Enabling traffic prediction in virtual switching: A case study

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.

We propose a solution that can accelerate the slow path by predicting upcoming flows and preemptively installing the corresponding rules in the data plane. More concretely, our approach enhances packet processing latency and the overall system performance by populating the data plane ahead of time. Using a workload containing different levels of so-called "coflows", our experimental results demonstrate that the effectiveness of this approach varies with the flow rate of traffic traces. Notably, a system that could even just predict 25% of the traffic flows would decrease the average latency by up to ~24% and reduce CPU utilization by ~12%.

Keywords

Open vSwitch, Software-defined networking, Slow path, Coflows

1 Introduction

SDN has revolutionized the way network traffic is managed, by separating the data plane, which is responsible for forwarding and handling data packets, from the control plane, which makes network decisions and routing policies. This separation is integral to the architecture of modern data centers, where the control plane's computational overhead is offloaded to a general-purpose CPU that determines flow rules, which are then installed and applied to data packets in the data plane [7, 12]. This traditional SDN model emphasizes efficiency and scalability, but when put into practice, it has revealed a critical intermediary: The *slow path*.

The slow path is an entity that resides on the data plane device of SDN devices that install data plane forwarding rules in a *reactive* manner, *i.e.*, only when the first packet of a flow arrives at the device. The slow path is responsible for (1) processing packets of flows that have not been handled by the fast data path and (2) installing rules in the fast data path of the device for handling these flows. Open Virtual Switch (OVS) is a common data plane virtual switch that installs flow rules reactively and has been widely adopted due to its flexibility, scalability, and ability to manage network traffic in dynamic environments. It supports key networking features such as virtual networking and flow control, making it ideal for orchestrating containerized applications [7, 10]. An OVS switch can be controlled by an SDN controller through a well-defined interface, *e.g.*, OpenFlow. Its data plane consists of a fast path, *e.g.*, a high-throughput 5-tuple classifier cache on a NIC or in the kernel, and a slow path, consisting of different types off *caches* holding rules for those active flows that have not yet been installed in the fast path. The slow path typically resides in the user space logic of virtual switches whereas the fast path resides in kernel space.

Optimizing OVS is crucial for advancing SDN. In OVS, the *slow path* periodically validates the entries installed in the data plane caches against current OpenFlow rules to ensure their accuracy, a process that must be smoothly completed to maintain network performance. However, as network bandwidth expands and topologies grow in complexity, the slow path has emerged as a significant bottleneck, particularly when flows arrive and the data plane does not have any rules installed to handle these flows [8].

Our research in this paper suggests a novel flow prediction mechanism that proactively loads network flow information into the datapath cache, thus, predicting future network flows. More concretely, our paper contributions are:

- Exploring prediction techniques to proactively manage and enhance the performance of the OVS cache, reducing latency from cache misses.
- Conducting extensive performance benchmarks of OVS in a production-like container management platform across diverse traffic scenarios to show its capabilities and limitations.
- Developing a benchmarking framework that can be applied by future research to evaluate and refine the performance of OVS under varying conditions.

In the following, Section 2 presents the background and motivation for our work. Section 3 summarizes the research about the *slow path*. Then, Section 4 details our experimental setup and methodology while Section 5 discusses our evaluation results. Finally, Sections 6 and 7 comment on the future work and our conclusions.

2 Background & Motivation

We now describe two essential SDN components relevant to our work: OVS in Section 2.1, and OpenShift Container Platform (OCP) in Section 2.2. Finally, in Section 2.3, we describe the definition of a *coflow* adopted throughout the paper.

2.1 Open vSwitch

OVS operates primarily at the slow path in virtualized environments, especially within SDN. OVS uses a split architecture, where the fast path resides in the Linux kernel for high-speed packet processing, and the control path operates in user space, managing more complex operations such as flow setup and policy enforcement. A key component of this design is the use of so-called *upcalls*, which occur when the kernel cannot match an incoming packet to an existing flow rule, triggering a request to the control path (via the `ovs-vswitchd`) for further processing, see Figure 1[1], which illustrates the OVS components and interfaces [7, 10].

upcalls are particularly costly in terms of latency and resource consumption. Each time an *upcall* is triggered, packets transition

[1]Zooming into `ovs-vswitchd`'s behavior in Figure 1, this is where the interactions with the megaflow cache, hosted in kernel space, happens [8].

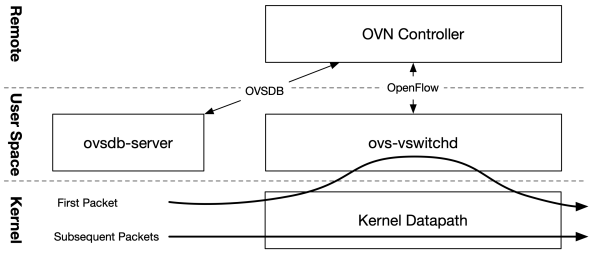


Figure 1: Components and interfaces of OVS including the kernel data path in charge of forwarding packets.

from the kernel to user space, where more extensive packet classification takes place. This process adds considerably to the delay, as packets must wait for the user space component to install a flow rule before they can be forwarded. As network traffic increases, the rate of *upcalls* grows, exacerbating the performance impact, as the overhead from kernel-to-user space transitions increases. Moreover, handling these *upcalls* greatly burdens the `ovs-vswitchd` user space daemon, consuming significant CPU and memory resources, especially in high-throughput environments [8, 12].

