Degree Project in Information and Communication Technology

Second cycle, 30 credits

Improving Megaflow Cache Performance in Open vSwitch with Coflow-Aware Branch Prediction

EMIL STÅHL

Improving Megaflow Cache Performance in Open vSwitch with Coflow-Aware Branch Prediction

EMIL STÅHL

Degree Programme in Information and Communication Technology
Date: October 22, 2024

Supervisors: Eelco Chaudron, Hamid Ghasemirahni, Simone Ferlin-Reiter
Examiner: Marco Chiesa
School of Electrical Engineering and Computer Science
Host company: Red Hat AB
Swedish title: Förbättring av prestandan för Megaflow-cache i Open vSwitch med coflow-medveten spekulativ prediktion

Abstract

Software-defined Networking (SDN) has increasingly shifted towards hardware solutions that accelerate packet processing within data planes. However, optimizing the interaction between the data plane and the control plane, commonly referred to as the *slow path*, remains a significant challenge. This challenge arises because, in several major SDN applications, the control plane installs rules in the data plane reactively as new flows arrive, requiring time-consuming transitions to user space, which can become a significant bottleneck in high-throughput networks.

This thesis explores potential optimizations of Open vSwitch (OVS) by employing coflows to anticipate imminent network traffic, thus reducing the latency-inducing upcalls to the control plane, which are typically triggered by cache misses in the OVS megaflow cache. The study involves a series of benchmarks conducted on an OVN-simulated, single-node OCP cluster. These benchmarks utilize XDP to timestamp packets at both ingress and egress points of the cluster, measuring latency across various traffic scenarios. These scenarios are generated using synthetic coflow traffic traces, which vary in flow size distribution. The findings provide a comprehensive analysis of how OVS's performance is influenced by accurately predicting varying proportions of future flows under different traffic conditions.

Results indicate that the benefits OVS gains from the ability to predict and preload flows are contingent upon the flow rate of the traffic trace. Notably, even a modest ability to foresee flows can result in enhancements in both maximum and mean latency, as well as reductions in CPU utilization. These improvements underscore the potential of predictive techniques in boosting data plane responsiveness and overall system efficiency. Suggestions for future work include developing a real-time coflow predictor that could dynamically load flows into the datapath during runtime. Such advancements could reduce latency and resource consumption in OVS production deployments.

Keywords

Open vSwitch, Software-Defined Networking, Slow path, Coflows

Sammanfattning

Utvecklingen av mjukvarubaserade nätverk har i allt högre grad förlitat sig på hårdvaruimplementationer för paketbearbetning i dataplan. Trots dessa framsteg är optimering av interaktionen mellan dataplan och kontrollplan fortfarande en kritisk utmaning. Denna interaktion riskerar att bli en betydande flaskhals i nätverk med hög överföringshastighet.

Detta arbete utforskar potentiella optimeringar av Open vSwitch (OVS) genom användning av coflows för att förutse nära förestående nätverkstrafik, vilket minskar de latensinducerande anropen till kontrollplanet, som vanligtvis initieras av cachemissar i OVS megaflow-cache. Studien omfattar en serie av prestandatester utförda på en enskild nod i ett OCP-kluster simulerat med OVN. Dessa prestandatester använder XDP för att tidsstämpla paket vid både in- och utgångspunkter i klustret, och mäter latens över olika trafikscenarier. Dessa scenarier genereras med hjälp av syntetiska coflow-nätverkstrafikprofiler med varierande storleksfördelning. Resultaten ger en omfattande analys av hur OVS-prestandan påverkas beroende på hur stor andel av den framtida trafiken som förutses korrekt under olika trafikförhållanden.

De erhållna resultaten visar att de fördelar OVS erhåller från förmågan att förutse och förinläsa flöden beror på flödestakten i nätverkstrafiken. Märkbart är att även en begränsad förmåga att förutse flöden kan leda till betydande förbättringar i både maximal och genomsnittlig latens, samt minskningar i processoranvändning. Dessa förbättringar understryker potentialen hos prediktiva tekniker för att öka dataplanets responsivitet och övergripande systemeffektivitet. Förslag till framtida arbeten inkluderar utveckling av en realtidsprediktor för coflows som dynamiskt kan förinläsa flöden i dataplanet i realtid. Sådana framsteg skulle kunna minska latens och resursanvändning i produktionsmiljöer av OVS.

Nyckelord

Open vSwitch, Software-Defined Networking, Slow path, Coflows

Acknowledgments

I am deeply grateful to my supervisors at Red Hat, Simone Ferlin-Reiter and Eelco Chaudron, for their invaluable support throughout this project. Simone, thank you for giving me the opportunity to pursue this research, which has provided me with unparalleled experience. Eelco, your prompt and tireless assistance with technical challenges and questions has been instrumental to the success of this work. Your expertise and knowledge are truly remarkable, and it has been a great honor and privilege to collaborate with such a distinguished expert in the field. Your guidance has pushed me beyond my limits, imparting knowledge that I will value throughout my career.

I also extend my gratitude to KTH for the resources and support provided during this project. I am particularly thankful to my examiner, Marco Chiesa, for facilitating my connection with Red Hat, and to my supervisor, Hamid Ghasemirahni, whose insights into scientific research have been a cornerstone of my work. His feedback has been crucial in elevating the quality of this thesis to its current standard of excellence.

I am grateful to those who have assisted me throughout this journey. Special thanks to Xavier Simonart at Red Hat for his support in designing and configuring the cluster used in this research. Additionally, I am thankful to Saksham and Agarwal Rachit at Cornell University for their guidance in utilizing the Sincronia coflow workload generator, which played an important role in this work.

Lastly, I want to acknowledge the personal support from those in my life whose contributions, though not mentioned individually, have been vital in making this work possible. You know who you are — thank you.

Stockholm, October 2024
Emil Ståhl

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

ACK	Acknowledgement
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BGP	Border Gateway Protocol
BGP-LS	Border Gateway Protocol - Link State
BPF	Berkeley Packet Filter
CAIDA	Cooperative Association for Internet Data Analysis
CDF	Cumulative Distribution Function
CDN	Content Delivery Network
CLI	Command Line Argument
CNI	Container Network Interface
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
DNS	Domain Name System
DPDK	Data Plane Development Kit
DSA	Domain-Specific Accelerator
eBPF	extended Berkeley Packet Filter
ECDF	Empirical Cumulative Distribution Function
ELF	Executable and Linkable Format
FB Hadoop	Facebook Hadoop
FPGA	Field Programmable Gate Array
FSD	Flow Size Distribution
Gbps	Gigabits per second
Geneve	Generic Network Virtualization Encapsulation
GPU	Graphics Processing Unit
GSRPC	Google Search RPC
HBM	High-Bandwidth Memory
HoL	Head-of-Line

HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IXP	Internet Exchange Point
JSON	JavaScript Object Notation
KVM	Kernel-based virtual machine
L1	Layer 1
L2	Layer 2
L3	Layer 3
LLC	Last Level Cache
MAC	Media Access Control
NAT	Network Address Translation
NB DB	Northbound Database
NETCONF	Network Configuration Protocol
NFV	Network Function Virtualization
NIC	Network Interface Controller
NM	NuevoMatch
NMU	NuevoMatchUP
NUMA	Non-uniform memory access
NVGRE	Network Virtualization using Generic Routing Encapsulation
OCP	OpenShift Container Platform
OVN	Open Virtual Networking
OVN-K	OVN-Kubernetes
OVS	Open Virtual Switch
OVS-CCACHE	OVS with Computational Cache
OVS-CFLOWS	OVS with Computational Flows
OVSDB	OVS Database
PCAP	Packet Capture

PDF	Probability Density Function
pps	Packets Per Second
QoS	Quality of Service
RAM	Random-Access Memory
RPC	Remote procedure call
RQ-RMI	Range Query Recursive Model Index
SB DB	Southbound Database
SD-WAN	Software-Defined Wide Area Network
SDN	Software-Defined Networking
SDX	Software-Defined Internet Exchange
SIMD	Single Instruction Multiple Data
SR-IOV	Single-root input/output virtualization
STP	Spanning Tree Protocol
SUT	System-Under-Test
SYN	Synchronize
TCP	Transmission Control Protocol
TE	Traffic Engineering
TLV	Type-Length-Value
TSS	Tuple Space Search
TX	Transmit
UDP	User Datagram Protocol
veth	Virtual Ethernet Device
VLAN	Virtual Local Area Network
VM	Virtual Machine
VXLAN	Virtual Extensible Local Area Network
WAN	Wide Area Network
XDP	eXpress Data Path

Chapter 1

Introduction

This thesis explores the potential benefits of employing branch prediction to enhance megaflow cache performance in Open Virtual Switch (OVS) by pre-loading upcoming flows into the datapath of OVS. This chapter introduces the thesis. Section 1.1 presents the area to be researched. In Section 1.2, we define the problem the work aims to address. The purpose and goal of the work are described in Section 1.3 and 1.4. Further on, Section 1.5 describes the potential benefits of the work and how these relate to ethical and sustainability aspects. Next, Section 1.6 presents the methodologies used while Section 1.7 describes the delimitations for the work. Lastly, Section 1.8 presents the outline for the rest of the thesis.

1.1 Background

In modern networking architectures, the operational landscape typically consists of a hardware-executed packet-processing data plane, a software-centric centralized control plane, and an intermediary layer known as the *slow path*. The slow path facilitates communication between the control and data planes, with OVS prominently positioned as an open-source orchestrator. OVS plays an important role in disseminating control plane updates to data plane caches [1]. Within the context of networking research, historical neglect has characterized the treatment of the slow path, as predominant efforts have traditionally concentrated on optimizing data plane pipelines. Presently, networking landscapes witness an escalating trend toward hardware implementation of these pipelines. Paradoxically, the slow path has received limited attention, mainly due to the prevailing notion that it lacks inherent performance-critical attributes. Recent studies, however, underscore an

imminent bottleneck in emerging networks, emphasizing the importance of a dedicated accelerator for the slow path, analogous to advancements in data plane optimization. This highlights the necessity to extend optimization endeavors beyond the data plane [2]. Consequently, the focus of this research is to contribute to the enhancement of the slow path within modern networking frameworks. Critical to the optimization of the slow path is the mitigation of cache miss bottlenecks. These bottlenecks emanate from constraints within the memory and feature support of the Network Interface Controller (NIC) data plane, resulting in heightened misses during the processing of flows with diverse fields. Compounded by altered traffic patterns and complex OpenFlow pipeline configurations, these challenges further underscore the significance of addressing cache miss bottlenecks within the slow path layer. This research aims to enhance the efficiency of the slow path in modern networking environments by minimizing the overhead associated with traversing the slow path every time a flow that is not currently present in the data plane caches arrives [1, 2, 3].

1.2 Problem

Optimizing OVS is crucial for advancing Software-Defined Networking (SDN). A significant challenge in this process is managing unknown flows, which introduces complexity in packet processing. When OVS encounters packets from new, unrecorded flows, the system often experiences cache misses in the megaflow cache. This cache plays a vital role in OVS by storing outcomes from previous flow analyses, which allows for faster decision-making. By retaining these results, the megaflow cache enables OVS to quickly match new packets against previously seen flows, streamlining the process. A cache miss occurs when an incoming packet corresponds to a flow not present in the megaflow cache, prompting the system to engage in a resource-intensive process of upcalls to the control path. Upon encountering a megaflow cache miss, OVS initiates a sequence involving a packet upcall to the `ovs-vswitchd`. This operation entails navigating the OpenFlow tables and seeking the relevant rules for the unknown flow. The results are subsequently cached into the megaflow cache for potential reuse in subsequent instances of the same flow [2]. The inefficiencies arising from this process are notably exacerbated in scenarios typified by ephemeral flows, a characteristic feature of workloads in the Red Hat OpenShift Container Platform (OCP). The frequency of cache misses, combined with the need to traverse OpenFlow tables, triggers upcalls to `ovs-vswitchd` in user space, imposing a significant

computational burden on the system. The expense of these cache misses lies in the prolonged latency introduced in decision-making processes for unknown flows. The delay in flow identification and rule retrieval hampers the overall performance of OVS, adversely affecting the efficiency of SDN environments [1, 2, 3]. Addressing these challenges is crucial for enhancing the overall efficiency of OVS in dynamic SDN contexts, and the subsequent optimizations sought in this research aim to alleviate these specific pain points.

1.3 Purpose

This thesis explores ways OVS can be made more efficient by anticipating upcoming network traffic. By predicting the flows about to reach the switch, OVS can preload these flows into its datapath cache. This proactive approach helps circumvent the latency typically incurred during slow path upcalls when processing new flows [2, 3]. To achieve this, the project will conduct a series of benchmarks on OVS across various network traffic scenarios. By analyzing OVS performance under these conditions, we aim to identify opportunities for enhancements. This research seeks to offer the open-source community a solid groundwork for integrating a flow prediction mechanism into OVS. Such a mechanism, informed by insights from understanding the upper limits of performance with perfect foresight of future flows, holds the potential to significantly improve OVS efficiency.

1.4 Research question

The thesis aims to address the following research question:

To what extent can the performance of the OVS megaflow cache be enhanced by applying branch prediction techniques that utilize coflows to predict forthcoming traffic flows?

The primary research objective of this study is to determine the potential upper bounds of performance achievable in OVS when provided with perfect knowledge of future traffic patterns. This involves:

Objective 1 *Analyzing the effectiveness of incorporating branch prediction strategies within the OVS megaflow cache.*

Objective 2 *Assessing performance variations in OVS across different traffic scenarios to establish a comprehensive understanding of the system's capabilities.*

1.5 Benefits, Sustainability, and Ethics

The performance optimization of OVS holds significant implications across various dimensions. As a core component of the Internet's infrastructure, OVS is vital for the functioning of contemporary society. Enhancing our understanding of OVS operations is essential for improving its performance, reliability, and availability. These improvements are not only technical necessities but also align with broader sustainability objectives, reflecting the role of the Internet in societal progress. Increased digitization elevates the dependence on electricity; however, a robust and efficient Internet underpins social sustainability and supports the achievement of the United Nations' Sustainable Development Goals [4]. Moreover, examining the limitations of OVS raises ethical concerns. It could affect the reputation of developers and unveil new vulnerabilities within organizations that depend on OVS technologies.

1.6 Methodology

The decision of methodology is important in a scientific study. In this work, the research method used is defined as *experimental* since the method

focuses on finding the potential upper-bound performance limit achievable in OVS given perfect knowledge about future flows. Moreover, the research is also *applied* since it makes use of OVS as well as a wide variety of tools, which means that the methodology is based on work performed by other researchers and developers [5]. The methodology for this research involves three key components: the selection and configuration of a benchmarking environment, the simulation of a production-like OCP cluster, and the implementation of various traffic patterns for experimental testing. To assess the system's performance, timestamps and sequence numbers are embedded within individual network packets, enabling the collection of precise performance statistics. This approach is designed to align with the thesis's primary objective, which is to determine the upper performance limit of OVS under different traffic conditions, assuming perfect knowledge of future traffic patterns.

1.7 Delimitations

Due to the scope of this project, as well as constraints related to knowledge, time, and resources, several limitations need to be acknowledged. These limitations primarily revolve around our utilization of hardware and software configurations. It is crucial to note that the methodology and analysis employed in this study do not attempt to compensate for environmental variations that may arise due to the specific choices made regarding hardware and software configurations on the experimental servers.

Firstly, this research does not rely on access to real-world applications and traffic traces. Instead, the benchmarks exclusively utilize synthetically generated traffic to assess performance metrics. This approach allows for controlled testing conditions but may not fully capture the complexities of real-world network behaviors. Moreover, our investigation is based on a single network traffic trace, which may limit the generalizability of the findings to diverse network scenarios. The benchmarking environment is established using a virtual cluster tailored to emulate the behavior and configuration of a real-world production OCP cluster. This virtualized setup enables controlled experiments but may not perfectly replicate all nuances of a physical cluster deployment. Lastly, the benchmarking traffic trace workload is intentionally restricted to 200,000 distinct flows. This limitation aims to prevent the consideration of cache revalidation and flow replacement in OVS.

1.8 Outline

The thesis is structured as follows: Chapter 2 lays the theoretical groundwork, discussing the key concepts of SDN, OVS, OCP, coflows, and relevant prior research. This foundation is critical for understanding the more complex methodologies and results that follow. Chapter 3 describes the methodologies employed throughout the study. It details the experimental approaches, data collection methods, and analytical techniques used, providing a clear framework for the experiments conducted. In Chapter 4, the results of the benchmarks conducted on OVS are presented. This chapter begins to analyze these results, setting the stage for a more detailed examination in the following chapter. Chapter 5 goes deeper into the analysis of the experimental data. It discusses the limitations and impacts of the findings, exploring why the results took their particular form and how they affect the current understanding and performance capabilities of OVS. Finally, Chapter 6 concludes the thesis. It summarizes the research findings, discusses their significance, and suggests potential avenues for future research based on the insights gained.

Chapter 2

Background

This chapter provides an introduction to SDN and its associated subfields. Following this, the discussion progresses to explore OVS and its closely related system, Open Virtual Networking (OVN). Additional specific topics discussed include OCP and the concept of Coflows. The chapter concludes with a review of related work in this area.

2.1 Software-Defined Networking

In the context of computer science and networking, SDN stands as a paradigm shift that has redefined the way networks are designed, managed, and optimized. At its core, SDN introduces an architecture marked by a clear separation between the control plane and the data plane, providing a restructuring of traditional networking models. In the traditional networking paradigm, these elements are tightly integrated, making network management a complex and rigid process. The control plane in SDN is responsible for making global decisions about the network. This includes tasks such as route calculation, traffic engineering, and overall network management. By centralizing these functions, SDN facilitates a more holistic and efficient approach to network control. This centralized intelligence is encapsulated in an SDN controller. Conversely, the data plane in SDN is responsible for the actual forwarding of network packets. Network devices in the data plane, often referred to as switches or forwarding devices, execute instructions received from the centralized SDN controller. This separation of control and data planes enhances flexibility, adaptability, and programmability, enabling dynamic adjustments to network behavior without the need for configuring each network device separately [6].

2.1.1 Motivation for SDN

Traditional networking exhibits several inherent limitations that impede the dynamic evolution of network architectures. These drawbacks lead to the emergence of SDN. This section explores the motivations driving the transition to SDN, outlining key challenges in traditional networking and how SDN addresses them.

- **Rigid Network Infrastructure** Traditional networks are characterized by rigid, hardware-centric architectures that lack the flexibility to swiftly adapt to changing requirements. SDN, through its decoupling of the control plane and data plane, introduces a dynamic framework where network behavior can be programmatically adjusted. This flexibility allows for rapid adaptation to varying workloads, optimizing resource utilization [6].

- **Limited Adaptability** Conventional network configurations struggle to adapt to diverse and dynamic traffic patterns. SDN leverages centralized control, enabling real-time monitoring and adjustment of network routes. This adaptive capability ensures efficient utilization of network resources, optimizing data flow based on actual usage patterns [6].

- **Complex Configuration and Management** The complexity associated with configuring and managing traditional networks is a significant hindrance. SDN simplifies network management by centralizing control and providing a unified view of the entire infrastructure. This streamlining not only reduces administrative overhead but also facilitates quicker troubleshooting and optimization [6].

- **Inefficient Resource Utilization** Traditional networking often leads to suboptimal resource utilization due to static routing and configurations. SDN's programmable nature enables dynamic resource allocation, ensuring that resources are efficiently distributed based on real-time demands contributing to improved overall network efficiency [6].

- **Lack of Application-Aware Networking** Traditional networks lack the granularity to cater to the specific requirements of diverse applications. SDN introduces application awareness by allowing developers to program network behavior based on application needs [6].

- **Limited Scalability** Scalability challenges are inherent in traditional networking architectures, especially when confronted with the increasing demands of modern applications. SDN, with its centralized control, facilitates scalability by abstracting the underlying hardware details [6].

2.1.2 Flows

In SDN, a flow represents a sequence of data packets traveling between specific source and destination points within a network. Flows vary in properties such as size, duration, and rate, which encompass characteristics such as total data transfer and packet transmission speed. [7, 8, 9]. SDN controllers install flow-level rules by dynamically programming forwarding rules on network devices. These rules dictate how traffic should be handled, directing packets based on predefined criteria such as source and destination addresses or packet attributes. This dynamic programming allows for centralized control over network behavior, enabling efficient management and optimization of traffic flows [10, 11]. Flow-based routing techniques play a pivotal role in load balancing and traffic engineering, optimizing resource utilization and network performance. Leveraging flow information, these techniques distribute traffic across multiple paths to prevent congestion and ensure efficient bandwidth usage [12, 13, 14, 15, 16].

2.1.3 OpenFlow

Within SDN, the integration of OpenFlow facilitates the separation of the control and data planes, establishing a standardized interface for communication between the SDN controller and OpenFlow-compatible network switches. OpenFlow's core function is to prescribe rules for the SDN controller to communicate instructions to switches regarding the processing and forwarding of network packets. Operating on a flow-based forwarding model, OpenFlow diverges from traditional networks with fixed routing tables. Armed with a comprehensive network view, the SDN controller directs switches on how to handle specific flows, ensuring dynamic and programmable control over network behavior. Upon receiving instructions, switches update their flow tables accordingly, enabling them to make forwarding decisions based on the defined rules. The OpenFlow protocol consists of a set of well-defined message types, each serving a specific purpose in facilitating communication between the controller and the switches [6, 10].

In SDN, Southbound Application Programming Interfaces (APIs) expand

communication beyond OpenFlow, enabling interaction between the SDN controller and network devices, particularly switches in the data plane. They standardize instructions, policies, and command transmission. Examples of Southbound APIs include OpenFlow, Network Configuration Protocol (NETCONF), RESTful, Border Gateway Protocol - Link State (BGP-LS), and P4 [6]. Similarly, Northbound APIs facilitate communication between the control plane and applications, abstracting network functionality for upper-layer interaction. They enable the extraction of network information and command issuance, enhancing dynamic network management. Examples of Northbound APIs include RESTful, WebSocket, and OpenStack Neutron.

2.1.4 SDN Controller Functionality

As the orchestrator and brain of SDN architecture, the SDN controller performs a multitude of functionalities. Following is an explanation of common SDN controller functionalities, including key aspects such as flow table management and policy enforcement [6, 17].

- **Topology Discovery:** One of the fundamental tasks of an SDN controller is to gain a comprehensive understanding of the network's topology. Through various southbound APIs, such as OpenFlow, the controller discovers network devices, their interconnections, and the overall structure of the network.
- **Flow Table Management:** SDN controllers maintain flow tables within switches, specifying how each switch should handle different types of network traffic. The controller is responsible for dynamically populating and updating these flow tables based on network policies, traffic patterns, and the overall state of the network.
- **Routing and Path Computation:** Leveraging the global view of the network, the SDN controller determines optimal paths for data packets to traverse. By utilizing path computation algorithms and considering factors like link capacities, latency, and network policies, the controller ensures efficient and effective routing.
- **Policy Enforcement:** SDN allows for the centralized definition and enforcement of network policies. The controller ensures that network-wide policies, such as Quality of Service (QoS), security, and access control, are consistently applied across the entire infrastructure.

- **Dynamic Adaptation:** In response to changing network conditions, the SDN controller dynamically adapts the network's behavior. This includes rerouting traffic to avoid congested links, adjusting QoS parameters based on demand, and reacting to events such as link failures or network attacks.
- **Service Chaining:** SDN controllers can facilitate service chaining, a mechanism where network services are orchestrated to create specific service paths for traffic. This enables the insertion of services like firewalls, load balancers, and intrusion detection systems into the network flow.
- **Integration with Applications:** Through northbound APIs, the SDN controller provides a standardized interface for applications to interact with the network. This integration allows applications to express their requirements, enabling innovative solutions such as network optimization, monitoring, and analytics.
- **Fault Detection and Recovery:** The SDN controller monitors the health of the network and detects faults or failures. In the event of a network anomaly, the controller can initiate recovery mechanisms, such as rerouting traffic or isolating malfunctioning components, to maintain network reliability.
- **Load Balancing:** By intelligently distributing network traffic across multiple paths, the SDN controller optimizes resource utilization and avoids congestion. Load balancing strategies are implemented to ensure even distribution and efficient utilization of network resources [6].

2.1.5 Network Virtualization

In SDN, network virtualization aims to decouple the logical network functions from the underlying physical infrastructure. This abstraction allows for the creation of multiple virtual networks, each with its own set of network services, policies, and addressing schemes. The primary components involved in network virtualization are [18, 17]:

- **Hypervisor:** Network virtualization often starts at the hypervisor level in virtualized environments. The hypervisor creates virtual switches that operate within the virtualization host, and Virtual Machines (VMs) are connected to these switches. These virtual switches serve as the entry points for network traffic into the virtualized environment.

- **Virtual Network Overlays:** Virtual network overlays provide a logical abstraction that enables the creation of multiple independent virtual networks on top of the physical network infrastructure. Technologies like Virtual Extensible Local Area Network (VXLAN), Network Virtualization using Generic Routing Encapsulation (NVGRE), and Geneve are commonly used to implement these overlays. These overlays encapsulate network packets, allowing them to traverse the physical network while maintaining logical isolation between virtual networks.
- **SDN Controller Integration:** SDN controllers play a crucial role in network virtualization by providing a centralized point for managing virtual networks. The controller orchestrates the creation, modification, and removal of virtual networks.

Network virtualization enables a spectrum of functionalities and is identified as Network Function Virtualization (NFV). This section explores the technical nuances of NFV, concentrating on both the creation and management of virtual networks. Examples of network virtualization functionalities include [18, 17, 19]:

- **Tenant Isolation:** Network virtualization allows for the creation of isolated virtual networks for different tenants or customers. Each tenant can have its own set of virtual switches, routers, and addressing schemes, thereby providing a secure and segregated environment.
- **Service Chaining:** Virtual networks dynamically chain with services like firewalls, load balancers, and intrusion detection systems. This ensures traffic follows predefined paths for efficient service use. NFV facilitates deployment by decoupling services from hardware, using virtualization to instantiate and manage services on VMs.
- **Integration with Cloud Environments:** In cloud environments, network virtualization facilitates the creation and management of virtual networks for different cloud tenants. It aligns with multi-tenancy principles, enabling cloud providers to offer customized networking solutions to their customers.
- **Network Slicing:** Network slicing takes the concept of network virtualization further by creating slices of the network optimized for specific use cases or applications. This fine-grained segmentation allows for tailored network configurations [17].

2.1.6 Performance Considerations of SDN

While SDN has enhanced concepts like flexibility, adaptability, and control over network resources, it has also impacted the performance aspects compared to traditional networking approaches [20]. The following section explores the balance between the advantages and challenges that SDN introduces concerning latency and throughput.

2.1.6.1 Latency

Latency is a crucial aspect of all networks. The separation of control and data planes in SDN architecture has a dual effect on latency. On the positive side, it enables dynamic and optimized traffic routing, potentially reducing end-to-end latency. The swift local decision-making, facilitated by this separation, enhances data transmission efficiency, contributing to improved network responsiveness. However, challenges arise when necessary flows or rules are not pre-installed in the data plane. In such cases, the switch must consult the central controller for guidance on routing a specific flow, introducing additional latency. This dependency on real-time decisions from the central controller can lead to delays [6, 20].

2.1.6.2 Throughput Dynamics

Unlike traditional networks, SDN's decoupling of the control and data planes allows for dynamic adjustments, optimizing throughput through efficient resource allocation. Despite these advantages, challenges exist. The centralized control in SDN may cause delays in decision-making, impacting real-time adaptability, especially in scenarios with rapid changes. When many switches send a high volume of control messages, this centralized control can become a bottleneck. Distributed controllers like Onix offer an alternative by dispersing control responsibilities and mitigating potential bottlenecks associated with a single centralized controller [21, 6, 22, 20].

2.1.7 SDN in Real-world Deployments

The emergence of SDN has changed how networking is done in everything from small home offices, and large corporations to data centers, Internet Exchange Points (IXPs), and Wide Area Networks (WANs). This section describes two real-world applications of SDN, namely Software-Defined

Internet Exchange (SDX) and B4, showcasing their impact on the global internet.

2.1.7.1 SDX: A Software Defined Internet Exchange

SDN transforms traditional IXPs through SDX. SDX replaces the static configurations of traditional IXPs with a dynamic, programmable framework. The controller in SDX facilitates real-time adjustments and interactions among network participants. This dynamic adaptability offers a significant contrast to traditional IXPs, enabling faster adaptation to network demands and fostering innovation in service offerings. SDX effectively addresses challenges posed by diverse internet traffic patterns by allowing administrators to dynamically allocate resources based on application-specific needs. Among the key contributions of SDX are enhancing network performance and user experience through application-specific routing and presenting a virtual SDN switch for each participating network. This setup allows for the implementation of flexible policies that interact safely with the existing Border Gateway Protocol (BGP), ensuring efficient and secure data flow across networks [23].

2.1.7.2 Revolutionizing WANs with B4

WANs face several challenges including scalability, flexibility, underutilization of links, and cost-efficiency. Addressing these issues, Sushant Jain *et al.* explores Google's Software-Defined Wide Area Network (SD-WAN), known as B4. B4 confronts traditional WAN limitations such as the inefficient use of links and the growing demands for bandwidth and performance. Key SDN techniques employed by B4 include centralized control, dynamic path computation, Traffic Engineering (TE), OpenFlow, and traffic splitting. These methods ensure optimal resource use and address the inefficiencies of conventional WANs. The effectiveness of B4 is highlighted by its ability to drive WAN links to near 100% utilization for extended periods, showcasing the potential of SDN to meet the high bandwidth demands in WANs [24].

2.2 Open vSwitch

The evolution of network virtualization within SDN has marked a significant transition from primarily physical to primarily virtual network ports. In this new paradigm, network switches often operate within the hypervisor to manage workloads. As the popularity and complexity of virtual networking

and network virtualization have increased, several limitations of traditional virtual switches have become apparent. These include restricted scalability, which is often tied to specific hypervisors, a lack of flexibility in supporting advanced network features and protocols, and increased management complexity. This complexity is due to intricate configuration interfaces and limited transparency into virtual network traffic. It is within this context that OVS emerged as a significant player in the domain. OVS represents a solution to the challenges posed by the intricate nature of virtualized networking environments and the shortcomings inherent in conventional virtual switches. The interplay between SDN and OVS becomes apparent as OVS integrates with the overarching SDN framework. In virtualized environments, the integration of hypervisors with network switches naturally leads to the hypervisor serving as the initial switch for incoming traffic from workloads. OVS plays an important role in enhancing the flexibility, scalability, and efficiency of network management within virtualized environments [1, 2]. Continuing, the motivation and architecture of OVS are presented, with a primary focus on its integration with the operating system kernel, as well as key aspects like packet classification, flow caching, and cache invalidation.

2.2.1 Open vSwitch and OpenFlow

From its inception, OVS has distinguished itself by embodying OpenFlow capabilities. This has positioned OVS as a versatile and adaptable solution within the dynamic landscape of networking. By being OpenFlow-capable, OVS avoids being tethered to a singular network control stack, offering re-programmability that sets it apart from other virtual switches with fixed packet processing pipelines, often mirroring the rigid nature of forwarding Application-Specific Integrated Circuits (ASICs). OpenFlow empowers OVS with the ability to dynamically reconfigure its forwarding behavior. This flexibility is a key attribute that enhances the adaptability and responsiveness of OVS to varying network requirements. OpenFlow allows for the dynamic manipulation of packet flows, enabling the implementation of custom forwarding and routing decisions as well as offering functions enabling TE, load balancing, bandwidth management, and security policies [1, 3].

