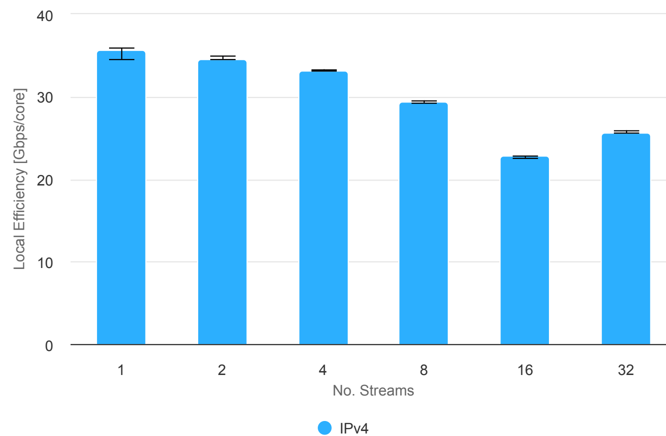


Figure 6.1: The bar chart illustrates the local efficiency (expressed as Gbps per core) of an IPv4 configuration on an AMD Genoa processor with Nvidia ConnextX-6 network interface card under varying numbers of streams, ranging from 1 to 32. Each bar represents the local efficiency observed at different stream counts, showing a generally high efficiency at lower stream counts, which slightly declines as the number of streams increases.

Figure 6.1 presents local efficiency measurements on the previously mentioned test scenario with the Nvidia ConnectX-6 card. It shows a trend where local efficiency scales almost linearly as the number of streams rises from 1 to 4, suggesting that parallel processing of network streams up to a certain point doesn't impact throughput efficiency per core. However, the efficiency starts falling at 8 streams, after which the efficiency begins to decline as the number of streams increases to 16 and further to 32. This decline could indicate the system is encountering resource constraints, such as CPU limitations or insufficient network bandwidth.

The remote efficiency holds steady as the number of streams increases from 1 to 4, which indicates that up to this point, the system scales well with additional streams, improving its remote efficiency. However, upon reaching 4 streams, the efficiency plateaus between 4 and 8 streams. Beyond 8 streams, efficiency dramatically decreases as the number of streams rises to 32.

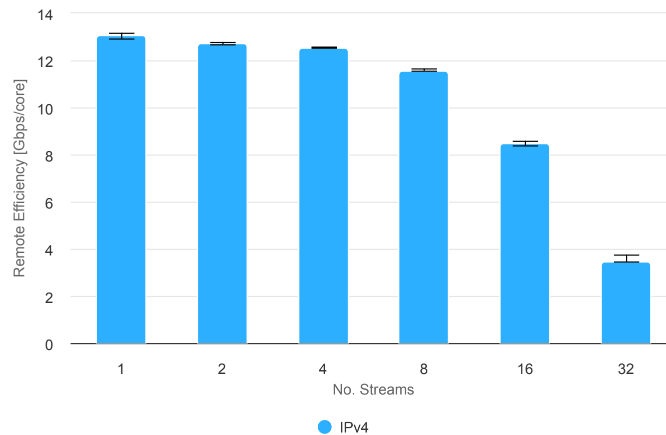


Figure 6.2: The chart illustrates the remote efficiency of an IPv4 configuration on an AMD Genoa processor with a Nvidia ConnextX-6 network interface card measured across different numbers of streams from 1 to 32. The remote efficiency, presented in gigabits per second per core (Gbps/core), demonstrates strong performance across 1 to 4 streams, maintaining close to peak values with slight variances. However, as the number of streams increases to 8 and 16, a progressive decline in efficiency is observed, culminating in a significant drop at 32 streams.

Finally, observing the throughput measurement, there's a clear trend of increasing as the number of streams grows, however when the stream count is 32, there's a pronounced drop in throughput. All these results point to interesting performance issues related to remote server efficiency, which are further discussed in Section 6.6. But, since these performance results are achieved similarly to all the others, including the same Linux kernel, they will be taken at face value.

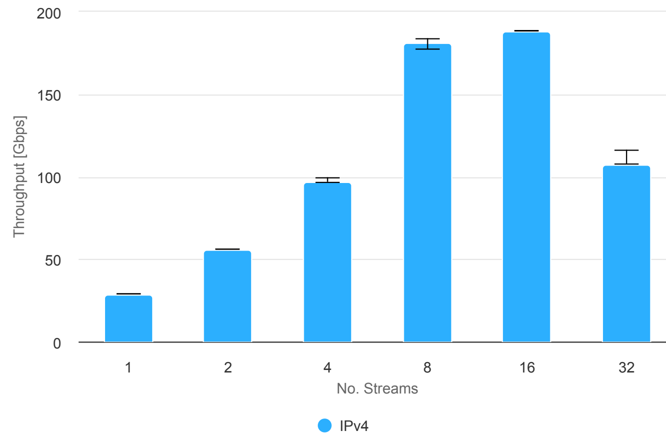


Figure 6.3: The chart presents the throughput performance of an IPv4 configuration measured in gigabits per second per core (Gbps/core) across different numbers of streams (1 to 32). Initially, at one stream, the throughput is relatively low. As the number of streams increases to 2 and then 4, there is a notable increase in throughput, demonstrating the network's capacity to handle higher loads effectively. The throughput peaks significantly at 8 and 16 streams. However, a decrease at 32 streams suggests a potential saturation point where the setup struggles with the load, leading to reduced throughput.

Now let's review the result from Intel E810 on the same setup in Figure 6.4. The local efficiency shows that as the number of streams increases from 1 to 4, the local efficiency decreases minimally. This indicates that the system effectively utilises its resources to process more data in parallel, increasing efficiency. The peak efficiency is observed at 4 streams, suggesting this is the optimal number of streams for this system configuration, as it achieves the highest throughput per core. After reaching the peak at 4 streams, there is a decline in efficiency when the number of streams is increased. This decline is due to the network bottleneck, the card's rate of 100 Gbps. However, until the line rate is saturated, ranges of 1-4 streams can be used to compare the two cards. Both local efficiency measurements indicate that cards at the same workload with IPv4 configuration scale similarly, however difference in median values is around 1.5 Gbps per core higher for Intel card.

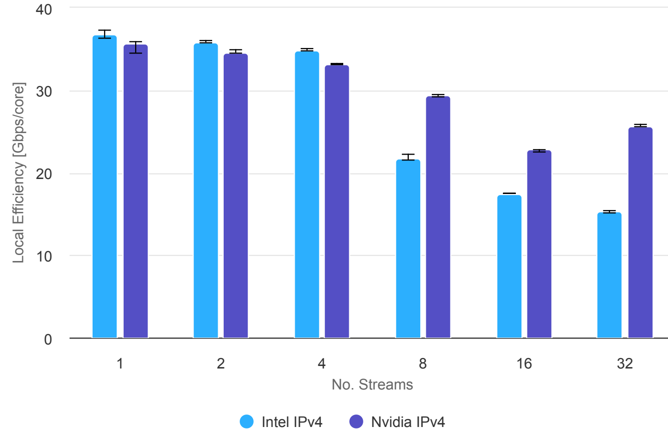


Figure 6.4: The chart illustrates the local efficiency in gigabits per second (Gbps) for Intel and Nvidia network cards with IPv4 under different numbers of streams ranging from 1 to 32. Both brands show a generally decreasing trend in efficiency as the number of streams increases, with notable drops at higher stream counts. Intel's performance remains consistently higher than Nvidia's across most stream configurations, particularly noticeable at lower stream counts (1 to 4).

Next, the focus will be on IPv4 with VLAN configuration shown in Table 6.1. These results for local efficiency appear to range between -4% and 1% compared to the results without VLAN. All the measurements show the same trends and the difference at the peak throughput is around 1%, which is at 16 streams. This minimal difference in peak performance indicates that the overhead introduced by VLAN tagging in an IPv4 environment is negligible in the context of the tested system. VLANs, often used for network segmentation and improving security, can sometimes introduce a slight overhead due to additional bytes in the frame for the VLAN tag. However, the observed data suggests that the network interface cards can handle VLAN encapsulation and de-encapsulation with little impact on the network throughput and efficiency.

		1	2	4	8	16	32
IPv4	Local efficiency	35.66	34.60	33.12	29.37	22.79	25.70
	Throughput	28.74	56.05	96.76	181.47	188.19	107.63
IPv4 VLAN	Local efficiency	-4%	0%	+1%	-2%	-1%	-4%
	Throughput	-2%	-5%	+2%	-2%	0%	+5%

Table 6.1: The table presents comparative data on local efficiency (as Gbps per core for IPv4 and fraction of IPv4 for VLAN) and throughput (in gigabits per second for IPv4 and fraction of IPv4 for VLAN) for IPv4 and IPv4 VLAN under varying numbers of streams (1, 2, 4, 8, 16, 32). IPv4 VLAN configuration differs from IPv4 in only a few percentage points.

As mentioned in Section 5.2.1, the last set of measurements is for IPv6 and IPv6 with VLAN configurations. The results discussed are presented in Table 6.2. Interestingly, the performance characteristics of IPv4 and IPv6 and IPv6 with VLAN are strikingly similar, suggesting that the underlying Nvidia ConnectX-6 card and software are equally adept at

handling both protocols. This parity is expected, as IPv6 is designed to be the successor of IPv4, retaining operational similarities while providing a larger address space and some additional features.

		1	2	4	8	16	32
IPv6	Local efficiency	34.68	34.97	32.79	29.03	21.88	24.12
	Throughput	27.87	53.17	95.93	176.26	185.14	116.09
IPv6 VLAN	Local efficiency	-1%	-4%	-4%	0%	-1%	0%
	Throughput	-1%	-2%	0%	-3%	0%	+3%

Table 6.2: The table presents comparative data on local efficiency (as Gbps per core for IPv6 and fraction of IPv6 for VLAN) and throughput (in gigabits per second for IPv6 and fraction of IPv6 for VLAN) for IPv6 and IPv6 VLAN under varying numbers of streams (1, 2, 4, 8, 16, 32). IPv6 VLAN configuration differs from IPv6 in only a few percentage points.

Moreover, the similarity in the IPv6, both with and without VLAN, throughout the range of streams reinforces the conclusion that VLAN configuration does not significantly impact performance. It reflects a state of optimization in the VLAN implementation that allows network traffic to be segmented without losing efficiency. It is critical for complex network infrastructure that relies on such segmentation for security and management purposes.

		1	2	4	8	14	28
IPv4	Local efficiency	66.21	58.18	33.00	27.84	22.74	19.18
	Throughput	45.11	72.73	94.03	94.04	94.06	94.15
IPv4 VLAN	Local efficiency	+2%	-2%	-1%	+2%	0%	-2%
	Throughput	-3%	+2%	0%	0%	0%	0%
IPv6	Local efficiency	+4%	+1%	+9%	-2%	0%	-2%
	Throughput	-14%	-11%	-1%	-1%	-1%	-1%
IPv6 VLAN	Local efficiency	+1%	-2%	+2%	0%	0%	-1%
	Throughput	-11%	-10%	-2%	-2%	-2%	-2%

Table 6.3: The table provides a detailed comparison of local efficiency and throughput for both IPv4 and IPv6 configurations, with VLAN configurations, across varying numbers of streams (1, 2, 4, 8, 14, 28). This table demonstrates the varying impacts of configurations, illustrating key performance trends within a Sapphire Rapids processor with the Intel E810 network card.

The networking hardware for the tests included Nvidia ConnectX-6 and Intel E810 cards. The results are methodically laid out in Table 6.3. This table is crucial as it encapsulates the empirical evidence of the performance metrics gathered. The results are compelling because they corroborate the pattern observed from prior measurements taken using AMD Epyc processors. These earlier measurements indicated no discernible performance impact when comparing IPv4 to IPv6 configurations, with or without VLAN. The absence of performance degradation between the protocols, irrespective of the VLAN configuration, is a significant finding. It highlights the efficiency of modern networking stacks and the proficiency with which they handle traffic, regardless of the complexity introduced by VLANs.

		1	2	4	8	14	28
IPv4	Local efficiency	49.99	46.67	34.07	23.17	18.76	18.17
	Throughput	40.08	72.26	94.02	94.03	94.05	94.14
IPv4 VLAN	Local efficiency	0%	-1%	-3%	-2%	-2%	-2%
	Throughput	-2%	-1%	0%	0%	0%	0%
IPv6	Local efficiency	+2%	+5%	-1%	-1%	-1%	-1%
	Throughput	-3%	-3%	-1%	-1%	-1%	-1%
IPv6 VLAN	Local efficiency	0%	+8%	-4%	-3%	-1%	-2%
	Throughput	-3%	-3%	-2%	-2%	-2%	-2%

Table 6.4: The table provides a detailed comparison of local efficiency and throughput for both IPv4 and IPv6 configurations, with VLAN configurations, across varying numbers of streams (1, 2, 4, 8, 14, 28). This table demonstrates the varying impacts of configurations, illustrating key performance trends within a Sapphire Rapids processor with the Nvidia ConnectX-6 network card.

Moreover, the findings of these tests are not isolated incidents but are consistently supported across all machine setups detailed in Chapter 4. This consistency across different platforms lends considerable weight to the results, underscoring the robustness of the network performance under varying conditions. It is also worth noting that the performance parity holds not only for Nvidia and Intel network cards but also for the Broadcom card that was part of the test suite. The extensive results from various machine configurations have been organized and presented in the Appendix A.1.

6.2 Duplex test scenario

Table 6.5 presents results from the test scenario described in Section 5.2.2 on the fourth generation of AMD Epyc processor using the BestNode test scenario.

Figure 6.5 displays data from a Duplex TCP test scenario using an IPv4 configuration, which shows similar results as the other configurations corresponding to findings from Section 6.1, comparing it to results obtained from that same section. A quick analysis of the data reveals a few trends and insights about network performance in these conditions.

The duplex test scenario data indicates that local and remote efficiency generally increases as streams rise from 1 to 8. This suggests that the system scales well with additional parallel streams, likely due to better utilization of available CPU resources and network bandwidth. Peak efficiency is achieved with 8 streams, beyond which there is a sharp decline as the number of streams increases to 16 and 32. The full utilization of CPU resources explains this downturn.

While not directly comparable to the Duplex scenario, the unidirectional stream TCP scenario from Section 5.2.1 shows a different pattern. The efficiency starts high and remains relatively stable as the number of streams increases. This steadiness may be attributed to the lack of parallel streams competing for resources, allowing for a more predictable and controlled utilization of the network and CPU.