To minimize the need for *upcalls*, OVS implements two types of caching mechanisms in the kernel data path: The microflow cache and the megaflow cache. The microflow cache is designed for exact flow matches, ensuring that once the control path has processed a flow, subsequent packets belonging to the same flow can be forwarded directly without triggering another upcall. This significantly reduces latency for sustained flows, as packets can bypass expensive classification processes after the initial packet. However, for bursty traffic patterns characterized by frequent new or short-lived flows, such as port scans or peer-to-peer applications, the microflow cache provides limited benefit, as cache misses occur more frequently, leading to repeated *upcalls* [7].

The megaflow cache offers a more general solution by storing flow entries that can match a wider variety of traffic patterns, reducing the need for *upcalls* in more dynamic environments. Although megaflow caches require multiple hash lookups per packet, they mitigate the performance impact of short-lived flows by covering a broader range of traffic. The combination of microflow and megaflow caching balances efficiency and flexibility, helping to reduce the reliance on the slow path by minimizing *upcalls* frequency [7, 10]. Despite these optimizations, *upcalls* continue to be a major bottleneck in OVS, especially as traffic scales or becomes more dynamic [8, 11]. Cache misses in both microflow and megaflow caches inevitably result in *upcalls*, introducing latency and consuming resources. Furthermore, as the number of flow rules increases, the size and efficiency of the caches are constrained, leading to more frequent invalidations and a higher chance of cache misses. This, in turn, elevates the rate of *upcalls*, causing performance degradation as the user space control path becomes overwhelmed [8].

2.2 Openshift Container Platform

The OCP, developed by Red Hat, leverages Kubernetes to orchestrate containerized applications across hybrid and multi-cloud environments. Within OCP, OVS is deployed to manage network traffic

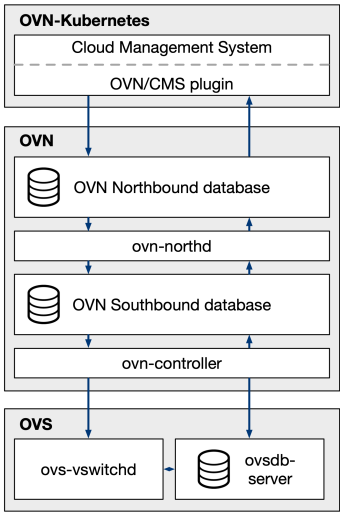


Figure 2: OVN-Kubernetes integration in an OCP cluster: Interaction between cloud management systems, OVN databases, and OVS components.

across Kubernetes clusters, with network virtualization facilitated by the Open Virtual Networking (OVN), integrated as the Container Network Interface (CNI) through OVN-Kubernetes (OVN-K). This framework supports secure communication and isolation among pods. The traffic characteristics of containerized applications inherently include many small and short-lived flows, leading to bursty traffic patterns [4, 9]. This burstiness significantly loads the slow path of the OCP network infrastructure, requiring more software forwarding resources. During peak rate periods, bursty flows can cause a notable increase in queue size and end-to-end latency [11].

2.3 Coflows

Cluster computing environments, essential for data-intensive applications often struggle with optimizing communication patterns. This inefficiency stems from traditional networking approaches that fail to manage complex, concurrent data flows, leading to suboptimal performance and resource utilization [1]. Addressing these challenges, Chowdhury *et al.* introduced the concept of *coflows* a networking abstraction designed to enhance scheduling and resource allocation by recognizing the collective communication patterns of applications [3]. A *coflow* is a semantically related collection of flows between two groups of machines. Mathematically, a coflow c from a source group S to a destination group D can be written as:

$$c(S, D) = \{f_1, f_2, \ldots, f_{|c|}\}$$

where $|c|$ is the number of flows, each flow f_i has a size, typically measured in bytes, critical for setting transmission rates and optimizing completion times. By understanding these dimensions, cluster applications can achieve better performance through more effective network management strategies [3].

A *coflow* can be thought of as the collective network activity generated by a single application. For example, when loading a webpage, this involves several related network flows: First, the

DNS resolution, which constitutes the base flow. Second, associated flows, initiated by and related to this base flow, include an HTTPS GET request and, if the page has dynamic content, database server queries. Finally, further interactions, like submitting data on the page, add additional associated flows. All these flows together, essential for completing the task, constitute a so-called *coflow* [3].

3 Related Work

In the paper [12] the authors highlight an important concern regarding the slow path within OVS. It is asserted that the slow path 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 centers, alongside the ever-increasing complexity of network topologies. 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 to scale alongside the large number of tenants and services.

While broad literature on the slow path as a key bottleneck exists, various solutions focus on alleviating its limitations through hardware-based accelerators [10, 12]. These works emphasize the need for stateful and event-driven processing enhancements, yet specific implementation methods remain unexplored. Other research focuses on optimizing the slow path by partitioning the flow table between hardware and software, which has been shown to significantly reduce software CPU usage and improve tail forwarding latency [11]. For instance, NuevoMatch employs neural network-based packet classification to enhance OVS scalability and performance, presented in two integration approaches: One as an additional caching layer before OVS's megaflow cache, and another as a complete replacement of the OVS data path that performs direct classification on OpenFlow rules, thereby avoiding control-path upcalls entirely [8]. Although these designs offer promising performance improvements, they are restricted to analyzing past and present data, without the capability to predict future events.

Moreover, other research started recognizing that having future insights into caches can bring performance advantages. In Seer, this is demonstrated by enhancing distributed system caching and predicting future packet arrivals to preemptively manage cache states. This method facilitates advanced pre-fetching and eviction strategies, significantly reducing cache miss ratios by up to 65% [5]. Drawing on the benefits shown in both NuevoMatch, known for its real-time inference capabilities, and Seer, distinguished by its predictive caching techniques, we explore the feasibility of combining these approaches within OVS to enhance performance.