However, while OpenFlow excels in dynamic packet manipulation and forwarding decisions, it has limitations in controlling more abstract or logical aspects of a network. Tasks such as adding or removing virtual network switches and ports, associating OpenFlow controllers and bridges, and enabling or disabling the Spanning Tree Protocol (STP) fall outside

the purview of OpenFlow. To address these broader network configuration requirements, OVS incorporates another component known as the OVS Database (OVSDB). This component serves as a configuration database, extending the capabilities of OVS beyond the confines of OpenFlow [1].

2.2.2 Open vSwitch Database

The OVSDB is the central component of OVS, serving as the repository for all network configuration details. Primarily, OVSDB functions as a configuration database, meticulously handling the storage and retrieval of essential information. This information encompasses not only the basic configuration of network switches and ports but also extends to more complex elements such as QoS queues. These capabilities surpass those provided by OpenFlow, which is primarily concerned with the dynamic manipulation of packet flows. The structured storage within OVSDB facilitates efficient modifications and retrieval of configuration parameters, thereby enabling OVS to dynamically respond to evolving network conditions [1].

2.2.2.1 OVSDB Server

At the heart of the OVSDB architecture lies the OVSDB server, a component responsible for orchestrating the interactions between the various entities within the OVS ecosystem. The server is tasked with managing the database, handling requests from OVSDB clients, and ensuring the integrity and consistency of the stored configuration data. Figure 2.1 shows the location of the OVSDB server within OVS and how it utilizes OpenFlow to connect to the SDN controller [1].

2.2.2.2 OVSDB Client

On the other side of the equation, OVSDB clients play a crucial role in interacting with the OVSDB server. These clients act as the interface through which external entities, such as SDN controllers, communicate with the OVSDB server to query or modify the network configuration. By facilitating communication between the external entities and the OVSDB server, OVSDB clients enable the integration of OVS into broader network management frameworks [1].

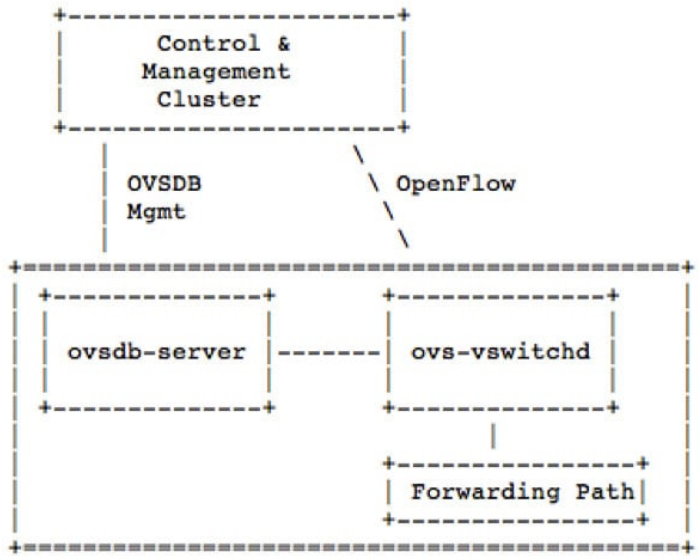


Figure 2.1: The location of the OVSDB server within OVS, depicting the relationship between OpenFlow and the SDN controller [1].

2.2.3 Open vSwitch Daemon

At the heart of managing and controlling OVS switches on the local machine lies the OVS Daemon, commonly referred to as `ovs-vswitchd`. This daemon plays a crucial role in configuring and operating OVS datapaths, ensuring efficient switching across bridges as described in its configuration files. Responsibilities of `ovs-vswitchd` include:

1. **Datapath Setup:** Upon startup, `ovs-vswitchd` retrieves its configuration from the specified OVSDB server. It then proceeds to set up OVS datapaths.

2. **Dynamic Configuration Updates:** As the OVSDB changes, `ovs-vswitchd` updates its configuration to reflect the modifications. This dynamic adaptability ensures that the OVS bridges remain in sync with evolving network requirements.

3. **Switch Feature Configuration:** `ovs-vswitchd` configures switches with a range of features, including:

    - Layer 2 (L2) switching with Media Access Control (MAC) learning.

- NIC bonding with automatic fail-over and source MAC-based Transmit (TX) load balancing.
- 802.1Q Virtual Local Area Network (VLAN) support.
- Port mirroring, with optional VLAN tagging.
- Connectivity to an external OpenFlow controller, such as NOX or OVN.

2.2.3.1 Integration with OVSDB

`ovs-vswitchd` relies on the OVSDB to obtain its initial configuration and to adapt to dynamic changes. The specified OVSDB server serves as the source of truth for `ovs-vswitchd`, ensuring that the daemon operates based on the latest network configurations. This integration with OVSDB involves the retrieval of configurations related to bridges, switches, and ports. The communication between `ovs-vswitchd` and OVSDB ensures that the daemon remains in lockstep with the broader network management framework.

2.2.4 Datapath kernel module

For performance-related considerations, OVS adopts a split design, separating the OVS architecture into a user space process and a datapath kernel module. In this structure, policies are set in user space, while the datapath kernel module handles tasks such as packet processing and forwarding, flow table management, and interaction with the underlying networking stack [25]. Figure 2.2 shows how the kernel's datapath module manages incoming packets from physical or virtual NICs, following instructions from `ovs-vswitchd`. These instructions, called actions, dictate transmission details and forwarding, packet modifications, sampling, or dropping. If no specific instructions exist, the kernel datapath defers to `ovs-vswitchd` through an upcall for decision-making. In user space, `ovs-vswitchd` analyzes the packet and instructs the datapath on the appropriate action [1, 2, 3]. Notably, the flow rules from `ovs-vswitchd` are cached in the datapath kernel module to expedite future retrievals, a concept elaborated further in Section 2.2.6 and 2.2.7.

2.2.5 Packet Classification

In OVS, packet classification serves the purpose of directing traffic according to the OpenFlow rules orchestrated by a central SDN controller. Unlike

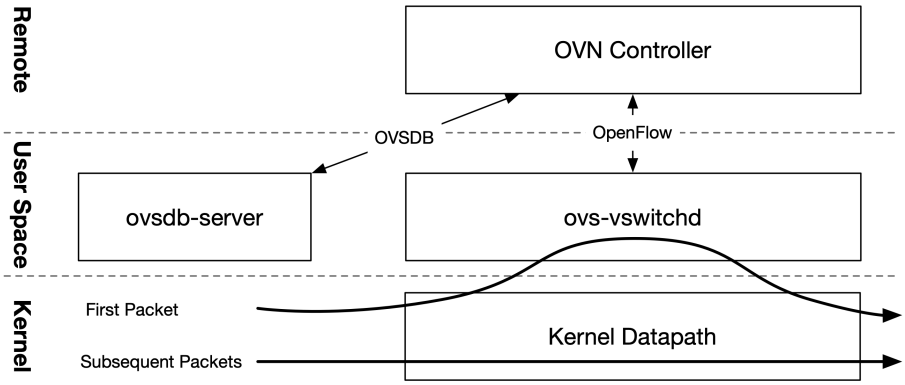


Figure 2.2: The components and interfaces of OVS including the kernel datapath responsible for forwarding networking packets [1].

traditional routing tables, OVS dynamically enforces policies, enables traffic management and facilitates load balancing through a flow-based approach for packet classification. The underlying mechanism for packet classification in OVS is a Tuple Space Search (TSS) classifier that resides in the kernel space. This classifier relies on a series of hash lookups to efficiently categorize incoming packets based on their attributes. To generate the necessary hashes for classification, a 5-tuple operation is used. This involves considering the source and destination addresses, protocol type, and source and destination ports to produce unique identifiers for efficient packet categorization. To classify packets according to the policy determined by OpenFlow, the TSS classifier employs a pipeline of flow tables implemented as simple hash tables [1].

Figure 2.3 shows a pipeline consisting of n hash tables, a search of the classifier must examine every hash table requiring n hash lookups. If the search returns empty, there are no matches for the specific flow. If there is a match in a single table, that match is returned. If more than one table returns a match, the classifier returns the match with the highest priority. The initial table in the hash table pipeline commonly handles the physical-to-logical translation, followed by a second table for L2 lookups within the logical space. The search then proceeds to the actual tables that govern how switches and routers are configured in the system. The search culminates in the last table for logical-to-physical conversion before dispatching the packet through the egress port [1]. By implementing the TSS classifier with hash tables, rather than alternative decision tree classification algorithms, the classifier enables favorable algorithmic complexity properties such as updating the hash tables

in $O(1)$ time and achieving $O(f)$ space complexity, where f represents the number of flows. However, packet classification may still be expensive in real-world production environments. For example, a typical deployment of the VMware NSX virtual networking platform requires 25 hash tables. This results in ~100 hash lookups needed to classify a single packet, representing a considerable number of hashes for this type of application. Consequently, packet classification in OVS becomes expensive on general-purpose Central Processing Units (CPUs) [1, 2, 3]. To mitigate this challenge, OVS implements flow caching, an approach elaborated upon in Section 2.2.6.

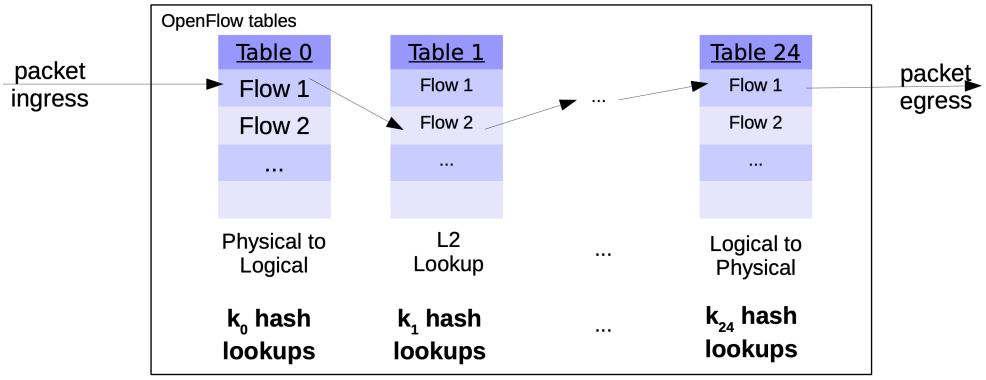


Figure 2.3: The pipeline of OpenFlow flow tables implemented as hash tables for use in the TSS classifier [1].

2.2.6 Microflow cache

OVS employs flow caching to address the challenges associated with the extended packet processing pipelines and the high computational cost involved in packet classification within network virtualization. Flow caching, although not conventionally utilized in networking due to a tendency to consider worst-case scenarios, proves to be a useful solution for mitigating these challenges. Unlike traditional hardware switches that have ASIC cores dedicated solely to handling networking tasks, virtual switches, which coexist with other VM components on the hypervisor, necessitate efficient resource utilization to avoid impeding the performance of other concurrent tasks within the virtualized environment. Flow caching facilitates this resource sharing by optimizing the handling of packet flows and minimizing the computational overhead associated with packet classification [1]. In OVS, flow caching is implemented within the kernel module, streamlining hash lookups to a single operation. Termed the “microflow cache”, this caching mechanism is designed

such that a single cache entry precisely matches all packet header fields supported by OpenFlow. In this context, a microflow refers to an individual stream of packets within the network. Figure 2.4 illustrates how, through the utilization of the microflow cache, packets can be classified with a single hash operation instead of traversing the pipeline consisting of n hash tables. This optimization effectively reduces the time complexity needed for packet classification from $O(n)$ to $O(1)$. The operational essence of the microflow cache involves the continuous population of new entries each time a previously unseen flow arrives. This process is highly efficient during the classification of packets after the initial arrival of a flow, as the cached information enables a speed improvement of up to $100\times$ compared to traversing the full hash table pipeline explained in Section 2.2.5 [1, 2, 3].

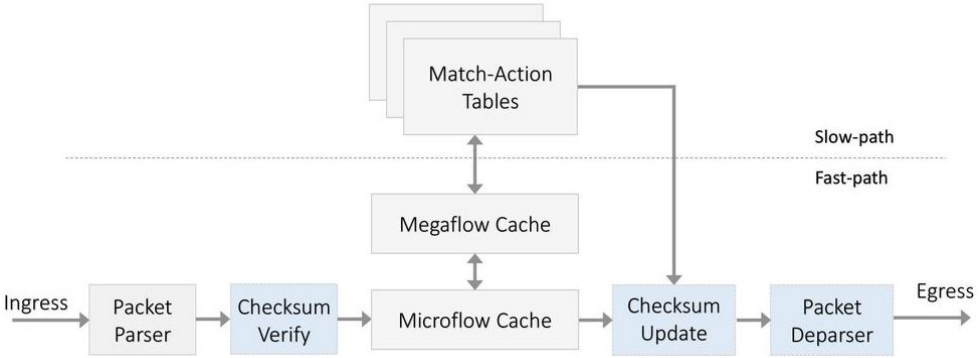


Figure 2.4: Utilizing the microflow cache reduces the number of hash operations required for packet classification to a single operation [1].

The implementation of the microflow achieves a significant speedup of two orders of magnitude when compared to examining every hash table in the packet classification pipeline. However, its effectiveness is limited in handling traffic patterns characterized by short-lived flows. This limitation is particularly evident in scenarios involving port scans, malicious activities, and applications with rendezvous patterns, such as peer-to-peer networking and distributed systems requiring coordination traffic. The inherent challenge lies in the fact that the flow size of such traffic is too small for the microflow cache to effectively reduce the associated overhead, necessitating at least one TSS classifier search. In practical terms, the gain achieved by the microflow cache in these scenarios is not commensurate with the incurred cost, resulting in a notably low hit rate [1, 2].

2.2.7 Megaflow cache

In addressing the challenges associated with classifying packets within traffic patterns characterized by short-lived flows, OVS introduces a second cache located within the datapath kernel module, known as the "megaflow cache". To achieve the effective classification of packets in short-lived flows, the megaflow cache exhibits a greater degree of generality than the microflow cache. It accomplishes this by supporting the caching of forwarding decisions for larger aggregates of traffic and more general-purpose flows compared to its microflow counterpart. The increased generality requires a handful of hash lookups to retrieve a flow from the megaflow cache, as opposed to the single lookup required for the microflow cache. To extend the cache's coverage to a larger set of flows, one approach involves the megaflow cache taking the cross-product of all OpenFlow tables residing in user space and consolidating these flows into a single table. However, this results in an exponential growth in the number of entries leading to impractical levels that cannot be accommodated in memory. To mitigate this issue and reduce memory consumption, the megaflow cache employs a lazy approach for populating the cache. Instead of precomputing the cross-product in advance, the `ovs-vswitchd` dynamically computes cache entries incrementally and reactively as packets arrive in OVS. This approach means that incoming packets drive cache population, adapting to evolving traffic aggregates by adding new entries and removing old ones. The on-the-fly population of the megaflow cache not only maintains performance comparable to or better than the naive precomputation approach but also leverages traffic locality to achieve a practical cache size that fits into memory efficiently [1].

Distinguishing itself from the microflow cache, the megaflow cache operates without priorities, expediting packet classification. Using the TSS algorithm, megaflows with identical masks are organized into corresponding hash tables, denoted as H_m, wherein entries consist of masked flow keys. When presented with a packet header h, the TSS algorithm traverses each hash table to locate an entry that aligns with the header h. This alignment is determined by a match between the masked header and the masked key within the entry. The time complexity of a lookup in the megaflow cache using the TSS algorithm is $O(n)$, where n represents the number of hash tables [3]. Unlike the TSS classifier covered in Section 2.2.5, the megaflow cache's search for a match can terminate as soon as it finds a match, eliminating the need to search through all hash tables. Figure 2.5 illustrates how, due to the additional cost associated with retrieving cache entries from the more

general megaflow cache, the microflow cache is preserved as an Layer 1 (L1) cache, resembling CPU caching. The megaflow cache is consulted only in the event of a microflow cache miss, where the microflow cache functions as a hash table pointing to the location of the corresponding flow within the megaflow cache. Only when a cache miss occurs in both caches is the user space `ovs-vswitchd` queried, and the result is cached in the kernel [1].

While the megaflow cache avoids upcalls to user space for packet classification, it introduces added hash lookups. The efficiency of the megaflow cache relies on these added hash lookups outweighing the cost of accessing OpenFlow tables in user space. In summary, the microflow cache offers the advantage of requiring a single hash operation for retrieving a cached flow but involves the TSS classifier, which demands a large number of hash operations slowing down performance. Conversely, the megaflow cache requires more than one hash for all packets. Through the synergistic use of these two caches, OVS achieves an optimal balance, minimizing lookups for the first packet of a flow to only a handful while streamlining subsequent packets of the same flow with a single lookup [1].

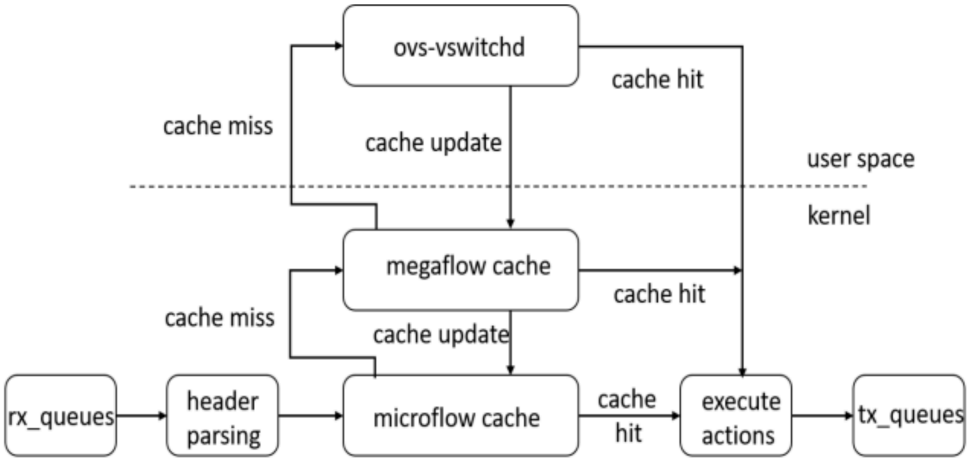


Figure 2.5: OVS datapath including the interplay between the microflow and megaflow caches in the kernel and the `ovs-vswitchd` residing in user space [26, 1].

2.2.8 Cache invalidation

Cache invalidation is a critical aspect introduced with caching mechanisms in any system. Within the OVS architecture, the responsibility of cache invalidation falls upon the revalidator process, which is invoked by

`ovs-vswitchd`. This process is responsible for maintaining the relevance of entries in the caches, ensuring its synchronization with dynamic changes occurring in the OpenFlow table. The tasks performed by the OVS revalidator process include a range of operations, including the updating of OpenFlow packet and byte statistics, removal of idle datapath flows, and verification that the installed datapath flows align with the configured OpenFlow rules. Changes in the dynamics of the OpenFlow table can occur due to various reasons, with the primary cause being alterations to the OpenFlow table itself[1, 27]. With the introduction of revalidation threads in OVS, the performance of cache revalidation can match the flow setup rate, resulting in increased kernel cache sizes. A revalidation thread operates as an independent entity processing datapath flows, updating OpenFlow statistics, and effecting necessary updates or removals. These threads are spawned during the initialization of the OVS datapath and the creation of a bridge. The number of revalidation threads can be adjusted using the `n-revalidator-threads` setting [1, 27]. Following is an overview of how the revalidation process for each kernel cache works.

2.2.8.1 Microflow revalidation

Microflow revalidation involves a straightforward process for maintaining the relevance of flows within the microflow cache. The microflow cache, characterized by a fixed size, adopts a simple yet effective pseudo-random replacement policy. This policy dictates that as new microflows arrive, they replace older ones in the cache. The microflow cache essentially functions as a mapping from the microflow hash table to another hash table within the kernel TSS classifier. This means that erroneous entries in the microflow cache are updated when new packets match in the microflow cache [1].

2.2.8.2 Megaflow revalidation

Due to the aggregated and general nature of the megaflow cache, its revalidation process is more complex. To streamline this process, the `ovs-vswitchd` in user space actively acquires datapath cache statistics through periodic polling of the kernel module for flow packet and byte counters, typically performed at approximately one-second intervals. These obtained statistics help in determining the fate of flows within the megaflow cache. Flows are evaluated based on their level of activity, with those considered less active facing potential removal to make room for more relevant flows in the cache. The decision-making process is multifaceted; not only

are flows removed when the cache approaches full capacity, but they are also subject to removal if they remain idle for a default duration of 10 seconds. Furthermore, as the megaflow cache nears its maximum size, a dynamic threshold is employed to regulate the idle duration for a flow. This mechanism ensures that as the cache approaches its limit, the tolerance for idle flows is reduced, forcing the cache to shrink and prioritize more active and pertinent flows [1, 27].

2.2.8.3 Dump and sweep

The orchestration of the microflow and megaflow revalidation processes is managed by the revalidation threads, operating in two distinct phases known as the "dump" and "sweep" phases. To ensure synchronization, all revalidator threads synchronize between these phases, ensuring that each thread is exclusively engaged in either the dump or sweep phase, but not a combination of both [27]. The decision on when to execute the dump and sweep phases is a prerogative of the leader thread, with a minimum time interval of 5 milliseconds between successive runs [27]. The leader thread is tasked with the following responsibilities:

1. Determine the `flow_limit`, defaulting to 200,000 flows.
2. Retrieve the current number of flows in the datapath and set `n_flows` to that value.
3. Set `max_n_flows` to the maximum ever observed, meaning it is set to `n_flows` if `n_flows` is greater.
4. Reset `avg_n_flows` to mirror the number of flows in the current run.
5. Initiate a flow dump, initiating the start of the revalidation process for the upcoming phases.

Dump Phase

In the initial dump phase, `ovs-vswitchd` retrieves all cache entries from the kernel, locating the associated user space data structures. Subsequently, it marks idle flows for deletion, removing them from the cache. If a flow is not marked for deletion, the actual revalidation process takes place, ensuring the cache remains synchronized with the network's current state [27].

Sweep Phase

During the sweep phase, `ovs-vswitchd` cleans up internal data structures after removing kernel cache entries. Each revalidator is assigned a specific set of data structures to manage. These data structures contain unique identifiers for network traffic. The revalidators systematically go through these data structures to determine if any entries need to be removed. This process involves checking individual network flows that were not included in the dump phase to verify their validity and collect statistical data [27].

2.2.9 Consequences of cache misses

As the volume of OpenFlow rules in OVS expands, the megaflow cache becomes the primary bottleneck impeding datapath performance within OVS. This bottleneck arises due to the inherent limitation of the megaflow cache, which cannot accommodate all OpenFlow rules simultaneously. With an expanding rule set, the cache inevitably reaches its capacity, leading to a higher probability of encountering flows that match rules not currently stored in the cache. This mismatch between incoming flows and cached rules necessitates upcalls to the control path for handling and processing. Therefore, as the number of rules increases, the likelihood of encountering such cache misses rises, resulting in a corresponding escalation in upcall frequency [3]. A performance analysis of OVS by Alon Rashelbach *et al.* is depicted in Figure 2.6 and showcases OVS throughput alongside the rates of deletions and upcalls sampled at 100 ms intervals for a dataset comprising 100,000 rules. Notably, as the frequency of upcalls rises, there is a corresponding decrease in throughput. Similarly, the system experiences performance degradation due to deletions, prompted by periodic megaflow cache cleanup activities aimed at removing idle flows. This performance trend remains consistent across various rule-set configurations [3, 21].

2.3 Open Virtual Networking

OVN emerges as an extension to OVS due to its limitations in addressing the complexities of modern network infrastructures. While OVS excels in virtual switching, its feature set fails to meet the demands of complex network environments. OVN rectifies this shortfall by augmenting OVS with advanced functionalities such as distributed routing, logical switching, and network virtualization enriching the capabilities of OVS to deliver a comprehensive

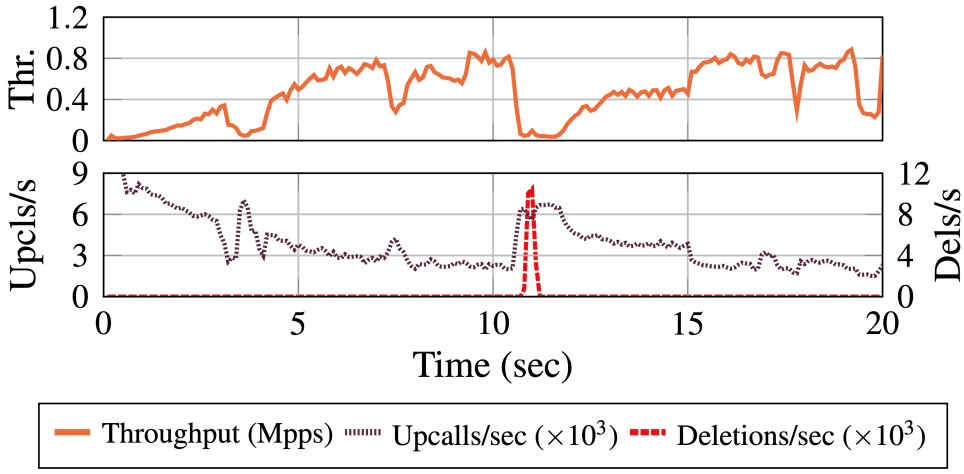


Figure 2.6: The correlation between the number of upcalls and throughput in OVS [3].

virtual networking solution tailored to modern networking demands. This evolution is imperative as virtual networking necessitates abstraction from physical constraints while retaining parity in features with physical networks [28, 29, 30, 31].

2.3.1 Architecture

Developed by the same open-source community as OVS, OVN introduces L2 and Layer 3 (L3) network virtualization for OVS. By building upon OVS, OVN extends its reach to provide virtual networking solutions suitable for cloud environments, data centers, and virtualized network functions. The architecture of OVN comprises several key components that collaborate to deliver comprehensive network virtualization capabilities [28, 30, 31].

- **Northbound Database (NB DB):** This component is the interface for external systems to interact with OVN. The NB DB stores logical network configuration information, such as logical switches, routers, and ports, providing a high-level abstraction of the virtual network topology [28, 30].
- **Southbound Database (SB DB):** Unlike its northern counterpart, the SB DB stores the state of the physical and logical network infrastructure. It maintains information about physical switches, ports, and other

network elements, facilitating communication between the logical and physical layers of the network [28, 30].

- **OVN Controller:** Acting as the brain of the OVN architecture, the OVN controller orchestrates network operations based on the information stored in the NB DB and SB DB. It translates logical network configuration into physical network rules and distributes them to the relevant OVS instances running on hypervisor hosts [28, 30].
- **Logical Switches and Routers:** These components provide the foundational building blocks for creating virtual network topologies within OVN. Logical switches enable communication between VMs and containers within the same virtual network, while logical routers facilitate inter-network communication between different virtual networks [28, 30].
- **Traffic Forwarding:** OVN implements network overlays by employing tunneling protocols, such as Generic Network Virtualization Encapsulation (Geneve), to encapsulate and transport network traffic between VMs and containers. The primary advantage of utilizing Geneve is its support for Type-Length-Value (TLV) header options, enabling the inclusion of additional metadata with the encapsulated packets. Upon the arrival of a packet at a logical switch, OVN utilizes flow tables to determine the appropriate actions for forwarding the packet, including routing it to the correct destination port or applying network policies [28, 30].
- **Integration with OVS:** OVN leverages OVS as the underlying data plane for forwarding network traffic. OVS instances running on hypervisor hosts process packets based on the rules distributed by the OVN controller [28, 30].

2.3.2 Control plane

The control plane operation of OVN is responsible for managing network configuration, synchronizing network state, and enforcing network policies within the virtualized network infrastructure. OVN's control plane orchestrates these tasks to ensure the consistent and reliable operation of the network. Network configuration management is facilitated through the NB DB, where administrators define and modify logical network elements such as switches, routers, ports, and network policies. The OVN controller translates these

high-level instructions into physical network rules and distributes them to relevant components, ensuring the network configuration reflects the desired state [28, 31]. OVN employs a distributed control plane architecture to synchronize network state across all components. The controller monitors changes in configuration and state and propagates them to the SB DB and OVS instances on hypervisor hosts. This synchronization guarantees that all network elements remain in sync and adhere to the defined policies and configurations. Policy enforcement is another crucial aspect managed by the control plane where network policies dictate the rules governing traffic within the virtualized environment. The OVN controller translates these policies into flow rules distributed to OVS instances, guiding how network traffic is processed and forwarded based on predefined criteria [28, 29, 30, 32].

2.3.3 OVN-Kubernetes

In modern cloud-native environments, OVN integrates with popular cloud platforms and orchestration frameworks like OpenStack and Kubernetes. Specifically, within Red Hat's Kubernetes distribution, known as OpenShift, OVN-Kubernetes (OVN-K) functions as the Container Network Interface (CNI) cluster network provider. It offers a rich set of features to manage network traffic flows effectively. OVN-K leverages OVN to provide scalable and flexible networking for Kubernetes clusters. By utilizing OVN, OVN-K ensures compatibility with various deployment scenarios and enables seamless integration with existing network infrastructures. One notable feature of OVN-K is its implementation of Kubernetes network policy support, including ingress and egress rules. This allows administrators to define fine-grained network policies to control the flow of traffic between pods and nodes [33, 34]. Figure 2.7 illustrates how OVN-K utilizes OVN for network management. Here, NB DB and SB DB run on controller nodes along with the OVN-controller, which communicates with `ovs-vswitchd` and `ovsdb-server` deployed on every node in the Kubernetes cluster [33, 29, 28, 31].

2.4 OpenShift Container Platform

OCP, a Kubernetes distribution developed by Red Hat, is a powerful solution for containerization, offering features that streamline the deployment, management, and scaling of containerized applications. It provides a consistent hybrid cloud foundation, enabling organizations to build and

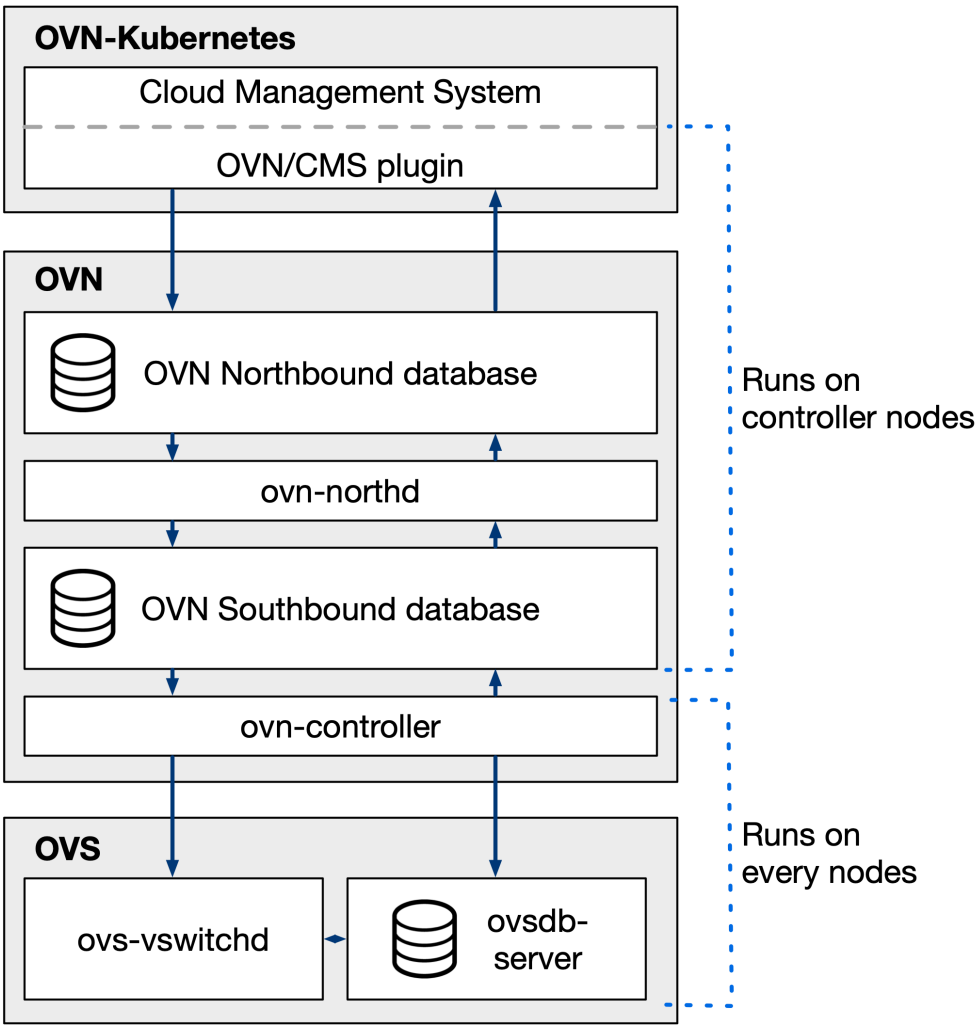


Figure 2.7: The OVN-K architecture and the location of the OVN controller, NB DB SB DB, and how it leverages OVS for packet forwarding [33].

expand their containerized workloads using a versatile infrastructure. As a container orchestration platform, OCP leverages Kubernetes to automate the deployment, scaling, and management of containerized applications, ensuring efficient and reliable operation across clusters. A key strength of OCP is its support for multi-cloud and hybrid cloud environments, allowing organizations to deploy and manage applications seamlessly across various environments, including on-premises data centers, public cloud providers, and edge locations [35, 36, 37, 31].

