On the Benefits of Predictable Traffic in Virtual Switches: A Case Study

Emil Ståhl, Marco Chiesa
KTH Royal Institute of Technology, Sweden

Simone Ferlin[1,2], Eelco Chaudron[1]
[1]Red Hat, [2]Karlstad University, Sweden

Abstract

Software-Defined Networking (SDN) has increasingly shifted toward 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 since the control plane installs rules in the data plane reactively as new flows arrive, which requires time-consuming transitions to user space. In this paper, we demonstrate how to prevent a bottleneck in the slow path by investigating the impact of predicting flows and preemptively installing their rules in the data plane. Using a workload containing different levels of "coflows", our results demonstrate that this predictive approach varies with the flow rate of traffic traces. Notably, a system that could predict 25% of the traffic flows would decrease the average latency by up to approximately 24% and reduce CPU utilization by approximately 12%.

CCS Concepts

• **Networks → Programmable networks**; *Network performance analysis*; *Network measurement*; Packet-switching networks.

Keywords

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

ACM Reference Format:
Emil Ståhl, Marco Chiesa and Simone Ferlin[1,2], Eelco Chaudron[1]. 2025. On the Benefits of Predictable Traffic in Virtual Switches: A Case Study. In *Applied Networking Research Workshop (ANRW 25), July 22, 2025, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3744200.3744762

1 Introduction

SDN has revolutionized the way network traffic is managed, by separating the data plane, 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, 13]. This 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 of SDN devices installing data plane forwarding rules in a *reactive* manner, *i.e.*, only when the first packet of a flow arrives. 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. Cache misses can significantly increase latency, often resulting in delays of tens or hundreds of milliseconds, especially for traffic flows not currently present in the cache. Open Virtual Switch (OVS) is a common data plane virtual switch that installs flow rules reactively and has been widely adopted due to its 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, 11]. 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, which consists of different types of *caches* holding rules for 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. In OVS, the *slow path* periodically validates the entries installed in the data plane caches against current OpenFlow rules, a critical process that guarantees network performance. However, as network bandwidth increases and topologies grow, the slow path has emerged as a bottleneck, particularly when new flows arrive and there are no rules installed [8].

In this paper, we demonstrate how a flow prediction mechanism can help the slow path. More concretely:

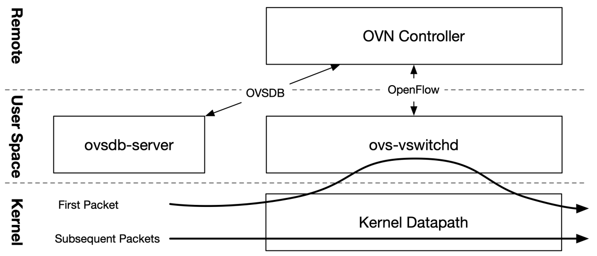


Figure 1: Components and interfaces of OVS including the kernel data path in charge of forwarding packets.

- Explore the benefits of predicting traffic to enhance OVS's cache performance and reduce cache misses.
- Conduct performance benchmarks of OVS in a production-like container management platform across diverse traffic scenarios.
- Develop a benchmarking framework that can be applied by future research to evaluate and refine OVS under varying conditions.

We focus on the *potential performance benefits* of predicting upcoming traffic with co-flow based workloads. Building accurate predictors for co-flows or arbitrary types of traffic is left as future work.

2 Background & Motivation

We now describe two 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. It 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 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, i.e., triggering a request to the control path (via the `ovs-vswitchd`), see Figure 1[1] [7, 11].

Upcalls are particularly costly, since packets transition from the kernel to user space, where more extensive packet classification takes place. This process adds 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, thus the rate of *upcalls* grows. Moreover, handling these *upcalls* burdens the `ovs-vswitchd` user space daemon, consuming significant CPU and memory resources [8, 13].

[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].

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 of the same flow can be forwarded directly. However, for bursty traffic, the microflow cache provides limited benefit [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 wider range of traffic. The combination of microflow and megaflow caching balances efficiency and flexibility [7, 11]. Despite these optimizations, *upcalls* continue to be a major bottleneck in OVS, especially as traffic scales or becomes more dynamic [8, 12]. Furthermore, as the number of flow rules increases, the size and efficiency of the caches are constrained [8].