4 Experimental setup

We now describe the experimental environment in Section 4.1, workload in Section 4.2, and the benchmarking setup in Section 4.3.

4.1 Experimental environment

To understand OVS's efficiency and performance, we set up a testbed, shown in Figure 3. This testbed contains two machines with Mellanox ConnectX-6 MT28908 100Gb Ethernet NICs. The System-Under-Test (SUT) runs CentOS Stream 9 with Linux kernel

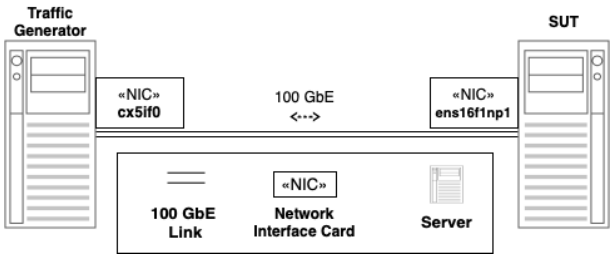


Figure 3: Experimental environment: Two servers connected with a 100 GbE link.

5.14.0, OVS 3.3, and OVN 23.06.2 in a KVM environment on an Intel Xeon Gold 6346 CPU (16 cores, 3.10GHz) and 96 GB RAM. The traffic generator operates on Ubuntu 20.04.4 with Linux kernel 5.4.0, FastClick [2], and DPDK 19.11.14 on an Intel Xeon Gold 5217 CPU (8 cores, 3.00GHz) and 49 GB RAM.

4.1.1 Openshift network simulation. To run our benchmarking on a production-like OCP cluster, without deploying a large-scale environment, we leverage OVN to set 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.

We deploy an environment on SUT that replicates a single-node OCP network configuration as shown in Figure 4. The goal is to enable access to an external IP address via the node's 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's IP within the `3.0.0.0/8` range. The network architecture is outlined below:

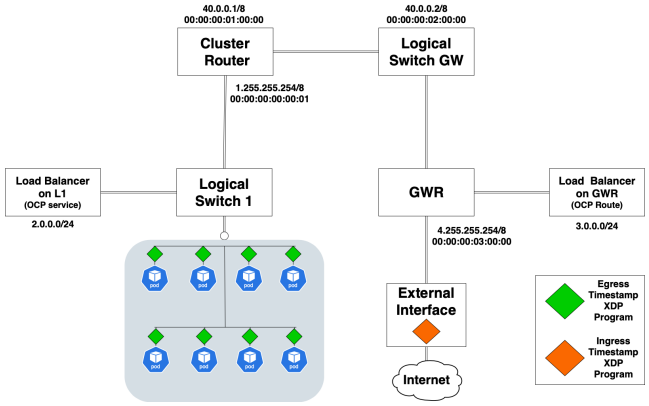


Figure 4: OVN network cluster of a single-node OCP: Ingress timestamp (orange) and egress timestamp (green).

The cluster consists of 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 UDP ports, with each POD allocated a unique service IP in the format of `2.0.0.<pod_id>`. We replicate the load balancer setup to allow external access to the OCP cluster network, balancing nine UDP ports from 2100 to 2180.

4.2 Coflow workload

To conduct the intended benchmarks, with a workload that mirrors the traffic patterns observed in real-world scenarios, particularly in OCP clusters, and containing a degree of coflows is imperative. Unfortunately, the absence of realistic trace files and the scarcity of *coflows'* information in conventional Internet backbone traces, pose a challenge to our experiments. Consequently, we explored alternative approaches for a suitable workload.

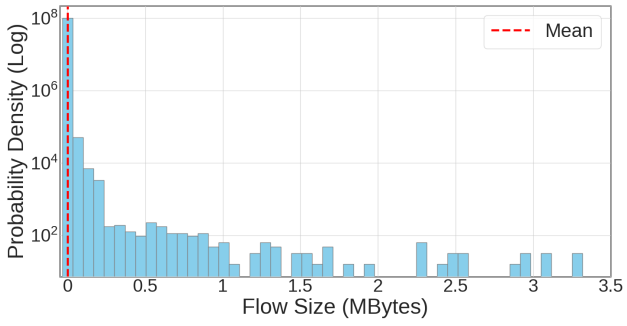
Our investigation led us to adopt the workload generator devised by Agarwal *et al.* This generator leverages an upsampling technique on a 526-coflow trace derived from a one-hour execution of MapReduce tasks on a 3000-machine Facebook cluster [1]. To adjust the trace to our benchmarking environment, we employ modulus reduction to map the original trace's larger set of destinations to our smaller set of pods deployed in the OCP cluster. To distribute traffic evenly across the pods, 50% is directed to the `3.0.0.0/8` subnet, and the remaining 50% to the `3.0.1.0/8` subnet. The trace encompasses 193,000 unique 5-tuple flows, which strategically approaches the flow table limit of 200,000 flows set by OVS. Further, 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.