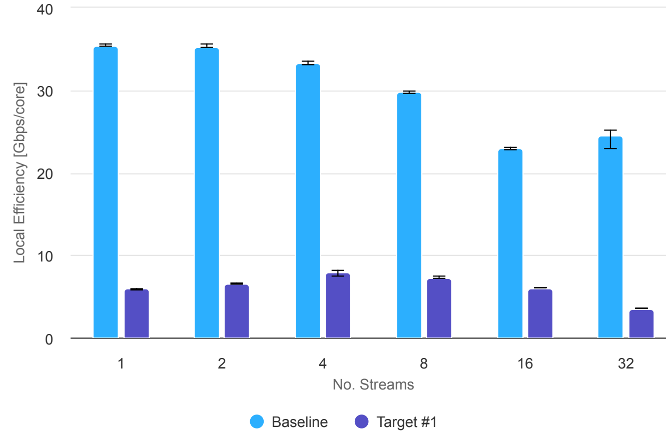


Figure 6.5: The chart compares the local efficiency in gigabits per second per core for two test scenarios across varying numbers of streams (1, 2, 4, 8, 16, 32). The blue bars represent the baseline one-directional test scenario, and the purple bars indicate the target bidirectional test scenario. The efficiency of the bidirectional tests is significantly lower, highlighting the increased computational demands of handling traffic in both directions simultaneously. The results are from an AMD Genoa processor with Nvidia ConnectX-6 NIC.

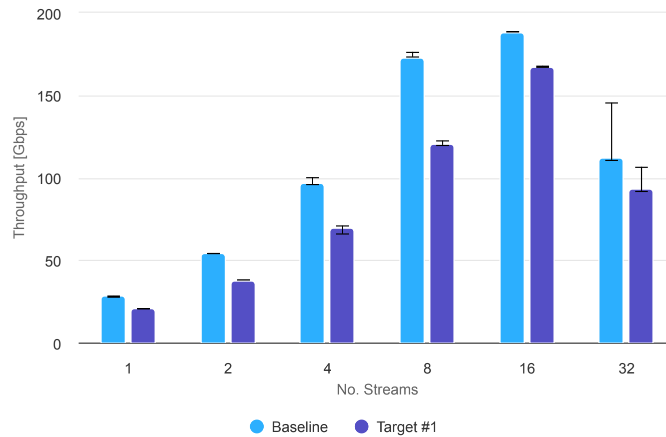


Figure 6.6: The chart displays the throughput, measured in gigabits per second (Gbps), for two test scenarios, one-directional (Baseline) and bidirectional (Target), across various numbers of streams (1, 2, 4, 8, 16, 32). This chart illustrates the impact of bidirectional communication on network throughput, highlighting the challenges and limitations of bidirectional data transmission under increasing loads. The results are from an AMD Genoa with Nvidia ConnectX-6 NIC.

The lower throughput in the Duplex scenario (see Figure 6.6) results from bidirectional traffic, creating more contention and complexity in packet scheduling and processing. Full-duplex communication requires the system to handle incoming and outgoing data simultaneously, which can lead to increased CPU usage and potential bottlenecks in the network

stack. Additionally, since full-duplex traffic doubles the number of packets (in and out), the associated handling overhead can become a limiting factor, especially as the number of streams grows beyond the optimal point. NICs' local and remote efficiency from Nvidia and Intel exhibit notable variations. For local efficiency, Nvidia NIC operates with an efficiency that falls between 14% and 26%, whereas Intel NIC displays a slightly higher efficiency range from 17% to 27%. When evaluating remote efficiency, they tend to be less affected and closer to their baseline capabilities. Nvidia NIC's remote efficiency ranges from 51% to as much as 90% of the baseline performance. Intel NIC similarly shows robust remote efficiencies, ranging from 47% to 85%. This performance metric suggests that although Intel NIC may offer marginally better local efficiency, Nvidia NIC consistently exhibit a stronger retention of baseline remote efficiency.

		1	2	4	8	16	32
Baseline Nvidia	Local efficiency	35.42	35.29	33.25	29.82	22.95	24.46
	Remote efficiency	11.04	10.95	12.35	11.23	8.87	3.57
	Throughput	28.23	54.28	96.91	173.17	188.19	111.86
Target Nvidia	Local efficiency	-84%	-82%	-77%	-76%	-74%	-86%
	Remote efficiency	-48%	-42%	-36%	-35%	-29%	-10%
	Throughput	-28%	-31%	-28%	-31%	-12%	-17%
Baseline Intel	Local efficiency	36.00	35.72	34.85	21.74	17.05	15.28
	Remote efficiency	13.04	13.01	12.15	7.99	5.59	4.48
	Throughput	24.12	48.73	89.42	94.06	94.12	94.29
Target Intel	Local efficiency	-83%	-80%	-80%	-73%	-76%	-76%
	Remote efficiency	-53%	-45%	-40%	-34%	-34%	-15%
	Throughput	-23%	-27%	-30%	0%	0%	0%

Table 6.5: The table compares local and remote efficiency and throughput measurements across a range of stream counts (1, 2, 4, 8, 16, 32) for both Nvidia and Intel network interface cards on AMD Genoa processors. The Baseline represents one-directional and Target bidirectional test scenarios. Target results are represented as fractions of the baseline measurements.

The next set of measurements presents collected data from the Intel Sapphire Rapids processor to understand its performance under the same test scenario configurations previously used. The results, as illustrated in Figure 6.7, show a distinct trend in efficiency based on the number of data streams processed. Initially, the efficiency is high when using a single stream but begins to taper off as additional streams are introduced. This decline can be attributed to the processor reaching its maximum throughput capacity, or line rate, at around eight streams. Beyond this point, adding more streams paradoxically reduces overall efficiency, as the system must allocate resources to manage the increased load, leading to diminished returns.

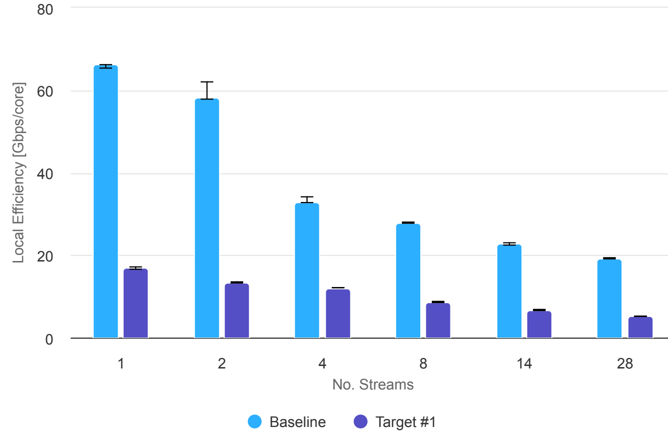


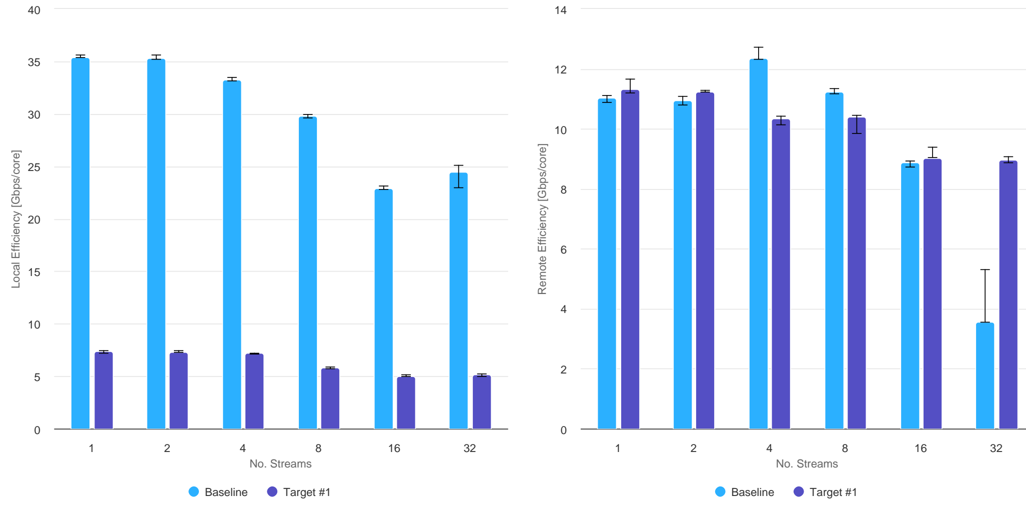
Figure 6.7: The chart compares the local efficiency, measured in gigabits per second per core, of the Intel E810 network interface card on the Intel Sapphire Rapids processor for two test scenarios, one-directional (Baseline) and bidirectional (Target). The tests were conducted across various streams (1, 2, 4, 8, 14, 28). This chart highlights the impact of bidirectional tests on the efficiency of network data handling.

The duplex scenario involving bidirectional traffic further complicates the efficiency dynamics. In this setup, the efficiencies recorded ranged from 23% to 36% for local efficiency and 58% to 83% for remote efficiency of the baseline measurements. This reduction is primarily due to the added contention and complexity in packet scheduling and processing when traffic flows in both directions. The duplex scenario inherently requires more sophisticated management of network resources, which challenges the processor's ability to maintain higher efficiency levels.

Comprehensive results from additional machines tested in this study are compiled in Appendix A.2, as referenced from Chapter 4. These results build upon the observations detailed above, with local efficiencies varying from 16% to approximately 37% of baseline performance and remote efficiencies 47% to 85%. It was observed that all machines reached the line rate when processing data streams, although the number of streams required to achieve this was higher than the baseline. Notably, some configurations could not attain line-rate performance on higher capacity 200Gb cards, highlighting the limitations of current hardware when dealing with extremely high data throughput demands. This detailed examination across different setups provides valuable insights into contemporary network processing units' scalability and performance.

6.3 NIC offloading

The design and execution of the baseline test scenario, as outlined in Section 2.3.1, provide a structured approach to evaluate the offloading capabilities discussed. This section delves deeper into the results from these tests, detailed further in Section 5.2.3 and illustrated in the forthcoming Figure 6.8. The tests utilized the fourth-generation AMD Epyc processor with the BestNode configuration on the Nvidia ConnectX-6 card, presenting a cutting-edge platform for assessing network performance and efficiency.



(a) Local efficiency

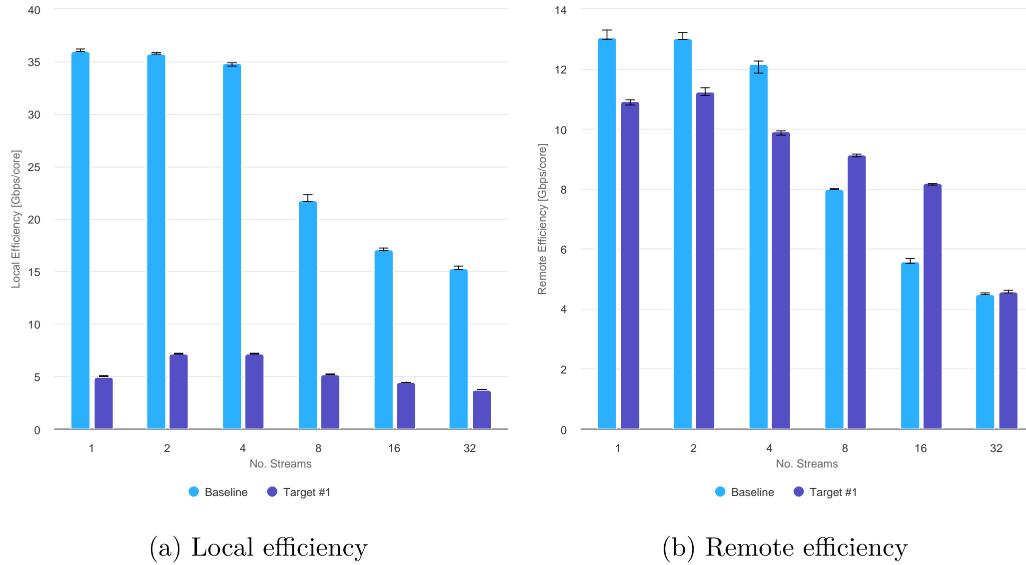
(b) Remote efficiency

Figure 6.8: The charts display local and remote efficiency in gigabits per second per core across varying numbers of data streams (1, 2, 4, 8, 16, 32) for network tests conducted under two different conditions: Baseline (TSO enabled) and Target (TSO disabled). The test setup comprises an AMD Genoa processor with a Nvidia ConnectX-6 network interface card. These charts underscore TSO's role in enhancing network performance, particularly in reducing CPU load and improving data handling efficiency on local machines, with variable impacts on remote machines.

The results present insightful findings regarding the performance metrics of local efficiency. It was observed that the trends in local efficiency metrics align closely with the baseline measurements. However, there is a notable disparity in performance levels. Specifically, the local efficiency metrics were significantly lower than the baseline, achieving only 20% to 21% of the baseline performance. This considerable reduction highlights the substantial importance of TSO offload for sender devices.

In contrast, the analysis of remote efficiency presents a different scenario. The impact on remote efficiency did not mirror the substantial decrease observed in local efficiency. Instead, the reduction in remote efficiency was relatively mild. The maximum drop recorded in remote efficiency was only up to 8%, as illustrated in Figure 6.8b. This figure visually represents the minimal decline in efficiency on the receiver end, suggesting a minimal performance impact of disabled TSO compared to the baseline.

To further explore the performance of network interface cards, a test was undertaken using the Intel E810 card. This experiment was conducted under identical conditions to those previously used for the Nvidia ConnectX-6. The findings from this test revealed that the Intel E810 card exhibited a local efficiency range similar to that of the Nvidia ConnectX-6, with values fluctuating between 14% and 25% of the baseline performance. This similarity suggests a consistent pattern of reduced local efficiency with disabled TSO offload.



(a) Local efficiency

(b) Remote efficiency

Figure 6.9: The charts display local and remote efficiency in gigabits per second per core across varying numbers of data streams (1, 2, 4, 8, 16, 32) for network tests conducted under two different conditions: Baseline (TSO enabled) and Target (TSO disabled). The test setup comprises an AMD Genoa processor with an Intel E810 network interface card. These charts underscore TSO's role in enhancing network performance, particularly in reducing CPU load and improving data handling efficiency on local machines, with variable impacts on remote machines.

However, the comparison of remote efficiency between the two cards highlighted a different trend. The Intel E810 card demonstrated less favourable remote efficiency performance than its Nvidia counterpart. Specifically, the Intel card experienced a performance drop of up to 20% in remote efficiency (see Figure 6.9b). This marked decline indicates a notable disparity in the capability of the Intel E810 card to maintain efficiency on the receiving end, setting it apart from the Nvidia ConnectX-6, which appeared to sustain closer to baseline levels under similar conditions.

In the next phase of the study, tests were conducted using the Intel Sapphire Rapids processor, employing the BestNode test scenario along with two types of network interface cards, the Nvidia ConnectX-6 and the Intel E810. These tests were designed to evaluate the performance nuances that might arise when varying the hardware within the same test parameters.