Additionally, OCP offers a robust ecosystem of tools, technologies, and integrations that enhance its functionality and adaptability. It includes container registries, monitoring solutions, logging frameworks, networking plugins, and service mesh architectures, providing a comprehensive suite of components that streamline integration with existing infrastructures and workflows. This ecosystem allows organizations to utilize their preferred tools and technologies while fully capitalizing on the benefits of containerization with OCP [38, 39, 31].

2.4.1 Architecture

The networking architecture of OCP is built on the integration of OVS and OVN, which work together to manage network resources and orchestrate data flow with precision and efficiency. At the center of this architecture, OVS functions as the virtual switch responsible for managing container networking within the OCP environment. As an open-source, software-based virtual switch, OVS operates at the data plane level, efficiently forwarding packets between network interfaces, VMs, and containers. OVS offers a robust feature set, including VLAN tagging, tunneling protocols, and flow-based forwarding, which integrates seamlessly with Kubernetes clusters to provide scalable and reliable networking solutions. It acts as the bridge between containerized workloads and the underlying network infrastructure, abstracting networking complexities to allow containers to communicate with each other and with external services effortlessly. The flexibility and extensibility of OVS make it an ideal choice for container networking, supporting dynamic allocation of network resources and efficient traffic management [38, 39, 40, 41, 31].

Complementing OVS, OVN adds another layer of sophistication to OCP's networking architecture, offering network virtualization capabilities tailored specifically for Kubernetes environments. Unlike traditional networking solutions, OVN abstracts network management tasks, simplifying the deployment and management of network resources within Kubernetes

clusters. OVN's role within OCP extends beyond basic packet forwarding; it facilitates network management of Kubernetes pods, nodes, and services [37]. By abstracting network policies, OVN empowers administrators to define and enforce fine-grained network policies, ensuring secure communication between pods while maintaining isolation and segmentation as per organizational requirements. In OCP, networking is streamlined through the integration of OVN-K as its CNI. OVN-K simplifies network configuration and management within Kubernetes clusters, offering native support for Kubernetes networking primitives. This choice enhances OCP's networking capabilities, ensuring reliable connectivity between pods, nodes, and external services while abstracting networking complexities for administrators [33, 34, 38, 39, 31].

2.5 Coflows

In the context of cluster computing environments, there is a growing demand for data-intensive applications that require efficient communication patterns to achieve optimal performance. Traditional networking approaches often face challenges in optimizing communication patterns for data-intensive applications like MapReduce, Spark, and distributed databases. These challenges include inefficient resource utilization, lack of coordination among distributed components, limited visibility into communication requirements, suboptimal routing decisions, and scalability issues. These shortcomings lead to suboptimal performance and resource utilization in cluster computing environments. These environments are commonly used for running large-scale applications that require significant computational resources, such as data processing, machine learning, and scientific simulations. Kubernetes is another type of cluster computing environment that allows users to deploy, manage, and scale containerized applications across a cluster of nodes [42, 33].

To address the challenges of applying traditional networking approaches in cluster computing environments, Chowdhury *et al.* introduced the concept of *coflows* as a networking abstraction in their work, "*Coflow: A Networking Abstraction for Cluster Applications*" [42]. The paper identifies the limitations of traditional networking approaches in addressing the diverse communication requirements of data-intensive applications. These applications often involve multiple concurrent flows with complex communication patterns, leading to inefficient resource utilization and increased completion times. Chowdhury *et al.* observe that many cluster computing applications are structured into multiple stages or involve machines grouped by functionality. In such

applications, communication occurs at the level of machine groups and is often dictated by application-specific semantics. This organizational structure leads to diverse communication patterns, with data flows between different groups of machines serving specific purposes and objectives. A coflow represents a collection of flows between two groups of machines, each with associated semantics and a collective objective. By grouping related flows into coflows, the networking infrastructure gains insights into the higher-level communication patterns of cluster applications, enabling more efficient scheduling and resource allocation strategies [42, 43].

A coflow can be thought of in more ordinary terms by considering the network behavior when initiating a network request, such as fetching a webpage through an Hypertext Transfer Protocol Secure (HTTPS) GET request, where multiple distinct data flows may occur. For instance, the process may begin with a Domain Name System (DNS) resolution request to translate the domain name to an Internet Protocol (IP) address. Subsequently, an HTTPS GET request is sent to retrieve the webpage's content from the server, followed by a query to a database server if the webpage relies on dynamic content. Finally, if the user interacts with the webpage, data may be pushed back to the server for storage or processing. All of these interactions, from DNS resolution to database queries, are flows belonging to the same application, which, when combined, form a coflow [42, 43].

2.5.1 Definition of a coflow

A semantically related collection of flows between two groups of machines is referred to as a coflow. Mathematically, a coflow c is defined as:

$$c(S, D) = \{f_1, f_2, ..., f_{|c|}\} \text{ where:}$$

- c represents the coflow,
- S is the source machine group,
- D is the destination machine group,
- $|c|$ represents the cardinality, denoting the number of flows within the set represented by c,
- f_i represents individual flows between machines in groups S and D, and
- $size(f_i)$ represents the size of individual flow f_i, typically measured in terms of bytes transmitted.

We can now represent a cluster computing application using a graph $G = (M, C)$, where:

- M is the set of machine groups, and
- C is the set of coflows connecting these machine groups.

The combined goal of a coflow influences not only the coordinated optimization strategies for its constituent flows but also dictates the essential information needed for such optimization [42]. This is evident when examining the primary objectives of cluster communication:

- **Meeting Deadlines:** The objective of meeting a deadline **D** requires ensuring that all flows within a coflow complete their transmission within the specified time constraint. To achieve this, rates for all flows can be set to ensure they finish by the deadline **D**.
- **Minimize Completion Time:** Minimizing the completion time of a coflow involves completing the transmission of all flows within the shortest possible duration. One approach to achieving this objective is to set the rates of individual flows such that the slowest progressing flow (f_{last}) finishes as quickly as possible, while ensuring that all other flows are complete by the time f_{last} finishes [42].

Both objectives require knowledge of $\mathit{size}(f_i)$ between S and D, for all f_i belonging to the set of flows represented by the cardinality $|c|$ in coflow c. This information is essential for determining transmission rates and optimizing flow completion times. However, the availability of additional information may vary depending on the application. For example, in long-running services where aggregation coflows are used, both the sources and destinations of each flow are known, along with the maximum size of individual flows. This additional information can further inform optimization strategies and help in making informed decisions about network resource allocation [42].

2.5.2 Constructing coflows

Several factors influence the formation of coflows, which are collections of flows with shared communication patterns and performance objectives:

- **Common Communication Patterns:** Flows that exhibit similar communication patterns or share common dependencies are grouped

into a coflow. These patterns may include data exchanges between different components of a distributed application, such as data producers and consumers, or between different stages of data processing pipelines [43].

- **Performance Objectives:** Flows with shared performance objectives, such as achieving a certain level of throughput, minimizing latency, or maximizing fairness, are often grouped into the same coflow. By grouping flows with similar performance requirements, the networking infrastructure can apply coherent scheduling and resource allocation strategies to optimize the overall performance of the coflow [43].

- **Temporal and Spatial Proximity:** Flows that originate from or are destined to nearby nodes within the cluster, either temporally or spatially, are more likely to be included in the same coflow. This consideration helps to minimize communication overhead and maximize resource utilization by optimizing the placement of data transfers within the cluster [43].

- **Application-Level Semantics:** Coflows may also be defined based on higher-level application semantics or data access patterns. For example, flows associated with a particular job or task within a distributed computation framework may be grouped into a single coflow to ensure coordinated data transfer and processing [43, 44].

2.5.3 Sincronia

In an effort focused on optimizing coflows, Saksham Agarwal *et al.* have in their work titled "*Sincronia: Near-Optimal Network Design for Coflows*" addressed the challenges related to efficient resource allocation and scheduling of network traffic in cluster computing environments. The paper introduces *Sincronia*, a framework designed to achieve near-optimal network performance by scheduling coflows more efficiently. Central to Sincronia's approach is its utilization of network congestion information to dynamically adapt scheduling decisions, thereby optimizing overall coflow completion times. Key technical aspects of Sincronia include its utilization of coflow-aware scheduling algorithms, which leverage both historical and real-time network congestion data to allocate resources efficiently. Additionally, the paper proposes a hierarchical scheduling architecture that allows for scalable and efficient coflow management in large-scale data center networks [44].

2.5.3.1 Workload generator

In the evaluation of the Sincronia framework, the authors employ two distinct workloads. The primary workload originates from previous network designs and consists of a 526-coflow trace derived from a one-hour operation of MapReduce tasks on a 3000-machine cluster within Facebook. However, this trace has several limitations, such as a limited number of coflows and a low network load, thereby constraining its representativeness as a cluster computing trace. To address these limitations, the authors leverage a second workload generated using a coflow workload generator. This generator facilitates the creation of traces that closely resemble the original Facebook trace while allowing for adjustments in the number of coflows, network load, and other parameters as needed. Throughout the evaluation of Sincronia, the results for these customized traces are based on 2000 coflows and a network load of 90% [44, 45, 46].

The workload generator, implemented in Python and publicly available on GitHub, is a versatile tool for generating coflow traces with customizable properties [46, 45]. These properties include:

- **Number of coflows**: An integer value greater than zero, specifies the desired number of coflows.
- **Coflow Size**: The aggregate size of constituent flows within a coflow.
- **Coflow Width**: The minimum number of sources or destinations utilized by the flows of a coflow.
- **Incast Ratio**: The ratio of the number of destinations to the number of sources of constituent flows.
- **Inter-arrival Time**: The duration between the arrival of two successive coflows, influencing the network load.

The output format of the workload generator produces a text file with the following format:

- **Line 1:** Provides metadata about the trace, including the number of input ports and the total number of coflows in the trace.
- **Subsequent Lines:** Each subsequent line represents a single coflow within the trace and follows a consistent format:
  - **Coflow ID:** Unique identifier for the coflow.

- **Arrival Time (in milliseconds):** Timestamp indicating the arrival time of the coflow.
- **Number of Flows:** The total number of constituent flows within the coflow.
- **Number of Sources:** Number of distinct source entities contributing to the flows within the coflow.
- **Number of Destinations:** Number of distinct destination entities receiving the flows within the coflow.
- **Flow Information:** For each flow within the coflow, the following details are provided:
  * **Source ID:** Identifier of the source entity for the flow.
  * **Destination ID:** Identifier of the destination entity for the flow.

This format enables straightforward interpretation and parsing of the generated traces, facilitating further analysis and evaluation of network behavior [44, 45, 46].

2.6 eXpress Data Path

eXpress Data Path (XDP) is a high-performance, programmable data plane framework within the Linux kernel that enables efficient packet processing at wire speed. It is designed to accelerate network functions by moving packet processing closer to the NIC, thereby reducing latency and improving overall network performance. Figure 2.8 illustrates how XDP operates at the lowest layers of the networking stack, allowing for direct access to incoming packets before they are processed by traditional kernel networking stack components such as socket buffers and network drivers. This direct access capability empowers developers to implement custom packet processing logic directly in the NIC's driver, bypassing the overhead associated with conventional networking stacks [25, 47].

One of the key advantages of XDP is its support for zero-copy packet handling. This means that packets can be processed and modified directly within the kernel space without the need to copy them to user space buffers. Traditional packet processing methods involve copying packets from kernel space to user space, which can introduce latency and consume CPU resources due to memory copying operations. With XDP's zero-copy capabilities, these

overheads are greatly reduced, leading to improved performance, especially in high-speed networking scenarios. By eliminating the need for packet copies to user space, XDP significantly reduces memory overhead and CPU utilization, making it well-suited for demanding networking applications where low latency and high throughput are crucial requirements. This zero-copy approach also benefits applications that require packet inspection, filtering, and manipulation, as it allows for direct access to packet data within the kernel, leading to faster processing and response times [25, 47, 48].

2.6.1 eBPF

An XDP program starts its execution with a pointer to an execution context object, which contains essential information such as pointers to the packet data and metadata. This execution context provides the necessary context for the XDP program to inspect, modify, or drop incoming packets based on custom logic implemented within the program. To organize code and data within an XDP program effectively, developers use the *SEC()* macro. This macro is crucial for adding annotations that guide the linker to place specific sections of code or data into named sections of the resulting Executable and Linkable Format (ELF) file generated during compilation and linking. These named sections are then recognized by the loader responsible for loading the program into the kernel [47].

Underlying XDP's packet processing capabilities are the concepts of Berkeley Packet Filter (BPF) and extended Berkeley Packet Filter (eBPF). BPF serves as a VM within the Linux kernel, executing sandboxed programs that efficiently filter and manipulate packets. These BPF programs, written in restricted C, are compiled into byte code format for execution within the BPF VM. eBPF extends the capabilities of BPF by introducing additional features and capabilities, such as the ability to handle more complex packet processing tasks while maintaining safety and security within the kernel environment. eBPF programs can interact with various kernel data structures, perform dynamic tracing, and execute custom logic directly within the networking datapath, making them powerful tools for network function acceleration and optimization [25]. When an eBPF program is loaded into the kernel, the kernel verifier plays a critical role in ensuring system safety. It checks for potential security vulnerabilities like out-of-bounds memory accesses, which could crash the kernel or compromise system integrity. The verifier also examines control flow, resource limits, and security policies to ensure that only safe and well-behaved eBPF programs run within the kernel [47, 49].

2.6.2 XDP actions

XDP actions are used to determine how incoming packets are handled within the networking datapath. These actions are specified within the eBPF program code and dictate the behavior of the packet processing logic [47, 50]. Below are some key XDP actions commonly used in eBPF programs:

- **XDP_PASS:** When an XDP program returns the `XDP_PASS` action, it indicates that the packet should continue its journey through the networking stack.
- **XDP_TX:** The `XDP_TX` action is used to transmit a packet directly from the XDP program without going through the normal networking stack.
- **XDP_DROP:** When an XDP program returns the `XDP_DROP` action, it instructs the system to discard the incoming packet immediately.
- **XDP_REDIRECT:** The `XDP_REDIRECT` action redirects the incoming packet to a different network interface or destination.
- **XDP_ABORTED:** The `XDP_ABORTED` action indicates that the XDP program encountered an error or exceptional condition during execution.

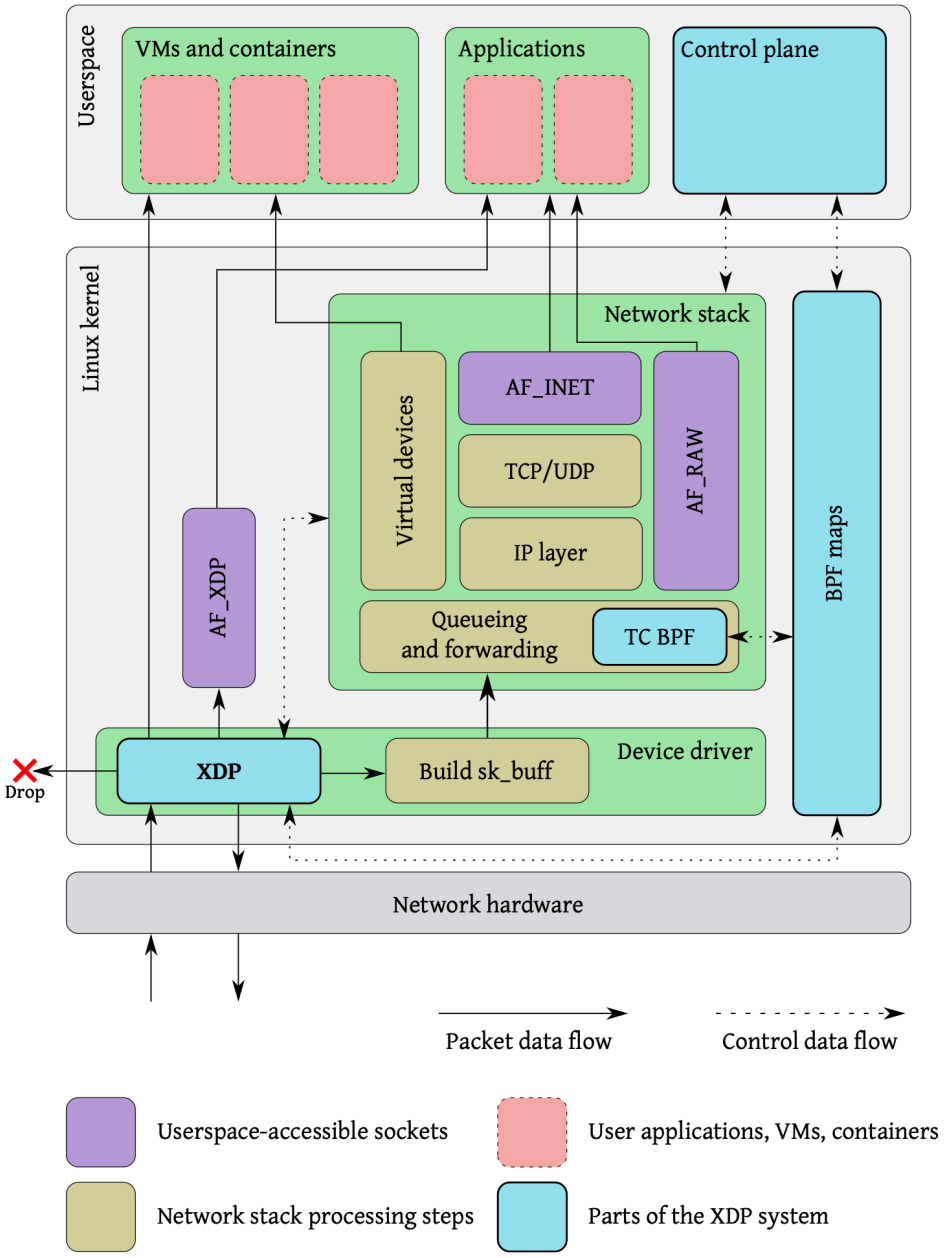


Figure 2.8: XDP's integration with the Linux network stack. To simplify the diagram, only the ingress path is shown [47].

2.7 Related work

Given that SDN and OVS are critical components of the Internet's infrastructure and container orchestration platforms such as Kubernetes and OCP, extensive research has been conducted in this area, with a focus on enhancing performance, latency, scalability, and reliability. This section presents previous related work in the field of SDN and OVS.

2.7.1 The Slow Path Needs an Accelerator Too

In the paper titled "*The Slow Path Needs an Accelerator Too!*" authored by Annus Zulfiqar *et al.*, an important concern regarding the slow path within OVS is highlighted. It is asserted that the slow path, which serves as the domain for OVS, is poised to emerge as a significant bottleneck in SDN environments. This projection is grounded in the escalating bandwidth capacities of physical networks, with 200 Gbps becoming increasingly prevalent in data center settings, alongside the ever-increasing complexity of network topologies. Traditionally, the networking community has directed its efforts towards optimizing per-packet processing by leveraging specialized silicon chips and Field Programmable Gate Array (FPGA)-based SmartNICs to accelerate the data plane. In contrast, the slow path predominantly relies on general-purpose CPUs for deployment. The accelerating pace of link rates poses challenges not only to the data plane but also to the slow path. Moreover, the evolving landscape of computing and applications, characterized by mega-scale, multi-tenant clouds, and highly disaggregated microservices, further exacerbates the strain on the slow path as it endeavors to scale alongside the burgeoning number of tenants and services [2].

Recent studies have revealed that the utilization of onboard CPUs introduces latency on the order of tens or hundreds of microseconds required for responding to network conditions, a significant contrast to the nanosecond-scale operations of ASICs and NICs [51]. In essence, the paper contends that for the slow path to achieve efficiency and performance scalability in next-generation networks, there must be a departure from reliance solely on CPU-based platforms towards the adoption of accelerated solutions, mirroring the advancements witnessed in the data plane with switching ASICs. The primary objective of the paper is to draw attention to this overlooked aspect of the networking stack, which teeters on the verge of becoming a bottleneck in the emergent landscape of next-generation networks. The authors advocate and demonstrate that contemporary CPU-based platforms, whether situated on end

hosts, SmartNICs, or switches, inadequately address the demands of the slow path [2].

2.7.1.1 Scaling the slow path

The conventional approach of scaling the slow path often involves allocating dedicated CPU cores to execute infrastructure tasks such as the slow path and hypervisor functions, while leveraging specialized hardware for offloading data plane operations. However, this strategy escalates operating costs for cloud providers. Furthermore, while offloading infrastructure processing to NICs may seem advantageous, it fails to address the inherent performance bottlenecks associated with CPUs. Moreover, the substantial influx of traffic into the virtual switch leads to control packets encountering excessive Head-of-Line (HoL) blocking, particularly when contending with other slow path traffic such as flow misses. Since tenant traffic, including control packets, is typically tunneled, the data plane of the NIC cannot prioritize these packets without resorting to deep-packet inspection. This limitation exacerbates the challenge of efficiently managing and prioritizing traffic within the slow path, ultimately impeding its scalability and performance in dynamic, high-throughput environments [2].

2.7.1.2 Findings

To assess the efficiency and performance ramifications of the slow path, the authors conducted benchmarking exercises, revealing several limitations associated with utilizing OVS on a general-purpose CPU as the slow path. Firstly, they observed that cache misses induce highly skewed slow-path processing times, characterized by long tails. This phenomenon underscores the challenges of maintaining consistent performance when cache access patterns deviate from the norm. Secondly, as the number of flow rules increases, caches become saturated, amplifying the cache revalidation cost. This escalation in revalidation overhead further strains CPU-based slow paths, impeding their ability to cope with expanding traffic volumes effectively. Lastly, with projected link rates surpassing 200 Gbps (equivalent to 288 million packets per second for minimum-sized packets), the slow path traffic is anticipated to reach approximately 92,000 requests per second. On a contemporary server CPU operating at 2.6 GHz, this translates to a staggering 28,000 clock cycles (equivalent to 10.8 microseconds) per request. These findings underscore the pressing need to address the performance limitations inherent in CPU-based slow paths [2].

2.7.1.3 Slow path accelerator properties

Addressing the bottlenecks associated with the slow path necessitates the adoption of a dedicated accelerator optimized explicitly for slow path processing, as advocated by the authors. The proposition put forth underscores the critical importance of constructing a Domain-Specific Accelerator (DSA) tailored specifically for the slow path. Consequently, the authors issue a rallying call to both academic and industry researchers and developers to collaborate in pursuit of this mission. To realize an effective DSA, the paper delineates the essential components and functionalities requisite for infrastructure control protocols and other slow-path operations. These common primitives serve as the foundational elements for the design of a programmable target specialized for the slow path [2]. The authors posit that an ideal DSA for the slow path should exhibit three fundamental properties:

1. **Predictable response times:** Given the prevailing trajectory towards deterministic response times in feed-forward networking architectures, the DSA must ensure bounded tail latency. This predictability is paramount in guaranteeing consistent and reliable performance under varying workload conditions.

2. **Fast table updates:** A crucial function of the slow path entails rapid, high-bandwidth table processing for flow caching. As such, the DSA should facilitate swift and efficient updates to accommodate the dynamic nature of network traffic patterns effectively.

3. **Large memory pools:** Given the inherent limitations of on-chip memory within data plane architectures, the DSA must support extensive memory pools, leveraging High-Bandwidth Memorys (HBMs) to augment available memory resources. This capability is indispensable for accommodating the burgeoning demands for memory-intensive operations within the slow path.

By adhering to these guiding principles and leveraging the capabilities of a purpose-built DSA, Zulfiqar *et al.* argue that the networking community can effectively mitigate the performance bottlenecks plaguing the slow path [2].

2.7.2 Elixir

The performance of SDN devices and the associated CPU resource utilization significantly depends on how the flow table is partitioned between hardware

and software. Previous research has primarily focused on utilizing traffic skewness to split the flow table. However, the presence of bursty flows, which are intermittent and irregular data transmissions, poses additional challenges to this process. It becomes crucial to identify the appropriate flows and timing for offloading flows between hardware and software to optimize overall performance while minimizing overhead. In their paper titled "*Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness*" Yanshu Wang *et al.* introduce Elixir, a solution for managing hybrid flow tables on commodity SDN devices [52].

2.7.2.1 Motivation

The rapid growth in the size of flow tables in modern network devices presents a challenge. As briefly touched on in Section 2.2, while hardware offers fast forwarding speeds, its on-chip memory capacity cannot accommodate all concurrent flows. Conversely, software possesses ample memory but limited forwarding capacity. To address this, many commodity network devices, including OVS, adopt a hybrid approach utilizing both hardware and software to manage large flow tables [1]. The critical task in this hybrid approach is determining the optimal split between hardware and software for the flow table. This split not only impacts forwarding performance but also dictates the CPU resources allocated for software forwarding. While previous methods have leveraged traffic skewness for flow splitting, such as offloading the top 10% largest flows to hardware to save approximately 90% of CPU resources, the prevalence of bursty flows introduces additional challenges [52]. In the realm of flow-table splitting, two critical windows are utilized:

1. **Flow Rate Identification Window:** This window serves as the timeframe for measuring flow throughput. Flows generating the highest traffic within this window are identified as large flows [52].

2. **Flow Replacement Window:** This window represents the time interval between consecutive decisions regarding flow replacement in hardware or software. Excessively frequent flow replacement can lead to performance degradation in both hardware and software components [52].

In scenarios involving high-speed packet forwarding, frequent flow table replacement within a small time window results in increased exchanges

of flows between hardware and software, leading to forwarding speed degradation in both components. Hardware performance is impeded by the constraints of the locking mechanisms required to prevent race conditions, thereby limiting sustained high-speed traffic forwarding when using a small replacement window. Conversely, employing a large replacement window to mitigate performance degradation leads to the potential retention of bursty flows in software. Accommodating peak rates of bursty flows requires excessive CPU resources, significantly surpassing average rates and resulting in resource wastage. Balancing these factors necessitates determining the optimal flow replacement window for hardware and software [52]. To tackle these challenges, the paper by Yanshu Wang *et al.* aims to address the following questions:

- How to accurately measure all the flow rates with low overhead on commodity devices?
- How to set the proper timing for flow replacement between hardware and software?

2.7.2.2 Design

To solve the challenges related to splitting the flow table, Elixir utilizes the following three techniques [52]:

- **Hybrid Measurement Integration:** Elixir adopts distinct methods for measuring the rates of bursty flows and large flows. Large flows, typically comprising numerous packets, can be accurately characterized with a small sample of traffic. Elixir utilizes low-rate sampling from hardware to software for identifying large flows. Recognizing the limited number of concurrent bursty flows, Elixir assigns a hardware counter to each bursty flow, with software polling these counters for rate measurement. Elixir also segregates the hardware flow table into two distinct areas: one for large flows and another for bursty flows. This separation allows for tailored measurement techniques for each flow type [52].
- **Flow Type Differentiation:** Elixir implements a distinct replacement strategy for large and bursty flows. For large flows, characterized by stable rate-changing patterns, Elixir conducts periodic exchanges between hardware and software. This process involves using a relatively large replacement window to offload large flows with the

highest average rates, mitigating throughput degradation resulting from excessive replacement frequency. Conversely, bursty flows, appearing irregularly in the system, prompt an event-driven offloading process in Elixir. Bursty flows are immediately offloaded to hardware upon detection. Elixir utilizes the significant increase in the size of the software queue when a bursty flow occurs as a trigger for determining when to initiate the offloading process. Through this approach, Elixir strikes a balance between burst-aware offloading and minimizing forwarding performance degradation due to frequent replacement [52].

- **Decoupled Window Management:** Elixir implements a decoupling of the flow rate identification window and the flow replacement window. This decoupling prevents the sacrifice of flow rate identification accuracy or lagged flow replacement that may occur when using a single size for both windows. In the case of large flows, Elixir sets the replacement window to the minimum interval for replacement decisions, minimizing the impact on forwarding performance. Simultaneously, the identification window is set to the minimum size necessary for accurate flow rate identification, as sampled traffic may inaccurately rank large flow rates. For bursty flows, Elixir adopts a small identification window to capture burstiness effectively, facilitating immediate offloading upon detection. This decoupling allows Elixir to strike a balance between timely flow replacement and accurate flow identification [52].

Elixir's integration of hybrid measurement techniques, flow type differentiation, and decoupled window management collectively contribute to the realization of a cost-effective approach to hybrid flow table management [52].

2.7.2.3 Evaluation

Through these techniques, Elixir delivers significant savings in CPU resources necessary for managing hybrid flow tables. Elixir achieves up to 50% savings in software CPU resources while maintaining tail forwarding latency at approximately 97.6% lower compared to alternative approaches [52].

2.7.3 Open vSwitch Computational Cache

Raschelbach *et al.* identify two primary scalability challenges within OVS. First, the performance of the megaflow cache significantly degrades as its size increases. For instance, OVS shows a substantial slowdown, roughly an

order of magnitude when managing 500,000 flows compared to scenarios with only 1,000 flows. Second, performance issues arise when an SDN controller inserts new rules into OVS. This decline occurs because maintaining the non-overlapping property of flows may necessitate evicting existing flows from the megaflow cache. Such evictions result in an increased number of slow path upcalls, which is especially problematic in environments with frequent rule updates [3].

In their seminal paper, "*Scaling Open vSwitch with a Computational Cache*", Raschelbach *et al.* propose a method to enhance the scalability of OVS to support a larger number of OpenFlow rules. The core of their approach is to accelerate OVS's packet classification mechanism, briefly introduced in Section 2.2.5. To accomplish this, the authors employ NuevoMatch (NM), an algorithm that utilizes neural network inference for packet matching. This approach shows potential for substantial scalability and performance improvements over traditional algorithms [3, 1]. Leveraging NM, they explore two distinct design options for integrating it with OVS:

1. Supplementing OVS's existing megaflow cache with NM as an additional caching layer.

2. Alternatively, replacing OVS's data path entirely with NM, enabling packet classification to be conducted directly on OpenFlow rules and precluding control path upcalls.