To evaluate OVS performance under different network traffic conditions, we build two coflow workloads with distinct Flow Size Distributions (FSDs). Here, FSD refers to the distribution of flow sizes measured in bytes, shown in Figure 5. Figure 5 depicts both histograms with the FSDs of both datasets with the same number of bins on the x-axis: Note that the Facebook Hadoop (FB Hadoop) dataset presents a longer tail with flows exceeding 10 MB whereas Google Search RPC (GSRPC) hardly exceeds 3 MB. These FSDs dictate the volume of flows generated, where the combined size of all flows conforms to the respective distribution [6]:

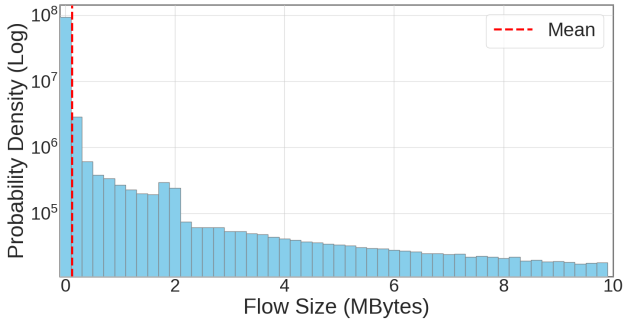
- **GSRPC:** Mean flow size of 426 bytes, bursty, short-lived RPC scenarios (Figure 5a)
- **FB Hadoop:** Mean flow size of 122,182 bytes, long-lived flows (Figure 5b).

While GSRPC's dataset behavior is key for evaluating OVS latency under high network loads, FB Hadoop's dataset represents large data exchanges in environments like Facebook's Hadoop clusters. Utilizing FastClick as a high-speed packet processor, the traffic generator transmits these coflow workloads towards the SUT's external interface in Figure 3 at a rate of 1 million packets per second, corresponding to 12 Gbps incoming throughput.

4.2.1 Coflowiness. To control the accuracy of flow prediction in OVS, we introduce *coflowiness*, a parameter quantifying the interdependency among multiple data flows within a network coflow,



(a) Google Search RPC



(b) Facebook Hadoop

Figure 5: Histogram of the flow size distribution. Note the difference in the x-axis.

based on common endpoints. coflowiness(x) measures the fraction of flows in coflow C with shared sources or destinations: coflowiness(x) = 0 indicates no shared attributes among flows in C, while coflowiness(x) = 1 implies all flows share at least one endpoint. Values between 0 and 1 represent partial sharing, where coflowiness(x) × 100% of flows match another within the same coflow. Our benchmarking strategy involves accurately defining the base and associated flows within coflows, where base flows represent primary data transfers and associated flows are supplementary, but directly related and dependent on the base flows. We define base flows by a destination port of 2100, which uniquely characterizes them in our benchmarks.

4.3 Benchmarking setup

To determine the maximum achievable performance in OVS with perfect knowledge of future flows, OVS is evaluated in the three following distinct scenarios:

(1) **Baseline:** This scenario starts with empty data path caches. It measures the default OVS performance using a trace comprising base and associated flows, without any predictive flow loading.
(2) **Optimal:** Before transmitting the coflow workload, this scenario pre-populates the OVS caches with the associated flows from the trace. Thus, only the base flows trigger *upcalls* to the controller during benchmarking. This setup

tests the OVS performance under conditions of optimal, albeit theoretical, flow prediction.

(3) **Varying Coflowiness:** While similar to the optimal scenario, this scenario incorporates a varying coflowiness value to adjust the ratio of associated flows to base flows. The goal is to assess the influence of different flow prediction accuracies on the performance of OVS.

Note that, in all scenarios, the flow cache capacity is configured at 200,000 entries, and the flow timeout is disabled. Table 1 shows coflowiness values in the corresponding benchmark scenario, and the percentage of associated flows preloaded into the OVS cache.

Table 1: Coflowiness levels and associated flows.

Coflowiness Level	(%) Associate Flows Preloaded
0.0	0% (Baseline, no preloading)
0.1	10%
0.25	25%
0.5	50%
0.75	75%
0.9	90%
1.0	100% (Optimal, complete preloading)

Our traffic generator continuously replays the generated coflow traffic trace towards the SUT until completion, defining this process as one benchmarking round. We conduct ten such rounds for statistical relevance for each coflow configuration across the two FSDs defined in Section 4.2. This approach results in seven unique benchmarks per FSD configuration. Data is collected by the receiving daemon, which analyzes packet details, including sequence numbers and the delay between ingress and egress at OVS. This analysis helps us derive maximum and average packet latencies for three 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.

Due to space constraints, we focus on presenting results for cases (2) and (3). Although latency results for the base packets are comparable to those of the associate packets, presenting results for associate flows is more crucial as they hold the potential for predictive improvement. It is also important to note that the latency of base flows cannot be significantly reduced through prediction and pre-loading strategies. This limitation exists because base flows would serve as the initial input used by a flow predictor to forecast the behavior of subsequent associate flows.

4.3.1 Measure latency. We now measure the per-packet latency of network traffic traversing through OVS. During the transmission of the *coflow* workload by 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. Then, to be able to time the traversal of packets, we developed an XDP program to insert timestamps into the packet payload.

These timestamps are inserted at specific interfaces, avoiding additional sources of latency. The ingress timestamp is added at the ingress point interface, depicted in orange in Figure 4, within the cluster deployed on the SUT machine. The egress timestamp is inserted at the egress point at the veth pair, shown in green in Figure 4, leading to individual pods. This precisely captures the moment before the packet exits 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 the system boot. Figure 6 shows how the UDP packet payload is structured to accommodate the three essential pieces of information for our experiments, 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 Internet Protocol (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 (24 bytes), facilitates extraction and interpretation of per-packet latency data.

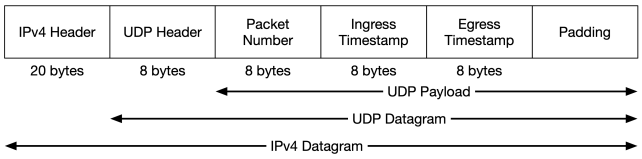


Figure 6: UDP packet structure: Packet number, ingress timestamp, and egress timestamp in the packet payload.