2.2 Openshift Container Platform

The OCP, developed by Red Hat, uses Kubernetes to orchestrate containerized applications across hybrid and multi-cloud environments. Within OCP, OVS is deployed to manage network traffic, with network virtualization facilitated by Open Virtual Networking (OVN), integrated as the Container Network Interface (CNI) through OVN-Kubernetes (OVN-K). 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 [12].

2.3 Coflows

Cluster computing environments often struggle to optimize communication patterns. This inefficiency is due to traditional networking approaches that do not manage complex, concurrent data flows [1]. Addressing these challenges, Chowdhury *et al.* [3] introduced the concept of *coflows*. 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. By understanding these dimensions, cluster applications can achieve better performance [3]. A coflow groups related network flows from a single application. Loading a webpage exemplifies this: a DNS base flow triggers associated flows such as HTTPS GETs, database queries, and data submissions [3].

3 Related Work

In [13] the authors highlight an important concern regarding the slow path within OVS, poised to emerge as a bottleneck in SDN. This projection is grounded in the increasing bandwidth demands alongside the growing complexity of network topologies. While literature on the slow path as a key bottleneck exists, various solutions focus on hardware-based accelerators to speed packet classification and flow rule matching [11, 13]. These works emphasize the need for stateful and event-driven processing, 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 reduce software CPU usage and improve tail forwarding latency [12]. For instance, NuevoMatch employs neural network-based packet classification to enhance OVS scalability and performance, presented in two 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 improvements, they are limited to analyzing past and present data.

Moreover, other research has begun to recognize that having future insights into caches can bring performance advantages. In Seer [5], by improving distributed system caching and predicting future packet arrivals to preemptively manage cache states. This method facilitates advanced pre-fetching and eviction strategies, reducing cache miss by up to 65% [5]. Drawing on the benefits shown in 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.

4 Experimental setup

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

4.1 Experimental environment

We evaluated OVS's performance using a testbed with two machines with Mellanox 100GbE Network Interface Controllers (NICs). The System-Under-Test (SUT) runs CentOS Stream 9, Linux 5.14, OVS 3.3, and OVN 23.06.2 in a KVM environment on an Intel Xeon Gold 6346 (16 cores, 3.10GHz, 96GB RAM), while the traffic generator runs Ubuntu 20.04.4, Linux 5.4, FastClick [2], and DPDK 19.11.14 on an Intel Xeon Gold 5217 (8 cores, 3.00GHz, 49GB RAM) [10].

4.1.1 Openshift network simulation. To run our benchmarking on a production-like OCP cluster, we use OVN to set up a cluster for benchmarking OVS. Given that OCP uses 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, see Figure 2. We enable access to an external IP via the node's IP.

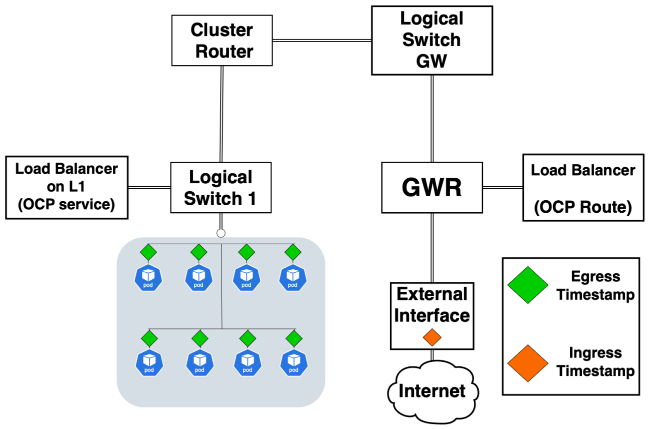


Figure 2: OVN network cluster of a single-node OCP: Ingress (orange) and egress (green) timestamps.