In an extensive evaluation of network interface cards (NICs) on different processors, significant variances in performance were observed, particularly between the Intel E810 NIC when used with Intel Sapphire Rapids and AMD Genoa processors. The results show that the local efficiencies of the Intel E810 NIC on the Intel Sapphire Rapids processor ranged from 30% to 40% of the baseline performance. This represents an improvement of approximately 15% to 20% compared to the same NIC's performance on an AMD Genoa machine. This difference underscores the impact that the choice of processor can have on the performance of networking hardware.

Regarding remote efficiency, the Intel E810 NIC also demonstrated robust performance, with only up to a 4% drop in efficiency. Comparatively, the Nvidia ConnectX-6 NIC showcased a broader range of local efficiency on Intel Sapphire Rapids, with performance metrics ranging from 30% to 50% of baseline performance. This suggests a higher peak

efficiency under optimal conditions than the Intel E810. The Nvidia ConnectX-6 also matched Intel's remote efficiency, with a performance decrease of only up to 4%.

		1	2	4	8	16	32
Baseline	Local efficiency	66.21	58.18	33.00	27.84	22.74	19.18
Intel	Remote efficiency	28.84	22.02	16.58	12.38	9.26	6.25
Target	Local efficiency	-68%	-69%	-60%	-60%	-59%	-58%
Intel	Remote efficiency	-4%	-2%	0%	0%	0%	0%
Baseline	Local efficiency	49.99	46.67	34.07	23.17	18.76	18.17
Nvidia	Remote efficiency	22.91	19.06	16.78	12.10	9.03	7.72
Target	Local efficiency	-65%	-68%	-60%	-53%	-49%	-52%
Nvidia	Remote efficiency	0%	+15%	+5%	-4%	0%	0%

Table 6.6: The table comprehensively compares local and remote efficiency for Intel and Nvidia under different configurations: Baseline and Target, where Baseline denotes TSO enabled, and Target signifies TSO disabled. This table effectively highlights the impact of TCP Segmentation Offload (TSO) on the network performance of Intel and Nvidia network cards with Intel Sapphire Rapids processor, showing varied effects based on local and remote server efficiencies.

The tests uniformly indicated that performance levels across all configurations were comparable to those recorded in the baseline scenario. Notably, there were no significant differences between the two network interface cards and across different network configurations. The results indicate that the IP stack configuration and the use of VLAN do not markedly influence the performance of these particular hardware setups.

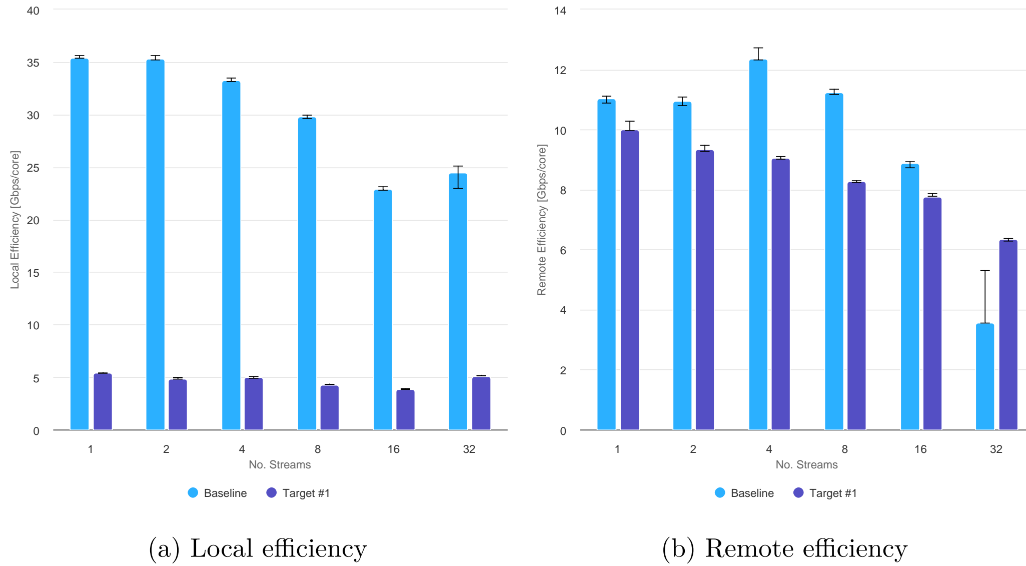
However, testing with older hardware architectures, as detailed in Appendix A.3, demonstrates a more predictable pattern where local and remote efficiencies fall below the baseline. This decline in performance is attributed primarily to increased CPU utilization, which averages about one core more than newer setups. These findings align with expectations that older systems with less efficient hardware will likely experience reduced performance due to higher demands on processing power.

Let's examine the network performance impact of another offload type as outlined in the test scenario from Section 5.2.3; the data presents a comprehensive view of the impact of checksum offloading on network efficiency. The results from this set of tests utilize the advanced fourth-generation AMD Epyc processor paired with the BestNode configuration for both Nvidia ConnectX-6 and Intel E810 network cards. Figure 6.10 focuses on the Nvidia ConnectX-6 card's performance under these conditions.

The performance metrics for the Nvidia ConnectX-6 card reveal significant insights into efficiency under TX and RX checksum offloading disabled. With the transmit checksum offload disabled, the local efficiency of the Nvidia card ranges between 15% and 20% of the baseline performance. This indicates a considerable reduction in efficiency, showcasing how critical the enablement of transmit checksum offloading is to maintaining higher local throughput and processing speed.

Moreover, with receive checksum offload disabled, the remote efficiency experiences a substantial drop, decreasing up to 23%. This drop is even more pronounced than the impact observed when TCP segmentation offload (TSO) is disabled. This result highlights a larger degradation in performance on both the transmit and receive sides, which aligns

with expectations given the increased processing burden placed on the CPU when checksum calculations are handled by the host rather than offloaded to the NIC.



(a) Local efficiency

(b) Remote efficiency

Figure 6.10: The charts depict the local and remote efficiency, measured in gigabits per second per core, under two different testing conditions across varying numbers of data streams (1, 2, 4, 8, 16, 32). They are compared between the Baseline, where RX and TX checksum offloads are enabled, and the Target, where these offloads are disabled. The results are from an AMD Genoa processor with a Nvidia ConnectX-6 card.

In a continuation of the network performance evaluation, further results have been analyzed for the Intel E810 Network Interface Card (NIC) under the same experimental setup as previously detailed. These results are systematically presented in Table 6.7 and reveal trends broadly consistent with earlier findings but with notably steeper performance declines.

The data on the Intel E810 NIC indicates that local efficiency ranges from 13% to 20%. This range suggests that the card performs at a lower efficiency than its baseline capability when operating under conditions where the transmit checksum offload is disabled. The decrease in local efficiency underlines the critical role that hardware-accelerated functions play in maintaining optimal operational performance, especially in scenarios demanding high data throughput.

Moreover, a more significant decline is observed in the remote efficiency, which drops by as much as 35%. This substantial reduction when receiving checksum offload is disabled points to the increased processing load transferred to the processor, highlighting a considerable degradation in the NIC's ability to handle incoming data efficiently under these conditions. This contrasts with the slightly milder impacts of similar configurations in previous tests on other hardware, emphasizing the Intel E810's sensitivity to disabling offloading features.

		1	2	4	8	16	32
Baseline Intel	Local efficiency	36.00	35.72	34.85	21.74	17.05	15.28
	Remote efficiency	13.04	13.01	12.15	7.99	5.59	4.48
Target Intel	Local efficiency	-87%	-87%	-85%	-82%	-80%	-79%
	Remote efficiency	-35%	-35%	-37%	-10%	+22%	-5%

Table 6.7: The table comprehensively compares local and remote efficiency across various stream counts (1, 2, 4, 8, 16, 32) for the Intel network card. The Baseline represents results with RX and TX checksum offload enabled, while the Target represents results with disabled offloads. The results are from the Genoa generation of AMD processor with Intel E810 card.

Tests were also conducted using the Intel Sapphire Rapids processor. These tests utilized the BestNode test scenario and employed two network interface card (NIC) types: the Nvidia ConnectX-6 and the Intel E810. The results of these tests are meticulously recorded in Table 6.8, providing a comprehensive view of each card's efficiency under specific conditions.

For the Intel E810 NIC, the findings indicate that the local efficiency ranges from 30% to 37% of the baseline performance. This level of efficiency represents a relatively smaller impact compared to the same card's performance on an AMD Genoa processor. Similarly, the remote efficiency experiences a decrease of up to 24%, which also presents a smaller impact compared to the results obtained with the AMD processor.

On the other hand, the Nvidia ConnectX-6 displayed a range of local efficiency between 27% and 47% of the baseline performance. Furthermore, the drop in remote efficiency was limited to up to 16%, indicating a stronger overall performance on the Intel processor compared to the AMD. These results underline that the Intel Sapphire Rapids processor may offer better compatibility and performance efficiency with high-end NICs.

		1	2	4	8	14	28
Baseline Intel	Local efficiency	66.21	58.18	33.00	27.84	22.74	19.18
	Remote efficiency	28.84	22.02	16.58	12.38	9.26	6.25
Target Intel	Local efficiency	-71%	-71%	-67%	-68%	-66%	-63%
	Remote efficiency	-19%	-15%	-15%	-20%	-14%	-11%
Baseline Nvidia	Local efficiency	49.99	46.67	34.07	23.17	18.76	18.17
	Remote efficiency	22.91	19.06	16.78	12.10	9.03	7.72
Target Nvidia	Local efficiency	-70%	-73%	-68%	-60%	-53%	-57%
	Remote efficiency	-14%	0%	-7%	-16%	-10%	-11%

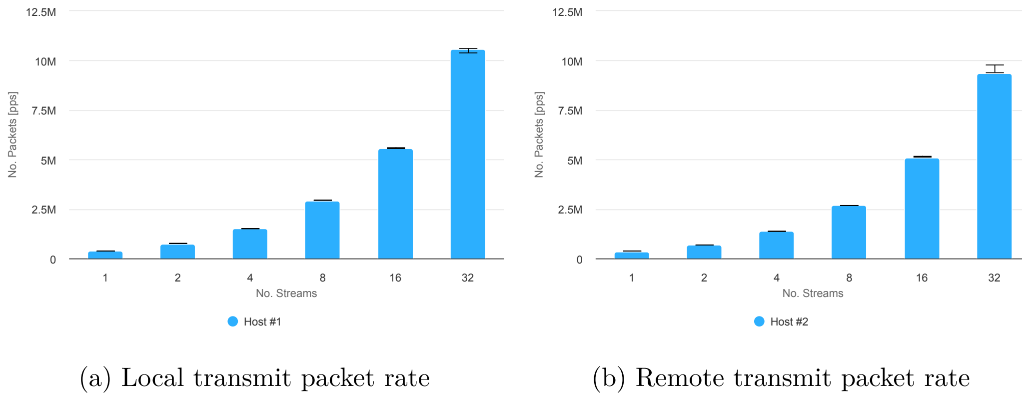
Table 6.8: The table comprehensively compares local and remote efficiency across various stream counts (1, 2, 4, 8, 16, 32) for the Intel network card. The Baseline represents results with RX and TX checksum offload enabled, while the Target represents results with disabled offloads. The results are from the Intel Sapphire Rapids processor with Intel E810 and Nvidia ConnectX-6 cards.

The comprehensive network interface card performance analysis continues with additional results detailed in Appendix A.4. Combined with those discussed earlier in this section, these findings consistently demonstrate an expected performance decline across different setups. Specifically, the local efficiency of the tested systems shows a notable drop, ranging from about 20% to 45% of baseline performance, depending on the specific

configuration and operating conditions. Furthermore, the impact on remote efficiency varies considerably, with changes spanning from a marginal 5% to as high as 40%. Notably, systems employing Intel processors exhibited less variability in performance outcomes than those using AMD processors. This suggests that Intel setups might offer a more stable and predictable performance environment, particularly under conditions that stress network throughput and efficiency.

6.4 UDP comparison

The subsequent results are derived from the UDP test scenario, detailed in Section 5.2.4, designed to place the highest possible workload on the network interface cards. This demanding test environment is critical for evaluating the cards under extreme operational conditions, pushing the hardware to its limits to assess its performance and scalability.



(a) Local transmit packet rate

(b) Remote transmit packet rate

Figure 6.11: The charts depict the transmit packet rates of two different hosts across varying numbers of data streams (1, 2, 4, 8, 16, 32). Chart (a) illustrates the local transmit packet rate for Host #1, which shows a significant increase in packet transmission as the number of streams escalates, peaking dramatically at 32. Conversely, Chart (b) represents the remote transmit packet rate for Host #2, demonstrating a similar trend of increasing packet rates with the number of streams, with a notable peak at 32. However, the rate increase appears more consistent and gradual than Host #1. These charts depict results from AMD Genora with Nvidia ConnectX-6 card on UDP test scenario.

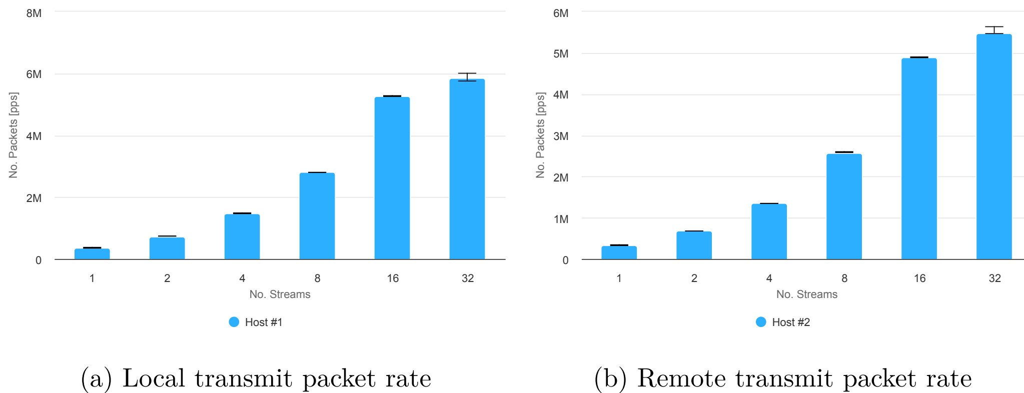
Focusing initially on the Nvidia ConnectX-6 and Intel E810 cards, these tests were conducted using the fourth generation of AMD Epyc processors within the BestNode test scenario configured for IPv4. The results, visually represented in Figure 6.11, detail both local and remote packet transmission rates. A notable observation from the data is the linear scaling of packet transmission with the increase in the number of streams. Specifically, the system peaks at approximately 10 million packets per second at the maximum number of streams, illustrating the processor's capacity to handle even greater loads potentially being limited by testing tool generation power. This trend underscores the robust capability of the AMD Epyc processor in conjunction with the efficient escalating network demands of both Nvidia.