2.7.3.1 NuevoMatch

The NM algorithm distinguishes itself by utilizing shallow neural networks, specifically incorporating a Range Query Recursive Model Index (RQ-RMI) model to understand the distribution of rules. This approach has shown superior performance compared to traditional methods when handling a large number of OpenFlow rules. Figure 2.9 illustrates how a rule lookup in OVS is transformed using neural network inference, replacing the traditional traversal of index data structures. When rule updates occur, they are initially integrated into a slow path remainder classifier, with periodic model retraining facilitating their integration into the fast path. As a result, the RQ-RMI model acts as a computational cache for these remainder rules, with model retraining effectively populating the cache. NM scales efficiently due to its low memory and optimized CPU usage. It further enhances performance through a hybrid training approach that combines Python, TensorFlow, and a custom library [3].

In the NM framework, rule sets are managed by partitioning them into distinct subsets known as iSets, where each iSet contains rules with non-overlapping header fields. The coverage of an iSet refers to the proportion of the total rule set it represents. Rules that cannot fit within iSets are handled by a remainder classifier, which can utilize alternative packet classification methods. Each iSet is trained using the RQ-RMI hierarchical model, which consists of multiple shallow neural networks. This model learns the rule ranges and predicts the index of the matching rule within an array. During inference, the estimated index is used to guide the search for the matching rule. The RQ-RMI training algorithm tightly controls the maximum error of these index estimates, ensuring fast and accurate lookups. During the search process, all packet fields are checked to validate candidate rules, and the highest priority rule is selected from those matching across iSets and the remainder [3].

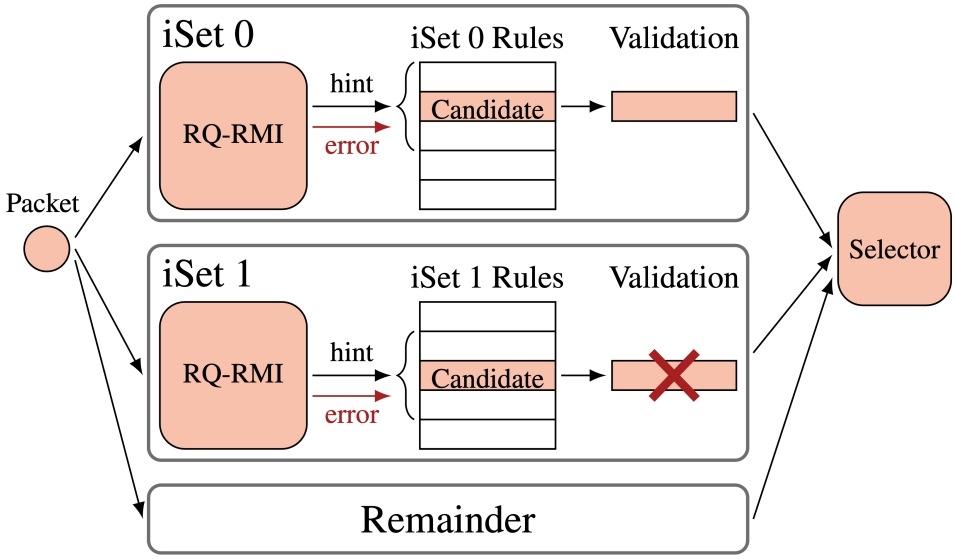


Figure 2.9: NuevoMatch algorithm utilizing RQ-RMI inference for packet classification [3].

2.7.3.2 NuevoMatchUP

The training time for RQ-RMI presents a significant challenge, especially concerning the required update rate in OVS when accommodating a substantial number of OpenFlow rules. Addressing this concern, the paper introduces NuevoMatchUP (NMU), an extension of the original NM algorithm,

dramatically enhancing the training rate by more than three orders of magnitude. Consequently, NMU achieves training times of mere milliseconds for tens of thousands of rules thereby making it possible to integrate NMU with OVS. In contrast to NM, NMU is implemented in C++, reducing its memory footprint and leveraging Single Instruction Multiple Data (SIMD) instructions to achieve enhanced performance. NMU also introduces crucial modifications to the RQ-RMI construction and training algorithms. Firstly, it allows for the creation of smaller RQ-RMI models by constructing iSets with overlapping rules, thereby expediting training. Secondly, it significantly improves training speed by minimizing the number of memory accesses. The paper presents two design options for integrating NMU with OVS. The first, termed OVS with Computational Cache (OVS-CCACHE), extends OVS's existing megaflow cache with NMU as an additional caching layer. The second design, OVS with Computational Flows (OVS-CFLOWS), replaces OVS's control path entirely with NMU, enabling packet classification to be conducted directly on OpenFlow rules in the data plane and thereby avoiding slow path upcalls [3].

2.7.3.3 OVS-CCACHE

Figure 2.10 shows the OVS-CCACHE setup where the control path remains unchanged. Incoming packets initially undergo matching against the microflow cache. Any misses are then directed to NM's computational cache, utilizing RQ-RMI models. The original megaflow cache handles lookups that don't match in RQ-RMI. If a packet remains unmatched, it proceeds with the standard OVS slow path upcall mechanism residing in user space. Upon addition of new flows into the OVS datapath, they are first inserted into the original megaflow cache. Subsequently, the RQ-RMI model undergoes periodic retraining in a separate thread, incorporating the new flows present in the megaflow cache. After training, the updated RQ-RMI models replace the previous models, and the megaflow cache is cleared. However, this solution still inherits the performance limitations of the slow path upcall mechanism [3].

2.7.3.4 OVS-CFLOWS

To address the performance constraints associated with frequent upcalls in the OVS-CCACHE solution. The second approach OVS-CFLOWS, which is shown in Figure 2.11, capitalizes on NMU to conduct efficient packet classification directly on OpenFlow rules, bypassing the necessity for non-

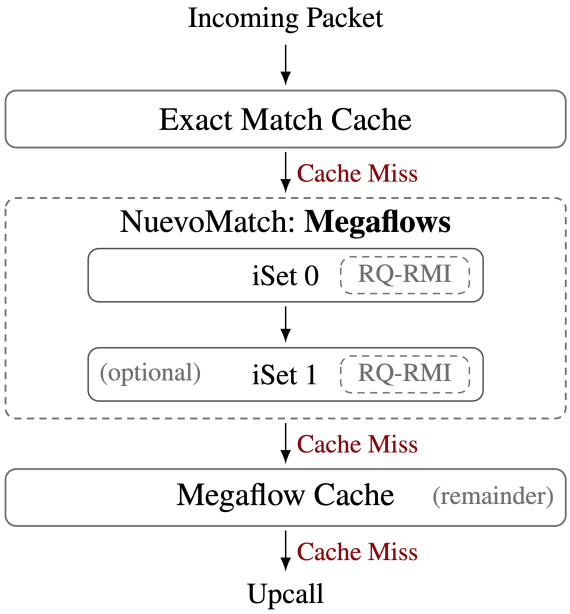


Figure 2.10: OVS-CCACHE with NM accelerating the megaflow cache [3].

overlapping megaflows. OVS-CFLOWS eliminates the reliance on the megaflow cache and the segregation of fast-slow paths characteristic of the original OVS design. This alteration eradicates the primary bottleneck limiting rule update rates in conventional OVS setups. However, adopting this option necessitates significant modifications to the existing OVS architecture [3].

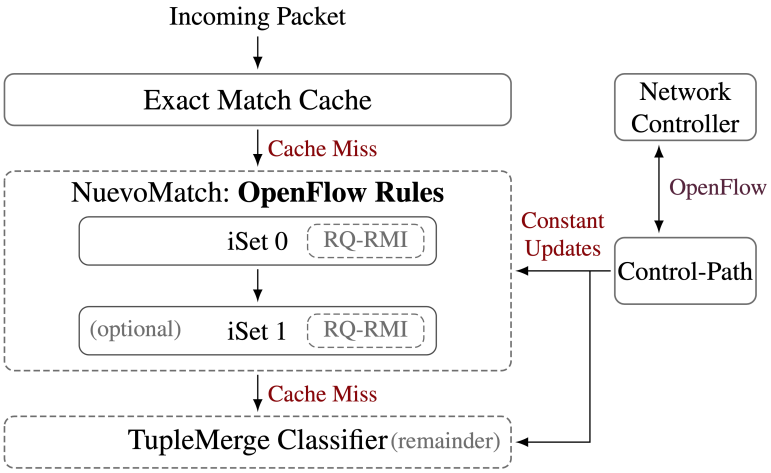


Figure 2.11: OVS-CFLOWS with NMU performing OpenFlow rule classification in the datapath avoiding slow path upcalls [3].

2.7.3.5 Evaluation

The paper by Rashelbach *et al.* conducted extensive evaluations of both the OVS-CCACHE and OVS-CFLOWS approaches, comparing them to the original OVS design. Results indicate a geometric mean speedup with OVS-CCACHE of $1.02\times$, $1.5\times$, and $1.9\times$ for 1,000, 100,000, and 500,000 OpenFlow rules, respectively, compared to the original OVS implementation. OVS-CFLOWS demonstrates even higher speedups, with improvements of $2.6\times$, $8.5\times$, and $12.3\times$ over the original OVS design in the same setups [3].

Chapter 3

Methodology

In this chapter, we explain the different parts of the theoretical method used to benchmark OVS and understand its upper-limit performance when having perfect knowledge of future flows. Specifically, we begin with Section 3.1 which presents the metrics used to evaluate the performance of OVS. Section 3.2 continues by describing the benchmarking environment, including the network configuration and OVS configuration. Section 3.3 presents the details of the workload used in the benchmarks. Finally, Section 3.4 explains the structure of our benchmarking experiments along with how the data gathered is processed.

3.1 Benchmarking metrics

To analyze the performance of OVS, we utilize several key metrics. First, we gauge OVS performance by measuring per-packet latency, spanning from ingress into OVS to egress from the virtual switch. This specific measure excludes any processing delays associated with the applications that transmit and receive packets. By focusing solely on the ingress to egress journey within OVS, we ensure that our measurements accurately reflect the switch's performance in handling packet traffic, aligning directly with the objectives of our research. Through this measure, we can quantitatively evaluate variations in OVS latency under different conditions: no flow prediction, partial flow prediction, and complete flow prediction.

Furthermore, we assess the CPU utilization required for OVS operation in various configurations. This utilization is expressed as a percentage and reflects different types of resource demands including kernel CPU load (system processes), software interrupts, and total CPU load. Special attention

is given to the resources consumed by OVS handler threads, which are pivotal during flow table lookups following a cache miss. Lastly, we examine the frequency of upcalls to `ovs-vswitchd` per second in different scenarios. By analyzing these metrics, we aim to precisely gauge the influence of flow prediction accuracy on OVS performance across diverse operational conditions.

3.2 Benchmarking environment

The selected benchmarking environment, depicted in Figure 3.1, comprises two interconnected machines utilizing Mellanox ConnectX-6 MT28908 100Gb Ethernet NICs. The System-Under-Test (SUT) machine operates on CentOS Stream 9, Linux kernel version 5.14.0, and utilizes OVS version 3.3 and OVN version 23.06.2 within a Kernel-based virtual machine (KVM) environment utilizing VT-x virtualization technology. The VM is hosted on an Intel Xeon Gold 6346 CPU clocked at 3.10GHz, featuring 16 cores with 1 thread per core, along with 512KiB L1 cache, 64 MiB L2 cache, and 16 MiB Last Level Cache (LLC). Additionally, the system is equipped with Non-uniform memory access (NUMA) architecture, with a single NUMA node comprising all 16 cores. The VM is also configured with 96 GB of Random-Access Memory (RAM).

The traffic generator is equipped with Ubuntu 20.04.4 as the operating system, running on Linux kernel version 5.4.0. It directly utilizes FastClick [53] with Data Plane Development Kit (DPDK) version 19.11.14 on an 8-core Intel Xeon Gold 5217 CPU clocked at 3.00GHz. The CPU features 8 cores with 2 threads per core, complemented by a 256KiB L1 cache, 8 MiB L2 cache, and 11 MiB LLC. Finally, the traffic generator server boasts a robust 49 GB of RAM.

3.2.1 System-Under-Test machine

To replicate benchmarking on a production-like OCP cluster without deploying a large-scale cluster comprising numerous physical nodes and pods, we opt to leverage OVN for setting up a cluster for benchmarking OVS. Given that OCP employs OVN-K as its default CNI, our approach to configuring a cluster using OVN closely simulates the communication dynamics between OVS and the cluster, particularly in terms of upcalls and installed OpenFlow rules. Following is a description of the OVN cluster network setup used to simulate a single-node OCP network.

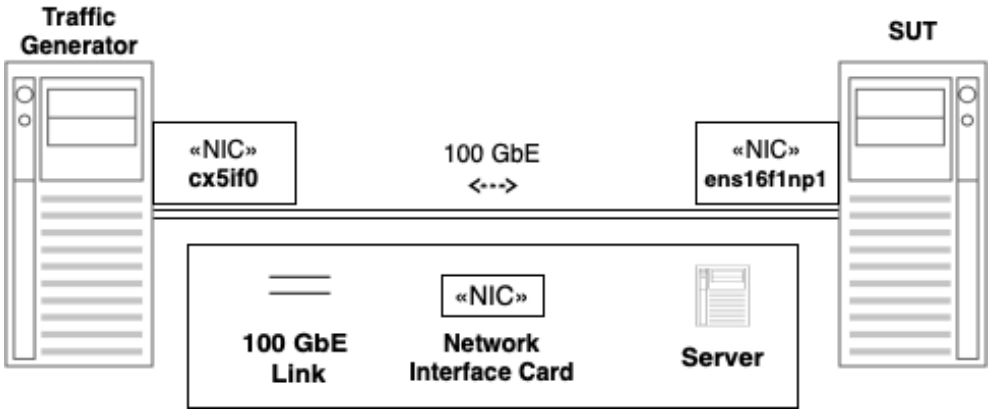


Figure 3.1: The benchmarking environment shows the two servers, including the traffic generator and the System-Under-Test, interconnected by a 100 GbE link.

3.2.1.1 OpenShift network simulation

We deploy an environment on SUT that replicates a single-node OCP network configuration as shown in Figure 3.2. The aim is to enable access to an external IP address via the node IP, mimicking the process of exposing a service externally using the `oc expose service <service>` command. In an actual cluster, the kernel manages the Network Address Translation (NAT) translation from an external virtual IP to the node IP. However, in this particular scenario, we bypass this translation mechanism, thereby permitting direct access to the external service IP within the `3.0.0.0/8` range. The network architecture is outlined below:

The script defaults to generating 8 PODs, each assigned a distinct IP address ranging from `1.0.0.1` to `1.0.0.8`. Notably, the last digit of the IP address correlates with the POD number. Every POD resides within its dedicated Linux namespace, denoted as pod$X$. Furthermore, we establish a service IP to emulate POD-to-POD interaction on the `2.0.0.0/24` network. Load balancers are configured for nine User Datagram Protocol (UDP) ports, with each POD allocated a unique service IP in the format of `2.0.0.<pod_id>`. For example, for POD5, the service IP is `2.0.0.5`. We replicate the load balancer setup to simulate external access to the OCP network. While each POD is assigned two IPs in the format of `3.0.0.<pod_id>` and `3.0.1.<pod_id>`, both IPs balance the nine UDP ports ranging from 2100 to 2180. The complete set of deployment scripts and accompanying resources utilized in our methodology are hosted in a publicly accessible GitHub repository [54].

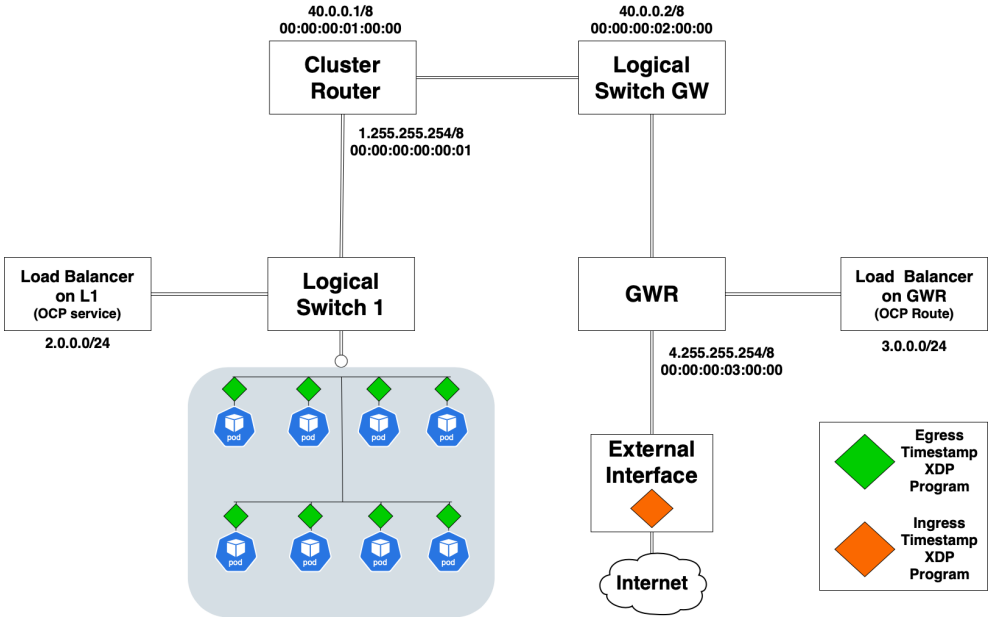


Figure 3.2: OVN network cluster of a single-node OCP: Ingress timestamp (orange) and egress timestamp (green).

3.2.2 Traffic Generator

The traffic generator transmits traffic towards the external interface on the SUT, which hosts the OCP network and is undergoing benchmarking. The traffic generator replays a preconfigured coflow traffic trace retrieved from a Packet Capture (PCAP) file, as detailed in Section 3.3. The traffic generator can replay this trace at a speed commensurate with its NIC performance of 100 Gigabits per second (Gbps). For network packets to reach the pods in the OCP cluster, they need to have a physical address of `00:00:00:03:00:00` and a destination IP address of either `3.0.0.0/8` or `3.0.1.0/8` [54].

3.3 Coflow workload

To conduct the intended benchmarks of OVS, it is imperative to utilize a workload that mirrors the traffic patterns observed in real-world scenarios, particularly in OCP clusters or similar cluster computing environments employing OVS and containing a degree of coflows. Unfortunately, the absence of realistic trace files and the scarcity of coflows in conventional Internet backbone traces like Cooperative Association for Internet Data Analysis (CAIDA), pose a challenge to our experiments, which heavily depend

on the concept of coflows. Consequently, we explored alternative approaches to procure a suitable workload for use in our benchmarks of OVS. Our investigation led us to adopt the workload generator devised by Agarwal *et al.*, as featured in their work on the development of the Sincronia framework and covered in Section 2.5.3 of this work [44, 45, 46]. This generator leverages an upsampling technique on a 526-coflow trace derived from a one-hour execution of MapReduce tasks on a 3000-machine Facebook cluster. Although this trace has limitations regarding coflow count and network load, the generator offers the flexibility to adjust these parameters while maintaining the essential characteristics of the original Facebook trace. For our benchmarking experiments with OVS, we rely on the foundation provided by Agarwal *et al.'s* workload generator, enhancing it with additional properties and functionalities to suit our specific requirements [44, 45, 46]. The following sections describe how our enhanced workload generator operates.

3.3.1 Extended workload generator

To manage the coflows generated by the original workload generator, we undertake the task of parsing the output file, as detailed in Section 2.5.3.1, into a JavaScript Object Notation (JSON) format. Below, we introduce a series of changes and properties that we integrate into the original workload generator. These properties effectively modify the fundamental characteristics and arrangement of the resulting coflow traffic trace [54].

3.3.1.1 Destination IP addresses

The original Sincronia workload generator has a limitation: it does not support a different number of source and destination addresses. This constraint stems from its reliance on the Facebook MapReduce trace. To address this limitation for our benchmarks, where we require more source IPs than destination IP addresses, we've made updates to the workload generator. When creating the coflow trace, we employ modulus reduction to map the larger set of destination IDs to our smaller set of pods deployed in the OVN cluster [54]. The resulting value is referred to as the "pod index". This calculation is expressed mathematically as follows:

$$(\text{dst}_{id} \mod \text{pods}) + 1$$

Where:

- dst_{id} represents the destination ID provided by the Sincronia coflow workload generator.
- $pods$ denotes the number of deployed pods in the OVN cluster.

With the newly derived pod indices, we proceed to construct the destination IP addresses. Our objective is to evenly distribute traffic across the pods, directing 50% of the traffic to the `3.0.0.0/8` subnet and the remaining 50% to the `3.0.1.0/8` subnet. This distribution is achieved by randomly selecting a "host id" from the set $\{0, 1\}$, where $P(0) = P(1) = \frac{1}{2}$. This ensures that approximately half of the destination IPs are routed to `3.0.0.0/8` and the other half to `3.0.1.0/8`. Consequently, all destination IPs in the trace follow the format:

$$3.0.(h, p)/8$$

Where:

- h represents the host ID, which can be either 0 or 1.
- p denotes the pod index, corresponding to one of the eight pods in the OVN cluster.

3.3.1.2 Coflowiness

We introduce a parameter termed *coflowiness*. Coflowiness denotes the extent of inter-dependency among multiple data flows within a network coflow, specifically characterized by the presence of a shared destination or source. A higher level of coflowiness is evidenced when these individual flows share a common endpoint or origin, whereas a lower degree of coflowiness indicates fewer shared attributes among flows within a coflow. Our enhanced workload generator allows for the adjustment of coflowiness via a Command Line Argument (CLI) argument during its execution. Let coflowiness(x) denote the coflowiness value of coflow C. If coflowiness$(x) = 0$, it signifies that all flows within coflow C are distinct, implying that the destination and source attributes of each flow are unique within the coflow. Conversely, if coflowiness$(x) = 1$, it indicates that all flows within coflow C share either a destination or a source with at least one other flow within the same coflow. Additionally, if $0 <$ coflowiness$(x) < 1$, it implies that coflowiness$(x) \times 100\%$ of all flows in coflow C match at least one other flow in coflow C.

Since our benchmarking and broader work rely on the concept of coflows, we need to define which flows in our trace should be interpreted as base flows and which flows should be seen as associated flows. In networking, base flows in coflows represent primary data transfers, while associated flows are supplementary data transfers related to the base flows. Although the original Sincronia workload generator outputs a coflow that certainly has base and associated flows, there's no easy way to discern which flows are base flows and which are associated flows. Therefore, we choose another approach that relies on the fact that we manually define what flows should be interpreted as base flows and what flows should be associated flows. In our trace and benchmarking, a base flow is defined as a flow that has a destination port of 2100. This destination port of 2100 is only used for base flows [54].

3.3.1.3 Network ports

Given that the original workload generator lacks source or destination network ports, we have augmented our extended version of the generator to include these. The port inclusion operates as follows: Our workload generator differentiates between source and destination ports. Each destination IP present in the trace is assigned 9 destination ports as described by the set

$$\{x \mid x = 2100 + 10n, \text{ where } n = 0, 1, 2, 3, \ldots, 8\}$$

For each unique destination IP within a coflow, a destination port is randomly selected from the set. The assigning of source ports to flows does not matter other than that by altering the total number of unique source ports, S, we can control the number of unique 5-tuple flows present in the trace without having to change the OVN cluster configuration. Moreover, for every coflow in the trace, we ensure that all flows sharing either a destination or source IP utilize the same destination port. Formally, this can be expressed as:

$$\forall f \in F_c, \text{IP}_f = \text{IP}_a \text{ and } \text{Port}_f = \text{Port}_b$$

where

- F_c represents the coflow,
- f represents an individual flow within F_c,
- IP_f represents the IP address of flow f,
- $Port_f$ represents the port of flow f,

- IP_a represents the common IP address,
- $Port_b$ represents the common port.

This approach of assigning source and destination ports simulates the presence of S open ports for each source IP in the trace, where processes exchange traffic with processes running at any of the nine ports open at the pods deployed in the OCP cluster. This is crucial to emulate the behavior where flows within a coflow originate or are destined for the same network entity. Across different coflows, the selected port for a given IP may vary from the set of selected ports allocated to that IP [54].

3.3.1.4 Flow size distribution

To more accurately control the Flow Size Distribution (FSD) of our trace of coflows, we enhanced the original workload generator to generate flow sizes that follow an arbitrary Cumulative Distribution Function (CDF). The generator reads and parses the designated CDF file for utilization. To produce a distribution comprising n values adhering to the provided CDF, we use the inverse transform sampling method. This method relies on the fact that if F is the CDF of a continuous random variable with support on $[a, b]$, then $F(X)$, where X is a uniform random variable on $[0, 1]$, has the same distribution as X. Following is a detailed breakdown of each step involved in the inverse transform sampling method, which is employed to generate the desired FSD:

- **Interpolation** Given a CDF $F(x)$, we aim to find its inverse function $F^{-1}(y)$, where y is a uniformly distributed random variable on $[0, 1]$. However, in many cases, $F^{-1}(y)$ may not have a simple, closed-form inverse function. To address this, we use interpolation to approximate $F^{-1}(y)$. By constructing a piecewise interpolation between the CDF points, we can estimate the inverse function at any given value of y. This interpolation approach allows us to approximate the inverse transform needed for generating samples from the desired distribution. Specifically, we use cubic polynomials between each pair of adjacent points to create a piecewise cubic spline. Cubic spline interpolation is preferred to linear interpolation for its smoother curve. It ensures continuity in function values, and first and second derivatives, reducing oscillations and providing a more accurate representation of data, especially when curvature is significant.

- **Quantile Function**: Once we have the interpolated CDF $F(X)$, we can compute its inverse $F^{-1}(y)$, also known as the quantile function, by inverting the relationship between x and y. This involves solving for x in the equation $F(X) = y$. Since the interpolated CDF is piecewise linear, we can efficiently find the corresponding x for any given y within the range of probabilities.

- **Inverse Transform Sampling**: Once we have the quantile function $F^{-1}(y)$, we utilize the concept of inverse transform sampling to generate n uniformly distributed random numbers y_i from the interval $[0, 1]$. This method leverages the uniformity of the y_i values to ensure an even sampling across the entire range of the CDF. Mathematically, let y_i denote the ith uniformly distributed random number, where $i = 1, 2, \ldots, n$. These y_i values are independent and identically distributed random variables with a uniform distribution on the interval $[0, 1]$, i.e., $y_i \sim U(0, 1)$. Inverse transform sampling involves applying the inverse function $F^{-1}(y)$ to each y_i to obtain the corresponding random variable x_i from the desired distribution. Formally, we have:

$$x_i = F^{-1}(y_i)$$

  where x_i represents the ith sample drawn from the desired distribution. By sampling from a uniform distribution and transforming these samples through the inverse CDF, we ensure that the generated x_i values follow the specified distribution described by the original CDF.

- **Expected Value Analysis**: Increasing the number of generated values provides a better representation of the CDF because it results in a denser sampling of the probability space. Mathematically, as the sample size n increases, the Empirical Cumulative Distribution Function (ECDF) constructed from the generated values converges to the true CDF. To quantitatively assess how close the generated distribution is to the CDF, we can compute the expected value $E_{\text{sample}}(\bar{x})$ of the observed values and compare it with the theoretical expected value $E_{\text{theoretical}}(X)$ of the distribution described by the CDF. If the observed values closely match the desired distribution, their average, $E_{\text{sample}}(\bar{x})$, should approximate the theoretical expected value $E_{\text{theoretical}}(X)$ calculated from the CDF. Formally, the expected value of the observed values can be computed as:

$$E_{\text{sample}}(\bar{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

where x_i represents the ith generated value. Similarly, the theoretical expected value $E_{\text{theoretical}}(X)$ of the random variable X can be computed as:

$$E_{\text{theoretical}}(X) = \int_{-\infty}^{\infty} x \cdot \frac{dF(x)}{dx}\, dx$$

where $\frac{dF(x)}{dx}$ represents the Probability Density Function (PDF) derived from the CDF of X.

Comparing the calculated sample mean, $E_{\text{sample}}(\bar{x})$, with the theoretical expected value, $E_{\text{theoretical}}(X)$, obtained from the CDF allows us to gauge the fidelity of the generated distribution. A close match between these values indicates that the generated distribution closely aligns with the distribution described by the CDF.

To derive the actual cumulative FSD utilized in our trace, we undergo a systematic process. First, we traverse through all flows within the JSON trace file. For each flow encountered, we generate a random number, denoted as y, from the interval $[0, 1]$. This random value is then applied to the cubic spline interpolation function provided by the SciPy library, yielding a value aligned with the given CDF; this value is referred to as the *byte size*. Subsequently, this byte size is added to the `flow_size_bytes` field for each corresponding flow in the JSON file. When creating the actual trace file comprising real IP packets, this `flow_size_bytes` field serves as a crucial reference point for generating the packet payload [54].

3.3.1.5 Generating trace file

Having formulated the JSON trace fields with additional attributes like coflowiness, network ports, and FSD, the next step involves constructing an actual synthetic network trace file comprising real network packets. To accomplish this, we employ the Scapy Python library, renowned for its capabilities in crafting and manipulating network packets at a granular level, enabling custom packet creation, modification, and analysis. Scapy offers a versatile interface, affording precise control over packet headers and payloads [55, 56, 57]. In the generation process, we opt for stateless

UDP packets encapsulated within the PCAP file format. To achieve this, we iterate through all coflows and flows in the JSON file, extracting essential fields such as `source_IP`, `destination_IP`, `source_port`, `destination_port`, and `flow_size_bytes`. With this information, we can construct individual UDP packets with Scapy. Each packet adheres to a maximum payload size of 1458 bytes, resulting in a UDP packet size of 1500 bytes when considering a 14-byte Ethernet header, a 20-byte IP header, and an 8-byte UDP header. The payload itself is a dummy representation of zeros in byte format, as it serves solely to construct the desired FSD and is not utilized for any other function. If the flow size exceeds the maximum payload capacity, the flow is logically partitioned into multiple UDP packets until the desired flow size is attained. Subsequently, upon generating the packet header and payload, the packet is persistently written to the PCAP file on disk. In the generation process, each coflow's constituent flows are sequentially arranged within the resulting PCAP file, ensuring that the order of coflows and their flows is preserved. This approach maintains the coherence of the network trace, as flows belonging to the same coflow are consecutively organized without intermingling with other coflows. Consequently, the resulting PCAP file mirrors the structure of coflows as generated by the original workload generator devised by Agarwal *et al.*, augmented with our additional attributes such as coflowiness, network ports, and FSD [54].

3.3.2 Replay coflow trace

To replay the coflow traffic trace generated by our extended workload generator, detailed in Section 3.3, we leverage FastClick, an enhanced iteration of the Click modular router that incorporates advanced DPDK support. FastClick is adept at achieving high throughput and minimal latency by directly accessing NICs and bypassing the traditional networking stack for specific operations. This approach optimizes packet handling and reduces processing overhead [53]. To ensure that all the packets of the trace arrive in the same order each time we do a benchmark, we limit FastClick to utilizing a single CPU core. However, this limitation results in a maximum transmit rate of ~2 million Packets Per Second (pps) resulting in a throughput of ~13 Gbps.

3.4 Benchmarking methodology

This section presents the benchmarking methodology used to benchmark the performance of OVS as deployed in our OCP network cluster shown in Figure 3.2.

3.4.1 Measuring Per-Packet Latency

We employ a systematic methodology to measure the per-packet latency of network traffic traversing through OVS. This section outlines the steps taken to gauge per-packet latency and compile relevant statistics.

Packet Number Insertion

During the replay of the trace on the traffic generator, FastClick inserts a 64-bit packet number into the UDP packet payload. This packet number serves as a unique identifier for each packet and facilitates tracking its journey through the network, including the cluster and OVS [54].