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. This information is stored in an array for later statistical analyses.

5 Evaluations

We now describe the results from benchmarking OVS using the cluster configuration described in Section 4.1, while transmitting a *coflow* workload under various scenarios. More concretely, we quantify performance by considering three primary metrics: Maximum and average packet latencies from ingress to egress, CPU utilization of OVS handler threads and the overall system, and the frequency of upcalls to `ovs-vswitchd` per second.

5.1 End-to-End Packet Latency

This section presents latency statistics for network flows within the OCP cluster deployed on the SUT machine in Figure 3. We analyze two types of latency measurements: 1) the maximum latency of the first packet in each unique associated flow, and 2) the mean latency across all packets in both base and associated flows. Each type of latency measurement is derived from ten runs, with each

measurement data being collected by a daemon running on each pod in the cluster.

5.1.1 First associate packet latency statistics. Figure 7 shows the coflowiness level and corresponding latency. We can see that, although GSRPC and FB Hadoop are very distinct datasets, latency is drastically reduced as the coflowiness level increases beyond 0.5, see Table 1. In the base case with a coflowiness value of 0.0, the GSRPC trace has ~58% higher maximum latency compared to FB Hadoop. Moreover, for intermediate coflowiness values (e.g. 0.1 to 0.75) GSRPC shows ~1.6× to ~2× higher maximum latency when compared to its counterpart. When all flows are present as in the optimal scenario of coflowiness 1.0, their latency distributions converge which is expected since no upcalls are required.

5.1.2 Aggregate packet latency statistics. The mean end-to-end latency for FB Hadoop, as shown in Figure 8a, exhibits a clear and consistent decreasing trend as coflowiness increases. This indicates that the prediction of flows has a positive impact in the overall mean latency. For the GSRPC FSD shown in Figure 8b, 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. Moreover, in the baseline scenario, the mean latency for GSRPC is ~10× higher than that for FB Hadoop. For intermediate coflowiness values, GSRPC's mean latency remains about 5× to 10× higher than FB Hadoop demonstrating a ~30% improvement when 25% of the flows are accurately predicted. At a coflowiness of 0.75, the mean latency for GSRPC is ~15× higher than that for FB Hadoop. In the optimal scenario, the mean latencies for both FSDs converge, which is expected since all flows are fully cached.

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, only a small fraction of flows are accurately predicted and preloaded into the cache. Consequently, incoming packets are more likely to encounter cache misses, triggering upcalls to the `ovs-vswitchd` user space daemon. 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. Conversely, the GSRPC trace presents challenges for caching due to its high flow rate, defined as the number of new flows arriving per time unit, particularly with new, small flows and bursty traffic patterns. The 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.

5.2 Resource Allocation and CPU Usage

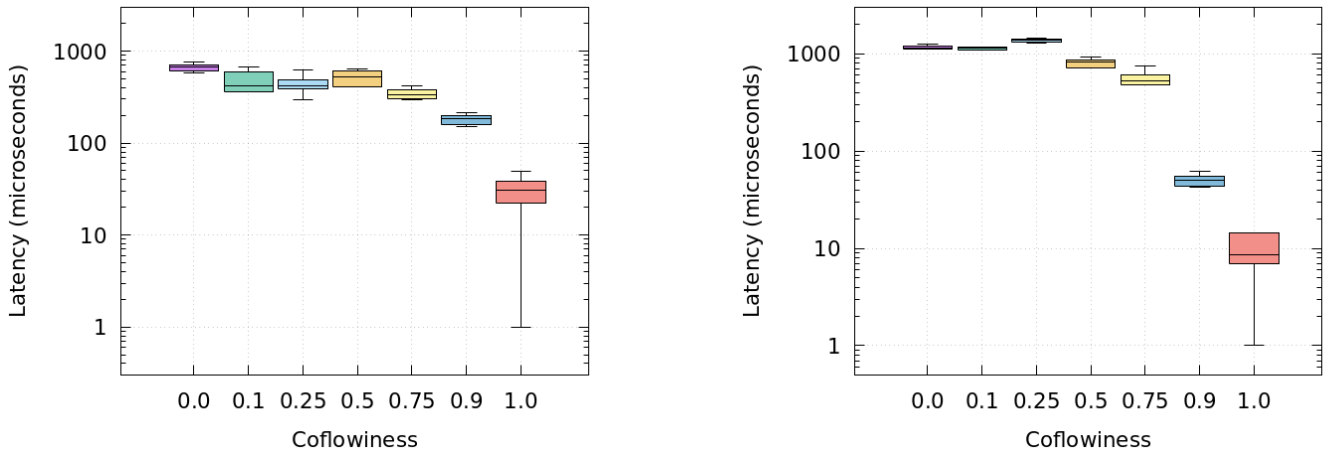
This section presents the CPU resource utilization required to achieve varying end-to-end packet latencies across different coflowiness configurations. These statistics are gathered while benchmarking OVS using the two FSDs. Each figure plots the CPU utilization as a percentage on the y-axis against time in seconds on the x-axis, under various coflowiness configurations.

The examination of total CPU utilization for FB Hadoop, as illustrated in Figure 9a, reveals a consistent trend where higher coflowiness values correspond to lower CPU utilization, with the optimal scenario at a coflowiness of 1.0 exhibiting the lowest utilization of ~7% on average. Conversely, the baseline scenario with a coflowiness of 0.0 experiences the highest CPU utilization, ranging from ~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 ~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 9b, 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% to ~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 transmitting the GSRPC trace, there is no discernible reduction in CPU utilization due to flow caching. Importantly, even with perfect flow prediction, the total CPU utilization with GSRPC is ~6× 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.

