Single-Core Hotspots on Your VNF? Break Them Up!

Changgang Zheng[†‡], Bowen Yang[†], Jin Ke[†], Ye Yang[†], Enge Song[†], Haonan Li[†], Zihao Fan[†¶], Mingxin Li[†‡], Yilong Lv[†], Yisong Qiao[†], Jun Liang[†], Donglin Lai[†], Bengbeng Xue[†], Yang Song[†], Xing Li[†⋄], Rong Wen[†§], Zhigang Zong[†], Shunmin Zhu[†]

[†]Alibaba Cloud, [‡]Nanjing University, [¶]Shanghai Jiao Tong University, [⋄]Zhejiang University, [§]Fudan University

ABSTRACT

Current NFVs assign packets to CPU cores at flow granularity, where each flow is pinned to a single CPU. This approach is efficient under most scenarios but has exposed limitations when handling elephant flows. These "heavy hitters" overwhelm single cores, creating bottlenecks that affect overall throughput and degrade service quality. As networks scale to higher-speed links and core-rich CPUs, these imbalances become more severe. In this paper, we propose **ParaFlowO**, an architecture that **Para**llelizes processing elephant **Flow**s across multiple CPU cores while preserving in-**O**rder delivery. ParaFlowO breaks elephant flows into flowlets and dynamically rotates them across multiple cores. It integrates a lightweight reordering mechanism to preserve packet order and controls parallelism to mitigate contention on shared state. Preliminary evaluations show that ParaFlowO offers a practical solution to mixed-grained parallelism in stateful middleboxes.

CCS CONCEPTS

• **Networks** → **Packet scheduling**; **Network resources allocation**; *Middle boxes / network appliances*; Cloud computing.

KEYWORDS

Network Function Virtualization; Stateful Middleboxes; Elephant Flows; Flowlet Switching; Parallel Packet Processing; Intra-Server Load Balancing

ACM Reference Format:

Changgang Zheng, Bowen Yang, Jin Ke, Ye Yang, Enge Song, Haonan Li, Zihao Fan, Mingxin Li, Yilong Lv, Yisong Qiao, Jun Liang, Donglin Lai, Bengbeng Xue, Yang Song, Xing Li, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2026. Single-Core Hotspots on Your VNF? Break Them Up!. In *The 10th Asia-Pacific Workshop on Networking (APNet 2026), August 06–07, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3820441.3820442

1 INTRODUCTION

Network Function Virtualization (NFV) has emerged as a foundational technology for modern cloud infrastructure [12]. Cloud providers such as Amazon Web Services and Alibaba Cloud deploy a wide range of virtualized network functions (VNFs), including load balancers, NAT gateways, and firewalls, on general-purpose CPUs running inside virtual machines (VMs) [32, 33, 37]. These VNFs serve millions of users across large-scale, multi-tenant data centers and are expected to sustain high throughput and low latency under diverse workloads.

Current NFV platforms typically adopt a per-core processing model [30, 31], where each CPU core independently handles a subset of network flows. This design choice mainly focused on avoiding 1) packet out-of-order [16, 24] and 2) resource contention [4, 17, 21]. However, this model is highly sensitive to traffic patterns. In the case of handling elephant flows, some cores may become overloaded while others remain underutilized [26, 35], forming single-core hotspots, resulting in degraded throughput, increased latency, and poor user experience.

To mitigate core imbalance, several intra-server load balancing techniques have been implemented. Hardware-based Receive Side Scaling (RSS) [22] uses static hash-based mapping to distribute flows across cores, but lacks responsiveness to workload dynamics and fails under elephant flow [26, 30, 47]. Userspace dispatchers [15, 25, 28, 29], such as those implemented with DPDK [36], allow dynamic flow migration based on runtime telemetry, but introduce high CPU overhead, engineering complexity, and packet reordering challenges. Kernel-space or hardware-assisted alternatives [21] often suffer from limitations in flexibility or generality. As a result, many production systems apply coarse-grained static mappings for simplicity, performance, and generality, at the cost of single-core hotspots.

These trade-offs leave a question: *how can NFVs achieve fine-grained load balancing toward CPU cores without sacrificing simplicity, performance, and generality?*

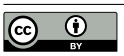
We observe that, in most cloud environments, traffic destined for VM-based VNF must first traverse a virtual switch (vSwitch) [3, 23, 40]. This creates an opportunity to offload intra-server load balancing to vSwitch, using its own compute and memory resources, before packets reach the VM.