Timestamp Insertion with XDP

We developed an XDP program to insert timestamps into the UDP packet payload. These timestamps are inserted at two critical points in the OCP cluster shown in Figure 3.2:

- **Ingress Timestamp:** Inserted at the ingress point of the physical port within the cluster deployed on the SUT machine. This point, highlighted in orange in Figure 3.2, marks the initial packet entry into our network infrastructure, capturing the arrival time at the very beginning of the datapath.
- **Egress Timestamp:** Inserted just as packets are about to leave OVS, through the Virtual Ethernet Device (veth) pair leading to individual pods. This egress point, shown in green in Figure 3.2, captures the time immediately before the packet exits the switch, providing a precise measure of the packet's traversal through the switch.

These timestamps are obtained using the `bpf_ktime_get_ns()` function in XDP, which returns an unsigned 64-bit value representing the number of nanoseconds since system boot. By capturing these timestamps, we gain insights into the exact time at which packets traverse specific network interfaces [54].

Packet Payload Structure

Figure 3.3 shows how the UDP packet payload is structured to accommodate the three essential properties, each occupying 64 bits of space:

1. **Packet Number:** Located at the beginning of the packet payload, starting at byte 28 when counting from the IP header.
2. **Ingress Timestamp:** Immediately follows the packet number in the payload, occupying the next 8 bytes.
3. **Egress Timestamp:** The final 8 bytes in the payload, positioned after the ingress timestamp.

The arrangement of these values in the packet payload, totaling 24 bytes, facilitates efficient extraction and interpretation of per-packet latency data.

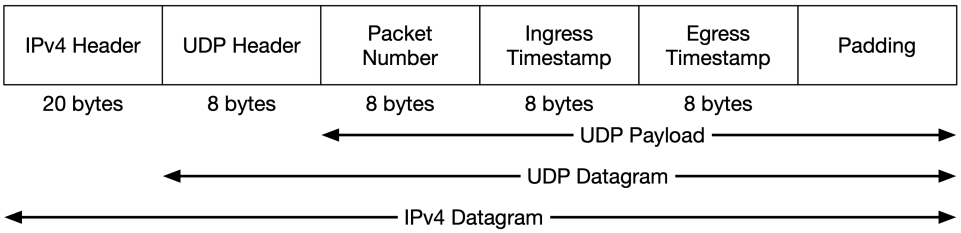


Figure 3.3: The UDP packet structure including packet number, ingress timestamp, and egress timestamp located in the UDP packet payload.

Receiving daemon

Upon packet reception, a dedicated receiving daemon operates within each cluster pod. This daemon is responsible for parsing each incoming UDP packet, extracting the ingress and egress timestamps along with the corresponding packet number, and calculating the latency for each packet by computing the delta between the egress and ingress timestamps. Furthermore, the daemon records the latency of the first packet of each distinct flow. With this information, we can analyze the impact of OVS upcalls' costs when future flow prediction is not utilized and compare it to scenarios where it is employed. The latency measurements and associated packet numbers parsed during the benchmark are stored in an array. Upon completion of the benchmark, the array's contents are written to disk in a structured JSON format, facilitating subsequent latency analysis and statistical processing. The source code for the pod daemon is publicly accessible on GitHub [54].

3.4.2 Benchmarking setup

To measure the upper limit of achievable performance when anticipating future flows accurately, the following methodology is employed: Utilizing the definition of a coflow, outlined in Section 2.5, where a user initiates an Hypertext Transfer Protocol (HTTP) request for a service page, considered as a base flow, this action generates one or more associated flows to entities such as DNS servers, databases, cloud hosting sites, or Content Delivery Networks (CDNs). It is imperative to note that the objective of this study is to quantify the upper limit of performance when accurately predicting future associated flows based on a set of base flows. In our simulation, we prioritize the processing of the first UDP packet of a base flow before addressing any packets from associated flows. This sequencing mirrors real-world scenarios where there would be a processing delay as OVS executes the necessary predictive computations. These computations determine which associated flows to preload into the cache. In contrast, our benchmark setup includes no such delay, allowing the immediate initiation of the next base flow's transmission once the initial packet of the current base flow has been processed by OVS. Ideally, in a production environment, once OVS processes the initial packet of a base flow, it would proactively program the datapath rules to manage the associated flows based on anticipated traffic patterns. Bearing this in mind, we benchmark the following scenarios:

Baseline scenario, without flow prediction To establish a baseline for OVS performance in the absence of flow prediction, the methodology begins by clearing the datapath caches and setting the maximum flow capacity to 200,000. Following this configuration, the system undergoes benchmarking where the trace, including base and associated flows, is transmitted through OVS. Performance metrics are continuously collected and subsequently analyzed in detail.

Optimal scenario, with applied flow prediction: Again, at the start of the test, we reset OVS to clear all datapath flows, set the maximum number of flows to 200,000, and maximize the datapath flow timeout value. We then populate the OVS caches by sending the trace containing only the associated flows. This action programs the datapath cache with all the associated flows. Now, we initiate the benchmarking process by sending the full trace containing both base and associated flows and collecting statistics. Since the datapath cache is programmed with the associated flows, only the base flows will trigger upcalls, resulting in maximum performance gain.

Varying the percentage of associated flows: The third objective of the benchmarking setup is to vary the fraction of associated flows to the number of base flows, as well as other trace properties such as FSD, to investigate its impact on performance. This is further elaborated upon in Section 3.4.2.1.

3.4.2.1 Varying coflowiness

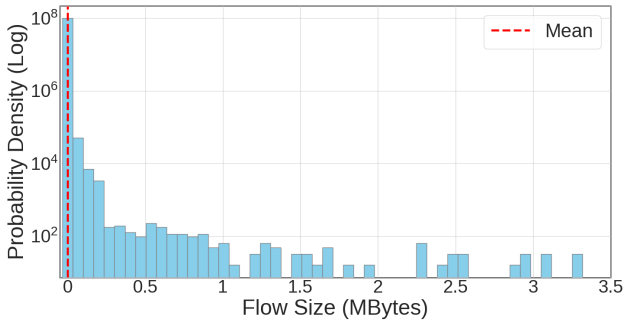
In our methodology, we define *coflowiness* as the percentage of associate flows that are predicted and prepopulated in the OVS cache. To explore the impact of different coflowiness levels on performance, we generate a series of network traces. A coflowiness level of 0.0 serves as our baseline scenario, where no flow predictions are made, and thus, no flows are preloaded into the cache. Conversely, a coflowiness level of 1.0 represents an ideal scenario with 100% accurate flow prediction, where every possible flow is preloaded into the cache. This setup aims to establish the absolute upper limit of performance achievable under optimal predictive conditions. For intermediate levels, coflowiness of 0.1 indicates that 10% of the associate flows are predicted and preloaded, while 0.9 coflowiness means 90% of associate flows are preloaded, with the remainder being base flows that are not preloaded. Additional levels examined include 0.25, 0.5, and 0.75, each representing an incremental increase in the percentage of associate flows predicted and preloaded, allowing for a systematic evaluation of how different levels of coflowiness affect OVS performance.

3.4.2.2 Coflow trace properties

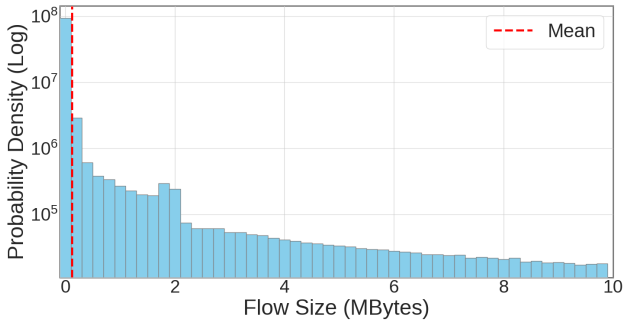
In our research, we assess the performance of OVS across various network traffic scenarios using FSDs derived from Montazeri *et al.*'s study on the Homa transport protocol [7]. The selected FSDs for our benchmarks are as follows:

1. **Google Search RPC (GSRPC):** This distribution, as shown in Figure 3.4a, has a mean flow size of 426 bytes, typical of Remote procedure call (RPC) scenarios, where flows are often bursty and short-lived. In such cases, the request or the response is generally small, as data often flows predominantly in one direction and the payload is minimal. The "Homa" paper highlights that existing data center transport designs struggle to achieve minimal latency for such small and transient messages under high network loads [7, 54]. This characteristic makes the GSRPC environment an essential FSD for evaluating the latency performance of OVS.

2. **Facebook Hadoop (FB Hadoop) Cluster:** Unlike the GSRPC, this distribution, shown in Figure 3.4b, has a significantly larger average flow size of 122,182 bytes. It represents scenarios involving substantial data exchanges, typical in large-scale data processing environments like those used by Facebook's Hadoop clusters [7, 54].



(a) Google Search RPC



(b) Facebook Hadoop

Figure 3.4: Histogram of the flow size distribution. Note the difference in the x-axis.

These contrasting flow sizes enable a comprehensive benchmark, covering different network scenarios. This approach helps understand how OVS performs under varied traffic conditions, from minimal to larger data flows. Moreover, the network coflow traffic trace employed in the benchmarks possesses several distinct characteristics:

The trace encompasses 193,000 unique 5-tuple flows, a figure strategically selected to approach the flow table limit of 200,000 flows set by OVS. This compilation of unique flows comprises 23,525 unique source IPs and 1,030 unique source ports. As delineated in Section 3.3, it also includes 16

unique destination IPs and 9 destination ports. Collectively, these properties aggregate to a total of 236,007 5-tuple flows across 1,000 individual coflows. The configuration remains consistent across all benchmarks, independent of the FSD applied. The only variations between benchmarks are the coflowiness value and the mean packet payload, which are adjusted according to the specified size distribution [54]. In Section 5.1.2.1, we provide a detailed analysis of the trace files. This analysis includes a discussion on the number of unique flows for each configuration of coflowiness. For a comprehensive overview, refer to Table 5.1, which illustrates the number of flows corresponding to each configuration.

3.4.2.3 Benchmark preparation and optimizations

To address the challenge of excessive CPU utilization, it is crucial to understand the underlying processes within OVN. When a new source MAC address is introduced into the network, OVN undertakes extensive bookkeeping activities. These activities include the revalidation of network configurations, which significantly increases CPU load. Given that our primary focus is to evaluate the performance of OVS rather than OVN, a strategic approach is required to mitigate this issue. The adopted solution involves pre-learning all MAC addresses within the OVN forwarding database before initiating performance benchmarks.

The pre-learning process is facilitated by generating Internet Control Message Protocol (ICMP) packets from the generator server, each having a different source MAC address that needs to be learned by OVN. These packets are directed towards the pods within the simulated OCP cluster. This configuration ensures that each ICMP packet's source MAC address is registered in the OVN forwarding database before the onset of benchmarking. By populating the database in advance with the source MAC addresses, the system minimizes the need for extensive revalidation, thereby optimizing CPU usage during the benchmark trials. Furthermore, to ensure a clean state for each benchmarking session, OVS is restarted between trials. This restart is strategically employed to clear any cached flows while preserving the MAC addresses within the OVN forwarding database. This approach guarantees that the integrity of the MAC address data is maintained, avoiding the need for revalidation and thereby optimizing CPU usage for subsequent benchmarks [54].

Additionally, to address the issue of packet drops at high transmission speeds of 2 million pps, we implemented several optimizations in the

network stack of the SUT. These adjustments were aimed at augmenting the packet handling capacity of the system, thereby improving overall network performance. Firstly, we expanded the ring buffer of the SUT's Ethernet device from 1,024 bytes to 8,192 bytes. The ring buffers, located at the base of the network stack, play a critical role in temporarily storing incoming packets until they can be processed by the device driver. By increasing the size of these buffers, we significantly reduced the likelihood of packet drops at the initial point of entry into the network stack. Secondly, we doubled the capacity of the backlog queues of the network device from 1,000 to 2,000 packets. This modification allows the kernel to store a larger number of packets temporarily before they are handled by the kernel's protocol stack, further mitigating the risk of packet loss. Additionally, we maximized the size of the system-wide UDP buffers to their highest configurable limits. This expansion provides more buffer space for the daemons running on each pod, thereby extending the time available for processing incoming data and preventing packet loss.

Following these optimizations, the OVS on the SUT achieved a stable processing rate of incoming packets at 1 million pps, corresponding to an incoming throughput of 12 Gbps, without experiencing any packet drops. Upon further debugging, we identified that the primary cause of packet drops was not the Ethernet device, the operating system, or the CPU capacity, but rather OVS itself. The bottleneck was found to be the socket buffer to user space, which is fixed in size within the OVS source code and cannot be increased. This limitation restricts OVS to handling only a certain number of new flows per second, which varies depending on the packet size. To ensure the avoidance of packet loss during our evaluations, we conducted our benchmark tests of OVS at a reduced rate of 1 million pps, achieving a throughput of 12 Gbps. This testing utilized the FSD from Facebook's Hadoop clusters.

3.4.2.4 Gathering CPU utilization and upcall statistics

To gather additional benchmark metrics as outlined in Section 3.1, specifically CPU utilization and upcall statistics, we utilize two scripts with the following tools:

1. **CPU Utilization**: We use `mpstat` to report on the activity of individual CPU cores. This tool provides detailed information on how each core is utilized during the benchmarking process.

2. **OVS Handler Thread Statistics**: For monitoring the tasks managed by the Linux kernel specific to OVS, we employ `pidstat`. This command

allows us to track the performance and activity of individual handler threads.

3. **Upcall Statistics**: We extract upcall statistics using `ovs-appctl dpctl/show -s`. This command retrieves detailed upcall data directly from OVS.

All data collected is then parsed and formatted for enhanced readability and is presented in the results section [54].

3.4.2.5 Statistical Analysis of Benchmark Runs

To establish a reliable benchmarking methodology, our traffic generator continuously transmits the generated coflow traffic trace towards the SUT until completion, defining this process as one benchmarking round. Given the critical importance of scientific accuracy, we conduct ten such rounds for each coflow configuration across two distinct FSDs. This approach results in seven unique benchmarks per FSD configuration.

Data collection is facilitated by the receiving daemon that runs on each cluster pod, as detailed in Section 3.4.1. We employ a Python script to parse JSON-formatted data files, enabling us to calculate the minimum, maximum, and average end-to-end packet latencies from each benchmarking round. Further, we analyze packet details, including sequence numbers and the delay between ingress and egress at OVS [54]. This analysis helps us derive minimum, maximum, and average statistics for three specific packet types:

1. The first base packet of each unique 5-tuple flow,
2. The first associate packet of each unique 5-tuple flow, and
3. Aggregate statistics based on all packets processed by FastClick and observed by OVS.

We visually represent this end-to-end packet latency data using box plots in Section 4.1. Each plot incorporates data from ten benchmarking rounds and is categorized by packet type. Three figures are produced for each category and FSD to depict the minimum, maximum, and mean latency values.

Chapter 4

Results and Analysis

This chapter presents the benchmarks' results on OVS. Section 4.1 shows the end-to-end packet latency obtained by benchmarking OVS with various FSDs. We then move to Section 4.2, which illustrates how CPU utilization varies when OVS handles different traffic traces under varying conditions. Lastly, Section 4.3 presents statistics on the number of upcalls per second performed during all of the benchmarks.

4.1 End-to-End Packet Latency

This section presents the end-to-end packet latency measured by a daemon on each pod within the OVN cluster deployed on the SUT machine. The latency measurements focus on different FSDs, as described in Section 3.4.2.2. Specifically, we report on three key latency metrics: the latency of the first packet of each base flow, the first packet of each unique 5-tuple associate flow, and the latency for all received packets. We include box plots for each metric illustrating the minimum, maximum, and mean latencies to provide a comprehensive overview. These plots are derived from ten experimental runs for each coflowiness level, as defined in Section 3.4.2.1. Each sub-graph displays box plots for each benchmark, with different coflowiness values represented on the x-axis and corresponding latency on the y-axis. The y-axis units are either in microseconds or nanoseconds, depending on the specific measurement. This section is divided into three subsections, each dedicated to one of the latency metrics. Each subsection includes a main graph composed of six sub-graphs, organized in pairs for each metric - minimum, maximum, and mean latency. These sub-graphs allow for a direct comparison between the two FSDs: FB Hadoop and GSRPC. This structured layout facilitates the

evaluation of latency performance across different scenarios and coflowiness values.

4.1.1 First base packet latency statistics

In Section 3.3.1.2, we defined base flows as those flows used to predict which associate flows to preemptively load into the OVS megaflow cache. Essentially, these base flows are not present in the cache upon their initial arrival in OVS. Figure 4.1 presents latency statistics for the first packet of each unique 5-tuple base flow that arrives in OVS.

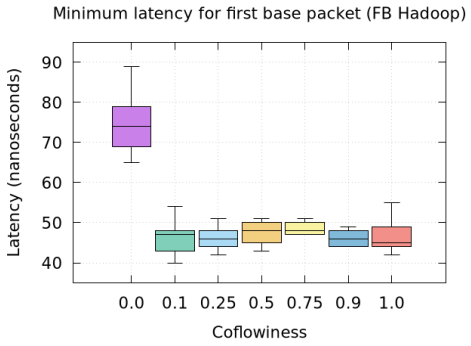
Analyzing Figure 4.1 yields several key observations. First, Figures 4.1a and 4.1b demonstrate that, when setting the coflowiness value at 0.0, which serves as the baseline scenario described in Section 3.4.2, the minimum latency for the GSRPC trace ranges from 2 to 300 times that of the baseline scenario in the FB Hadoop trace. For the remaining benchmarks with various degrees of associate flows loaded into the cache, the minimum latency for classifying a base flow ranges from 40 to 60 nanoseconds for the FB Hadoop trace, whereas the more bursty GSRPC trace exhibits a range from 50 nanoseconds to over ~200 nanoseconds.

Regarding maximum latency, Figures 4.1c and 4.1d indicate that both the FB Hadoop and GSRPC traces exhibit a trend where an increase in coflowiness results in a decrease in end-to-end packet latency, although this trend is more pronounced for the larger FB Hadoop trace than for its more bursty counterpart. Notably, the maximum latency for the GSRPC trace begins to decrease at a coflowiness of 0.5, compared to a coflowiness of 0.1 for the FB Hadoop trace. Moreover, the maximum latency for coflowiness values of 0.5 and 0.75 is approximately the same for both traffic traces. It is also evident that the reduction in maximum latency between the base case and scenarios where a subset of flows is predicted is more substantial for the FB Hadoop trace compared to the GSRPC trace, which shows a less clear trend due to its bursty nature and higher flow rate.

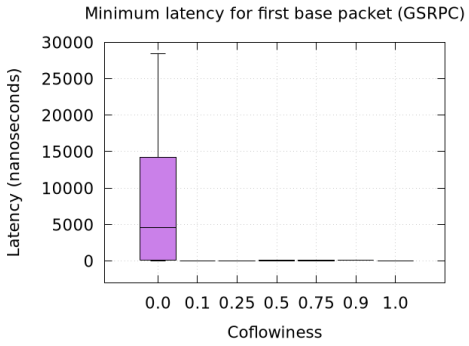
Finally, an examination of the mean latency of the first base packet for each distribution, as depicted in Figures 4.1e and 4.1f, reveals that the GSRPC trace results in a mean end-to-end latency up to $10\times$ higher across all coflowiness values compared to the FB Hadoop trace. The FB Hadoop trace, characterized by larger flow sizes, displays a clear decreasing trend where increased coflowiness steadily reduces mean latency. In contrast, the GSRPC trace shows a decrease in mean latency only starting at a coflowiness of 0.75. The observation that the mean latency at a coflowiness of 0.1 for

GSRPC exceeds that of the baseline scenario for the FB Hadoop trace is attributed to the increased flow rate, which consequently impacts the number of upcalls, CPU utilization, and latency of the GSRPC trace. This issue is further discussed in Section 5.1.4.

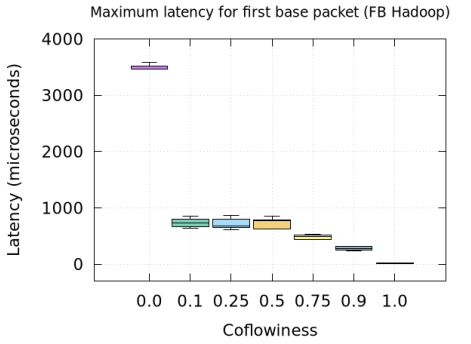
The results indicate that an increase in coflowiness generally reduces both the maximum and mean end-to-end base packet latencies for both the FB Hadoop and GSRPC traces, with the FB Hadoop trace showing a more consistent decreasing trend in latency compared to the GSRPC trace. However, the GSRPC trace exhibits significantly higher latencies overall due to its bursty traffic patterns which affect the flow rate of the trace as later discussed in Section 5.1.4.



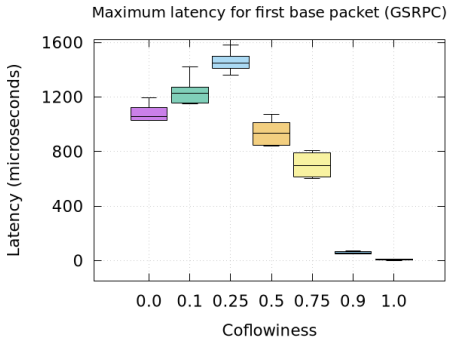
(a) Minimum end-to-end latency for the first base packet using the Facebook Hadoop flow size distribution



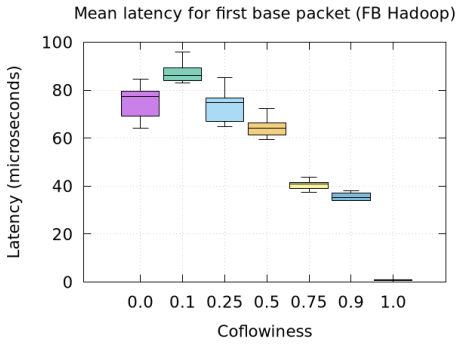
(b) Minimum end-to-end latency for the first base packet using the Google Search RPC flow size distribution



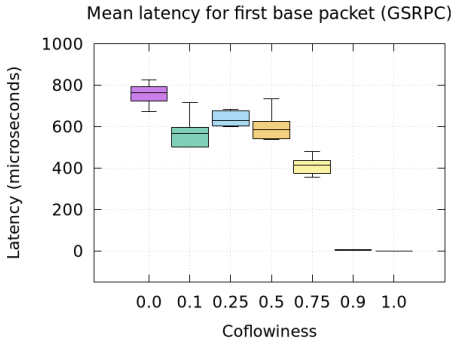
(c) Maximum end-to-end latency for the first base packet using the Facebook Hadoop flow size distribution



(d) Maximum end-to-end latency for the first base packet using the Google Search RPC flow size distribution



(e) Mean end-to-end latency for the first base packet using the Facebook Hadoop flow size distribution



(f) Mean end-to-end latency for the first base packet using the Google Search RPC flow size distribution

Figure 4.1: Statistical measures of end-to-end packet latency based on the first packet of each unique base flow processed by OVS using the two different flow size distributions.

4.1.2 First associate packet latency statistics

As defined in Section 3.3.1.2, an associate flow comprises related flows sharing common characteristics. Section 3.4.2 details our benchmarking methodology, which pre-loads these flows into the OVS datapath to simulate accurate prediction. Figure 4.2 visualizes the minimum, maximum, and mean end-to-end packet latency of the first packet of each unique 5-tuple associate flow.

The analysis of Figure 4.2, reveals several observations. First, Figures 4.2a and 4.2b indicate that at a coflowiness level of 0.0 (baseline), the GSRPC FSD experiences a significantly higher latency, approximately 300 to 4000 times greater, compared to the FB Hadoop FSD. However, for other coflowiness values, where a subset of flows are pre-loaded into the cache, the minimum latency for both FSDs converges to a similar range, approximately between 30 and 40 nanoseconds. This trend is expected because the minimum latency for the first packet of an associate flow always corresponds to a flow that is already present in the cache, thereby achieving the lowest possible latency.

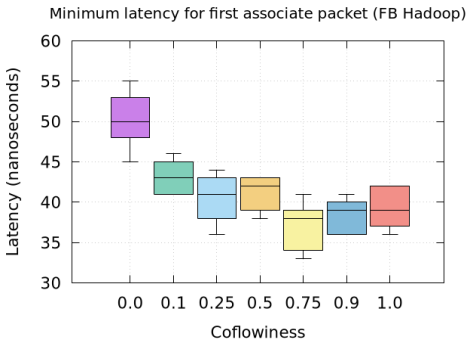
Moreover, analyzing Figures 4.2c and 4.2d, we observe that the maximum latency for the GSRPC FSD reaches up to ~1600 microseconds, whereas the maximum latency for the FB Hadoop FSD is only ~800 microseconds, indicating a $2\times$ difference in worst-case latency. Interestingly, for the GSRPC distribution, the worst-case latency occurs at a coflowiness of 0.25, which deviates from the expected pattern seen in the FB Hadoop trace, where the baseline scenario (coflowiness of 0.0) represents the highest latency. This anomaly in the GSRPC distribution can likely be attributed to the variation in the number of unique flows across different coflowiness configurations, as detailed in Table 5.1. There is an 18% increase in the number of unique flows between coflowiness values of 0.0 and 0.25. This issue and its impacts are further discussed in Section 5.1.2.1.

In contrast, the FB Hadoop trace exhibits a more predictable pattern, with a gradual decrease in latency as coflowiness increases. When all flows are pre-loaded into the cache (coflowiness of 1.0), both FSDs perform similarly, which is consistent with our expectations. Additionally, both distributions show a general decrease in maximum latency as coflowiness increases, although this trend is more pronounced and consistent in the less bursty FB Hadoop trace. However, Figure 4.2c also reveals anomalies due to the variations in the number of unique flows. Notably, when the coflowiness is at 0.25, the latency remains similar to that observed at 0.1. In contrast, at a coflowiness of 0.5, there is a slight increase in latency, which deviates from expectations. This

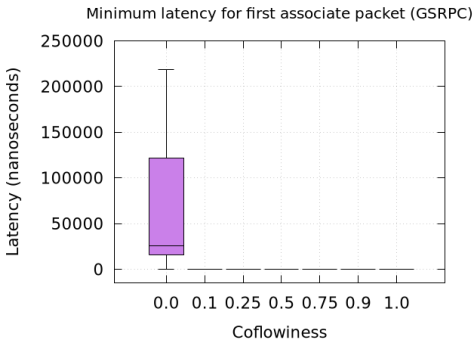
unexpected behavior is discussed in further detail in Section 5.1.2.1.

Finally, observing Figures 4.2e and 4.2f we observe that the GSRPC FSD exhibits a mean latency that is approximately 13 times higher than the FB Hadoop FSD in the baseline scenario of 0.0 coflowiness. In the optimal scenario, where all flows are pre-loaded into the cache, both distributions demonstrate similar mean latency. When analyzing the intermediate coflowiness levels from 0.1 to 0.75, the GSRPC distribution shows a significantly higher mean latency, ranging from approximately 50 to 70 times greater than the FB Hadoop distribution. Interestingly, the analysis reveals an unexpected increase in the mean latency of the GSRPC trace when coflowiness ranges between 0.25 and 0.75, compared to a coflowiness of 0.1. This increase can be attributed to variations in the number of unique flows. Specifically, lower coflowiness values have fewer unique flows, which leads to misrepresented latency measurements. This occurs because fewer flows result in a reduced load on OVS, as detailed in Section 5.1.2.1.

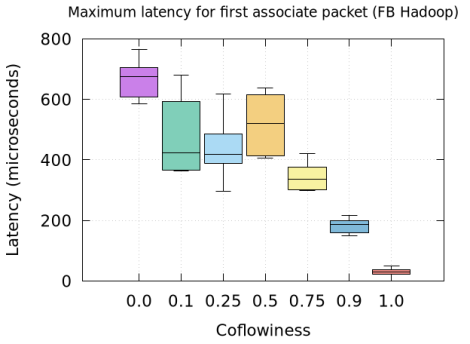
Both FSDs show a decreasing trend in mean latency as coflowiness increases. However, this trend is more pronounced and consistent in the FB Hadoop distribution, which maintains lower mean latency across all coflowiness configurations. Overall, the results indicate that while the GSRPC FSD generally exhibits higher latency across all metrics compared to the FB Hadoop distribution, both distributions show a decreasing trend in latency as coflowiness increases, with performance converging in the optimal case where all flows are cached. The key takeaway from Figure 4.2 is that accurately predicting and pre-loading just 10% of the flows significantly decreases end-to-end packet latency for the first associate packet of each unique flow.



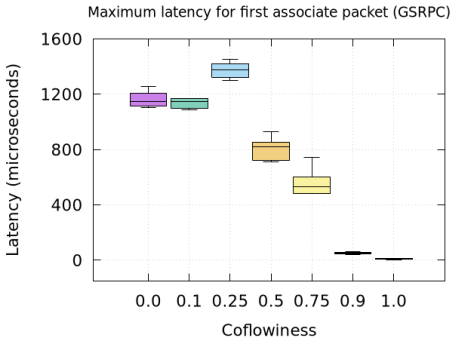
(a) Minimum end-to-end latency for the first associate packet using the Facebook Hadoop flow size distribution



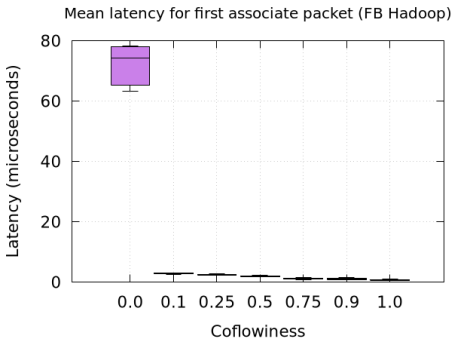
(b) Minimum end-to-end latency for the first associate packet using the Google Search RPC flow size distribution



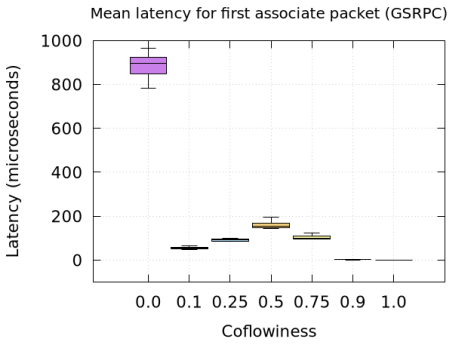
(c) Maximum end-to-end latency for the first associate packet using the Facebook Hadoop flow size distribution



(d) Maximum end-to-end latency for the first associate packet using the Google Search RPC flow size distribution



(e) Mean end-to-end latency for the first associate packet using the Facebook Hadoop flow size distribution



(f) Mean end-to-end latency for the first associate packet using the Google Search RPC flow size distribution

Figure 4.2: Statistical measures of end-to-end packet latency based on the first packet of each unique associate flow processed by OVS using the two different flow size distributions.

4.1.3 Aggregate packet latency statistics

This section presents latency statistics for all received packets, encompassing both base and associate packets, to provide an overview of the actual performance of OVS when applying prediction techniques using coflows. Figure 4.3 displays the minimum, maximum, and mean end-to-end packet latencies of all packets, beyond just the initial ones, using the two distinct FSDs: the FB Hadoop FSD and the GSRPC FSD.