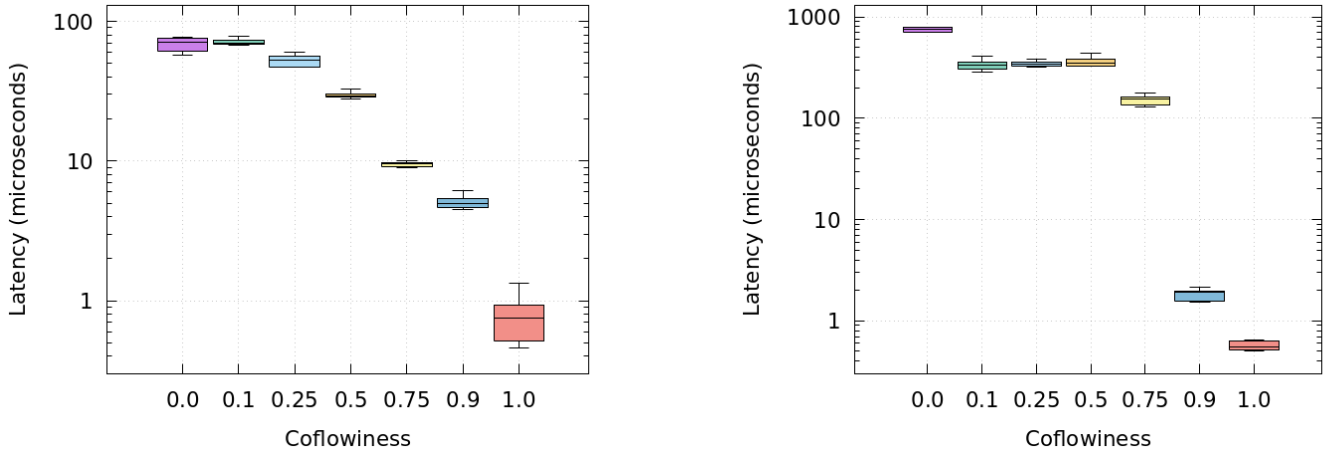
Similar to the impact on latency with the FB Hadoop trace, once a flow is established and cached in the kernel datapath, subsequent packets of the flow are processed efficiently, minimizing the need for additional upcalls. This caching mechanism effectively reduces upcall frequency as more flows become cached. As the system encounters fewer new flows, this amplifies the reduction in upcall frequency and, consequently, lowers CPU demand.

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.



(a) Facebook Hadoop **(b) Google Search RPC**

Figure 7: Maximum ingress-to-egress packet latency for the first packet of each unique associated flow processed by OVS, comparing latency across two distinct flow size distributions.



(a) Facebook Hadoop **(b) Google Search RPC**

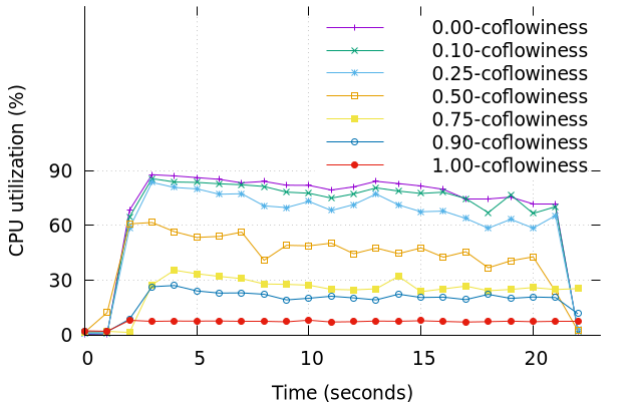
Figure 8: Mean ingress-to-egress packet latency statistics calculated from the first packet of each unique flow processed by OVS, using two different flow size distributions. Note the difference in the y-axis.

5.3 CPU Utilization of OVS Handler Threads

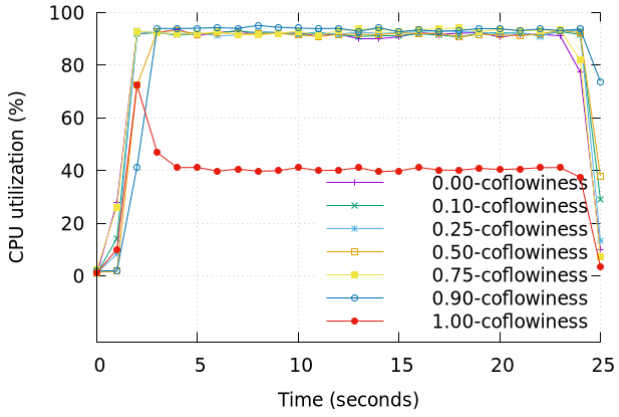
In this section, we present an analysis of CPU utilization specifically attributable to OVS handler threads, which are responsible for processing initial packets of new flows before flow entries are cached. This utilization is included in the total CPU utilization figures shown in the graphs in the previous section.

In Figure 10a, 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 baseline scenario. Similarly, Figure 10b, which analyzes the GSRPC setup, illustrates that there is no consistent trend of decreasing total CPU utilization with increasing coflowiness, except in the optimal scenario of 1.0 coflowiness where utilization is as low as ~0.1%, mirroring the result seen in the FB Hadoop setup. All coflowiness cases, aside from the optimal scenario, exhibit utilization levels between ~20% and ~40%. Moreover, only the optimal scenario of predicting all flows correctly at 1.0 coflowiness demonstrates a significant utilization benefit.