Moreover, whether VLAN is employed or not, this linear scaling trend is consistent across all IP stack configurations. This uniformity in performance across different networking setups indicates that the hardware's efficiency and scalability are not adversely affected

by complexity variations in the network layer. Such consistency is crucial for network design, suggesting that systems can be configured for high throughput demands without significant performance degradation, regardless of IP stack and VLAN settings.

These results also align with findings from Section 6.1, reinforcing the observed behaviours and performance benchmarks established earlier.

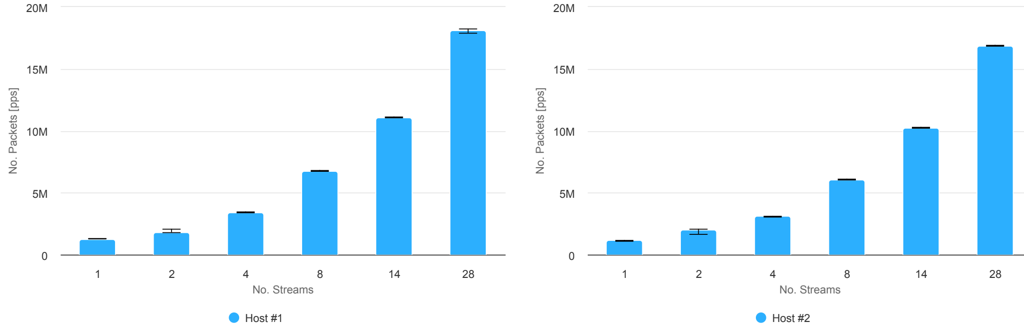
In a continuation of our evaluation of network interface cards under intense demand, the Intel E810 card demonstrates a comparable trend to its counterparts, with an increasing number of streams correlating directly to a higher number of packets sent per second (see Figure 6.12). This observation, similar to what was seen with the Nvidia ConnectX-6 card, reflects a linear scaling in performance. However, a critical distinction arises: the linear increase in packet throughput with the Intel E810 card holds steady only up to 16 streams, beyond which the growth plateaus. This suggests that the Intel E810 card, within this specific setup, begins to approach its maximum performance capacity when handling more than 16 streams, achieving around 55% of packet rate compared to the Nvidia ConnectX-6 card.



(a) Local transmit packet rate (b) Remote transmit packet rate

Figure 6.12: The charts illustrate the local and remote transmit packet rates measured in millions of packets per second (Mpps) for two hosts equipped with AMD Genoa processors and Intel E810 NICs across varying numbers of data streams (1, 2, 4, 8, 16, 32). In chart (a), Host #1 displays a progressive increase in local packet transmission rates as the number of streams increases, with a notable spike at 32 streams. Chart (b) shows similar trends for Host #2 in remote packet transmissions, where rates increase significantly as the number of streams rises, particularly achieving the highest transmission rate at 32 streams.

The results derived from testing on the Intel Sapphire Rapids processor, utilizing the BestNode test scenario with two different network interface cards, such as the Nvidia ConnectX-6 and the Intel E810, provide intriguing insights into network performance dynamics. Notably, the Intel E810 card demonstrated a high packet per second rate (see Figure 6.13). This anomaly has prompted a deeper investigation, which is extensively discussed in Section 6.6 to understand and possibly rectify what might be influencing these results.



(a) Local transmit packet rate

(b) Remote transmit packet rate

Figure 6.13: The charts show the local and remote transmit packet rates for two hosts equipped with Intel Sapphire Rapids processors and Intel E810 NICs, measured across various data stream counts (1, 2, 4, 8, 14, 28). Chart (a) shows Host #1's local transmit packet rates, which steadily increase as the number of streams rises, peaking impressively at 28 streams. Conversely, Chart (b) captures Host #2's remote transmit packet rates, which also show a progressive increase with the number of streams, culminating in a significant peak at 28 streams. These charts highlight the scalability of the Intel Sapphire Rapids with Intel E810 NIC setup in efficiently handling increasing network loads, both locally and remotely, showcasing the capability to manage large volumes of data traffic efficiently.

Additionally, a significant observation was made regarding the IP configuration's impact on performance, specifically concerning the use of VLAN. Tests showed configurations without VLAN could deliver about 1 million more packets per second at the peak of maximum streams compared to those with VLAN enabled. This suggests that VLAN implementation might introduce a performance degradation of up to 5%.

Lastly, comprehensive investigations across various machine setups detailed in Chapter 4 reveal that implementing VLAN configuration at maximum streams typically results in a performance decrease of approximately 5%. An exception is noted with the Broadcom card, where the performance impact is dramatically higher, nearly 50%. This significant discrepancy highlights the Broadcom card's unique sensitivity to VLAN configurations compared to its counterparts. Additionally, when comparing performance metrics head-to-head, the Nvidia ConnectX-6 consistently outperforms the other network interface cards, including the Intel E810 and Broadcom, as further elaborated in Appendix A.5. This superior performance of the Nvidia ConnectX-6 suggests its enhanced efficiency and robustness in handling high-load network scenarios, making it a preferable choice in environments where maintaining peak performance is crucial.

6.5 NUMA impact

As described in Section 5.2, the test scenarios involve multi-socket processor architectures, making it pertinent to assess the impact on performance using the diverse setups outlined in Chapter 4. The NeighbourNode measurement approach is notably relevant to Intel processors due to their architectural specifics and how they manage inter-processor communications and network traffic.

Detailed results involving the Inter Sapphire Rapids processors equipped with Nvidia ConnectX-6 and Intel E810 network interface cards are compiled in this context. These

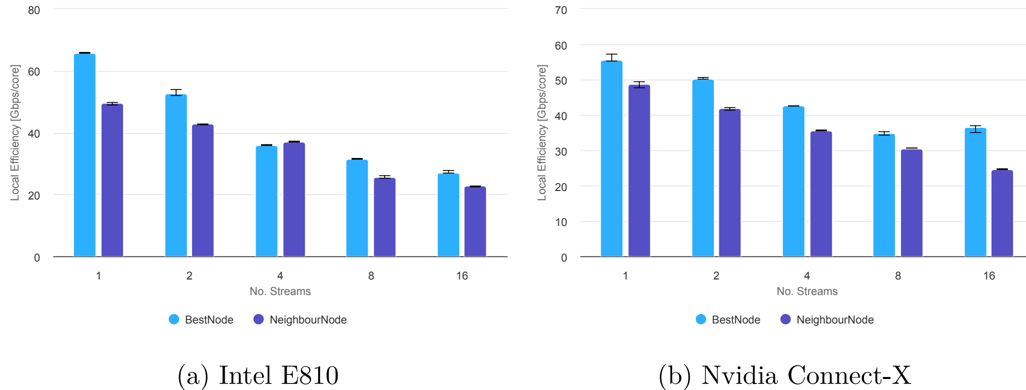
findings are scheduled for presentation in Table 6.9. When analyzing the data, it is evident that the BestNode configuration yields significantly better performance than the NeighbourNode configuration. Specifically, the Intel network card has an efficiency gain ranging between 5 to 8 Gbps per core. In contrast, using the Nvidia network card, the efficiency improvement is somewhat lower, between 3 to 5 Gbps per core. This differential highlights the varied efficiencies between network cards under similar test conditions.

		1	2	4	8	14	28
BestNode Intel	Local efficiency	66.21	58.18	33.00	27.84	22.74	19.18
	Remote efficiency	28.84	22.02	16.58	12.38	9.26	6.25
	Throughput	45.11	72.73	94.03	94.04	94.06	94.15
Neighbour Node Intel	Local efficiency	-24%	-17%	-13%	-21%	-18%	-22%
	Remote efficiency	-46%	-34%	-20%	-27%	-27%	-34%
	Throughput	-42%	-34%	-6%	0%	0%	0%
BestNode Nvidia	Local efficiency	49.99	46.67	34.07	23.17	18.76	18.17
	Remote efficiency	22.91	19.06	16.78	12.10	9.03	7.72
	Throughput	40.08	72.26	94.02	94.03	94.05	94.14
Neighbour Node Nvidia	Local efficiency	-12%	-10%	-20%	-12%	-3%	-17%
	Remote efficiency	-32%	-35%	-23%	-17%	-11%	-19%
	Throughput	-16%	-30%	0%	0%	0%	0%

Table 6.9: The table provides a detailed comparison of local and remote efficiency, along with throughput metrics, across various stream counts (1, 2, 4, 8, 14, 28) for two types of node configurations—BestNode and Neighbour Node—each utilizing Intel and Nvidia NICs. The Neighbour Node configurations demonstrate significantly lower efficiencies and varied throughput results, suggesting potential latency or resource-sharing impacts in these less optimal node selections. This table highlights the influence of Non-Uniform Memory Access (NUMA) configurations on network performance, showcasing the variable efficiency and throughput of Intel Sapphire Rapids processors across different operational scenarios and hardware setups.

Moreover, the Intel network card requires more streams to achieve the maximum line rate when configured under the NeighbourNode setup. This contrasts with the Nvidia card, which reaches the line rate with fewer streams. This observation underlines a recurring theme found in previous tests: Nvidia network cards generally offer better performance than their Intel counterparts, especially in configurations that demand high throughput from multiple network streams.

The next section of the analysis introduces performance metrics for the Intel IceLake processor paired with both Nvidia and Intel network interface cards, which will be illustrated in Figure 6.14. The initial focus is on the Nvidia card, which, in the BestNode configuration, demonstrates a significant local efficiency advantage, achieving approximately 7 Gbps per core more than its counterparts. Interestingly, this enhanced performance does not stem from lower CPU utilization; the Nvidia card registers higher CPU usage. Despite this, it still delivers superior throughput, leveraging the increased processing power to handle greater data volumes more efficiently.



(a) Intel E810

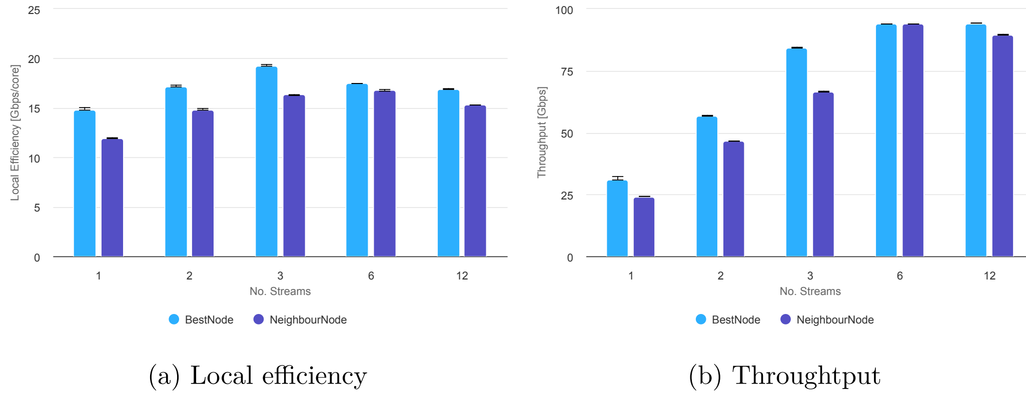
(b) Nvidia Connect-X

Figure 6.14: The charts compare the local efficiency in gigabits per second per core of two network interface cards (Intel E810 and Nvidia Connect-X) across various data streams (1, 2, 4, 8, 16) under two different configurations: BestNode and NeighbourNode. In the chart (a), the Intel E810 shows higher local efficiency for BestNode configurations across all stream counts, with noticeable performance degradation in NeighbourNode settings, particularly at higher stream counts. Similarly, chart (b) illustrates that the Nvidia Connect-X also performs better in BestNode settings, maintaining superior efficiency across the same range of data streams. These charts highlight the impact of optimal NUMA node selection on the performance of network interface cards, emphasizing the significant advantage of matching network hardware with the appropriate memory access configurations to optimize performance. These results were obtained on Intel Icelake.

This counterintuitive result, where higher CPU utilization correlates with better performance, suggests that the Nvidia card's architecture or driver optimizations might be more adept at processing large amounts of data at the correct socket. This could indicate a well-tuned system where the card and processor efficiently work together to maximize data handling and throughput despite the increased demand for system resources.

Meanwhile, the performance of the Intel card on the same Icelake processor offers insights that align with findings from other recent tests involving newer generations of Intel processors. Like its counterparts, the Intel card exhibits similar trends in terms of efficiency and throughput. This consistency across different generations of hardware suggests that Intel network cards maintain a stable performance profile, which is crucial for network planning and operations, ensuring that systems can be scaled or upgraded without unexpected deviations in network performance.

The final set of results, featuring the Intel Skylake processor equipped with a Broadcom network interface card, is detailed in Figure 6.15. These findings corroborate previously observed patterns, particularly highlighting that the local efficiency is reduced by approximately 20%. Notably, this performance drop is more pronounced in the NeighbourNode configuration, which fails to achieve the full line rate capacity of the 100Gb Broadcom card. This inefficiency is attributed to the maximum remote CPU utilization being reached, beyond which the system cannot process additional traffic.



(a) Local efficiency

(b) Throughtput

Figure 6.15: The charts present the local efficiency and throughput performance of a Broadcom Network Interface Card (NIC) on an Intel Skylake processor under two different configurations, BestNode and NeighbourNode, across various numbers of data streams (1, 2, 3, 6, 12). In chart (a), the local efficiency in gigabits per second per core is shown for both configurations. BestNode consistently outperforms NeighbourNode across all stream counts, highlighting the benefits of optimal NUMA placement in enhancing data processing efficiency. Chart (b) demonstrates throughput in gigabits per second, with a similar trend where BestNode configuration maintains superior performance, particularly at higher stream counts.

This situation underscores a significant limitation when older processor architectures manage high-bandwidth network traffic. The Skylake processor, despite its capabilities, struggles under the demands of 100 Gb throughput, particularly in less optimized configurations like NeighbourNode. This indicates a mismatch between the network card's capabilities and the processor's ability to handle such high levels of data flow effectively without specific configuration adjustments.

The observation that these older architectures require configuration optimization, such as switching to a BestNode setup, to manage high traffic effectively points to the necessity of tailored system configurations to harness the full potential of high-capacity network cards. BestNode configurations, which optimize processing by strategically managing data flow and CPU utilization across the system, can help mitigate these bottlenecks and improve the network's overall efficiency.

The results from various network interface card tests reveal both the capabilities and limitations inherent in each model, emphasizing the critical role of selecting the optimal configuration tailored to the specific demands of the processor architecture and the anticipated network load. This selection process is vital in complex environments that utilize multi-socket processor architectures, where the goal is to maximize per-core network throughput. The insights from these tests provide invaluable guidance for optimizing network infrastructure, ensuring that each component is appropriately matched and configured to handle the expected data volumes efficiently. Such optimization helps achieve the best possible performance, reduces bottlenecks, and enhances the overall reliability and responsiveness of the system.