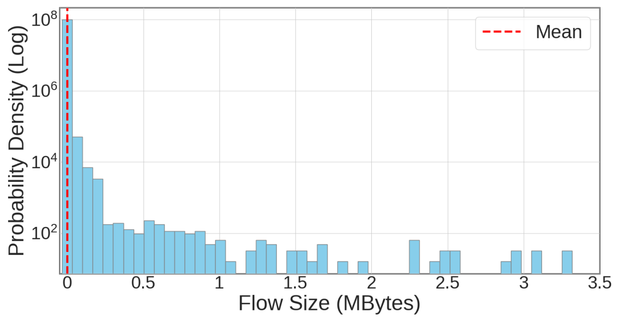
The cluster runs 8 pods, each assigned a distinct IP address. Every POD resides within its dedicated namespace, denoted as pod$X$. Furthermore, we establish a service IP to emulate pod-to-pod on the privately routed `2.0.0.0/24` network. Load balancers are configured for 9 User Datagram Protocol (UDP) ports, with each pod assigned a unique service IP in the format of `2.0.0.<pod_id>`. We replicate the load-balancer setup for external access, balancing 9 UDP ports [10].

4.2 Coflow workload

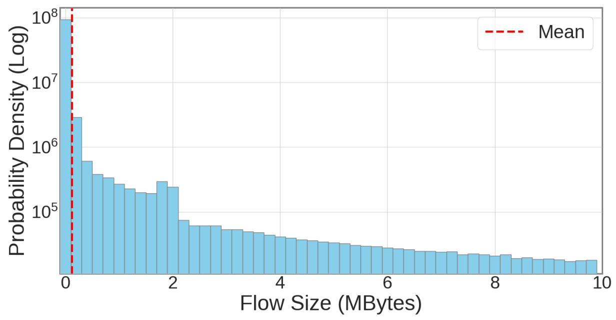
We use the workload generator by Agarwal *et al.* to create UDP-only traffic, upscaling a 526-coflow trace from a one-hour MapReduce execution on a 3000-machine Facebook cluster to 193,000 unique 5-tuple UDP flows, approaching the flow table limit of 200,000 flows set by OVS. This dataset represents real-world data center traffic, since many flows share the same source or destination. Each packet in the trace results in an UDP packet size of 1500 bytes.

We build two coflow workloads with distinct Flow Size Distributions (FSDs). Figure 3 shows both histograms with FSDs of both datasets with the same number of bins on the x-axis: Note that the Facebook Hadoop (FB Hadoop) dataset contains flows exceeding 10MB whereas Google Search RPC (GSRPC) hardly exceeds 3MB. The FSDs dictate the volume of generated flows, where the combined size of all flows matches the respective distribution [6]:

- **GSRPC:** Mean flow size of 426 bytes, bursty, short-lived (Figure 3a)
- **FB Hadoop:** Mean flow size of 122,182 bytes, long-lived flows (Figure 3b).



(a) Google Search RPC



(b) Facebook Hadoop

Figure 3: Flow size distribution (note the difference in the x-axis).

While GSRPC's dataset evaluates OVS latency under high network loads, FB Hadoop's dataset represents large data exchanges. Using FastClick as a high-speed packet processor, the traffic generator transmits these coflow workloads to the SUT's external interface at a rate of 1 million packets per second, equivalent to 12 Gbps of incoming throughput [10].

4.2.1 Coflowiness. To control the accuracy of flow prediction in OVS, we introduce *coflowiness*, a parameter quantifying the inter-dependency among multiple data flows, 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$(x) \times 100\%$ 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 [10].

4.3 Benchmarking setup

To determine the achievable performance in OVS with perfect knowledge of future flows, we test these scenarios:

(1) **Baseline:** It starts with empty data path caches, measuring default OVS performance using a trace with base and associated flows.
(2) **Optimal:** It pre-populates the OVS caches with the associated flows from the trace. Thus, only the base flows trigger *upcalls*. This setup tests the OVS performance under conditions of optimal, albeit theoretical, flow prediction to define a performance ceiling. The goal is to benchmark against this best-case scenario.
(3) **Varying Coflowiness:** While similar to the optimal scenario, it incorporates a varying coflowiness 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.

In all scenarios, the flow cache capacity is set to 200,000 entries, and the flow timeout is disabled. Table 1 shows coflowiness levels, and the percentage of associated preloaded flows.

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)

We run ten rounds for each coflow configuration in FSDs defined in Section 4.2. This approach results in seven unique benchmarks per FSD configuration. Data are collected by the receiving daemon, which helps us derive maximum and average packet latencies for three packet types: (1) First base packet of each unique 5-tuple flow, (2) first associate packet of each unique 5-tuple flow, and (3) aggregate statistics based on all packets processed by FastClick and observed by OVS [10]. Due to space limitation, we focus on (2) and (3)[2].