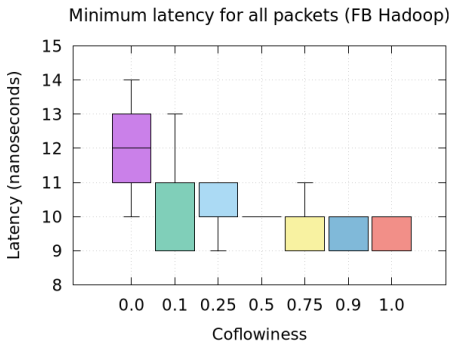
Figures 4.3a and 4.3b illustrate that the minimum end-to-end latency for the GSRPC baseline scenario with coflowiness 0.0 is significantly higher by a factor of $4\times$ than the baseline scenario for FB Hadoop. This suggests that GSRPC has a markedly higher base case minimum latency compared to FB Hadoop. For FB Hadoop, the minimum latency exhibits relatively low variance across different coflowiness values, indicating a consistent performance. In contrast, the GSRPC baseline scenario latency remains substantially elevated above other coflowiness values. The data presented in the two figures do not exhibit a definitive downward trend, suggesting minimal potential for optimization. This observation concludes that the mean aggregate minimum latency is consistently achieved through cache hits, limiting opportunities for further reduction.

Regarding maximum end-to-end latency, Figures 4.3c and 4.3d demonstrate notable differences when comparing the FB Hadoop baseline scenario, which features a coflowiness of 0.0, to other scenarios with higher coflowiness values. Specifically, the maximum latency in the baseline scenario is approximately $5\times$ higher than that observed in the scenario with a coflowiness of 0.1, and $60\times$ higher compared to the scenario with a coflowiness of 1.0. This pattern clearly illustrates a significant decrease in latency as coflowiness increases from 0.1 to 1.0. This trend indicates that higher coflowiness values can effectively reduce maximum latency in FB Hadoop. In contrast, the GSRPC FSD demonstrates a more stable maximum latency of around 1,000 microseconds for lower coflowiness values (0.0, 0.1, and 0.25). The impact of flow prediction on reducing maximum latency in GSRPC is less pronounced, with a particularly notable observation at a coflowiness of 0.25, where the latency worsens. As discussed in Sections 4.1.1 and 4.1.2, the increase in latency for configurations with higher coflowiness values is a direct consequence of the variance in the number of unique flows. This issue, which results in disproportionately higher latency due to the increased flow count, is further explored in Section 5.1.2.1. At higher coflowiness values (0.9 and 1.0), both FSDs converge to similar latency levels, which aligns with expectations

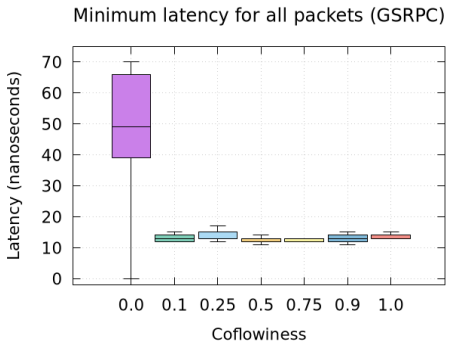
since most flows are cached at these levels.

The mean end-to-end latency for FB Hadoop, as shown in Figure 4.3e, exhibits a clear and consistent decreasing trend as coflowiness increases. This indicates that the prediction of flows positively impacts the overall mean latency. For the GSRPC FSD shown in Figure 4.3f, the mean latency also shows a decreasing trend; however, this trend is less pronounced and only becomes evident once at least 75% of the flows are predicted accurately. The observed trend can be attributed to the relationship between coflowiness and flow caching. Specifically, when coflowiness values are below 0.75, the advantages of caching additional flows are offset by the rise in the number of unique flows, compared to earlier configurations. However, at higher coflowiness values, the variation in unique flows across configurations diminishes, allowing OVS to effectively leverage the increased number of cached flows. Moreover, in the baseline scenario, the mean latency for GSRPC is approximately $10\times$ higher than that for FB Hadoop. For intermediate coflowiness values (0.1, 0.25, and 0.5), GSRPC's mean latency remains about 5 to 10 times higher than FB Hadoop. At a coflowiness of 0.75, the mean latency for GSRPC is approximately $15\times$ higher than that for FB Hadoop. At the highest coflowiness level (1.0), the mean latencies for both FSDs converge, which is expected since all flows are fully cached.

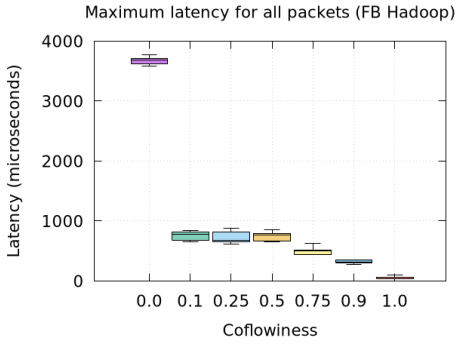
In summary, these observations indicate that accurately predicting a small portion of the flows, such as 10%, can significantly reduce the worst-case aggregate latency by up to $5\times$ when using the less bursty FB Hadoop trace. In contrast, the GSRPC trace requires a higher proportion of correctly predicted flows to achieve a noticeable reduction in latency. Regarding the mean aggregate latency, the FB Hadoop trace demonstrates a ~30% improvement when 25% of the flows are accurately predicted. While increasing coflowiness generally leads to reductions in both mean and maximum latencies, the degree of these reductions and their predictability are heavily influenced by the specific FSD employed.



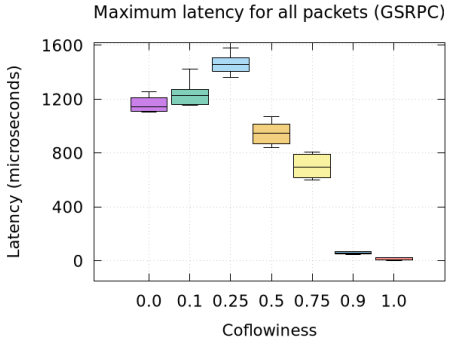
(a) Minimum end-to-end latency based on all packets using the Facebook Hadoop flow size distribution



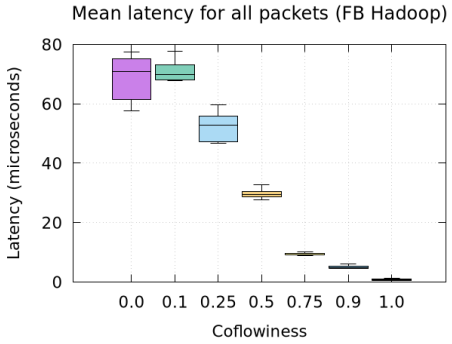
(b) Minimum end-to-end latency based on all packets using the Google Search RPC flow size distribution



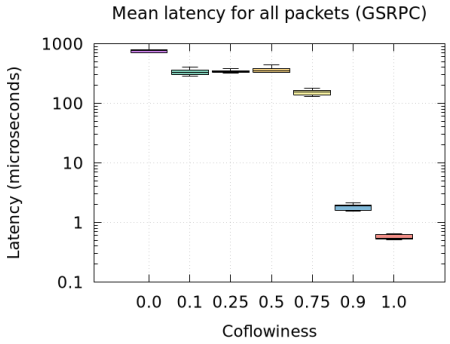
(c) Maximum end-to-end latency based on all packets using the Facebook Hadoop flow size distribution



(d) Maximum end-to-end latency based on all packets using the Google Search RPC flow size distribution



(e) Mean end-to-end latency based on all packets using the Facebook Hadoop flow size distribution



(f) Mean end-to-end latency based on all packets using the Google Search RPC flow size distribution

Figure 4.3: Statistical measures of end-to-end packet latency based on all packets of each unique flow processed by OVS using the two different flow size distributions.

4.2 Analysis of CPU Load Distribution

This section aims to illustrate the impact of varying coflowiness values and FSDs on CPU utilization and resource distribution within OVS. Section 4.2.1 begins by presenting figures that depict CPU utilization in terms of system processes, software interrupts, and total system CPU load across different OVS configurations. Subsequently, Section 4.2.2 explores the effects of OVS handler threads on CPU utilization. This analysis includes CPU usage in both user and kernel spaces, as well as the total CPU utilization attributed to these threads.

4.2.1 Resource Allocation and CPU Usage

This section presents the CPU resource utilization required to achieve varying end-to-end packet latencies across different configurations, as discussed in Section 4.1. Figure 4.4 comprises six sub-figures that delineate the CPU usage for system processes, software interrupts, and the overall system load. These comparisons are made while benchmarking OVS using the two FSDs. Each sub-figure plots the CPU utilization as a percentage on the y-axis against time in seconds on the x-axis, under various coflowiness configurations.

4.2.1.1 CPU utilization of system processes

Analyzing Figure 4.4, we can observe significant differences in CPU utilization between the system processes of FB Hadoop and GSRPC. For FB Hadoop, depicted in Figure 4.4a, there is a discernible trend where lower coflowiness values correspond to higher CPU utilization. This trend is accentuated by a gradual decrease in CPU utilization over time, more pronounced at lower coflowiness values due to increased flow caching. The most notable reduction occurs between coflowiness values of 0.5 and 0.25, where CPU utilization drops from around 18% to 10%. At maximum coflowiness (1.0), the system requires only about 1.5% of CPU utilization to manage system processes, whereas the baseline scenario (0.0 coflowiness) peaks at ~29%. Conversely, the GSRPC trace, shown in Figure 4.4b, exhibits no consistent trend of decreasing CPU utilization with increased coflowiness. Only the optimal scenario at a coflowiness of 1.0 shows significantly lower CPU utilization compared to other scenarios, which maintain around 30-40% utilization. Contrary to the FB Hadoop results, CPU utilization does not decrease over time in the GSRPC setup; in some instances, it even increases. Notably, a coflowiness of 0.9 registers the highest CPU utilization across

all scenarios. The lack of reduced CPU utilization, despite improved flow prediction, stands in sharp contrast to the observations from the FB Hadoop study. This discrepancy could stem from CPU overload while processing the GSRPC trace, which might impede the system's ability to manage new flows effectively. As a result, the expected trend becomes obscured. With additional processing power, it is conceivable that the graphs would reveal a more discernible trend. The detailed examination of CPU utilization, along with other related topics, is presented in Section 5.1.3.

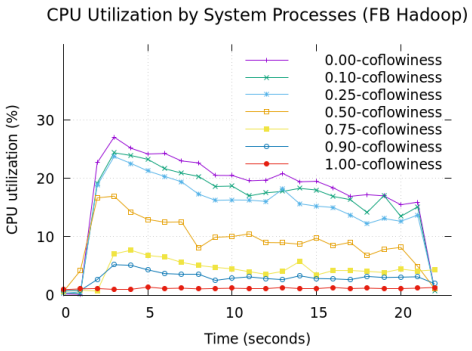
4.2.1.2 CPU utilization of software interrupts

Moving on to Figures 4.4c and 4.4d, which illustrate the CPU utilization due to software interrupts, we observe that in the FB Hadoop benchmarks in Figure 4.4c, the trend in CPU utilization is less distinct than that observed in the system processes graph. Although a general decreasing trend in CPU utilization is evident, it is not as pronounced. The optimal scenario at a coflowiness of 1.0 shows the lowest CPU utilization at approximately 5%, with the remaining cases clustering between 10% and 15%. A subtle pattern suggests that lower coflowiness values correlate with an increase in software interrupts, with the optimal scenario experiencing minimal upcalls and, consequently, fewer software interrupts to user space. Unlike the trends observed in system process utilization, there is no clear decrease in CPU utilization over time; in fact, there is a slight increase towards the end of the benchmark. Transitioning to the GSRPC FSD, depicted in Figure 4.4d, the scenario contrasts with that of FB Hadoop. Here, the optimal coflowiness value does not result in significantly lower CPU utilization for software interrupts compared to other cases. All scenarios exhibit similar performance, with CPU utilization ranging from just over 30% to 45%. There is no observable trend where lower coflowiness values lead to increased software interrupts. The graph for GSRPC is more volatile, aligning with the latency results for GSRPC shown in Section 4.1.

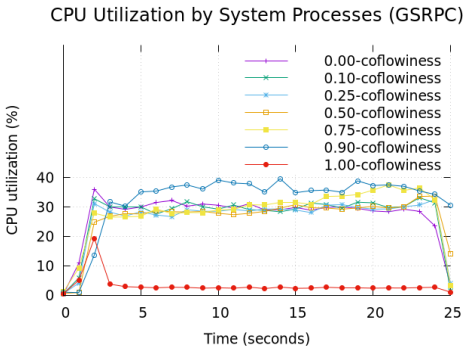
4.2.1.3 Total CPU utilization

Lastly, the examination of total CPU utilization for FB Hadoop, as illustrated in Figure 4.4e, reveals a consistent trend similar to that observed in the system processes graph. Lower coflowiness values correspond to higher CPU utilization, with the optimal scenario at a coflowiness of 1.0 exhibiting the lowest utilization of approximately 7% on average. Conversely, the baseline scenario with a coflowiness of 0.0 experiences the highest CPU

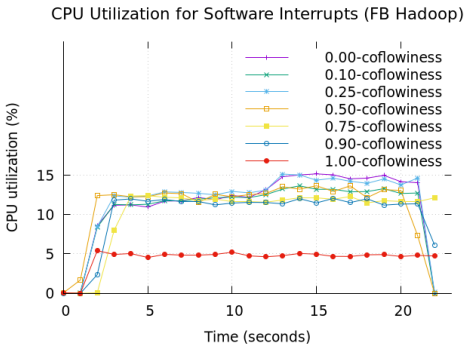
utilization, ranging from approximately 70% to 96%. A gradual decrease in CPU utilization over time is noted, attributed to increased caching of flows. However, this trend becomes less distinct at higher coflowiness levels, where most flows are already cached. Predicting 50% of flows accurately results in a significant reduction in total CPU utilization, lowering it to about 40-60% compared to the 70-96% observed in the baseline scenario where no flows are predicted. In contrast, when analyzing the total CPU consumption using the GSRPC trace, shown in Figure 4.4f, there is no evident trend of decreasing CPU utilization with increased coflowiness, except in the optimal scenario at 1.0 coflowiness. All coflowiness values that fail to predict all flows correctly maintain high total CPU utilization, typically between ~90-97%. Even with an accurate prediction of all flows, total CPU utilization remains substantial, starting at ~40% and peaking at ~70%, which is markedly higher than the mere ~7% for the optimal scenario in FB Hadoop. Throughout the duration, there is no discernible reduction in CPU utilization due to flow caching. Additionally, all coflowiness values, excluding the baseline scenario, exhibit similar total CPU utilization levels to the baseline scenario in FB Hadoop. The 0.5 coflowiness value in GSRPC is twice as high compared to its counterpart in FB Hadoop. Importantly, even with perfect flow prediction, the total CPU utilization in GSRPC is up to ~6$\times$ higher than that observed in the optimal FB Hadoop scenario. The analysis presented indicates that the advantage of accurate flow prediction for decreasing overall CPU utilization in the GSRPC environment is marginal. This observation is particularly evident when the CPU is operating at full capacity, which suggests that the system is already overloaded. Under such conditions, it is apparent that flow prediction has minimal impact on CPU utilization, especially at lower coflowiness configurations. Finally, the initial spike in CPU utilization observed at the beginning of the experiments can be attributed to several factors. Firstly, OVS begins to necessitate upcalls to its daemon in user space. Secondly, the pod daemons demand additional processing power due to the parsing of incoming packets. These topics are further explored in Section 5.1.3.



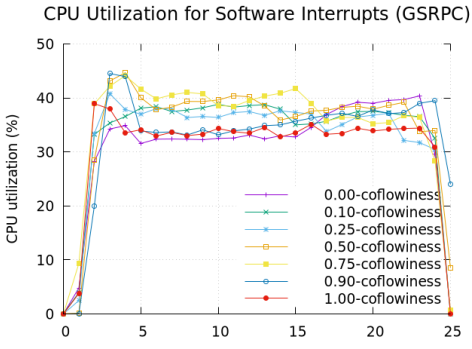
(a) System Processes CPU Utilization using the Facebook Hadoop flow size distribution



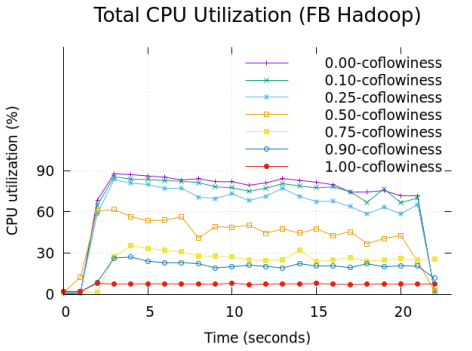
(b) System Processes CPU Utilization using the Google Search RPC flow size distribution



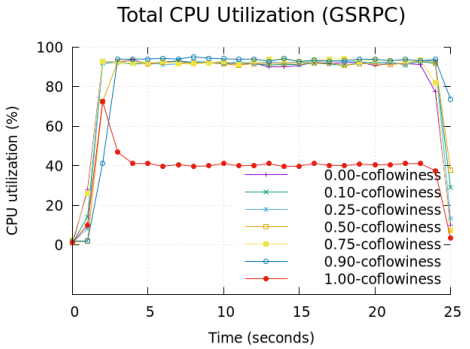
(c) Software Interrupts CPU Utilization using the Facebook Hadoop flow size distribution



(d) Software Interrupts CPU Utilization using the Google Search RPC flow size distribution



(e) Total CPU Utilization using the Facebook Hadoop flow size distribution



(f) Total CPU Utilization using the Google Search RPC flow size distribution

Figure 4.4: Comparison of CPU utilization between Facebook Hadoop and Google Search RPC flow size distributions for system processes, software interrupts, and total usage.

4.2.2 CPU Utilization of OVS Handler Threads

In this section, we present an analysis of CPU utilization specifically attributable to OVS handler threads, which is included in the total CPU utilization figures shown in the graphs in the previous section. Figure 4.5 depicts the CPU resources required for managing OVS handler threads. These sub-figures differentiate CPU usage observed in user space, kernel space, and their cumulative total. The CPU utilization percentage is plotted on the y-axis against time in seconds on the x-axis, across various configurations of coflowiness.

4.2.2.1 User processes handler threads utilization

In Figure 4.5, two distinct patterns of CPU utilization by user threads are observed, as detailed in Figures 4.5a and 4.5b. Initially, Figure 4.5a reveals a pronounced trend where user thread CPU utilization decreases with increasing coflowiness, although this trend is less pronounced at coflowiness values of 0.1 and 0.0. The optimal scenario, at 1.0 coflowiness, registers the lowest average CPU utilization at only 0.2%. This reduced utilization is expected as OVS eliminates the need for upcalls to user space when all flows are efficiently cached. The base case shows a substantial 50% CPU utilization, which is halved to 25% at a coflowiness of 0.5. Over time, there is a marginal decrease in utilization across all coflowiness levels, more noticeable at lower values, attributed to increased flow caching. The base case is approximately 180 times more CPU intensive than the optimal case at 1.0 coflowiness.

In contrast, the pattern in the GSRPC environment shown in Figure 4.5b differs notably. No consistent trend of decreasing CPU utilization with increasing coflowiness is observed, except in the optimal scenario at 1.0 coflowiness. Here, utilization consistently ranges between 7-13%, significantly lower than the maximum 50% seen in the FB Hadoop benchmark. This discrepancy is likely due to the bursty nature of traffic, where OVS handles multiple upcalls in batches, potentially utilizing CPU cache and other resources more efficiently. Unlike the FB Hadoop case, no clear trend of decreasing utilization over time is evident, as the caching of more flows does not significantly impact overall CPU utilization.

4.2.2.2 System processes handler threads utilization

Moving to the analysis of CPU utilization from a system process perspective, as depicted in Figures 4.5c and 4.5d, we observe distinct trends. In Figure

4.5c, there is a consistent decrease in system threads CPU utilization with increasing coflowiness, mirroring the trend observed in user threads. Notably, this trend of decreasing utilization over time is evident across all coflowiness levels except at 0.0. In the optimal case of 1.0 coflowiness, the CPU utilization is minimized to an average of only ~0.1%, underscoring efficient flow caching. The baseline scenario registers the highest system thread CPU utilization at 25%. At a coflowiness of 0.5, there is a reduction to below 10% utilization, with the baseline scenario being ~250× more CPU intensive than the optimal scenario at 1.0 coflowiness, indicating a significant variation greater than that seen in user process utilization.

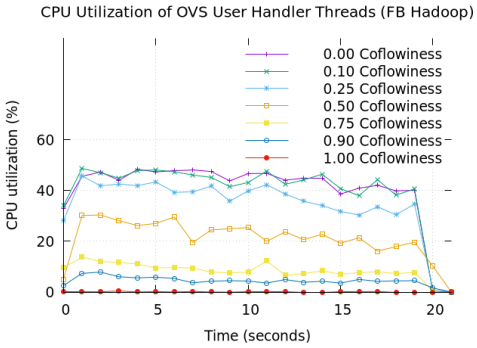
In contrast, Figure 4.5d which captures a more bursty trace, reveals no consistent trend in decreasing system threads CPU utilization with increased coflowiness, except in the optimal case at 1.0 coflowiness where the utilization again bottoms out at 0.1% on average. In cases where not all flows are predicted correctly, utilization spans from ~15-30%, matching the maximum values observed in the FB Hadoop setup. Notably, a coflowiness of 0.9 results in the highest utilization, the cause of which remains unclear. Contrary to the FB Hadoop results, no decreasing trend in utilization is evident as time progresses, potentially due to the nature of flow caching in this context. The difference between the highest and lowest utilization here is vast, at ~300×, a discrepancy more pronounced than observed in the FB Hadoop analysis. Furthermore, unlike in the FB Hadoop scenario, predicting all flows correctly offers benefits only in the optimal case at 1.0 coflowiness.

4.2.2.3 Total handler threads utilization

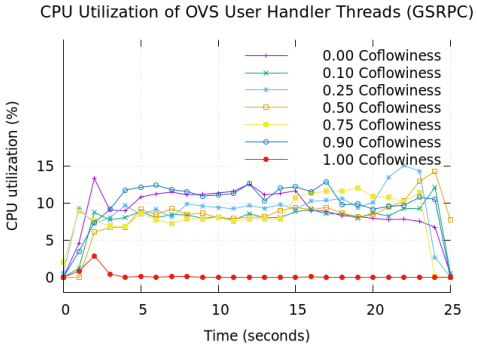
Lastly, Figures 4.5e and 4.5f, which summarize the total CPU load required by the handler threads, exhibit trends consistent with those observed in the individual analyses of user and system utilization. In Figure 4.5e, there is a discernible trend of decreasing total CPU utilization with increasing coflowiness, complemented by a general decline in utilization over time. The total utilization in this figure ranges from a minimal ~0.1% in the optimal scenario to a substantial ~70% in the baseline scenario. Notably, achieving a 50% flow prediction rate results in a halving of the total utilization compared to the base scenario.

Similarly, Figure 4.5f, which analyzes the GSRPC setup, aligns with the observed patterns in its preceding figures. This figure illustrates that there is no consistent trend of decreasing total CPU utilization with increasing coflowiness, except in the optimal case of 1.0 coflowiness where utilization

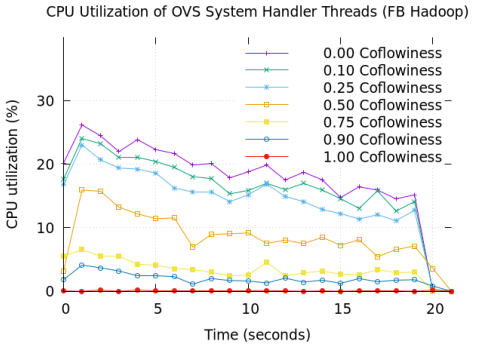
is as low as ~0.1%, mirroring the result seen in the FB Hadoop setup. Unlike the decreasing trends over time noted in other figures, this setup shows no such trend due to the caching dynamics. All coflowiness scenarios, aside from the optimal, exhibit utilization levels between approximately 20% and 40%. Moreover, only the optimal scenario of predicting all flows correctly at 1.0 coflowiness demonstrates a significant utilization benefit. Lastly, it should be noted that the benchmark with 0.9 coflowiness results in the highest CPU utilization, even higher than the baseline scenario, which is unexpected. This can be attributed to the varying number of unique flows in different coflowiness configurations, as discussed further in Section 5.1.2.1.



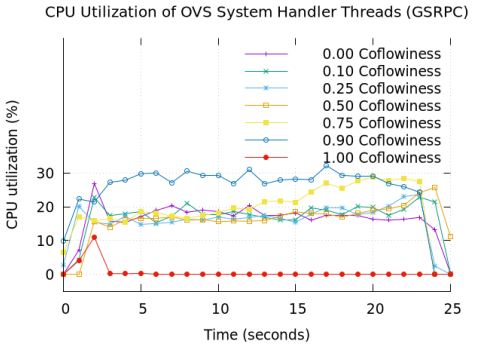
(a) CPU Utilization of OVS Handler Threads in User Space using the Facebook Hadoop flow size distribution



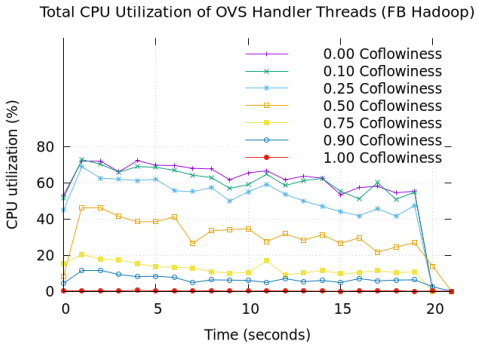
(b) CPU Utilization of OVS Handler Threads in User Space using the Google Search RPC flow size distribution



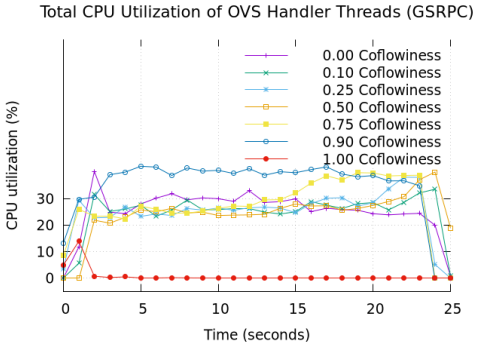
(c) CPU Utilization of OVS Handler Threads in Kernel Space using the Facebook Hadoop flow size distribution



(d) CPU Utilization of OVS Handler Threads in Kernel Space using the Google Search RPC flow size distribution



(e) Total CPU Utilization of OVS Handler Threads using the Facebook Hadoop flow size distribution



(f) Total CPU Utilization of OVS Handler Threads using the Google Search RPC flow size distribution

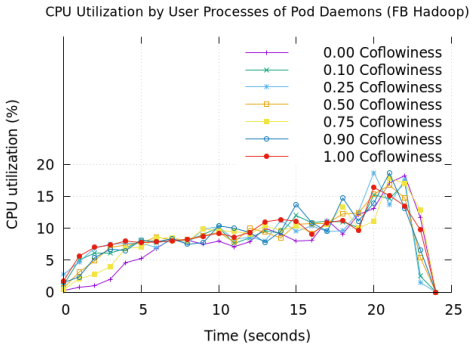
Figure 4.5: Comparative analysis of CPU utilization for OVS handler threads in user space, kernel space, and combined total usage under varying flow size distributions

4.2.3 Resource Utilization of Pod Daemons

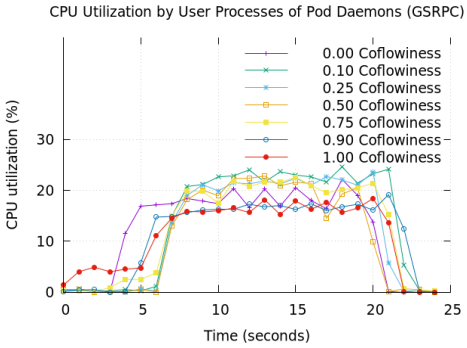
This section analyzes the CPU resource utilization by the daemons running on each cluster pod, which are responsible for parsing timestamps and sequence numbers from UDP packet payloads. Figure 4.6 is divided into six sub-figures, each depicting the CPU usage for user processes, system processes, and the total system load caused by the pod daemons while benchmarking OVS using two FSDs. These sub-figures plot the CPU utilization as a percentage on the y-axis against time in seconds on the x-axis, under various coflowiness configurations.

Sub-figures 4.6a and 4.6b highlight that the peak CPU utilization for user processes is notably higher with the GSRPC trace, reaching ~25%, compared to just under ~20% for the FB Hadoop trace. Additionally, the increase in CPU utilization occurs later with GSRPC than with FB Hadoop. CPU utilization remains relatively low for system processes for FB Hadoop, peaking at slightly over ~5%. In contrast, it is significantly higher for GSRPC, which reaches ~20% utilization. The total CPU utilization demanded by the pod daemons reaches ~25% with the FB Hadoop trace, whereas it escalates to ~45% with GSRPC. It is observed that the peak in CPU utilization consistently occurs later in all measured statistics for FB Hadoop, unlike the relatively stable pattern seen with GSRPC. The increased overhead on the system arises from managing several simultaneously open sockets, particularly impacting socket lookups. In the case of the FB Hadoop benchmark, there is a notable decrease in latency due to more efficient handling of a greater number of cached flows. This, in turn, imposes an additional load on the pod daemons. Conversely, the GSRPC benchmark benefits less from flow prediction, resulting in a more stable packet delivery rate to the pod daemons. Consequently, CPU utilization of the pod daemons does not escalate over time. The influence of pod daemons on overall CPU utilization is examined in further detail in 5.1.3.2. To further elucidate the specific causes of the observed CPU utilization patterns, detailed profiling using tools like *perf* is recommended.

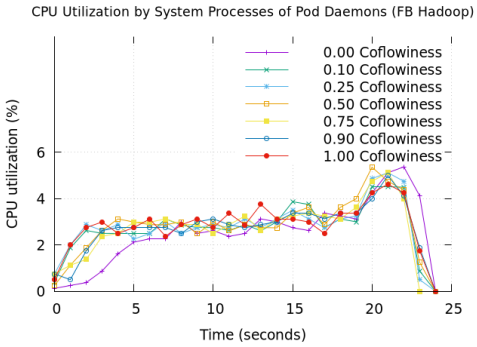
Furthermore, for all three types of statistics, CPU utilization begins to rise immediately with FB Hadoop, whereas it starts later with GSRPC, indicating delayed resource demands. In summary, the more bursty GSRPC trace necessitates increased effort from the pod daemons, resulting in higher CPU resource consumption. These findings should be considered alongside the overall CPU utilization metrics to determine the contributions of the daemons compared to other processes on the SUT.



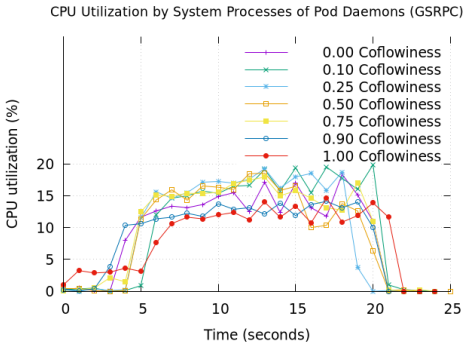
(a) Pod Daemons User Processes CPU Utilization using the Facebook Hadoop flow size distribution



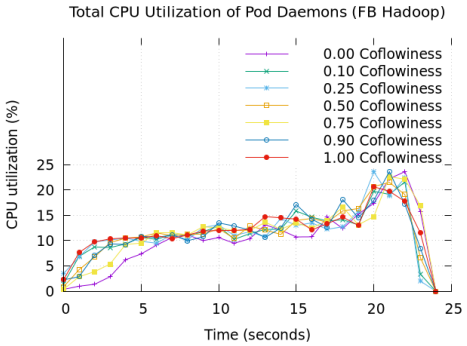
(b) Pod Daemons User Processes CPU Utilization using the Google Search RPC flow size distribution



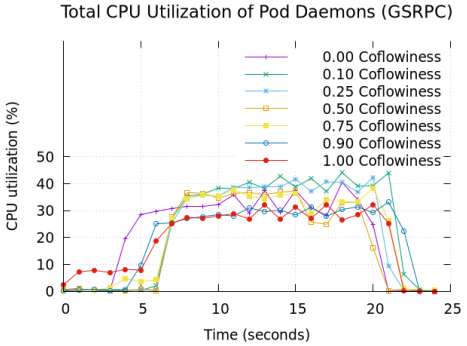
(c) Pod Daemons System Processes CPU Utilization using the Facebook Hadoop flow size distribution



(d) Pod Daemons System Processes CPU Utilization using the Google Search RPC flow size distribution



(e) Total CPU Utilization of Pod Daemons using the Facebook Hadoop flow size distribution



(f) Total CPU Utilization of Pod Daemons using the Google Search RPC flow size distribution

Figure 4.6: Comparison of CPU utilization by pod daemons between Facebook Hadoop and Google Search RPC flow size distributions for user processes, system processes, and total usage.