Further analysis underscores that while local efficiency is an important metric, the more significant limitation often stems from the remote machine's CPU utilization, which, when maxed out, can severely restrict overall network throughput. High CPU utilization on the remote side indicates insufficient processing power to handle incoming traffic at optimal

speeds, causing a drop in throughput and efficiency. This finding points to the necessity of focusing on the network cards and local configurations and ensuring that the remote setups are equally capable of handling the projected loads. Managing and synchronizing the capabilities of both local and remote resources is essential for maintaining smooth and efficient network operations, especially in high-demand scenarios where data transmission rates are critical.

6.6 Performance impact of mitigations

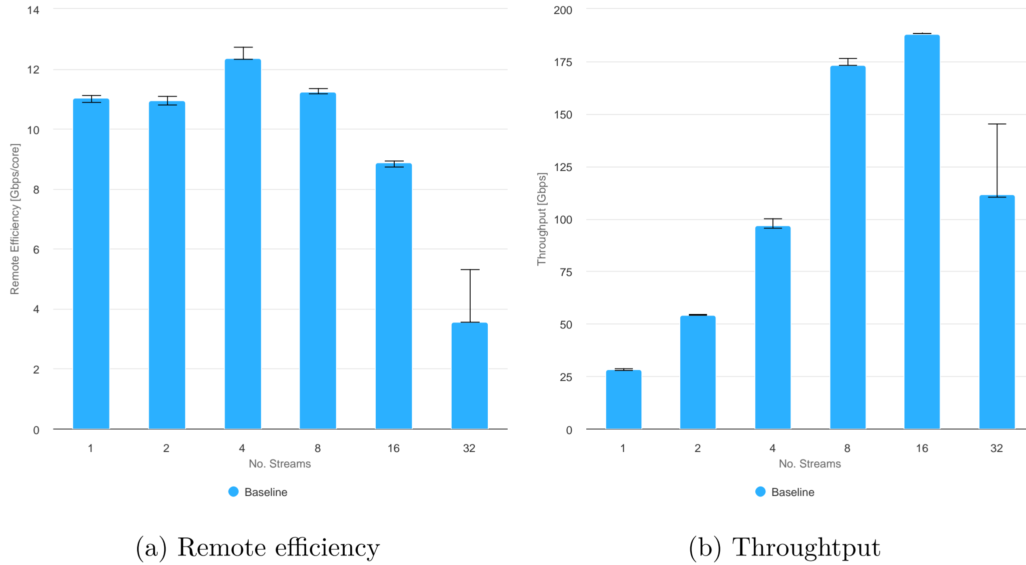
As mentioned in Section 6.1, comparative analyses between AMD and Intel processors revealed notable differences in stability, particularly under high loads. This instability is especially evident in configurations using AMD processors, where increased CPU utilization on the receiving side was observed during the tests.

More specifically, when utilizing the fourth-generation AMD processor paired with an Nvidia ConnectX-6 network card, the impact of escalating data streams on remote efficiency was monitored. Initially, as the number of streams increased from 1 to 4, the system's remote efficiency remained relatively stable. This suggests that up to this point, the AMD processor and Nvidia card combination could handle the increased load without significant performance degradation.

However, the situation changed markedly when the number of streams increased. As the streams expanded from 8 to 32, a noticeable decline in efficiency began to manifest. This degradation in performance became more pronounced with each increase in stream count, indicating a direct correlation between the number of streams and the strain placed on the system's resources.

The culmination of this trend was observed when the system reached 32 streams. At this level, the throughput experienced a significant drop (see Figure 6.16), which was attributed to the remote server reaching its maximum processing capacity. This was evident from utilising all 32 AMD processor cores, which became a bottleneck.

The investigation into a specific performance issue began with a straightforward approach, focusing on developing test scenarios and conducting continuous evaluations across different kernel versions. In this case, the evaluations centred on kernel version 5.14.0-284. Subsequent comparisons were made with what was suspected to be a problematic kernel, version 5.14.0-408, as shown in Figure 6.17. These comparisons revealed that while there were observable performance improvements in scenarios involving a smaller number of streams, the same could not be said for configurations with a higher number of streams; these did not experience the same degree of performance degradation.



(a) Remote efficiency (b) Throughtput

Figure 6.16: The charts display the remote efficiency and throughput of a network interface card (NIC) across various numbers of data streams (1, 2, 4, 8, 16, 32). The chart shows relatively stable performance across lower numbers of streams but sees a significant drop at 32 streams. The results were collected on an AMD Genoa processor.

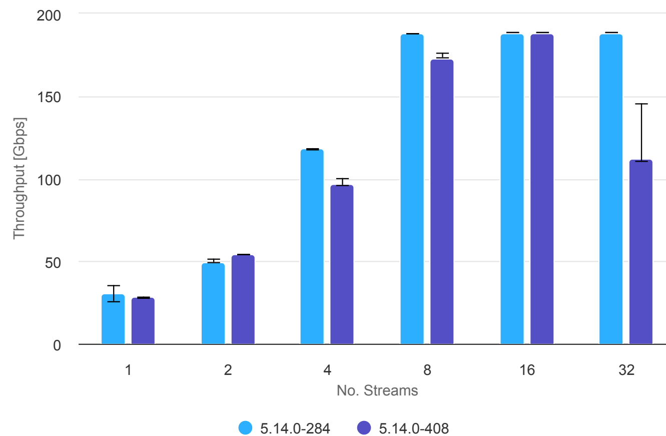
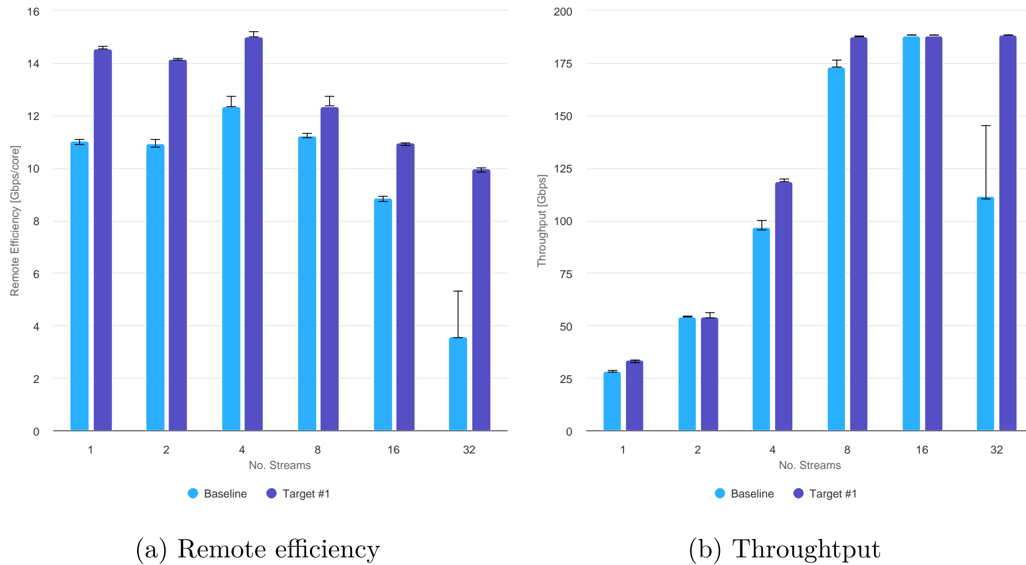


Figure 6.17: The bar chart compares the network throughput in gigabits per second across various numbers of streams (1, 2, 4, 8, 16, 32) for two different versions of a kernel: 5.14.0-284 (blue bars) and 5.14.0-408 (purple bars). The throughput generally increases with the number of streams for both kernel versions, indicating scaling efficiency. However, the newer kernel version (5.14.0-408) tends to perform similarly or slightly below the older version (5.14.0-284) at lower stream counts but shows a notable decline in performance at 32 streams.

This difference in performance led to the hypothesis that a specific change introduced between kernel versions 284 and 408 was responsible for the observed discrepancies. Further in-depth analysis confirmed this hypothesis, identifying the cause of the performance regression as a vulnerability mitigation mechanism, formally named the Speculative Return

Stack Overflow (SRSO)[5], colloquially known as AMD Inception. This mitigation aimed to enhance security but at the cost of performance under certain conditions.

The mitigation can be selectively disabled using the kernel parameter `spec_rstack_overflow=off`, or more comprehensively through `mitigations=off`, depending on the desired scope of vulnerability protection versus performance optimization. A subsequent set of measurements, referenced in Figure 6.18, provided empirical support that disabling this SRSO mitigation alleviated the performance degradation. This discovery underscored the delicate balance between securing systems against potential vulnerabilities and maintaining optimal performance, particularly in environments with critical high throughput and low latency.



(a) Remote efficiency

(b) Throughput

Figure 6.18: The charts compare the performance of network throughput and remote efficiency across various numbers of data streams (1, 2, 4, 8, 16, 32) under two different configurations: Baseline, with Speculative Return Stack Overflow (SRSO) mitigation enabled, and Target, with SRSO mitigation disabled. In the chart (a), remote efficiency, measured in gigabits per second per core, is consistently higher in the Baseline setup across all stream counts, suggesting that disabling SRSO mitigation helps maintain higher data efficiency, particularly noticeable at higher streams (16 and 32). Chart (b) shows throughput in gigabits per second, where the Baseline configuration generally outperforms the Target, except at 32 streams, where performance drops significantly, indicating possible performance limitations due to the mitigation's overhead. These charts illustrate the trade-offs between security enhancements through mitigation techniques and network performance, highlighting the potential impact on throughput and efficiency under varying operational loads.

6.7 Summary

The analysis presented in Section 6.1 indicates that utilising different IP versions and implementing VLAN results in a negligible performance impact. This suggests that all configurations tested are adequately equipped to handle these network settings effectively.

Further insights from the Duplex test scenario detailed in Section 6.2 reveal that while the computational demands for handling bidirectional communication are significantly greater than those for unidirectional communication, modern processors are capable of achieving a throughput of 100Gb in both directions within this testing framework.

The findings from Section 6.3 elucidate the consequences of disabling TCP Segmentation Offload (TSO), which leads to a reduction in local efficiency by approximately 60% to 70% compared to when TSO is enabled. Conversely, the impact on remote efficiency is negligible. Additionally, the performance across network interface cards from different manufacturers remains consistent, irrespective of the vendor.

Measurements obtained from scenarios where both transmit (TX) and receive (RX) offloads are disabled show that the performance penalties mirror those observed when TSO is disabled. Notably, this includes a tangible effect on remote efficiency. The decrease in performance is relatively uniform across different network interface card brands.

In Section 6.4, the findings reinforce the earlier observation from Section 6.1 that variations in IP configurations, whether with or without VLAN, have a minimal effect on performance. However, it is noted that the primary bottleneck in this scenario is attributed to `iperf3` [51], as its performance metrics do not align with those achievable using DPDK[1], suggesting that the testbed is capable of much higher performance when processing is offloaded from the kernel.

The examination of NUMA architectures in Section 6.5 emphasizes the importance of managing data processing and interrupt request (IRQ) handling within the same node that hosts the network card being tested. Performance reductions ranged from as little as 10% to as much as 40%, highlighting that older processors might struggle to achieve 100Gb throughput if data handling is assigned to an adjacent socket.

Section 6.6 discusses the critical nature of being cognizant of vulnerability mitigations for different processor brands within the Linux kernel. It highlights a mitigation strategy for AMD processors that substantially increases CPU utilization and decreases efficiency relative to their Intel counterparts. The section also underscores the value of continuous integration in performance testing, which allows for rapid detection and response to kernel patches. This approach benefits from triggering a testing pipeline for various configurations and demonstrates the utility of analytical tools in aggregating and comparing extensive data sets.

The comprehensive comparison of performance metrics across all test setups from the testbed, as outlined in Table 6.10, provides a detailed overview of processor capabilities. The Intel Sapphire Rapids emerges as the top performer, followed closely by the Intel Ice Lake. The AMD Genoa and Milan processors are ranked next, showing strong performance but slightly hindered by the SRSO mitigation, which, if absent, might have positioned them closer to the Sapphire Rapids in terms of efficiency. Notably, the ARM setup ranks just above the last place, which is held by the Intel Skylake. The positioning of the Skylake processor at the bottom of the list is attributed to its older technology base and the burden of multiple vulnerability mitigations.

Furthermore, the comparative analysis highlights that the ARM setup exhibits efficiency metrics in single and multistream tests comparable to AMD Genoa, indicating a competitive performance. In contrast, Intel processors distinctly outperform AMD and ARM in single-core efficiency. However, when it comes to multi-core processing, Intel's advantage

[1]https://core.dpdk.org/perf-reports/

diminishes, bringing its performance closer to that of AMD and ARM, suggesting a more levelled field in scenarios that demand multi-core capabilities.

The Intel E810 is the best overall network interface card, closely followed by the Nvidia ConnectX-6. Broadcom BCM57508 cards are last. However, it aligns with the marketing depiction as power efficient rather than a high-performance leader. When considering the optimal combinations of processors and network cards, setups involving Intel Sapphire Rapids and Ice Lake paired with Intel E810 cards lead to performance. These combinations are followed by setups incorporating the same Intel processors but using the Nvidia card instead. This hierarchical arrangement underscores the significant impact of the choice of network interface cards on the overall system performance, particularly in compatibility and efficiency with different processors.