Interestingly, our results show that for low coflowiness values, the OVS handler threads exhibit lower resource consumption when using the GSRPC FSD compared to FB Hadoop. This observation is attributed to the challenges OVS faces at high flow rates, where

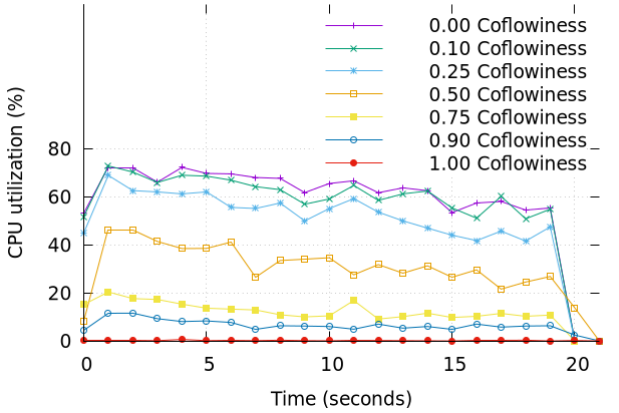


(a) Facebook Hadoop

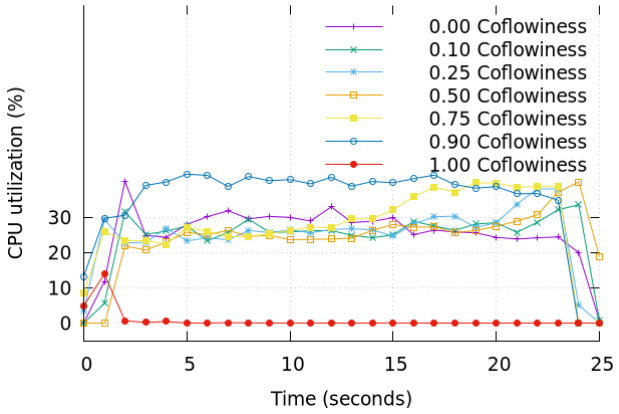


(b) Google Search RPC

Figure 9: CPU utilization between Facebook Hadoop and Google Search RPC: Flow size distributions for system processes, software interrupts, and total usage.



(a) Facebook Hadoop



(b) Google Search RPC

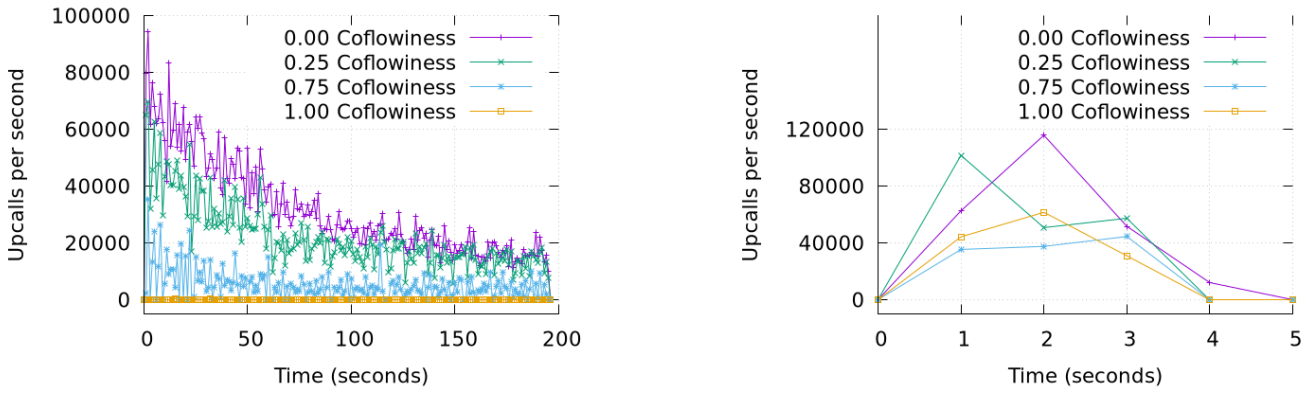
Figure 10: Total CPU utilization for OVS handler threads under varying flow size distributions.

it is bombarded with new flows more frequently than it can efficiently process them. In response, OVS employs upcall batching as a mitigation strategy. While this approach helps amortize the cost per upcall by distributing it across multiple packets, it queues packets before they can be classified, thus increasing latency.

5.4 Upcall statistics

Now we evaluate 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. Figures 11a and 11b show 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 11a, 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 higher coflowiness values, as more flows are cached over time. This behavior aligns with previous CPU utilization results, as the number of upcalls directly correlates with CPU usage when handling new flows. For the optimal scenario, 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 ~70,000 and ~35,000 upcalls per second, respectively. Over time, the number of upcalls gradually decreases and converges toward the optimal scenario. The GSRPC benchmark, as shown in Figure 11b, 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. In the GSRPC baseline scenario, the number of upcalls per second peaks at ~115,000, making it the highest among all configurations. Notably, the upcalls in the FB



(a) Facebook Hadoop **(b) Google Search RPC**

Figure 11: Number of upcalls made to `ovs-vswitchd` under varying flow size distributions.

Hadoop benchmark commence immediately, indicating that packets are processed individually or in small batches. In contrast, the GSRPC benchmark (Figure 11b) 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 again indicates that OVS attempts to handle the high GSRPC flow rate through batch processing of upcalls to use CPU resources more efficiently.