4.3.1 Measure latency. We now measure the per-packet latency traversing OVS. During transmission of the *coflow* workload, FastClick inserts a 64-bit packet number into each payload of UDP packets. To time the traversal of packets, we developed an eXpress Data Path (XDP) program to insert timestamps into the packet payload Also, the timestamps

[2]Note that the latency of base flows cannot be significantly reduced, since base flows serve as the initial input for the flow predictor to forecast the behavior of subsequent associated flows.

are inserted at specific interfaces: The ingress timestamp is shown in orange in Figure 2, while the egress timestamp is shown in green in Figure 2. For our latency measurements, the UDP payload was structured to include three 64-bit fields: the unique packet number inserted by FastClick, followed by the ingress timestamp, and finally the egress timestamp.

Upon receiving the packet, a dedicated daemon operates within each cluster pod, extracting the ingress and egress timestamps, along with the corresponding packet number, and calculating the latency for each packet [10].

5 Evaluations

We now show the results with the cluster from Section 4.1. via two metrics: Average packet latencies from ingress to egress, and CPU utilization of OVS handler threads.

5.1 End-to-End Packet Latency

This section presents latency statistics for network flows within the OCP cluster deployed on the SUT machine.

The mean end-to-end latency for FB Hadoop in Figure 4a, shows a consistent decreasing trend as coflowiness increases. This indicates that the prediction of flows has a positive impact on the overall mean latency. For GSRPC FSD in Figure 4b, the mean latency shows a decreasing trend; however, it becomes evident once at least 75% of the flows are predicted. 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 showing 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.

The FB Hadoop trace shows improvements in latency with increased coflowiness. Larger flows benefit from caching because once a flow rule is installed, subsequent packets of that flow benefit from packet processing within the kernel space. 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.

5.2 CPU Usage of OVS Handler Threads

The OVS handler threads process the initial packets of new flows before caching rules. In Figure 5a, there is a trend of decreasing total CPU with increasing coflowiness, with a general decline in utilization. The total utilization ranges from a minimal ~0.1% in the optimal scenario to ~70% in the baseline scenario. Notably, achieving a 50% flow prediction rate results in halving the total utilization compared to the baseline scenario. Similarly, Figure 5b, which analyzes the GSRPC setup, shows that there is no consistent trend of decreasing total CPU with increasing coflowiness, except for the optimal scenario of 1.0 coflowiness where utilization is as low as ~0.1%, mirroring the result seen in the FB Hadoop setup. Moreover, only the optimal scenario of predicting all flows correctly at 1.0 coflowiness demonstrates a significant utilization benefit. Our results show that for low coflowiness, the OVS handler threads exhibit lower resource consumption with GSRPC FSD compared to FB Hadoop. This observation is attributed to the challenges that OVS faces at high flow rates. Although OVS uses upcall batching, amortizing the cost per upcall, it queues packets, which adds to latency.

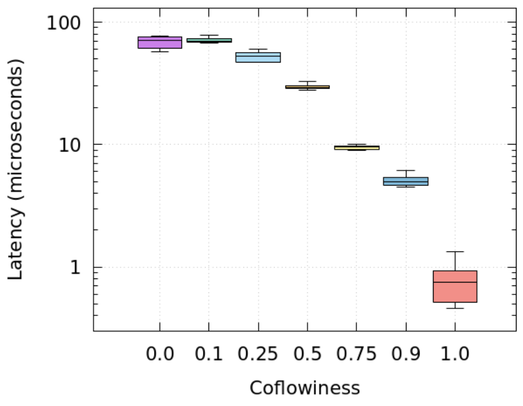
5.3 Bottlenecks and Trade-offs

Our benchmarks show high flow rates and cache misses causing a bottleneck in OVS CPU utilization. The way OVS handles upcalls (individually or in batches) significantly affects CPU utilization and packet latency.