4.3 Upcall statistics

In this section, we analyze the number of upcalls per second made to the `ovs-vswitchd` across various coflowiness configurations and FSDs. These upcalls indicate when new flows are processed over time, capturing differences in packet/flow order and batch sizes. Figure 4.7 presents two subfigures: 4.7a and 4.7b. Each sub-figure shows the number of upcalls per second (y-axis) as a function of time (x-axis) for different FSDs. To enhance readability, we include a subset of the coflowiness values, specifically the baseline scenario, the optimal scenario, and the 0.25 and 0.75 coflowiness quantiles. The term "number of upcalls" specifically refers to those performed during the benchmarking process. This count excludes the upcalls necessary for preloading associated flows into the datapath cache before benchmarking.

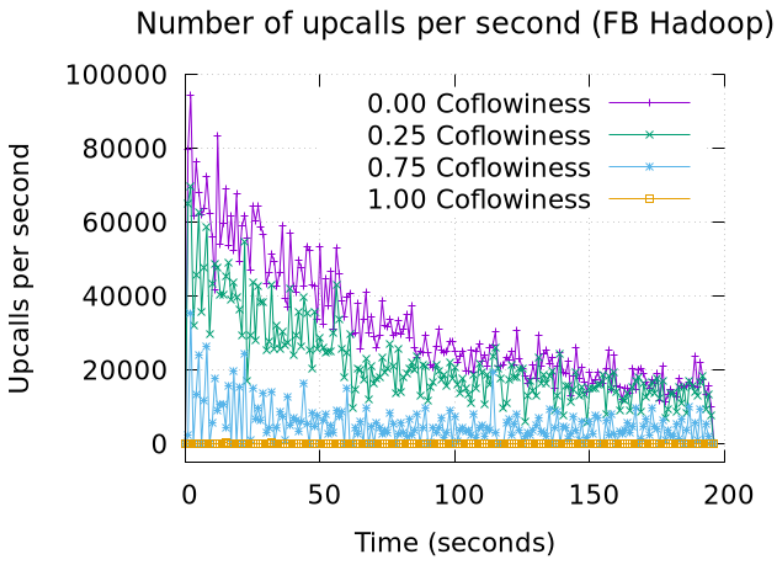
In Figure 4.7a, we observe that, except for the optimal scenario, where all flows are predicted accurately, the number of upcalls per second decreases as time progresses. This trend is more pronounced for lower coflowiness values, as more flows are cached over time. This behavior aligns with previous CPU utilization results, as the number of upcalls is directly correlated with CPU usage when handling new flows. Initially, in the FB Hadoop scenario, the baseline scenario configuration reaches a peak of approximately 95,000 upcalls per second when traffic begins. For the optimal configuration, the number of upcalls remains consistently low, around 50 upcalls per second. Intermediate configurations, such as the 0.25 and 0.75 coflowiness values, experience initial peaks of approximately 70,000 and 35,000 upcalls per second, respectively. Over time, the number of upcalls gradually decreases and converges toward the optimal scenario. After the initial decline, the graphs exhibit noticeable fluctuations in the number of upcalls per second, particularly for lower coflowiness values. These fluctuations suggest variability in the processing of new flows, although their magnitude diminishes over time.

The GSRPC benchmark, as shown in Figure 4.7b, exhibits distinct behavior compared to the FB Hadoop benchmark, primarily due to the much smaller average flow size in the distribution. This results in a significantly shorter benchmark duration, approximately 5 seconds, compared to ~200 seconds for the FB Hadoop benchmark. For the baseline configuration, the number of upcalls per second peaks at around 115,000, making it the highest among all configurations. Notably, unlike in the FB Hadoop benchmark, where the peak in upcalls occurs immediately upon traffic initiation, the GSRPC benchmark experiences a delay, with the peak occurring between 1 and 2 seconds after the start. This delayed peak suggests that upcall

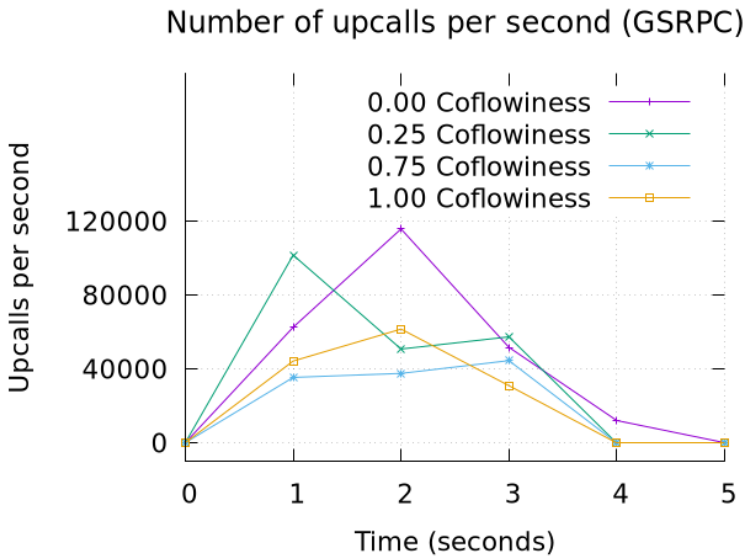
batching occurs in this scenario. Furthermore, the GSRPC benchmark lacks the frequent fluctuations observed in the FB Hadoop benchmark, although this may be attributed to the much shorter duration of the test.

Interestingly, unlike the FB Hadoop benchmark, the number of upcalls per second does not show a gradual decrease over time in the GSRPC benchmark. This behavior suggests that fewer flows are being cached, which may be due to the different nature of the FSD in this test. As seen in the CPU utilization plots for the GSRPC, the number of upcalls per second in the optimal configuration is not necessarily lower than in other coflowiness configurations. The optimal configuration results in a higher peak in upcalls per second than the 0.75 coflowiness value. This outcome indicates that the characteristics of the optimal case for the GSRPC benchmark differ from the FB Hadoop benchmark, possibly due to the unique traffic patterns and flow dynamics in each benchmark.

The FB Hadoop benchmark shows an early peak in upcalls followed by a gradual decrease, with frequent fluctuations in lower coflowiness values. In contrast, the GSRPC benchmark has a shorter duration, delayed upcall peak, and lacks fluctuations, with consistently higher upcall rates. Overall, the GSRPC generates more upcalls across all configurations compared to FB Hadoop.



(a) Number of upcalls made to `ovs-vswitchd` per second, using the Facebook Hadoop flow size distribution.



(b) Number of upcalls made to `ovs-vswitchd` per second, using the Google Search RPC flow size distribution

Figure 4.7: Comparative analysis of the number of upcalls made to `ovs-vswitchd` under varying flow size distributions

Chapter 5

Discussion

This chapter consists of three major sections that provide an analysis of the results presented in Chapter 4 and respectively, an analysis regarding the different parts of the project as well as how they may have impacted the results. Specifically, in Section 5.1, we analyze the significance of the results, potential underlying causes, the impact of varying traffic conditions on OVS performance, and the real-world implications of these findings. Further, in Section 5.2, we discuss how our benchmarking environment and methodology could be improved as well as alternative benchmarking metrics that might give a better representation of the performance of OVS. Lastly, in Section 5.3, we shortly touch upon the potential sustainability and ethical impacts of our findings.

5.1 Analysis of results

In this section, we analyze latency, CPU utilization, and upcall statistics to better understand the underlying factors influencing the results. Section 5.1.1 initiates this analysis by exploring the correlation between coflowiness and latency; specific latency anomalies are then discussed in Section 5.1.2. Sections 5.1.3 and 5.1.4 examine how coflowiness affects CPU utilization and upcall intensity, respectively. Finally, Section 5.1.5 considers the potential impacts of increased latency on connection-oriented sessions with various configurations.

5.1.1 Impact of Coflowiness on Latency

Coflowiness, defined as the percentage of associated flows predicted and preloaded into the OVS cache, plays a critical role in determining the latency experienced by packets traversing the network. As coflowiness increases, more flows are preemptively loaded into the OVS megaflow cache, reducing the likelihood of cache misses. This proactive caching strategy directly influences the latency metrics observed in the benchmarks. In scenarios with low coflowiness (e.g., 0.0 to 0.25), only a small fraction of flows are preloaded into the cache. Consequently, incoming packets are more likely to encounter cache misses, triggering upcalls to the `ovs-vswitchd` user space daemon. These upcalls involve significant overhead due to context switching and additional processing, leading to increased end-to-end packet latency. The effect is particularly pronounced in the GSRPC trace, which is characterized by bursty traffic and a high rate of new flow arrivals.

As coflowiness increases to moderate levels (e.g., 0.5 to 0.75), a larger portion of the flows are cached. This reduces the frequency of cache misses and upcalls, leading to a noticeable decrease in both mean and maximum latencies. The FB Hadoop trace, with its larger and less bursty flows, benefits significantly from this increase in coflowiness. The reduction in latency is more gradual in the GSRPC trace due to its inherent bursty traffic. At high coflowiness levels (e.g., 0.9 to 1.0), the cache contains entries for nearly all anticipated flows. This maximizes the likelihood of cache hits and minimizes the need for upcalls. As a result, both the FB Hadoop and GSRPC traces exhibit converging latency metrics, approaching the optimal performance achievable with OVS. The convergence underscores the effectiveness of comprehensive flow prediction and caching in mitigating latency irrespective of the underlying traffic patterns.

5.1.1.1 Cache Efficiency with Larger Flows

The FB Hadoop trace demonstrates substantial improvements in latency with increased coflowiness. Larger flows inherently benefit more from caching mechanisms because once a flow entry is installed in the cache, subsequent packets belonging to that flow can utilize the cached actions without incurring additional overhead. This results in a sustained period of efficient packet processing within the kernel space, minimizing the need for upcalls.

The OVS megaflow cache operates by storing flow entries that can match multiple packets sharing common attributes. For larger flows, the cache entry remains relevant for a longer time, servicing multiple packets and thus

amortizing the cost of the initial cache installation. This efficiency is reflected in the observed latency reductions for the FB Hadoop trace as coflowiness increases. The decrease in both mean and maximum latencies indicates that preloading flows effectively leverage the cache's capabilities, leading to optimal in-kernel processing and reduced overall latency.

At the highest coflowiness level of 1.0, where all possible flows are accurately predicted and preloaded into the OVS cache, the latencies for both the FB Hadoop and GSRPC traces converge. This convergence is a direct consequence of eliminating cache misses for anticipated flows, ensuring that packets are processed entirely within the kernel datapath. With complete caching, the need for upcalls is effectively nullified for the flows in question. Packets benefit from the fastest possible processing path, utilizing the pre-installed flow entries in the megaflow cache. The latency experienced is thus minimized and becomes largely independent of the flow characteristics that previously differentiated the two FSDs. Any residual differences in latency are attributable to factors outside the caching mechanism, such as packet size variations and inherent processing delays.

5.1.1.2 Cache Challenges with Bursty Flows

Conversely, the GSRPC trace presents challenges for caching due to its high rate of new, small flows and bursty traffic patterns. Frequent arrival of new flows leads to a higher likelihood of cache misses, resulting in more upcalls and increased latency. This phenomenon adversely affects the cache's efficiency. Since new flows are short-lived and may not be predicted and preloaded into the cache, they continually trigger the slow path processing in OVS. The upcall overhead becomes a significant contributor to latency, as almost every packet requires user space intervention to install a corresponding cache entry. The benchmarks reflect this challenge, showing that the GSRPC trace maintains higher latencies even as coflowiness increases until a substantial proportion of flows are cached.

5.1.1.3 Caching Effects Summary

The discussion elucidates the complexities involved in optimizing OVS performance through flow prediction and caching. The impact of coflowiness on latency is evident, with higher levels leading to reduced cache misses and improved latency metrics. However, the degree of benefit is heavily influenced by the nature of the network traffic. The FB Hadoop and GSRPC traces

represent two extremes of network traffic patterns, providing valuable insights into how OVS handles different workloads.

- **FB Hadoop Trace**: The less bursty nature of the FB Hadoop trace, with its larger and longer-lived flows, makes it more amenable to caching. Even at lower coflowiness levels, preloading a modest portion of the flows yields significant improvements in latency. The cache entries remain relevant for longer durations, serving multiple packets and reducing the frequency of cache misses. This results in a consistent and predictable decrease in both mean and maximum latencies as coflowiness increases.
- **GSRPC Trace**: The GSRPC trace poses challenges due to its high burstiness and prevalence of small, short-lived flows. The effectiveness of caching is diminished because new flows continuously arrive, many of which are not preloaded into the cache. Consequently, higher coflowiness levels are required to achieve noticeable latency reductions. Even then, the improvements are less pronounced compared to the FB Hadoop trace. The results highlight the limitations of caching strategies in environments with bursty traffic patterns.

The analysis suggests that flow prediction and preloading are more beneficial in networks with predictable traffic patterns and stable flows. In such environments, operators can achieve significant performance gains with relatively modest flow prediction. For networks characterized by bursty and unpredictable traffic, alternative approaches may be necessary. These could include dynamic scaling of resources, more aggressive caching policies, or enhancements to OVS that better handle high rates of new flow arrivals [54].

5.1.2 Anomalies in Latency Results

When studying the results of Section 4.1, unexpected anomalies can be observed in Figures 4.1e, 4.1d, 4.2d, 4.2f, and 4.3d where some higher coflowiness levels resulted in increased latency, contrary to the anticipated performance improvement. For example, at a coflowiness level of 0.25 seen in Figure 4.2d, the latency metrics were worse than those at the baseline scenario of 0.0 coflowiness. This counterintuitive outcome warrants a thorough examination of the underlying causes, which can be attributed to methodological shortcomings in constructing the traffic trace, such as variations in the number of unique flows and the distribution of those flows.

5.1.2.1 Number of unique flows

The main contributor to the observed anomalies in latency results is the variance in the number of unique 5-tuple flows contained within the traces when changing the coflowiness configuration. A methodological weakness arises from how the traces are generated and the way coflowiness values are defined. As seen in Table 5.1, benchmarks with lower coflowiness values encompass fewer unique 5-tuple flows, whereas configurations with higher coflowiness values contain a greater number of unique flows. Despite these differences, all configurations maintain the same total number of flows when duplicates are counted. This phenomenon results in configurations with more unique flows imposing a greater load on OVS and the SUT. Consequently, higher coflowiness configurations are misrepresented as having higher latency than they would if all traces contained the same number of unique flows. The primary reason for this misrepresentation is that a greater number of unique flows trigger more upcalls.

However, the influence of this variance on the overall research outcomes might be less significant than initially perceived. The FB Hadoop and GSRPC benchmarks, which both employ the same trace data and the number of unique flows, merely adjust the FSD. This uniformity mitigates the potential adverse effects, validating the comparability of results across the two benchmark configurations. Notably, the FB Hadoop benchmark consistently demonstrates that higher coflowiness values are associated with reduced latency, establishing a clear positive correlation. Conversely, the GSRPC benchmark reveals more anomalies, deviating from expected outcomes. Significantly, all but one anomaly occurs exclusively in the GSRPC benchmarks, highlighting that bursty traffic patterns are particularly susceptible to variations in flow count. This issue is further explored in Section 5.1.4. It is important to recognize that while the disparity in unique flow numbers adversely affects latency comparisons between coflowiness configurations of benchmarks using the same FSD, it does not compromise the validity of comparing results across different FSDs. This resilience is due to the consistency of unique flow numbers across coflowiness configurations when switching between these two benchmarks [54].

5.1.2.2 Distribution of unique flows

An additional factor that may influence or exacerbate the observed latency anomalies relates to the distribution of unique associate and base flows throughout the trace. Ideally, unique flows should be evenly distributed across

Table 5.1: Coflowiness levels and their corresponding number of unique flows [54].

Coflowiness Level	Number of unique flows
0.0	108,566
0.1	108,566
0.25	128,294
0.5	176,918
0.75	197,910
0.9	193,001
1.0	193,001

the entire duration of each test, regardless of the coflowiness configurations. Such uniform distribution ensures consistent realization of caching benefits, leading to a progressive reduction in latency as coflowiness increases. However, when establishing the traces with varied coflowiness configurations, this factor was not taken into account, nor was the distribution of unique flows studied. The distribution of unique flows may vary among configurations. For example, if all unique flows are concentrated early in the trace, OVS can perform necessary upcalls over a brief period, subsequently benefiting from having all flows in the megaflow cache for the remainder of the benchmark. Conversely, if unique flows are spread throughout the test duration, the cache will not be maximally utilized at any specific time, necessitating continuous upcalls throughout the benchmark. Analysis of Figure 4.7a indicates that unique flows are relatively evenly distributed across all coflowiness configurations, as evidenced by a consistent decrease in the number of upcalls as the benchmark progresses. However, this observation does not confirm that the distributions are identical across all configurations, nor does it conclusively determine that these distributions do not impact the latency anomalies observed [54].

5.1.3 Flow Dynamics and Resource Efficiency

In analyzing the performance and resource utilization of OVS, the comparison between the FB Hadoop and GSRPC traces offers significant insights due to their contrasting traffic characteristics. The FB Hadoop trace is characterized by larger, longer-lived flows, which result in a lower flow arrival rate. This contrasts sharply with the GSRPC trace that comprises smaller, short-lived flows with a bursty arrival pattern, leading to a higher and more erratic flow

arrival rate.

With the FB Hadoop trace, once a flow is established and cached in the kernel datapath, subsequent packets of the flow are processed efficiently, minimizing the need for additional upcalls. This caching mechanism effectively reduces upcall frequency over time as more flows become cached. Furthermore, the presence of a high coflowiness level enhances this effect. It leads to reduced initial upcall frequency and lower CPU utilization in system processes, software interrupts, and overall CPU load, as depicted in Figures 4.4a, 4.4c, and 4.4e. Over time, the system encounters fewer new flows, which amplifies the reduction in upcall frequency and, consequently, CPU demand. This trend is further evidenced by the decreasing CPU utilization in handler threads in both user and kernel spaces, with significant reductions as coflowiness increases (Figures 4.5a and 4.5c).

In contrast, the GSRPC trace presents a scenario where the system continuously encounters new flows due to their short-lived nature, maintaining a high frequency of upcalls and software interrupts throughout. Even with increasing coflowiness, the rapid introduction of new flows largely negates the benefits of preloading flows. Consequently, the CPU utilization remains elevated across most coflowiness levels, indicating that flow prediction is less effective in reducing CPU load in such environments, as shown in Figures 4.4b, 4.4d, and 4.4f.

5.1.3.1 Batch processing of upcalls

The upcall statistics further elucidate the relationship between flow rate and CPU load. In the FB Hadoop benchmark (Figure 4.7a), the number of upcalls per second decreases over time for non-optimal coflowiness levels, aligning with the reduction in CPU utilization. This decrease is due to the effective caching of flows, which reduces the need for upcalls as the benchmark progresses. Notably, the upcalls in the FB Hadoop benchmark commence immediately, indicating that packets are processed individually or in small batches. In contrast, the GSRPC benchmark (Figure 4.7b) shows a consistently high number of upcalls per second across coflowiness levels, with a minimal reduction over time. The peak in upcalls occurs slightly after the benchmark starts, which shows that OVS attempts to handle the high flow rate through batch processing to use CPU resources more efficiently.

The analysis revealing that OVS processes the GSRPC packets in batches is also notable in the CPU utilization patterns observed in the OVS handler threads. Notably, the total CPU usage of these threads is lower when

processing the GSRPC trace than when handling the FB Hadoop trace due to larger batches. This reduces CPU utilization per upcall in the handler threads but introduces higher latency as packets wait for batch processing.

5.1.3.2 Daemons resource utilization and throughput

The noticeably higher daemon load observed in GSRPC, as compared to FB Hadoop, may stem from several factors.

First, since the GSRPC trace consists of smaller packets it leads to a higher pps rate and consequently, a reduced queue depth. Smaller packets occupy less buffer space, allowing the buffer to accommodate more packets before reaching capacity. As a result, under identical traffic loads, the queue depth, measured in bytes, is smaller with GSRPC's packet sizes since the buffers are not filled as rapidly. Additionally, smaller packets can be processed and transmitted more swiftly than larger ones, as they require less transmission time over the network medium. This facilitates faster buffer dequeuing, thereby diminishing the queue depth because packets spend less time waiting in the queue. The need to handle more packets increases CPU time spent in kernel mode, processing network stack operations, which in turn leads to higher CPU utilization. In contrast, the larger packets used in the FB Hadoop trace allow the application to handle larger blocks of data simultaneously. This reduces the frequency of context switches between the application and the operating system. Consequently, the data loaded into the CPU caches is utilized more effectively before replacement, thereby enhancing overall efficiency.

Another factor that may affect daemon load is the processing of the GSRPC trace by OVS in larger batches, which results in the installation of flow entries that enable OVS to forward buffered packets in batches. Consequently, despite a consistent entry rate of 1 million pps for both GSRPC and FB Hadoop traces, packets may exit OVS and arrive at the pod daemons in bursts in the case of GSRPC. This bursty behavior causes more packets to arrive at the pod daemons within shorter time intervals compared to the more uniform packet arrival pattern observed with FB Hadoop. However, it is important to note that, theoretically, packet batching should only occur for the first packet of flows that are not present in the cache. This means that batching should decrease as coflowiness increases. Bursty arrival of packets at the pod daemons would result in the following implications:

1. **Increased Instantaneous Processing Load**: During bursts, the pod daemons must handle a larger volume of packets in a brief period, which

leads to spikes in CPU utilization.

2. **Resource Contention**: The sudden surge in processing demand can cause contention for CPU resources, potentially resulting in inefficiencies such as increased context switching and cache misses within the CPU architecture.

3. **Higher Average CPU Usage**: These bursts contribute to elevated average CPU utilization as the pod daemons experience fluctuating, yet frequently peak, processing demands.

In contrast, the FB Hadoop benchmark exhibits less burstiness due to larger average flow sizes and more predictable traffic patterns. This allows OVS to cache flows more efficiently, resulting in:

- **Reduced Batch Processing**: Fewer cache misses mean that OVS processes smaller batches of packets, which promotes a more consistent packet forwarding rate.

- **Steady Packet Arrival at Pod Daemons**: The consistent flow of packets enables more efficient CPU utilization and reduces the instantaneous processing load on pod daemons.

These differences are evident in Figures 4.6e and 4.6f, where pod daemons show significantly higher CPU utilization in the GSRPC benchmark, requiring approximately 90% more CPU resources compared to the FB Hadoop benchmark. Furthermore, while the total CPU load for the GSRPC benchmark consistently reaches about 100% utilization, the FB Hadoop benchmark shows a marked decrease in CPU utilization with increased coflowiness, dropping to around 20% at a coflowiness level of 0.9. This represents a CPU utilization ~$5\times$ lower than that observed with GSRPC. However, the increase in daemon load for GSRPC is only ~90% higher compared with FB Hadoop. For future research, the significant difference in CPU utilization between these benchmarks could be easily verified using the `perf` tool on one of the daemons. Additionally, `perf` might indicate whether the discrepancies in utilization stem from issues within OVS, the kernel, or the daemon itself, potentially exacerbated by our use of Python.

This disparity highlights that while the daemons contribute significantly to the increased CPU load in GSRPC, they are not the sole contributors. The elevated CPU usage by the daemons accounts for only a portion of the total CPU load difference between the two benchmarks. Therefore, while the

daemon load is higher for GSRPC and increases the CPU load, it does not fully account for the entire difference in CPU load between FB Hadoop and GSRPC.

5.1.3.3 Bottlenecks and Trade-offs

The primary bottleneck in OVS CPU utilization, as revealed by the benchmarks, is the system's capacity to handle high flow rates with frequent cache misses, leading to excessive upcalls and elevated CPU utilization. The way OVS handles these upcalls, either individually or in batches, significantly affects CPU utilization and packet latency.

For the FB Hadoop trace, the bottleneck is mitigated by effective flow caching and prediction. The larger, long-lived flows allow the cache to remain relevant for a larger set of packets, enhancing the benefits of flow prediction and caching mechanisms. Additionally, the handler threads exhibit higher CPU utilization due to processing upcalls individually or in smaller batches. This leads to higher per-upcall CPU overhead but maintains lower end-to-end packet latency since packets are processed and forwarded promptly upon upcall handling. In the GSRPC traffic scenario, a persistent bottleneck arises from the intrinsic characteristics of the traffic. Due to the prevalence of small, short-lived flows, the cache benefits only one or two packets. Furthermore, the bursty nature of the traffic continuously introduces new flows, overwhelming the `ovs-vswitchd`. Although batch processing of upcalls in GSRPC reduces CPU utilization by grouping multiple upcalls, this method also increases packet latency. This delay occurs because packets must wait until the entire batch is processed, despite the improvements in CPU cache utilization and reductions in context switching overhead.

Lastly, despite lower CPU utilization in the handler threads due to batching, the overall CPU utilization remains high due to the high rate of new flows and the processing required in other system components (e.g., system processes and software interrupts). The batch processing does not sufficiently reduce the total CPU demand caused by the high volume of upcalls.

5.1.4 Flow rate implications on upcalls

The analysis of latency and upcall results within this research reveals a critical insight: it is the flow rate, not merely the distribution of flow sizes, that significantly influences the performance of OVS in terms of upcall frequency and latency.

Flow rate and FSD, though related, are distinct metrics. FSD characterizes the lengths of flows, quantified either by the number of packets or bytes per flow. Conversely, flow rate measures the number of new flow entries OVS must manage each second, directly affecting the frequency of upcalls and the overall processing burden on the switch. For instance, during the benchmarking exercise using the GSRPC trace, a high flow rate was observed of 236,007 flows within ~5 seconds. This contrasts with the FB Hadoop trace, which distributed the same number of flows across a longer duration of ~200 seconds. This discrepancy arises because the larger flows of the FB Hadoop trace take considerably longer to transmit at the same rate. The accelerated flow rate in the GSRPC trace significantly increased the number of upcalls and consequently elevated latency.

The sequence triggered by a packet from a new flow arriving at OVS is as follows: initially, a flow table miss occurs when the packet finds no matching entries in the datapath flow table in kernel space. This miss generates an upcall where the packet is then sent to the user space daemon `ovs-vswitchd` for further processing. Here, flow entry installation takes place as `ovs-vswitchd` processes the packet and decides on the necessary actions, and installs a new flow entry back in the kernel flow table. Once established, subsequent packets that match this new flow are processed in kernel space, eliminating the need for further upcalls. However, with a high flow rate, such as seen with the GSRPC trace, OVS is bombarded with new flows more frequently than it can efficiently manage, introducing various technical challenges.

These challenges include a CPU bottleneck in user space where the daemon becomes swamped with upcall processing, leading to higher CPU usage and potential delays in queueing upcalls. This bottleneck delays flow entry installation, during which time additional packets from the same flow may arrive, each generating a separate upcall if the flow entry is not yet established, compounding the processing load. The latency implications of these high upcall rates are substantial. The initial packets of new flows suffer from increased latency due to prolonged upcall processing times. Moreover, a backlog of upcalls can accumulate in the user space daemon's queue, introducing further latency through queuing delays.

The user space daemon's handling of upcalls can effectively be modeled using a queuing system framework. In this model, the arrival rates (λ) represent the flow rate, and the service rates (μ) correspond to the daemon's processing capacity. For the system to remain stable, meaning it avoids unbounded queuing delays, the condition $\lambda < \mu$ must be consistently met.

When flow rates are high, there is a risk of λ approaching or even exceeding μ, which can lead to increased latency and the potential for system overload. Notably, if the flow rates λ_1 and λ_2 for both traces were equal, the latency profiles would likely converge, assuming similar network conditions. This supports the conclusion that flow rate, rather than FSD, is the dominant factor affecting upcall frequency and latency in OVS.

5.1.5 Unexpected Upcall Counts and TCP Behavior

An interesting aspect of the results is that the total number of upcalls exceeded the observed number of flows (193,007). This suggests that multiple packets from the same flow caused upcalls before the first packet was fully processed and the flow entry was established in the cache. We utilized UDP packets in our OVS benchmarks. However, had the trace consisted of Transmission Control Protocol (TCP) sessions instead, the implications could have been markedly different. In typical TCP sessions, the sender transmits an initial Synchronize (SYN) packet to establish a connection and waits for an Acknowledgement (ACK) before sending additional data. Notably, no further data packets are sent until the SYN packet has been acknowledged, preventing any additional packets from arriving during this initial exchange. This would result in a single upcall packet (in each direction) for each TCP session, rather than a series of upcalls, as seen with UDP packets in our environment. This behavior would reduce the load on OVS, as only one initial packet would require processing to establish the flow entry in the cache, subsequently minimizing redundant upcalls.

With TCP Fast Open, the protocol allows data to be sent along with the initial SYN packet, potentially reducing connection setup time. This behavior could result in multiple packets arriving in OVS before the flow has been fully established, leading to multiple upcalls from the same flow. Such early transmission might complicate flow management and degrade performance as with UDP, particularly under high traffic conditions or when OVS is already under heavy load. However, TCP Fast Open might not always function in this manner, as it requires the exchange of a cookie before data transmission can begin. This initial exchange could delay the transmission of additional packets, thereby reducing the likelihood of multiple upcalls before the establishment of a flow entry in the cache.

5.2 Limitations

Our research possesses some limitations in terms of benchmarking methodology and implementation, these weaknesses and their impact are discussed in this section.

5.2.1 Synthetic traffic trace

A primary limitation of this research is its dependence on synthetic coflow traffic traces, which were generated using the extended workload tool outlined in Section 3.3. This reliance introduces several challenges to the accuracy and representativeness of our benchmarks.

Firstly, the synthetic traces, though comprising coflows, do not accurately mirror the complex nature of real-world coflow traffic observed in OCP deployments. The differences are notable in the structure of coflows, including the distribution of base and associated flows, as well as the quantity of coflows and flows per coflow. Furthermore, the experimental setup in our simulated OCP cluster diverges from likely real-world configurations. For instance, each pod in our cluster was equipped with nine open network ports, a figure that could vary significantly in actual deployments. Our simulated environment comprised a single-node with eight pods, whereas a typical operational OCP cluster might feature a larger array of nodes and pods, thus increasing the complexity of OpenFlow rules in OVS and the overall traffic trace complexity. Another critical aspect is the FSD within the traffic traces. Although the flow sizes in our benchmarks were derived from real-world data sets, they may not reflect the flow characteristics typical of an OCP cluster with containerized workloads. Real-world OCP clusters often handle frequent, lightweight service requests involving brief transactions and minimal data per query. The expected throughput may be characterized by high-demand bursts from individual microservices that do not necessitate sustained high bandwidth. Consequently, traffic patterns in an actual OCP deployment could consist predominantly of small, sporadic flows rather than extensive and voluminous flows. Thus, our usage of the GSRPC trace could, in some respects, be closer to what one might expect in a real-world OCP scenario than the FB Hadoop trace. Finally, the network protocols employed also affect the realism of our benchmarks. A production OCP cluster is more likely to utilize TCP sessions or QUIC, rather than UDP, altering the fundamental nature of the traffic patterns observed [54].