5.5 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. This leads to excessive upcalls and higher CPU utilization. The way OVS handles these upcalls (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. Moreover, handler threads show increased CPU utilization, processing upcalls immediately upon arrival at the ingress, either individually or in small 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. Batch processing leads to delays, as packets must await the processing of the entire batch. Although this method improves CPU cache efficiency and reduces context switching, it does not significantly lower overall CPU utilization. The persistent high CPU usage stems from the continuous influx of new flows and the extensive processing demands of other system components, such as system processes and software interrupts. Consequently, the benefits of batch processing are offset by the substantial CPU demands driven by the high flow rate and the extensive volume of upcalls.

5.6 Unexpected Upcall Counts and TCP

Our experiments with UDP demonstrated that the number of upcalls frequently exceeded the number of unique flows, as multiple packets from the same flow triggered upcalls before a flow entry was established. TCP, with its connection-oriented setup and flow control, spaces out packet arrivals, reducing the probability of such multiple upcalls. The initial SYN packet in a TCP connection prompts an upcall for flow entry installation, but subsequent packets wait for the handshake to complete, allowing time for the flow entry to be established. This mechanism, coupled with TCP's retransmission timers and the use of delayed acknowledgments, generally leads to fewer upcalls for TCP compared to UDP. This behavior results in more efficient upcall management, decreased CPU utilization, and improved overall performance of OVS.

6 Discussion and Future Work

This research establishes a theoretical upper limit of performance for OVS when provided with perfect knowledge of future flows and provides a methodology framework for future benchmarking once a flow prediction system is implemented. Future studies could explore statistical models like Markov chains or Bayesian networks, or machine learning models such as decision trees and neural networks, to predict flows. These models would utilize both historical and real-time traffic data to optimize prediction accuracy and resource utilization. Building on foundational work by Raschelbach *et al.* extending the computational cache to integrate flow prediction capabilities, such as shallow neural networks or hierarchical models, is suggested [8]. Additionally, the implementation of prediction algorithms may require specialized hardware like GPUs or FPGAs to meet computational demands and support real-time processing. This would enhance processing speed and energy efficiency, crucial for data center operations. If successful, integrating a flow prediction system into OVS could significantly enhance its performance and should be considered for deployment in data centers, IXPs,

and OCP settings. Future research should focus on validating these benefits and developing strategies for allocating resources between prediction tasks and regular packet processing.

Finally, researchers building on our work should focus on incorporating real-world coflow traffic traces from operational OCP clusters to provide a more accurate reflection of FSDs, flow counts per coflow, and levels of coflowiness. 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 OCP pods and nodes. 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.

7 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.

Our work has performed a set of benchmarks and reveals that the advantages of predicting and preloading flows in OVS depend heavily on the flow rate of the traffic trace. High flow rates, especially in bursty RPC traffic that rapidly introduces new flows, diminish the benefits of preemptive rule installation. This is due to the continued necessity for upcalls and software interrupts, despite employing upcall batching as a mitigation strategy. Although batching helps spread the upcall cost over multiple packets, it also delays packet classification, increasing latency.

Conversely, OVS significantly improves in handling larger, long-lived Facebook Hadoop flows characterized by lower flow rates. Predicting just 25% of these flows can decrease average latency by ~24% and reduce CPU usage by ~12%, as fewer new flows necessitate upcalls, allowing `ovs-vswitchd` to process incoming packets more efficiently. These findings underscore the potential of flow prediction and preloading in enhancing OVS's responsiveness and preventing the slow path from becoming a bottleneck in high-throughput scenarios.

References

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3230543.3230569

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. https://doi.org/10.1109/ANCS.2015.7110116

[3] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2390231.2390237

[4] Georgios Koukis, Sotiris Skaperas, Ioanna Angeliki Kapetanidou, Lefteris Mamatas, and Vassilis Tsaoussidis. 2024. Performance Evaluation of Kubernetes Networking Approaches across Constraint Edge Environments. http://arxiv.org/abs/2401.07674 arXiv:2401.07674.

[5] Jason Lei and Vishal Shrivastav. 2024. Seer: Enabling {Future-Aware} Online Caching in Networked Systems. 635–649. https://www.usenix.org/conference/nsdi24/presentation/lei

[6] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. 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 (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3230543.3230564

[7] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open vSwitch. USENIX Association, Oakland, CA. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf

[8] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. Scaling Open vSwitch with a Computational Cache. https://www.usenix.org/system/files/nsdi22-paper-rashelbach.pdf

[9] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. 2023. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing* 12, 1 (June 2023), 87. https://doi.org/10.1186/s13677-023-00471-1

[10] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open vSwitch dataplane ten years later. *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Aug. 2021). https://doi.org/10.1145/3452296.3472914 Conference Name: SIGCOMM '21: ACM SIGCOMM 2021 Conference ISBN: 9781450383837 Place: Virtual Event USA Publisher: ACM.

[11] Yanshu Wang, Dan Li, Yuanwei Lu, Jianping Wu, Hua Shao, and Yutian Wang. 2022. Elixir: A High-performance and Low-cost Approach to Managing {Hardware/Software} Hybrid Flow Tables Considering Flow Burstiness. Tsinghua University. https://www.usenix.org/system/files/nsdi22-paper-wang_yanshu.pdf

[12] Annus Zulfiqar, Ben Pfaff, William Tu, Gianni Antichi, and Muhammad Shahbaz. 2023. The Slow Path Needs an Accelerator Too! *ACM SIGCOMM Computer Communication Review* 53, 1 (April 2023), 38–47. https://doi.org/10.1145/3594255.3594259