<table>
<tr><td>Processor</td><td></td><td colspan="2">A4</td><td colspan="2">I4</td><td colspan="2">A3</td><td colspan="2">I3</td><td>I2</td><td colspan="2">ARM</td></tr>
<tr><td></td><td>NIC</td><td>Nvidia</td><td>Intel</td><td>Nvidia</td><td>Intel</td><td>Nvidia</td><td>Broadcom</td><td>Nvidia</td><td>Intel</td><td>Broadcom</td><td>Nvidia</td><td>Intel</td></tr>
<tr><td rowspan="2">A4</td><td>Nvidia</td><td>-</td><td>-2%/-2%</td><td>-30%/-25%</td><td>-47%/-40%</td><td>-24%/+88%</td><td>+80%/+80%</td><td>-37%/-30%</td><td>-47%/-34%</td><td>+138%/+105%</td><td>+3%/+8%</td><td>-1%/+8%</td></tr>
<tr><td>Intel</td><td>+2%/+4%</td><td>-</td><td>-29%/-24%</td><td>-46%/-39%</td><td>-23%/+91%</td><td>+83%/+82%</td><td>-36%/-29%</td><td>-46%/-33%</td><td>+142%/+107%</td><td>+5%/+9%</td><td>+1%/+9%</td></tr>
<tr><td rowspan="2">I4</td><td>Nvidia</td><td>+41%/+32%</td><td>+39%/+30%</td><td>-</td><td>-25%/-20%</td><td>+8%/+149%</td><td>+154%/+138%</td><td>-11%/-8%</td><td>-34%/-12%</td><td>+236%/+171%</td><td>+46%/+43%</td><td>+40%/+43%</td></tr>
<tr><td>Intel</td><td>+87%/+64%</td><td>+84%/+62%</td><td>+32%/+24%</td><td>-</td><td>+42%/+211%</td><td>+237%/+196%</td><td>+19%/+16%</td><td>0%/+10%</td><td>+345%/+238%</td><td>+93%/+78%</td><td>+86%/+78%</td></tr>
<tr><td rowspan="2">A3</td><td>Nvidia</td><td>+30%/-53%</td><td>+29%/-52%</td><td>-8%/-60%</td><td>-30%/-68%</td><td>-</td><td>+136%/-5%</td><td>-17%/-63%</td><td>-30%/-65%</td><td>+212%/+8%</td><td>+35%/-43%</td><td>+30%/-43%</td></tr>
<tr><td>Broadcom</td><td>-45%/-45%</td><td>-45%/-46%</td><td>-60%/-59%</td><td>-70%/-67%</td><td>-58%/+5%</td><td>-</td><td>-65%/-61%</td><td>-70%/-63%</td><td>+32%/+14%</td><td>-43%/-41%</td><td>-45%/-40%</td></tr>
<tr><td rowspan="2">I3</td><td>Nvidia</td><td>+57%/+42%</td><td>+54%/+40%</td><td>+11%/+7%</td><td>-16%/-14%</td><td>+20%/+168%</td><td>+183%/+152%</td><td>55.63 | 50.22</td><td>-16%/-5%</td><td>+274%/+192%</td><td>+62%/+53%</td><td>+56%/+54%</td></tr>
<tr><td>Intel</td><td>+85%/+50%</td><td>+82%/+47%</td><td>+31%/+13%</td><td>-1%/-10%</td><td>+42%/+182%</td><td>+234%/+169%</td><td>+18%/+5%</td><td>-</td><td>+342%/+207%</td><td>+92%/+61%</td><td>+84%/+62%</td></tr>
<tr><td>I2</td><td>Broadcom</td><td>-59%/-52%</td><td>-58%/-52%</td><td>-70%/-64%</td><td>-78%/-71%</td><td>-68%/-9%</td><td>-25%/-13%</td><td>-74%/-66%</td><td>-78%/-68%</td><td>14.85 | 17.18</td><td>-57%/-48%</td><td>-59%/-48%</td></tr>
<tr><td rowspan="2">ARM</td><td>Nvidia</td><td>-4%/-8%</td><td>-5%/-9%</td><td>-32%/-30%</td><td>-49%/-44%</td><td>-27%/+74%</td><td>+74%/+66%</td><td>-39%/-35%</td><td>-48%/-49%</td><td>+130%/+90%</td><td>-</td><td>-4%/0%</td></tr>
<tr><td>Intel</td><td>0%/-8%</td><td>-1%/-9%</td><td>-29%/-31%</td><td>-47%/-44%</td><td>-34%/+74%</td><td>+81%/+66%</td><td>-36%/-36%</td><td>-46%/-39%</td><td>+139%/+90%</td><td>+4%/0%</td><td>-</td></tr>
</table>

Table 6.10: The Comparison Matrix systematically evaluates all setups from the testbed detailed in Chapter 4, aligning the results with the baseline measurements discussed in Section 6.1. A4 is AMD Genoa, I4 is Intel Sapphire Rapids, A3 is AMD Milan, I3 is Intel Icelake and I2 is short for Intel Skylake. This matrix assesses local efficiency, measured as gigabits per second per core, across two performance metrics: single-stream and multi-stream. It reveals that Intel processors strongly dominate single-stream performance, leveraging their robust design to maximize throughput per core. However, this significant advantage diminishes in multi-stream scenarios, where the simultaneous handling of multiple data streams tempers their superiority. The Intel E810 emerges as the leading option among network interface cards, demonstrating superior performance across both metrics, followed closely by the Nvidia ConnectX-6. The Broadcom BCM57508, while positioned last, aligns with its market depiction as power efficient rather than a high-performance leader, indicating a strategic focus on energy efficiency over peak performance.

Chapter 7

Conclusion

The primary aim of this thesis was to develop and investigate continuous network performance testing test scenarios, with a particular focus on their application in modern Linux kernel environments. This research addressed the challenges and opportunities presented by the growth in network throughputs and the advancement of multi-core processor technologies.

The pivotal findings of this thesis are encapsulated in a detailed analysis of the fundamental contributors to network performance. This includes an in-depth examination of key hardware network offloading strategies, as thoroughly discussed in Chapter 2.

In Chapter 3, a variety of continuous integration (CI) tools were explored within the realm of network interface card (NIC) performance testing on Linux-based systems. This chapter evaluates numerous CI tools and platforms, examining their effectiveness in managing the intricacies and demands of network performance testing. It also explores established software testing methods to determine their adaptability in meeting the unique challenges faced in NIC performance testing and identifying the necessary tools to mitigate them.

Chapter 4 offers an exhaustive overview of the hardware configurations utilized in NIC performance testing. It outlines the specifications and setups of the chosen NICs, explaining the reasoning behind each selection. This chapter describes the CPU architectures used to test the NICs and introduces a complete testing matrix.

Drawing on insights from the preceding chapters, Chapter 5 presents a proposed approach for continuous network performance testing on Linux-based operating systems. This comprehensive approach encompasses test planning, execution, data collection, and analysis. Furthermore, it introduces a range of generic testing scenarios to facilitate an efficient, repeatable testing process that yields consistent and comparable results. Scalability is a key emphasis, ensuring that the test scenarios can be implemented across different network environments and hardware setups. Adaptability is equally important, allowing the test scenarios to stay applicable in changing hardware and software contexts.

Chapter 6 concludes that the test scenarios are flexible enough to accommodate a range of network configurations, including various IP versions (IPv4, IPv6), VLAN settings, and transport layer protocols like TCP or UDP. Future work could extend these scenarios to include configurations like MPTCP or VXLAN. The study demonstrates that variations in IP versions and VLAN setups minimally impact network performance, with modern processors efficiently handling up to 100Gb of bidirectional throughput. Disabling TCP Segmentation Offload (TSO) leads to a substantial reduction in local efficiency, between 60% and 70%, while the effects on remote efficiency are minimal and uniform across different NIC brands. Similar impacts are observed when both TX and RX offloads are disabled,

though remote efficiency declines. The importance of NUMA architecture management to avoid performance degradation and the significance of continuous integration in responding to kernel patches are highlighted. Intel Sapphire Rapids stands out as the top performer, with Intel E810 noted as the best NIC. However, performance greatly varies with different processor and NIC combinations, affecting overall system efficiency and optimization. These insights are critical for informed decision-making in network engineering and management. Looking ahead, there could be a focus on incorporating 400Gb cards or developing a metric for throughput per watt with a comparison of ARM processors to other architectures, facilitating comparisons that are especially relevant as large providers start to prioritize power efficiency [31].

Bibliography

[1] *User Datagram Protocol* [RFC 768]. RFC Editor, august 1980. DOI: 10.17487/RFC0768. Available at: https://www.rfc-editor.org/info/rfc768.

[2] *Internet Protocol* [RFC 791]. RFC Editor, september 1981. DOI: 10.17487/RFC0791. Available at: https://www.rfc-editor.org/info/rfc791.

[3] *Transmission Control Protocol* [RFC 793]. RFC Editor, september 1981. DOI: 10.17487/RFC0793. Available at: https://www.rfc-editor.org/info/rfc793.

[4] IEEE Standard for Ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012).* 2016, p. 1–4017. DOI: 10.1109/IEEESTD.2016.7428776.

[5] *Speculative Return Stack Overflow* [https://docs.kernel.org/admin-guide/hw-vuln/srso.html]. 2023. Accessed: (02.05.2024).

[6] Agarwal, A., Gupta, S. and Choudhury, T. Continuous and integrated software development using DevOps. In: IEEE. *2018 International conference on advances in computing and communication engineering (ICACCE).* 2018, p. 290–293.

[7] Ahuja, V., Farrens, M. and Ghosal, D. Cache-aware affinitization on commodity multicores for high-speed network flows. In: *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems.* 2012, p. 39–48.

[8] Belmont, J.-M. *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI.* Packt Publishing Ltd, 2018.

[9] Broadcom. *PCIe NIC Ethernet Adapters Specification Sheet.*

[10] Cox, A. and La Roche, F. *Skbuff.* Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/skbuff.h?h=v4.16.

[11] Draheim, D., Grundy, J., Hosking, J., Lutteroth, C. and Weber, G. Realistic load testing of web applications. In: IEEE. *Conference on Software Maintenance and Reengineering (CSMR'06).* 2006, p. 11–pp.

[12] Eddy, W. *Transmission Control Protocol (TCP)* [RFC 9293]. RFC Editor, august 2022. DOI: 10.17487/RFC9293. Available at: https://www.rfc-editor.org/info/rfc9293.

[13] Everett, G. D. and McLeod Jr, R. *Software testing: testing across the entire software development life cycle.* John Wiley & Sons, 2007.

[14] Floyd, S., Handley, M. J. and Kohler, E. *Datagram Congestion Control Protocol (DCCP)* [RFC 4340]. RFC Editor, march 2006. DOI: 10.17487/RFC4340. Available at: https://www.rfc-editor.org/info/rfc4340.

[15] Foong, A., Fung, J. and Newell, D. An in-depth analysis of the impact of processor affinity on network performance. In: IEEE. *Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004)(IEEE Cat. No. 04EX955).* 2004, vol. 1, p. 244–250.

[16] Gross, J., Ganga, I. and Sridhar, T. *Geneve: Generic Network Virtualization Encapsulation* [RFC 8926]. RFC Editor, november 2020. DOI: 10.17487/RFC8926. Available at: https://www.rfc-editor.org/info/rfc8926.

[17] Herbert, T. *Remote checksum offload for encapsulation.* Internet-Draft draft-herbert-remotecsumoffload-02. Internet Engineering Task Force, march 2016. Work in Progress. Available at: https://datatracker.ietf.org/doc/draft-herbert-remotecsumoffload/02/.

[18] Hinden, B. and Deering, D. S. E. *Internet Protocol, Version 6 (IPv6) Specification* [RFC 2460]. RFC Editor, december 1998. DOI: 10.17487/RFC2460. Available at: https://www.rfc-editor.org/info/rfc2460.

[19] Intel Corporation. *Intel Ethernet Controller E810 Datasheet.* Rev 2.7.

[20] Jacobson, I., Booch, G. and Rumbaugh, J. The unified process. *Ieee Software.* IEEE Computer Society. 1999, vol. 16, no. 3, p. 96.

[21] Jovanović, I. Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research.* 2006, vol. 30.

[22] Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J. et al. The highways and country roads to continuous deployment. *Ieee software.* IEEE. 2015, vol. 32, no. 2, p. 64–72.

[23] Li, T., Farinacci, D., Hanks, S. P., Meyer, D. and Traina, P. S. *Generic Routing Encapsulation (GRE)* [RFC 2784]. RFC Editor, march 2000. DOI: 10.17487/RFC2784. Available at: https://www.rfc-editor.org/info/rfc2784.

[24] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L. et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks* [RFC 7348]. RFC Editor, august 2014. DOI: 10.17487/RFC7348. Available at: https://www.rfc-editor.org/info/rfc7348.

[25] Mellanox Technologies. *Mellanox Adapters Programmer's Reference Manual.* Rev 0.53.

[26] Melman, D. T., Mizrahi, T. and 3rd, D. E. E. *Fibre Channel over Ethernet (FCoE) over Transparent Interconnection of Lots of Links (TRILL)* [RFC 6847]. RFC Editor, january 2013. DOI: 10.17487/RFC6847. Available at: https://www.rfc-editor.org/info/rfc6847.

[27] Molyneaux, I. *The art of application performance testing: from strategy to tools.* „O'Reilly Media, Inc.", 2014.

[28] Mukherjee, J., Wang, M. and Krishnamurthy, D. Performance testing web applications on the cloud. In: IEEE. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops.* 2014, p. 363–369.

[29] Patton, R. *Software Testing.* Sams Publishing, 2005.

[30] Postel, J. *Rfc0793: Transmission control protocol.* RFC Editor, 1981.

[31] Robinson, D. *Amazon has more than half of all Arm server CPUs in the world* [https://www.theregister.com/2023/08/08/amazon_arm_servers/]. 2023. Accessed: (02.05.2024).

[32] Rosen, R. *Linux kernel networking: Implementation and theory.* Apress, 2014.

[33] Sampath, S., Bryce, R. and Memon, A. M. A uniform representation of hybrid criteria for regression testing. *IEEE transactions on software engineering.* IEEE. 2013, vol. 39, no. 10, p. 1326–1344.

[34] Shaw, M. Prospects for an engineering discipline of software. *IEEE Software.* IEEE. 1990, vol. 7, no. 6, p. 15–24.

[35] Shi, S., Wang, Q., Xu, P. and Chu, X. Benchmarking state-of-the-art deep learning software tools. In: IEEE. *2016 7th International Conference on Cloud Computing and Big Data (CCBD).* 2016, p. 99–104.

[36] Steenkiste, P. A. A systematic approach to host interface design for high-speed networks. *Computer.* IEEE. 1994, vol. 27, no. 3, p. 47–57.

[37] Stewart, R. R., Tüxen, M. and Nielsen karen. *Stream Control Transmission Protocol* [RFC 9260]. RFC Editor, june 2022. DOI: 10.17487/RFC9260. Available at: https://www.rfc-editor.org/info/rfc9260.

[38] The kernel development community. *Accelerated RFS.* Available at: https://docs.kernel.org/networking/scaling.html#accelerated-rfs.

[39] The kernel development community. *Generic Receive Offload.* Available at: https://docs.kernel.org/networking/segmentation-offloads.html#generic-receive-offload.

[40] The kernel development community. *Generic Segmentation Offload.* Available at: https://docs.kernel.org/networking/segmentation-offloads.html#generic-segmentation-offload.

[41] The kernel development community. *Local Checksum Offload.* Available at: https://www.kernel.org/doc/html/next/networking/checksum-offloads.html#lco-local-checksum-offload.

[42] The kernel development community. *Partial Generic Segmentation Offload.* Available at: https://docs.kernel.org/networking/segmentation-offloads.html#partial-generic-segmentation-offload.