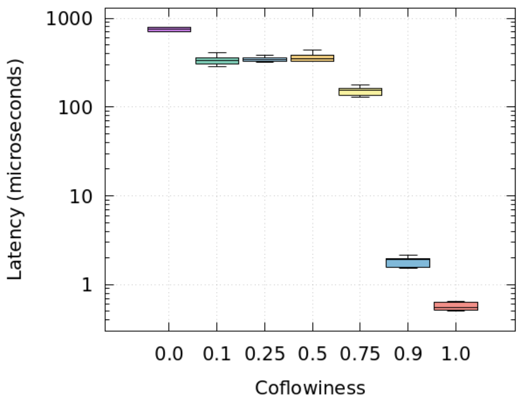
For the FB Hadoop trace, the bottleneck is mitigated by effective flow caching and prediction. The larger, long-lived flows allow the cache to remain relevant for a larger set of packets. Moreover, handler threads show increased CPU utilization, processing upcalls immediately upon arrival at the ingress. This leads to higher per-upcall CPU overhead, but maintains lower end-to-end packet latency. In the GSRPC traffic scenario, a persistent bottleneck arises from the 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, overwhelms the `ovs-vswitchd`. Although batch processing 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 processing demands of other components, such as system processes and software interrupts [10].

6 Discussion and Future Work

Preloading a subset of associate flows into the OVS cache improves performance, reducing cache misses and enabling direct data-plane processing with lower latency and CPU usage. The optimal scenario (with coflowiness of 1.0) serves as an empirical upper bound for OVS under theoretically perfect flow prediction. Future work could use statistical or machine learning models with historical and real-time data for flow prediction. Building on Raschelbach *et al.* [8], which extends the computational cache to integrate flow prediction capabilities, such as shallow neural networks or hierarchical models. Additionally, the implementation of prediction algorithms may require specialized hardware like GPUs or FPGAs to meet computational demands. Future

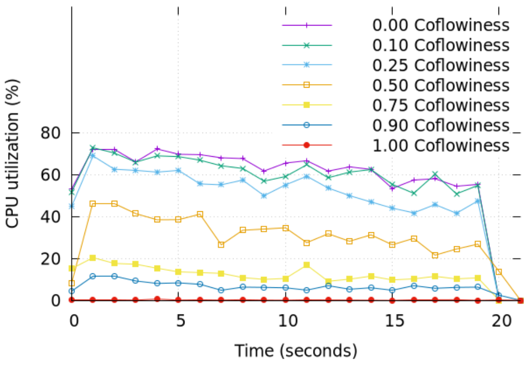


(a) Facebook Hadoop

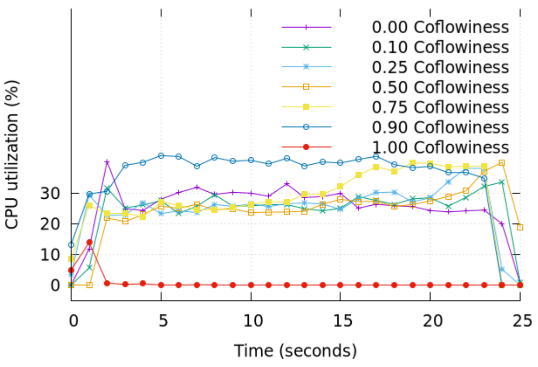


(b) Google Search RPC

Figure 4: Mean ingress-to-egress packet latency 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).



(a) Facebook Hadoop



(b) Google Search RPC

Figure 5: Total CPU utilization for OVS handler threads under varying flow size distributions.

research should focus on validating these benefits and developing strategies for allocating resources between prediction tasks and regular packet processing, since our single-node setup limits scalability insights. 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, 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. 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 realistically measures runtime performance of loading associate flows [10].

7 Conclusions

OVS is a key intermediary between the control and data planes. Our work shows that flow prediction and preloading benefits depend on traffic flow rates. High flow rates diminish the benefits of preemptive rule installation due to continued reliance on upcalls and software interrupts, even with batching as a mitigation. 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 [10].

Acknowledgments

This work was supported by the European Commission under the AC3 project (grant agreement ID 101093129). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official EU policies or endorsements, either expressed or implied, of the funding agencies.

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] Emil Ståhl. 2025. GitHub - RedHatResearch/ovscon24-Open-vSwitch-with-coflow-aware-prediction: Performance benchmark of Open vSwitch with perfect knowledge of future flows. https://github.com/RedHatResearch/ovscon24-Open-vSwitch-with-coflow-aware-prediction

[11] 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.

[12] 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

[13] 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