Further improvements to our extended workload generator could po-

tentially address several identifiable limitations. These limitations pertain primarily to the number of unique flows and the distribution of those flows within the trace, which may have contributed to the anomalies discussed in Section 5.1.2. The variation in flow numbers across different coflow configurations originated partly from an oversight during the development of the workload generator, in addition to other constraints imposed on the trace, such as the limit of 200,000 flows. This particular issue, as elaborated in Section 5.1.2.1, may have influenced the anomalies observed in the GSRPC benchmark results. Additionally, the final concern related to our benchmarks involves the distribution of these unique flows within the traces across various coflow configurations. Ideally, this distribution should have been uniform throughout the trace. However, as indicated in Section 5.1.2.2, deviations from this uniformity may have been responsible for the anomalies depicted in Figures 4.1e, 4.1d, 4.2d, 4.2f, and 4.3d.

5.2.2 OVN-Simulated OCP cluster

As briefly discussed in the previous section regarding the impact of using a simulated OCP cluster on the traffic trace, the simulated cluster brings additional considerations, covered below. While the simulation aims to replicate the traffic flow of a real-world OCP deployment, particularly in terms of which flows it produces when traffic hits the ingress point, quantifying the accuracy of this emulation presents several challenges. Below, we discuss these considerations, which likely impact the fidelity of our simulation:

- **Upcall Processing:** In genuine OCP environments, the rate of packets missing the kernel flow table varies widely due to diverse and unpredictable real-world traffic patterns. Our simulated environment may not accurately capture these rates, potentially leading to an underestimation or overestimation of upcall processing overhead. Production deployments also involve complex upcall processes, including security checks and policy enforcement, which our simulations might oversimplify. This could result in lower observed CPU utilization and a misrepresented performance impact of upcalls.

- **Flow Table Complexity:** Real-world OCP deployments typically feature intricate flow tables with numerous entries, wildcard entries, and group tables to manage diverse networking scenarios. Our simulation likely uses a smaller, more simplified flow table, which could mean fewer lookups and anomalously fast packet processing compared to

actual conditions. Moreover, real OCP environments dynamically adjust flows based on various factors such as application deployment, scaling, and network policies not fully replicated in our simulations.

- **Resource Contention:** In production, OVS competes for CPU resources with other processes and containers, possibly leading to performance degradation. Simulations may not account for such contention, presenting an overly optimistic view of CPU availability and utilization.

- **Traffic Pattern Complexity:** Our synthetic traffic likely does not capture the diversity of real traffic patterns, which include bursts, microbursts, and long-lived flows, along with factors like packet drops, retransmissions, and network congestion.

- **Hardware Optimization:** Unlike real-world deployments that might utilize hardware accelerations like DPDK or Single-root input/output virtualization (SR-IOV), our simulated setup does not incorporate these technologies. This omission could lead to notable discrepancies in CPU utilization and throughput measurements.

- **Software Interactions:** In actual OCP deployments, OVS interacts with Kubernetes and other control plane components, managing tasks like network policy enforcement and service discovery. Such interactions, which introduce additional overhead, are absent in our simulated environment.

Finally, as highlighted earlier, our single-node, 8-pod OCP cluster does not represent the scale of real-world deployments. This limitation impacts the size of the flow table and the frequency of upcalls, creating less stressful conditions for OVS. Additionally, our cluster does not run containers on the pods, which further skews the benchmarking in terms of CPU utilization and traffic complexity.

5.2.3 Throughput constraints

As discussed in Section 3.4.2.3, our benchmarks were conducted with a throughput of 12 Gbps and a packet rate of 1 million pps, despite the system's installed capacity of 100 Gbps. This limitation was primarily due to system resource constraints and the hard-coded buffer ring size in OVS. Operating at higher throughput levels led to packet drops at ingress within our

benchmarking setup. Conversations with experts involved in real-world OVS deployments suggest that throughput can reach up to 50 Gbps or even 100 Gbps when DPDK or hardware offload is enabled, demonstrating a significant variance from our test conditions. Running benchmarks at only 12 Gbps likely resulted in lower measured CPU utilization and reduced latency compared to configurations with higher throughput. Consequently, this disparity may lead to skewed results that do not accurately represent the performance observed in production environments. Moreover, it is important to note that real-world production environments seldom experience zero packet drops. This requirement imposed unrealistic constraints on our testing setup, potentially affecting the applicability of our findings to practical scenarios.

5.2.4 Flow Prediction Overhead and Challenges

While the results indicate substantial performance improvements, particularly in terms of reduced latency and CPU utilization, it is important to acknowledge that these findings are predicated on idealized assumptions that exclude several practical overheads associated with flow prediction and cache preloading. The omission of these factors offers a view that, while instructive for theoretical limits, may diverge significantly from real-world performance. This section discusses the implications of our research not accounting for such overheads in detail.

- **Computational Overhead of Flow Prediction**: The utilization of complex algorithms for the prediction of associate flows most likely presents a substantial computational challenge. These algorithms, which may involve advanced machine learning techniques or pattern recognition, require significant CPU resources. Under the high-throughput or bursty traffic conditions characteristic of the GSRPC trace, the real-time execution of these algorithms can induce considerable latency and elevate CPU utilization, potentially diminishing the performance gains observed.

- **Latency Due to Flow Prediction and Installation**: The processes of predicting flows and loading them into the datapath cache are non-trivial and introduce additional latency. This latency arises from the time taken to execute prediction algorithms and the subsequent interaction with the OVS control plane for flow installation. Such delays are critical yet unaccounted for in the obtained latency results, potentially offsetting the

observed benefits of reduced upcalls and enhanced packet processing latency.

- **Increased Control Plane Load**: The interaction required between the predicted flow data and the `ovs-vswitchd` daemon is significant. Especially under conditions of high coflowiness, the rapid and bulk installation of flows can lead to increased CPU utilization and create bottlenecks at the control plane, a factor not considered in our benchmarks.

- **Impact on CPU Utilization Metrics**: The metrics on CPU utilization in this study do not reflect the additional processing demands of flow prediction and cache management. These tasks are likely to increase CPU demands substantially, affecting the performance improvements reported and altering the efficiency of the OVS handler threads responsible for flow handling.

- **Variability of Real-world Traffic**: The predictability of network traffic in production environments is inherently uncertain. Flows may commence or terminate unexpectedly, deviating from historical patterns and complicating the accuracy of flow predictions. This variability can lead to wasted computational resources and negatively impact overall system performance.

- **Synchronization and Timing Challenges**: Ensuring precise synchronization between the timing of flow prediction, cache loading, and packet arrivals is critical to avoid cache misses and unnecessary upcalls resulting in wasted CPU resources. Delays or misalignments in this process can introduce additional overheads that our benchmarking methodology does not account for.

In conclusion, while the benchmark results provide valuable insights into the theoretical capabilities of OVS when predicting future flows under ideal conditions, it is important to acknowledge the limitations of this research. The practical implementation of such a system in real-world scenarios would entail significant overheads and presents several challenges. Ensuring that flow predictions are beneficial rather than merely consuming additional CPU resources and increasing latency are critical issues to overcome. Our research does not address these practical challenges, which remains a limitation of our work; however, future research may investigate these aspects to build upon and strengthen the applicability of our findings.

5.3 Sustainability and Ethics

Our research demonstrates that accurately predicting future flows can significantly reduce latency in OVS. This enhancement potentially improves the accessibility of network services relying on OVS, particularly in areas with underdeveloped Internet infrastructure. Enabling more users to access demanding Internet services aligns with sustainability goals by promoting equitable digital access. Additionally, our findings suggest that flow prediction can reduce CPU resource usage, thereby lowering energy consumption. This supports broader sustainability objectives [4]. However, it remains to be determined whether the benefits of flow prediction outweigh the additional processing power required for predicting and loading flows. While the flow prediction system consumes energy, the net impact on CPU resource utilization requires further investigation.

From an ethical perspective, our research uncovers potential vulnerabilities. The results indicate that sending bursty traffic traces can significantly impact OVS performance by increasing latency and CPU utilization. This vulnerability could be exploited in Distributed Denial-of-Service (DDoS) attacks, which would slow down or disrupt services dependent on OVS. Such attacks typically involve flooding OVS with numerous new flows, each necessitating a flow entry in the OVS cache. The initial packet of each new flow triggers an upcall to user space for flow setup. By inundating OVS with new flows and strategically managing TCP settings, attackers can overwhelm the system, causing delays and potential service disruptions for legitimate traffic. This could lead to packet loss when OVS's processing capacity is exceeded, thereby denying service to legitimate users.

Despite these concerns, the ethical implications of exposing such vulnerabilities are relatively limited. Our findings underscore the importance of scalability and robustness in network systems. Exposing potential weaknesses through benchmarking is a standard practice in software development, generally welcomed by developers as it contributes to improvements in software reliability and security. As such, the potential reputational risks to developers and the reluctance of entities to adopt OVS are unfounded. Benchmarking is widely accepted and valued in the software community for enhancing system performance and security. In conclusion, while our research highlights both potential enhancements and vulnerabilities in OVS, it ultimately contributes to the advancement of network technologies by prompting further research and development to address these issues. This approach does not pose significant ethical concerns but rather fosters

an environment of continuous improvement and resilience in networking solutions.

Chapter 6

Conclusions and Future work

This chapter is divided into two sections. In Section 6.1, we summarize the research findings and conclusions drawn from the benchmarking of OVS. Finally, Section 6.2 outlines potential objectives for future research that build upon the objectives, methodology, and results of this study.

6.1 Conclusions

In modern networking architectures, OVS serves as an essential intermediary between the control and data planes. Often referred to as the *slow path*, OVS's performance is hindered significantly by cache miss bottlenecks. These bottlenecks cause prolonged latency as new, unknown flows require upcalls to the user space daemon. Recent research highlights the need for dedicated accelerators to address this issue, similar to advancements in data plane optimization [2]. This research focused on enhancing the slow path's efficiency by preloading future flows into the datapath cache, thereby minimizing the overhead associated with handling new flows that are not already present in the data plane.

This research has performed a set of benchmarks with the purpose of establishing an upper-bound performance limit for OVS. The benchmarking methodology involved deploying a single-node OVN-simulated OCP cluster, which inserted timestamps at the ingress and egress points of OVS to measure latency across different network traffic types and varying conditions. The results demonstrate that coflowiness significantly benefits OVS performance. In particular, the FB Hadoop trace showed a clear reduction in latency as larger flows benefited from increased cache hits. Conversely, the GSRPC trace exhibited a more gradual latency reduction, driven by the high flow rate. As

coflowiness values approached ~90-100%, where the majority of flows were correctly predicted, the latency profiles of both traces converged. However, latency anomalies were observed, likely due to the inconsistent number and distribution of unique flows across different coflowiness configurations. In terms of CPU utilization, the FB Hadoop trace displayed a consistent decrease in total resource consumption as coflowiness increased. In contrast, despite higher coflowiness, the rapid introduction of new flows in the GSRPC trace diminished the benefits of preloading, as OVS frequently encountered new, short-lived flows. This led to sustained upcalls and software interrupts. Batching in GSRPC reduced the load on OVS handler threads but resulted in increased latency. Meanwhile, overall CPU utilization remained high due to the continuous influx of new flows and the processing demands on other system components, such as system processes and interrupts. The findings confirm that flow rate, rather than just the distribution of flow sizes, plays a critical role in OVS performance, particularly regarding upcall frequency and latency [54].

6.2 Future work

In this section, we explore potential avenues for future research that are designed to address the limitations identified in this study, with the ultimate goal of developing a system that enables OVS to predict and preload flows into its data plane cache during runtime. This section is organized into two primary segments. Initially, we suggest minor enhancements that could readily increase the realism and accuracy of the research presented in this work. Subsequently, we outline more comprehensive extensions to this research topic that could significantly advance the integration of OVS with a real-world DSA for the slow path [2].

6.2.1 Enhancing realism and accuracy

A primary drawback identified in Section 5.2.1 concerns the traffic traces utilized. Enhancing realism and accuracy would therefore benefit from conducting experiments with coflowiness configurations that ensure a consistent number of unique flows per configuration. This adjustment is likely to eliminate the latency anomalies noted in Section 4.1 and 5.1.2. Additionally, future studies should strive for an even distribution of unique flows across traces to prevent skewed cache utilization impacts on the results. Moreover, incorporating real-world coflow traffic traces from operational

OCP clusters with actively running containers would provide a more accurate reflection of FSDs, flow counts per coflow, and levels of coflowiness. Regarding the simulation limitations discussed in Section 5.2.2, benchmarking on an actual OCP deployment, rather than using stand-alone OVN, would better address issues related to flow table complexity, CPU utilization, upcall processing, and resource contention across pods and nodes. Lastly, the current methodology involves preloading associate flows into the cache before benchmarking, which presents an unrealistic latency advantage absent in actual systems. In real-world scenarios, there would be a delay between flow prediction and cache loading, necessitating additional processing power and upcalls. Future research should thus develop a methodology where associate flows are loaded into the cache as related base flows hit the ingress of OVS. This approach would allow for an examination of the latency and CPU utilization required for loading associate flows during runtime, thereby yielding more realistic results.

6.2.2 Implementing flow prediction mechanism

The objective of this thesis is to lay the groundwork for developing a domain-specific flow prediction accelerator that operates within the slow path of OVS. Herein, we propose logical extensions to this initial research by conceptualizing and benchmarking such a system.

Flow prediction in OVS should capitalize on the structure and characteristics of coflows, which group multiple network flows with shared objectives. Temporal and spatial correlations among these flows suggest that associate flows frequently occur shortly after a base flow and share common endpoints, prefixes, or other header elements. This correlation can be harnessed by maintaining a database that records associations between base flows and subsequent associate flows, built by monitoring traffic and identifying recurrent coflow patterns. When a base flow arrives and is not found in the datapath cache the system consults this database to predict and preload potential associate flows, thereby optimizing flow handling while preserving the required non-overlapping properties for packet matching in OVS. For the predictive modeling necessary to support this system, several approaches can be considered:

- **Statistical Models**: Utilize Markov chains or Bayesian networks to calculate the likelihood of associate flows based on a base flow.
- **Machine Learning Models**: Employ models such as decision trees,

support vector machines, or neural networks to predict associate flows. These models can leverage the complex, nonlinear relationships evident in network traffic patterns.

To continuously refine the flow predictor, it is advantageous to employ both offline and online learning strategies. Predictive models can be trained offline using historical traffic data to capture base and associate flow relationships. Concurrently, these models can be updated in real-time with new traffic data during system operation to adapt to changing network conditions, optimize the prediction horizon, and balance accuracy with resource utilization.

The foundational research referenced in Section 2.7.3, "Scaling Open vSwitch with a Computational Cache" by Rashelbach *et al.*, introduces NMU, which restructures OVS's datapath to facilitate direct packet classification based on OpenFlow rules, eliminating the need for control-path upcalls. Building on this, future research could extend the NMU's computational cache to incorporate flow prediction using coflows. A flow predictor could employ shallow neural networks or hierarchical models like RQ-RMI to predict flows across different fields and granularity levels effectively. Such integration allows NMU to utilize its efficient packet classification capabilities to manage both observed and anticipated flows within the data plane, enhancing the rapid retraining features of NMU to accommodate new coflow patterns with minimal latency and memory overhead. Upon implementing a flow prediction system, it is imperative to benchmark the new setup and analyze the trade-offs between cache size and system performance to identify optimal configurations. Additionally, strategies need to be developed to allocate computational resources effectively between prediction tasks and regular packet processing tasks [3].

Given the intensive computational demands of advanced prediction algorithms, especially those involving deep learning or extensive statistical models, it may be necessary to leverage specialized hardware like Graphics Processing Units (GPUs) or FPGAs. These platforms excel in parallel computation, enhancing the speed of prediction tasks and reducing latency, which is crucial for real-time flow preloading. Moreover, specialized hardware can offer a more energy-efficient solution compared to general-purpose CPUs, an essential consideration for data center operations. In conclusion, if the development of a flow predictor for OVS proves beneficial by future research, it would be prudent for the OVS community to consider its integration into production deployments within data centers, IXPs, and OCP environments. This would not only enhance the performance of OVS but also contribute to its evolution and applicability in modern network infrastructures [2].

References

[1] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch." Oakland, CA: USENIX Association, 2015. ISBN 978-1-931971-21-8. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf (Accessed 2024-01-18) [Pages ix, 1, 2, 3, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 44, and 47]

[2] A. Zulfiqar, B. Pfaff, W. Tu, G. Antichi, and M. Shahbaz, "The Slow Path Needs an Accelerator Too!," *ACM SIGCOMM Computer Communication Review*, vol. 53, no. 1, pp. 38–47, Apr. 2023. doi: 10.1145/3594255.3594259. [Online]. Available: https://benpfaff.org/papers/slow-path.pdf (Accessed 2024-01-18) [Pages 2, 3, 15, 18, 20, 21, 41, 42, 43, 117, 118, and 120]

[3] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "Scaling Open vSwitch with a Computational Cache" Apr. 2022. ISBN 978-1-939133-27-4. [Online]. Available: https://www.usenix.org/system/files/nsdi22-paper-rashelbach.pdf (Accessed 2024-02-19) [Pages ix, 2, 3, 15, 18, 20, 21, 22, 26, 27, 47, 48, 49, 50, 51, and 120]

[4] M. Bexell and K. Jönsson, "Responsibility and the United Nations' Sustainable Development Goals" *Forum for Development Studies*, vol. 44, no. 1, Jan. 2017. doi: 10.1080/08039410.2016.1252424 Publisher: Routledge _eprint: https://doi.org/10.1080/08039410.2016.1252424. [Online]. Available: https://doi.org/10.1080/08039410.2016.1252424 (Accessed 2024-01-22) [Pages 4 and 114]

[5] A. Håkansson, "Portal of Research Methods and Methodologies for Research Projects and Degree Projects." CSREA Press U.S.A, 2013,

pp. 67–73. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960 (Accessed 2024-03-20) [Page 5]

[6] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software-Defined Networking: State of the Art and Research Challenges" *Computer Networks*, vol. 72, May 2014. doi: 10.1016/j.comnet.2014.07.004. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1406/1406.0124.pdf (Accessed 2024-01-18) [Pages 7, 8, 9, 10, 11, and 13]

[7] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities" in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018. doi: 10.1145/3230543.3230564. ISBN 978-1-4503-5567-4. [Online]. Available: https://people.csail.mit.edu/alizadeh/papers/homa-sigcomm18.pdf (Accessed 2024-02-13) [Pages 9, 67, and 68]

[8] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-Performance Networks" vol. 41, Toronto, Ontario, Canada, Aug. 2011. doi: 10.1145/2018436.2018466. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9822fef4193fbd10303917738b0fe9678cf04829 [Page 9]

[9] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture" Oct. 1999. [Online]. Available: https://www.ietf.org/rfc/rfc2722.txt (Accessed 2024-02-13) [Page 9]

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks" *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008. doi: 10.1145/1355734.1355746. [Online]. Available: http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf (Accessed 2024-01-18) [Page 9]

[11] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)" Oct. 2004. [Online]. Available: https://www.ietf.org/rfc/rfc3917.txt (Accessed 2024-02-13) [Page 9]

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise" *ACM*

SIGCOMM Computer Communication Review, vol. 37, no. 4, Aug. 2007. doi: 10.1145/1282427.1282382. [Online]. Available: https://doi.org/10.1145/1282427.1282382 (Accessed 2024-02-13) [Page 9]

[13] A. Greenberg, G. Hjálmtýsson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management" *ACM SIGCOMM Computer Communication Review*, vol. 35, Oct. 2005. doi: 10.1145/1096536.1096541 [Page 9]

[14] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE" *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, Aug. 2010. doi: 10.1145/1851275.1851224. [Online]. Available: https://doi.org/10.1145/1851275.1851224 (Accessed 2024-02-13) [Page 9]

[15] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm" Internet Engineering Task Force, Request for Comments RFC 2992, Nov. 2000, Num Pages: 8. [Online]. Available: https://datatracker.ietf.org/doc/rfc2992 (Accessed 2024-02-13) [Page 9]

[16] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. M. Jr, P. Papadimitratos, and M. Chiesa, "A {High-Speed} {Load-Balancer} Design with Guaranteed {Per-Connection-Consistency}" Feb. 2020. ISBN 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/system/files/nsdi20-paper-barbette.pdf (Accessed 2024-02-13) [Page 9]

[17] N. M. Mosharaf Kabir Chowdhury and R. Boutaba, "Network virtualization: state of the art and research challenges" *IEEE Communications Magazine*, vol. 47, no. 7, 2009. doi: 10.1109/MCOM.2009.5183468 Conference Name: IEEE Communications Magazine. [Online]. Available: https://www.mosharaf.com/wp-content/uploads/nv-overview-commag09.pdf (Accessed 2024-01-18) [Pages 10, 11, and 12]

[18] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: enabling innovation in network function control" *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, Aug. 2014. doi: 10.1145/2740070.2626313. [Online]. Available: https://doi.org/10.1145/2740070.2626313 (Accessed 2024-01-18) [Pages 11 and 12]

[19] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization" *Computer*, vol. 38, May 2005. doi: 10.1109/MC.2005.136. [Online]. Available: https://homes.cs.washington.edu/~tom/support/impasse.pdf [Page 12]

[20] A. Garg, V. Saini, M. Imran, and M. Qadeer, "Performance analysis of software defined networks." San Francisco, CA, USA: IEEE, Aug. 2013. doi: 10.1109/CICN.2017.8319356. ISBN 978-0-7695-5102-9. [Online]. Available: https://ieeexplore.ieee.org/document/6730793 [Page 13]

[21] E. Chaudron, "Investigating the cost of Open vSwitch upcalls in Linux" Feb. 2022, Section: Python. [Online]. Available: https://developers.redhat.com/articles/2022/02/07/investigating-cost-open-vswitch-upcalls-linux (Accessed 2024-04-24) [Pages 13 and 26]

[22] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks" vol. 10, Vancouver, BC, Canada, Jan. 2010, pp. 351–364. [Online]. Available: https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Koponen.pdf [Page 13]

[23] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. G. Feamster, J. L. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A software defined Internet exchange" in *SIGCOMM 2014 - Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*. Association for Computing Machinery, 2014. doi: 10.1145/2619239.2626300 pp. 551–562. [Online]. Available: https://collaborate.princeton.edu/en/publications/sdx-a-software-defined-internet-exchange-3 (Accessed 2024-01-18) [Page 14]

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN" vol. 43, Aug. 2013. doi: 10.1145/2486001.2486019 pp. 3–14. [Online]. Available: https://cseweb.ucsd.edu/~vahdat/papers/b4-sigcomm13.pdf [Page 14]

[25] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, "Revisiting the Open vSwitch dataplane ten years later" *Proceedings of the 2021 ACM SIGCOMM 2021*

Conference, Aug. 2021. doi: 10.1145/3452296.3472914 Conference Name: SIGCOMM '21: ACM SIGCOMM 2021 Conference ISBN: 9781450383837 Place: Virtual Event USA Publisher: ACM. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2021/files/papers/3452296.3472914.pdf (Accessed 2024-01-22) [Pages 18, 37, and 38]

[26] G. Sun, W. Li, and D. Wang, "Performance Evaluation of DPDK Open vSwitch with Parallelization Feature on Multi-Core Platform" *Journal of Communications*, pp. 685–690, 2018. doi: 10.12720/jcm.13.11.685-690. [Online]. Available: https://www.jocm.us/uploadfile/2018/1026/20181026023438938.pdf (Accessed 2024-01-18) [Pages ix and 23]

[27] E. Chaudron, "Open vSwitch: The revalidator process explained" Oct. 2022. [Online]. Available: https://developers.redhat.com/articles/2022/10/19/open-vswitch-revalidator-process-explained (Accessed 2024-01-22) [Pages 24, 25, and 26]

[28] R. Bryant, K. Mestery, and J. Pettit, "OVN: Open Virtual Network for Open vSwitch" Vancouver, BC, Canada, Nov. 2015. [Online]. Available: https://www.openvswitch.org/support/slides/OVN-Vancouver.pdf [Pages 27, 28, and 29]

[29] "Open Virtual Network (OVN) Red Hat OpenStack Platform 13 | Red Hat Customer Portal." [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html/networking_with_open_virtual_network/open_virtual_network_ovn (Accessed 2024-02-07) [Pages 27 and 29]

[30] "OVN, Open Virtual Network :: OVN project documentation website." [Online]. Available: https://www.ovn.org/en/ (Accessed 2024-02-07) [Pages 27, 28, and 29]

[31] S. Ferlin-Reiter, "Cloud Networking with OpenShift (and OVN)" Red Hat, Oct. 2023. [Pages 27, 29, 31, and 32]

[32] B. Pfaff, "GitHub - ovn-org/ovn: Open Virtual Network." [Online]. Available: https://github.com/ovn-org/ovn (Accessed 2024-02-07) [Page 29]

[33] "OVN-Kubernetes architecture - OVN-Kubernetes network plugin | Networking | OpenShift Container Platform 4.12." [Online]. Available: https://docs.openshift.com/container-platform/4.12/networking/ovn_ku

bernetes_network_provider/ovn-kubernetes-architecture-assembly.html#ovn-kubernetes-architecture-con (Accessed 2024-02-07) [Pages ix, 29, 30, and 32]

[34] "ovn-org/ovn-kubernetes" Feb. 2024, Original-date: 2016-08-11T17:51:02Z. [Online]. Available: https://github.com/ovn-org/ovn-kubernetes (Accessed 2024-02-07) [Pages 29 and 32]

[35] J. Fernandes, "Why Red Hat Chose Kubernetes for OpenShift" Nov. 2016. [Online]. Available: https://content.cloud.redhat.com/blog/red-hat-chose-kubernetes-openshift (Accessed 2024-02-07) [Page 31]

[36] I. Roth, "Announcing OpenShift: the Platform as a Service for developers who love open source and CDI" May 2011. [Online]. Available: https://web.archive.org/web/20190926015515/https://blog.openshift.com/announcing-openshift-the-platform-as-a-service-for-developers-who-love-open-source-and-cdi/ (Accessed 2024-02-07) [Page 31]

[37] "OpenShift Container Platform architecture." [Online]. Available: https://docs.openshift.com/container-platform/4.11/architecture/architecture.html (Accessed 2024-02-07) [Pages 31 and 32]

[38] "Official Red Hat OpenShift Documentation." [Online]. Available: https://docs.openshift.com/ (Accessed 2024-02-07) [Pages 31 and 32]

[39] "Red Hat OpenShift" Apr. 2020. [Online]. Available: https://developers.redhat.com/products/openshift/overview (Accessed 2024-02-07) [Pages 31 and 32]

[40] "Red Hat OpenShift Container Platform." [Online]. Available: https://www.redhat.com/en/resources/openshift-container-platform-datasheet (Accessed 2024-02-07) [Page 31]

[41] "Red Hat OpenShift Container Platform." [Online]. Available: https://www.redhat.com/en/technologies/cloud-computing/openshift/container-platform (Accessed 2024-02-07) [Page 31]

[42] M. Chowdhury and I. Stoica, "Coflow: a networking abstraction for cluster applications" in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: Association for Computing Machinery, Oct. 2012. doi: 10.1145/2390231.2390237.

ISBN 978-1-4503-1776-4. [Online]. Available: https://conferences.sigcomm.org/hotnets/2012/papers/hotnets12-final51.pdf (Accessed 2024-02-13) [Pages 32, 33, and 34]

[43] L. Shi, Y. Liu, J. Zhang, and T. Robertazzi, "Coflow Scheduling in Data Centers: Routing and Bandwidth Allocation" *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, Dec. 2018. doi: 10.1109/TPDS.2021.3068424. [Online]. Available: https://arxiv.org/pdf/1812.06898.pdf [Pages 33 and 35]

[44] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: near-optimal network design for coflows" in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018. doi: 10.1145/3230543.3230569. ISBN 978-1-4503-5567-4. [Online]. Available: https://www.cs.cornell.edu/~ragarwal/pubs/sincronia.pdf (Accessed 2024-02-13) [Pages 35, 36, 37, and 57]

[45] S. Agarwal, R. Agarwal, and S. Rajakrishnan, "Sincronia Repository" 2018. [Online]. Available: https://github.com/sincronia-coflow (Accessed 2024-02-13) [Pages 36, 37, and 57]

[46] ——, "Sincronia Workload Generator for Coflows" 2018. [Online]. Available: https://github.com/sincronia-coflow/workload-generator (Accessed 2024-02-13) [Pages 36, 37, and 57]

[47] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: fast programmable packet processing in the operating system kernel" in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, Dec. 2018. doi: 10.1145/3281411.3281443. ISBN 978-1-4503-6080-7. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3281411.3281443 (Accessed 2024-03-16) [Pages ix, 37, 38, 39, and 40]

[48] B. Kataria, R. M P, L. Monis, M. P. Tahiliani, and K. Makhijani, "Programmable Data Plane for New IP using eXpress Data Path (XDP) in Linux" in *2022 IEEE 23rd International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2022.

doi: 10.1109/HPSR54439.2022.9831409 ISSN: 2325-5609. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9831409 (Accessed 2024-03-16) [Page 38]

[49] "Understanding the eBPF networking features in RHEL 9 Red Hat Enterprise Linux 9 | Red Hat Customer Portal." [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel-9_configuring-and-managing-networking (Accessed 2024-03-16) [Page 38]

[50] "Using the eXpress Data Path (XDP) in Red Hat Enterprise Linux 8." [Online]. Available: https://www.redhat.com/en/blog/using-express-data-path-xdp-red-hat-enterprise-linux-8 (Accessed 2024-03-16) [Page 39]

[51] L. Yu, J. Sonchack, and V. Liu, "Mantis: Reactive Programmable Switches" in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020. doi: 10.1145/3387514.3405870. ISBN 978-1-4503-7955-7. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405870 (Accessed 2024-02-16) [Page 41]

[52] Y. Wang, D. Li, Y. Lu, J. Wu, H. Shao, and Y. Wang, "Elixir: A High-performance and Low-cost Approach to Managing {Hardware/Software} Hybrid Flow Tables Considering Flow Burstiness." Tsinghua University, Apr. 2022. ISBN 978-1-939133-27-4. [Online]. Available: https://www.usenix.org/system/files/nsdi22-paper-wang_yanshu.pdf (Accessed 2024-02-19) [Pages 44, 45, and 46]

[53] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing" May 2015. doi: 10.1109/ANCS.2015.7110116. ISBN 978-1-4673-6633-5. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-233388 (Accessed 2024-03-04) [Pages 54 and 63]

[54] E. Ståhl, "emilstahl97/Open-vSwitch-with-Coflow-Aware-Prediction" Stockholm, Sweden, Nov. 2024, Original-date: 2024-10-03T15:55:04Z. [Online]. Available: https://github.com/emilstahl97/Open-vSwitch-with-Coflow-Aware-Prediction (Accessed 2024-11-14) [Pages xi, 55, 56, 57, 59, 60, 62, 63, 64, 65, 67, 68, 69, 71, 100, 101, 102, 109, and 118]

[55] “Scapy.” [Online]. Available: https://scapy.net/ (Accessed 2024-09-18) [Page 62]

[56] “Introduction — Scapy 2.6.0 documentation.” [Online]. Available: https://scapy.readthedocs.io/en/latest/introduction.html (Accessed 2024-09-18) [Page 62]

[57] “secdev/scapy” Sep. 2024, Original-date: 2015-10-01T17:06:46Z. [Online]. Available: https://github.com/secdev/scapy (Accessed 2024-09-18) [Page 62]

TRITA-EECS-EX-2024:917
Stockholm, Sweden 2024