[43] The kernel development community. *Receive Flow Steering.* Available at: https://docs.kernel.org/networking/scaling.html#rfs-receive-flow-steering.

[44] The kernel development community. *Receive Packet Steering.* Available at: https://docs.kernel.org/networking/scaling.html#rps-receive-packet-steering.

[45] The kernel development community. *Receive Side Scaling.* Available at: https://docs.kernel.org/networking/scaling.html#rss-receive-side-scaling.

[46] The kernel development community. *TCP Segmentation Offload.* Available at: https://docs.kernel.org/networking/segmentation-offloads.html#tcp-segmentation-offload.

[47] The kernel development community. *Transmit Packet Steering.* Available at: https://docs.kernel.org/networking/scaling.html#xps-transmit-packet-steering.

[48] The kernel development community. *TX Checksum Offload.* Available at: https://www.kernel.org/doc/html/next/networking/checksum-offloads.html#tx-checksum-offload.

[49] The kernel development community. *UDP Fragmentation Offload.* Available at: https://docs.kernel.org/networking/segmentation-offloads.html#udp-fragmentation-offload.

[50] Turull, D., Sjödin, P. and Olsson, R. Pktgen: Measuring performance on high speed networks. *Computer communications.* Elsevier. 2016, vol. 82, p. 39–48.

[51] Turull, D., Sjödin, P. and Olsson, R. Pktgen: Measuring performance on high speed networks. *Computer Communications.* 2016, vol. 82, p. 39–48. DOI: https://doi.org/10.1016/j.comcom.2016.03.003. ISSN 0140-3664. Available at: https://www.sciencedirect.com/science/article/pii/S0140366416300615.

[52] Yoo, S. and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability.* Wiley Online Library. 2012, vol. 22, no. 2, p. 67–120.

[53] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V. and Vasilescu, B. The impact of continuous integration on other software development practices: a large-scale empirical study. In: IEEE. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2017, p. 60–71.

Appendix A

Measurements

A.1 Baseline measurements

This section evaluates the performance of different processors and network interface card (NIC) configurations under various network conditions, continuing on the results presented in Section 6.1. These tables cover a range of processors, from Intel's Ice Lake and Skylake to AMD's Milan. The performance is examined with different network cards across multiple IP configurations, including IPv4 and IPv6, with and without VLAN. Performance metrics such as local efficiency and throughput are quantified across various concurrent streams, illustrating how each setup handles escalating network loads.

The first Table A.1 examines the Intel Ice Lake processor paired with Nvidia's ConnectX-6 network card. It contrasts the performance differences between IPv4 and IPv6 configurations, highlighting the scalability and efficiency shifts when VLAN is applied. This setup demonstrates how the processor and NIC handle increased network demands, with detailed measurements of local efficiency and throughput across up to 16 concurrent streams. Notably, the table shows a general decline in performance efficiency with increased loads but provides valuable insights into the network capabilities of this specific hardware combination.

The other tables continue this analysis with different hardware setups. Table A.3 assesses AMD Milan with Nvidia ConnectX-6, exploring similar metrics under various network loads and configurations. This approach is repeated across various combinations, including AMD Milan with Broadcom and Intel Skylake with Broadcom, in Table A.4 and Table A.5. These tables collectively outline the performance impacts of newer versus older technologies, different processor architectures, and the influence of network card selections. Each table provides specific insights into how these setups perform under different configurations, highlighting that different configurations have no significant impact on throughput and local efficiency.

		1	2	4	8	16
IPv4	Local efficiency	55.63	50.23	42.61	34.89	36.59
	Throughput	40.64	73.74	130.58	175.06	155.84
IPv4 VLAN	Local efficiency	+2%	-3%	-1%	0%	-2%
	Throughput	-3%	-3%	-1%	0%	-1%
IPv6	Local efficiency	+3%	-1%	-1%	-1%	+2%
	Throughput	-6%	-4%	-2%	0%	-11%
IPv6 VLAN	Local efficiency	+4%	-3%	-2%	-1%	-2%
	Throughput	-7%	-4%	-3%	0%	-3%

Table A.1: Performance metrics comparison for Intel Icelake with Nvidia ConnectX-6 network card under different IP configurations and VLAN setups across multiple concurrent streams. This table presents the local efficiency and throughput for IPv4 and IPv6, both with and without VLAN, across varying numbers of concurrent streams (1, 2, 4, 8, and 16). The first entry in the local efficiency and throughput rows indicates the actual performance measurement, measured in gigabit per second per core and gigabit per second, respectively. In contrast, the percentage changes in the subsequent rows reflect the impact compared to the base IPv4 configuration. The table underscores the scalability of network performance under increasing loads and the subtle variances introduced by different network protocols and configurations.

		1	2	4	8	16
IPv4	Local efficiency	65.79	52.77	35.94	31.62	27.15
	Throughput	42.40	74.36	94.04	94.05	94.08
IPv4 VLAN	Local efficiency	-1%	-3%	+2%	-2%	-1%
	Throughput	-8%	-5%	0%	0%	0%
IPv6	Local efficiency	-1%	-1%	0%	-1%	0%
	Throughput	-14%	-9%	0%	0%	0%
IPv6 VLAN	Local efficiency	-3%	-2%	-2%	-3%	-1%
	Throughput	-10%	-11%	-2%	-2%	-2%

Table A.2: Comparative analysis of network performance for Intel Icelake with IPv4 and IPv6 configurations on Intel E810 NIC, both with and without VLAN, across varying numbers of concurrent streams (1, 2, 4, 8, and 16). The table shows local efficiency and throughput measurements for each setup. Local efficiency and throughput are initially presented as an absolute value for IPv4 without VLAN, measured in gigabit per second per core and gigabit per second, followed by the percentage change. Throughput changes are similarly shown as deviations from the baseline IPv4 performance. This detailed matrix illustrates how network performance adapts under different IP versions with or without VLAN.

		1	2	3	6	12	24
IPv4	Local efficiency	47.36	21.37	31.63	21.74	13.22	24.21
	Throughput	37.39	51.52	89.80	84.56	134.66	186.28
IPv4 VLAN	Local efficiency	+1%	-11%	-4%	-1%	-3%	-1%
	Throughput	+1%	+5%	-2%	-2%	0%	0%
IPv6	Local efficiency	-1%	-12%	-7%	-1%	-4%	-1%
	Throughput	+2%	-5%	-2%	-2%	-2%	-1%
IPv6 VLAN	Local efficiency	0%	-13%	-5%	-2%	-3%	-1%
	Throughput	-1%	-7%	-1%	-2%	-2%	-3%

Table A.3: Performance evaluation of AMD Milan with Nvidia ConnectX-6 network card across IPv4 and IPv6 configurations, both with and without VLAN, for different numbers of concurrent streams (1, 2, 3, 6, 12, and 24). This table quantitatively details local efficiency and throughput for each network setup. For IPv4, initial values are presented. Local efficiency is measured in Gbps per core, and throughput is in Gbps. Followed by percentage changes that illustrate the impact of implementing VLAN and switching to IPv6. The effects of these changes on throughput are also provided, highlighting minimal performance impact across different stream counts.

		1	2	3	6	12	24
IPv4	Local efficiency	19.64	19.58	19.61	12.69	12.73	8.89
	Throughput	24.84	49.39	73.76	102.47	83.84	60.53
IPv4 VLAN	Local efficiency	-1%	0%	-1%	-2%	-2%	-5%
	Throughput	+1%	0%	0%	0%	+1%	0%
IPv6	Local efficiency	0%	-2%	-1%	-2%	-3%	-4%
	Throughput	0%	-2%	-1%	+1%	0%	-1%
IPv6 VLAN	Local efficiency	-4%	-4%	-3%	-1%	-5%	-6%
	Throughput	-2%	-3%	-2%	0%	0%	+1%

Table A.4: This table presents a performance analysis of AMD Milan using a Broadcom network card across various IPv4 and IPv6 configurations, with and without VLAN, across a range of concurrent streams (1, 2, 3, 6, 12, and 24). The data details local efficiency and throughput, measured in Gbps per core and Gbps, for each configuration and stream count, starting with baseline IPv4 values, followed by percentage changes due to VLAN implementation and IPv6 adaptation. The adjustments in local efficiency and throughput under each network condition provide insights into the network performance stability and scalability of AMD Milan in response to increased network demands and varied protocol environments.

		1	2	3	6	12
IPv4	Local efficiency	14.85	17.18	19.29	17.47	16.90
	Throughput	30.96	56.77	84.22	94.06	94.10
IPv4 VLAN	Local efficiency	-2%	-1%	-1%	-1%	-1%
	Throughput	+2%	+2%	0%	0%	0%
IPv6	Local efficiency	0%	0%	-2%	-1%	-2%
	Throughput	+2%	+1%	-1%	-1%	-1%
IPv6 VLAN	Local efficiency	-2%	-2%	-3%	-2%	-3%
	Throughput	+1%	+1%	-2%	-2%	-2%

Table A.5: Performance metrics of Intel Skylake with a Broadcom network card under IPv4 and IPv6 settings, both with and without VLAN, across multiple concurrent streams (1, 2, 3, 6, and 12). This table illustrates each configuration's local efficiency and throughput, providing baseline figures for IPv4, measured in gigabits per second per core and gigabits per second, and subsequent percentage changes reflecting the impact of different configurations. There seems to be no significant performance impact.

A.2 Duplex

The tables presented offer a comprehensive look at the performance characteristics of different network setups involving Intel and AMD processors with Broadcom and Nvidia network cards under varied duplex (unidirectional and bidirectional communication) scenarios. The first Table A.6 details the performance of Intel Sapphire Rapids processors in combination with Nvidia ConnectX-6 and Intel E810 network interface cards across multiple streams, quantifying local and remote efficiency, as well as throughput. This data shows the baseline performance in a unidirectional setting and the percentage reduction in efficiency and throughput in bidirectional scenarios, highlighting the processors' capabilities and limitations when handling increased network traffic.

The second Table A.7 focuses on the AMD Milan processor, comparing its performance using Nvidia and Broadcom network cards. This analysis is particularly insightful as it outlines the baseline performance and the significant drop in efficiency when switching to bidirectional communication across a range of concurrent streams. The data illustrates how performance degrades more severely under certain network card setups, notably with Broadcom showing a pronounced decline, especially in higher concurrency settings. The contrasting results between Nvidia and Broadcom setups provide valuable insights into the impact of network card selection on system performance in complex network scenarios.

Finally, using a Broadcom network card, the third Table A.8 centres on the Intel Skylake processor. This table is similar to the previous ones but focuses exclusively on Skylake's capabilities in handling unidirectional and bidirectional communications across fewer concurrent streams. It reports local and remote efficiency and throughput, showing a marked decline as the number of streams increases, indicative of Skylake's performance scalability issues under bidirectional stress. This detailed breakdown helps understand the specific performance bottlenecks and operational limitations faced by older Intel processors compared to newer models and different network cards.

		1	2	4	8	16	32
Baseline Intel	Local efficiency	66.21	58.18	33.00	27.84	22.74	19.18
	Remote efficiency	28.84	22.02	16.58	12.38	9.26	6.25
	Throughput	45.11	72.73	94.03	94.04	94.06	94.15
Target Intel	Local efficiency	-75%	-77%	-64%	-69%	-71%	-73%
	Remote efficiency	-42%	-40%	-28%	-30%	-28%	-17%
	Throughput	-40%	-39%	-16%	0%	0%	0%
Baseline Nvidia	Local efficiency	49.99	46.67	34.07	23.17	18.76	18.17
	Remote efficiency	22.91	19.06	16.78	12.10	9.03	7.72
	Throughput	40.08	72.26	94.02	94.03	94.05	94.14
Target Nvidia	Local efficiency	-72%	-77%	-74%	-63%	-64%	-66%
	Remote efficiency	-39%	-42%	-46%	-29%	-25%	-20%
	Throughput	-32%	-38%	-17%	0%	0%	0%

Table A.6: Comprehensive performance analysis of duplex configurations using Intel Sapphire Rapids and Nvidia ConnectX-6 and Intel E810 network interface cards, detailing both baseline (unidirectional communication) and target (bidirectional communication) metrics. This table presents local and remote efficiency measurements in Gbps per core, alongside total throughput in Gbps, across a range of concurrent streams (1, 2, 4, 8, 16, and 32). The baseline configurations for Intel and Nvidia illustrate initial performance metrics, while the target rows depict the percentage reduction in efficiency and throughput compared to the baseline. This stark contrast in performance across both platforms and various workload intensities highlights the scalability challenges and efficiency losses under increased operational demands. However, still shows the ability to reach a 100Gb line rate.

		1	2	3	6	12	24
Baseline	Local efficiency	47.68	19.02	30.48	21.57	12.87	23.93
Nvidia	Remote efficiency	15.32	14.45	15.22	7.03	6.37	8.38
	Throughput	37.60	48.91	88.25	83.29	134.94	185.79
Target	Local efficiency	-84%	-60%	-80%	-74%	-68%	-79%
Nvidia	Remote efficiency	-49%	-47%	-57%	-21%	-35%	-38%
	Throughput	-30%	-17%	-51%	-20%	-72%	-43%
Baseline	Local efficiency	19.64	19.58	19.61	12.69	12.73	8.89
Broadcom	Remote efficiency	23.50	22.12	21.02	14.35	6.78	4.64
	Throughput	24.84	49.39	73.76	102.47	83.84	60.53
Target	Local efficiency	-34%	-38%	-87%	-89%	-59%	-52%
Broadcom	Remote efficiency	-45%	-46%	-87%	-91%	-22%	-8%
	Throughput	+40%	+1%	-81%	-88%	-38%	-25%

Table A.7: This table presents a comparative analysis of duplex performance metrics for AMD Milan using Nvidia and Broadcom network cards, both baseline and target configurations, unidirectional and bidirectional communications, across multiple concurrent streams (1, 2, 3, 6, 12, and 24). Local and remote efficiency, in Gbps per core, alongside throughput in Gbps, are quantified for baseline settings, followed by percentage changes in the target settings to illustrate performance degradation under varied operational demands. The table distinctly highlights the variance in performance degradation between the two network card setups across a range of workload intensities, demonstrating the impact of different network cards on the overall system efficiency and throughput in duplex scenarios. Furthermore, results show that in this specific test scenario, the Nvidia ConnectX-6 is unable to reach a line rate of 200Gb in both directions.

		1	2	3	6	12
Baseline	Local efficiency	14.85	17.18	19.29	17.47	16.90
Broadcom	Remote efficiency	12.44	13.89	14.00	11.51	11.34
	Throughput	30.96	56.77	84.22	94.06	94.10
Target	Local efficiency	-27%	-37%	-47%	-54%	-56%
Broadcom	Remote efficiency	-12%	-23%	-28%	-30%	-34%
	Throughput	-48%	-45%	-47%	-24%	-7%

Table A.8: Performance comparison between baseline (unidirectional communication) and target (bidirectional communication) test scenarios for Intel Skylake using a Broadcom network card across various concurrent streams (1, 2, 3, 6, and 12). The table measures local and remote efficiency regarding gigabits per second per core and overall throughput in gigabits per second. Each baseline metric is followed by the percentage decrease observed in the target configuration, reflecting the performance impact under increased workload scenarios. This table effectively illustrates the decrease in local and remote efficiency and throughput as the number of streams increases, providing a clear view of the scalability challenges this specific setup faces.

A.3 TSO offloading

This section is a continuation of discussing results of the test scenario from Section 6.3, offers a detailed comparative analysis of network interface cards (NICs) focusing on the impact of enabling or disabling Transmission Segment Offloading (TSO) on efficiency and throughput. The first Table A.9 focuses on Nvidia and Intel NICs on the Intel Icelake processor, providing baseline data when TSO is enabled and target data with TSO disabled. It includes metrics such as local and remote efficiency (measured in Gbps per core) and throughput (measured in Gbps) across multiple concurrent streams. This presentation allows for a direct comparison of the two scenarios, highlighting how disabling TSO affects performance across different streams, reflecting a decrease in local efficiency and variable changes in remote efficiency.

The second Table A.10 extends this analysis to the AMD Milan processor with the Nvidia ConnectX-6 NIC. This table captures the fluctuations in local and remote efficiencies and throughput across multiple streams, showing how the baseline performance with TSO enabled compares against the target performance with TSO disabled. Significant losses in local efficiency are evident with marginal improvements in remote efficiency.

Lastly, the third Table A.11 presents a comparative study on Broadcom NICs on the Intel Skylake processor. This comparison follows the same methodology of contrasting performance metrics with TSO enabled and disabled. The data reveal a consistent pattern of decreased local efficiency when TSO is disabled, which aligns with findings from Nvidia and Intel NICs. However, it shows a more significant improvement in remote efficiency, which suggests that the Broadcom NICs might handle receiving traffic more effectively under certain conditions without TSO. Each table collectively emphasizes the critical role of TSO in optimizing network throughput and efficiency, providing valuable insights for network administrators and system architects.

		1	2	4	8	16
Baseline Nvidia	Local efficiency	55.63	50.23	42.61	34.89	36.59
	Remote efficiency	19.63	17.08	14.59	15.88	9.76
	Throughput	340.64	73.74	130.58	175.06	155.84
Target Nvidia	Local efficiency	-70%	-69%	-67%	-59%	-63%
	Remote efficiency	+24%	+26%	+29%	+9%	39%
	Throughput	-34%	-33%	-35%	-19%	0%
Baseline Intel	Local efficiency	65.79	52.77	35.94	31.62	27.15
	Remote efficiency	23.72	20.49	17.29	14.17	10.58
	Throughput	42.40	74.36	94.04	94.05	94.08
Target Intel	Local efficiency	-70%	-65%	-58%	-58%	-56%
	Remote efficiency	+4%	-5%	0%	0%	0%
	Throughput	-20%	-8%	0%	0%	0%

Table A.9: This table presents a comprehensive comparison on the Intel Icelake processor of baseline measurements with TSO enabled and target with TSO disabled for Nvidia ConnectX-6 and Intel E810 measured in terms of local and remote efficiency (measured in Gbps per core) and throughput (measured in Gbps) across multiple concurrent streams (1, 2, 4, 8, and 16). The Baseline rows show the initial performance levels for Nvidia and Intel network interface cards with TSO enabled, capturing local and remote efficiencies and throughput. In contrast, the Target rows display the percentage change from baseline when TSO was disabled, reflecting how each processor adapts to disabling TSO offload.

		1	2	3	6	12	24
Baseline Nvidia	Local efficiency	47.68	19.02	30.48	21.57	12.87	23.93
	Remote efficiency	15.32	14.45	15.22	7.03	6.37	8.38
	Throughput	37.60	48.91	88.25	83.29	134.94	185.79
Target Nvidia	Local efficiency	-77%	-38%	-61%	-54%	-28%	-66%
	Remote efficiency	+2%	+7%	-8%	+26%	+8%	+6%
	Throughput	-49%	-29%	-63%	-8%	+1%	-26%

Table A.10: This table displays the local and remote efficiency percentages and throughput values measured across many streams (1, 2, 3, 6, 12, and 24) measured on AMD Milan processor with Nvidia ConnectX-6 NIC. The data highlights changes in performance between the baseline with TSO offload enabled and the target with TSO disabled, indicating efficiency losses and gains.

		1	2	3	6	12
Baseline	Local efficiency	14.85	17.18	19.29	17.47	16.90
Broadcom	Remote efficiency	12.44	13.89	14.00	11.51	11.34
	Throughput	30.96	56.77	84.22	94.06	94.10
Target	Local efficiency	-27%	-50%	-61%	-48%	-49%
Broadcom	Remote efficiency	+27%	+8%	+7%	+32%	+33%
	Throughput	-41%	-50%	-55%	-23%	-4%

Table A.11: This table presents a comprehensive comparison of the Broadcom network interface card on the Intel Skylake processor, contrasting baseline measurements (enabled TSO) with the target (disabled TSO) where conditions are varied. The evaluation spans multiple streams (1, 2, 3, 6, and 12), focusing on local and remote efficiency (measured in Gbps per core) and throughput (measured in Gbps). The Baseline Broadcom rows capture the initial performance metrics, while the Target Broadcom rows show the percentage change in performance, indicating the card's performance when TSO offload is disabled .

A.4 TX and RX offloading

The section is a continuation of the performance evaluation from Section 6.3, which presents an in-depth performance evaluation of network interface cards (NICs) across different processors like Intel Icelake, AMD Milan, and Intel Skylake under conditions of receive and transmit checksum offloads. The first Table A.12 focuses on the Intel Icelake processor, comparing baseline metrics with checksum offloads enabled against target metrics with offloads disabled. This comparison uses Nvidia ConnectX-6 and Intel E810 NICs across multiple concurrent streams. The data notably demonstrate how local and remote efficiencies and throughput are affected when these offloads are turned off, with a significant drop in local efficiency and throughput as the primary outcome.

Similarly, the second Table A.13 examines the AMD Milan processor using an Nvidia network interface card. This table extends the analysis to include a broader range of streams, from 1 up to 24, providing a granular view of performance decay over an increased workload. Baseline measurements with enabled RX and TX offloads are juxtaposed with target scenarios where these functionalities are disabled. The results sharply highlight the degradation in local and remote efficiency and throughput across all streams, underlining the impact of disabling offloads in high-performance environments.

Lastly, the third Table A.14 focuses on the Intel Skylake processor, employing a Broadcom network card to ascertain the effects of enabling versus disabling RX/TX checksum offloads across fewer streams (1, 2, 3, 6, and 12). Like the previous tables, this illustrates the baseline and target scenarios' local and remote efficiencies and throughput, measured in gigabits per second per core and gigabit per second, respectively. Significant decreases in efficiency and throughput in the target configuration reveal the critical role of checksum offloads in maintaining optimal performance levels under various streaming conditions, thereby mapping out the scalability challenges and operational impacts in network communications.

		#1	2	4	8	16
Baseline	Local efficiency	55.63	50.23	42.61	34.89	36.59
Nvidia	Remote efficiency	19.63	17.08	14.59	15.88	9.76
	Throughput	40.64	73.74	130.58	175.06	155.84
Target	Local efficiency	-76%	-74%	-73%6	-66%	-63%
Nvidia	Remote efficiency	-5%	+9%	+22%	+2%	-9%
	Throughput	-48%	-49%	-50%	-35%	-12%
Baseline	Local efficiency	65.79	52.77	35.94	31.62	27.15
Intel	Remote efficiency	23.72	20.49	17.29	14.17	10.58
	Throughput	42.40	74.36	94.04	94.05	94.08
Target	Local efficiency	-74%	-68%	-68%	-68%	-65%
Intel	Remote efficiency	-15%	-11%	-12%	-13%	-8%
	Throughput	-34%	-27%	0%	0%	0%

Table A.12: This table presents a comprehensive comparison on the Intel Icelake processor of baseline measurements with receive and transmit checksum offloads enabled and target with these offloads disabled for Nvidia ConnectX-6 and Intel E810 measured in terms of local and remote efficiency (measured in Gbps per core) and throughput (measured in Gbps) across multiple concurrent streams (1, 2, 4, 8, and 16). The Baseline rows show the initial performance levels for Nvidia and Intel network interface cards with TX and RX offloads enabled, capturing local and remote efficiencies and throughput. In contrast, the Target rows display the percentage change from baseline when TX and RX offloads were disabled.

		#1	2	3	6	12	24
Baseline	Local efficiency	47.68	19.02	30.48	21.57	12.87	23.93
Nvidia	Remote efficiency	15.32	14.45	15.22	7.03	6.37	8.38
	Throughput	37.60	48.91	88.25	83.29	134.94	185.79
Target	Local efficiency	-81%	-51%	-74%	-72%	-56%	-79%
Nvidia	Remote efficiency	-19%	-14%	-47%	-16%	-6%	-21%
	Throughput	-69%	-50%	-69%	-40%	-35%	-48%

Table A.13: This table presents a comparative analysis of performance metrics for AMD Milan uses a Nvidia network interface card with baseline and target configurations. The baseline represents measurements with RX and TX offloads enabled, while the target represents those disabled. The results are measured across multiple concurrent streams (1, 2, 3, 6, 12, and 24). Local and remote efficiency, in Gbps per core, alongside throughput in Gbps, are quantified for baseline settings, followed by percentage changes in the target to illustrate performance degradation with these offload disabled, highlighting substantial local and remote efficiency declines.

		#1	2	3	6	12
Baseline	Local efficiency	14.85	17.18	19.29	17.47	16.90
Broadcom	Remote efficiency	12.44	13.89	14.00	11.51	11.34
	Throughput	30.96	56.77	84.22	94.06	94.10
Target	Local efficiency	-39%	-58%	-70%	-56%	-58%
Broadcom	Remote efficiency	-18%	-36%	-37%	-29%	-30%
	Throughput	-57%	-61%	-64%	-43%	-29%

Table A.14: Performance comparison of baseline and target scenarios for Intel Skylake using a Broadcom network card, assessed across multiple concurrent streams (1, 2, 3, 6, and 12). The baseline represents enabled RX/TX checksum offloads, while the target represents disabled offloads. The table captures local and remote efficiency in gigabits per second per core and overall throughput in gigabits per second. Each baseline measurement is followed by the percentage decrease in the target configuration, demonstrating the performance impact with these offloads disabled. This table effectively highlights the reduction in local and remote efficiency and throughput while receive and transmit checksum offloads are disabled.

A.5 UDP

This section continues the presentation of results from Section 6.4 and explores the performance of various network cards across different processor architectures during UDP traffic, focusing on the impact of increasing data streams on network throughput measured in packets per second (PPS). The first Table A.15 delves into the performance characteristics of Nvidia ConnectX-6 and Intel E810 network cards on Intel IceLake processors. This setup demonstrates how each network card manages bidirectional communication across different hosts, revealing significant differences in their capacity to handle escalating network traffic. Notably, Nvidia cards show superior packet handling capabilities at higher data stream counts than Intel, suggesting better scalability under load.

The second Table A.16 assesses the performance of Nvidia and Broadcom network cards on AMD Milan processors. This analysis extends to more varied stream counts (up to 24 streams), providing a broader perspective on each card's throughput efficiency under incrementally increased loads. The Broadcom cards particularly stand out at higher stream counts, showcasing their ability to sustain higher throughput rates, which could indicate better optimization for higher concurrency levels in network traffic.

Finally, the third Table A.17 shifts focus to an ARM-based setup, comparing Nvidia and Intel network cards across an even broader range of data streams, from 1 to 32. This setup highlights Nvidia's exceptional performance, maintaining robust throughput even as network demands scale significantly. Both host systems exhibit progressive performance degradation as the number of streams increases, yet Nvidia consistently outperforms Intel, especially at the highest concurrency levels. This indicates Nvidia's superior efficiency and throughput stability in high-demand scenarios, emphasizing its potential for use in environments with intense data transmission requirements.

Together, these tables provide valuable insights into the comparative performance of different network cards across various architectures and conditions, illuminating the complex dynamics of network scalability and efficiency in high-throughput environments.

		1	2	4	8	16
Nvidia	Host #1	1031303	1996631	3822444	7308213	9479147
	Host #2	1903709	1759808	3376013	6461508	7236961
Intel	Host #1	1044448	1911491	3721755	6933520	8539879
	Host #2	916203	1752248	3310653	6304425	6875822

Table A.15: Performance evaluation of Nvidia and Intel network cards across two host systems, measuring packets per second (pps) across various number of streams (1, 2, 4, 8, 16). This table showcases the capabilities of each network card on Intel IceLake processors during UDP transmissions, comparing the throughput of Host #1 and Host #2 under identical conditions in bidirectional communication. The data reveals the differing packet handling capacities of Nvidia and Intel hardware, providing insights into their performance scaling with increasing network traffic.

		1	2	3	6	12	24
Nvidia	Host #1	498840	991191	1483245	2401579	4815468	7252519
	Host #2	1440838	881628	1319874	2202576	4374334	4964795
Broadcom	Host #1	1040392	1690811	1553282	4706576	6927408	8299525
	Host #2	892036	1506397	2724234	3571938	7470404	8838950

Table A.16: This table compares the performance of Nvidia and Broadcom network cards in handling UDP traffic across multiple streams (1, 2, 3, 6, 12, and 24) on hosts within the AMD Milan processor. It quantifies the throughput in packets per second (pps) for two hosts under each network card, illustrating how each card manages network traffic as the load increases in bidirectional communication. The data provided shows variations in performance between the two brands and between different hosts, offering a detailed view of each card's efficiency and capability in progressively demanding network environments.

		1	2	4	8	16	32
Nvidia	Host #1	642853	1146881	2286337	4504058	8138258	15118067
	Host #2	569855	1029915	2045658	3986854	7457992	12648309
Intel	Host #1	572600	1085598	2139554	3824881	5040524	11370500
	Host #2	518966	994237	1940515	3499987	4589612	10476405

Table A.17: This table illustrates the performance of Nvidia ConnectX-6 and Intel E810 network cards on ARM-based hosts, measuring throughput in packets per second (pps) across increasing concurrent data streams (1, 2, 4, 8, 16, and 32). The table presents a side-by-side comparison of two hosts under each network card type, showcasing the scalability of each card as network loads intensify in bidirectional UDP communication. The data highlights the superior throughput capabilities of Nvidia cards across all stream counts, providing insights into network performance optimization in high-demand scenarios.