In this paper, we present the motivation, design, and preliminary analysis of a vSwitch-assisted intra-server load balancing system. Our design performs balancing in a finer-grained manner, mitigates VNF hotspots, and is transparent to VNFs. Although still in the early stages, this system is being developed as part of our next-generation VNF system, with the goal of deployment in a major cloud provider.

2 BACKGROUND

2.1 VNF Architecture

VNFs enable network services in cloud environments by leveraging VM-based deployments on Virtual Private Cloud (VPC) infrastructure, supporting multi-tenancy, elastic resource allocation, and rapid scaling [12]. A large portion of them, like NAT gateways, load balancers, and firewalls, require maintaining session states for correct operation.



APNet 2026, August 06–07, 2026, Singapore, Singapore

ACM ISBN 979-8-4007-2664-4/2026/08
https://doi.org/10.1145/3820441.3820442

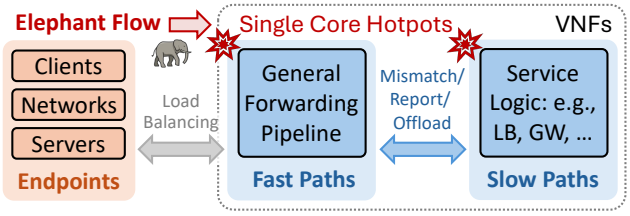
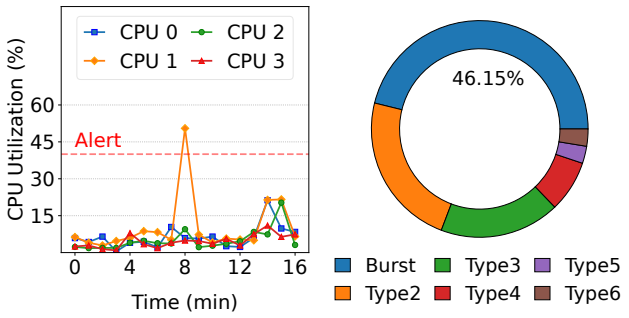


Figure 1: The fast and slow path processing architecture of VNF, and its potential single-core hotspots issue.



(a) CPU Hotspot Example (b) Alert Distribution

Figure 2: Example of single-core hotspot alerts and weekly alert breakdown in Alibaba Cloud's VNF system. Bursts indicate CPU hotspots caused by elephant flows. Once triggered, alerts initiate protective actions.

Major cloud providers employ a fast-slow path architecture [7, 9, 38] in VM deployment scenarios. The fast path handles the majority of packet forwarding and is data-intensive, and the slow path manages complex control logic and is compute-intensive. In this layered model, the fast path performs match-action processing using flow keys (e.g., 5-tuples). When a packet misses existing flow entries, it is redirected to the slow path through APIs, as shown in Figure 1. The slow path processes the packet and installs new rules back to the fast path for handling subsequent packets.

This separation improves modularity, scalability, and resource efficiency. However, despite advantages, VNFs still suffer from performance bottlenecks at finer granularity. We now turn to one such issue, single-core hotspots, that frequently arise in cloud-scale VNF deployments.

2.2 Single-Core Hotspot Problem

2.2.1 Per-Core Processing Model. Although modern VNFs are implemented on multi-core architectures, most still follow a flow-affinity execution model: packets belonging to the same session or flow are consistently steered to a single core within a VNF instance [6]. Flows are first distributed across VNF instances (e.g., via service-level load balancers). Then, within each instance, flow-level load balancing maps each flow to a fixed RX queue and consequently to a specific CPU core. This design choice is primarily driven by two factors:

- **Avoiding packet out-of-order** [16, 24]. In VM-based environments, maintaining in-order delivery when a flow is processed across multiple cores requires synchronization or software-based reordering. Such mechanisms consume significant CPU cycles and reduce overall resource availability, limiting the number of VNFs that can be co-located on a server and thereby increasing operational costs.
- **Avoiding shared-state contention** [4, 17, 21]. Since most of the VNFs maintain per-flow or per-session states, concurrent access from multiple cores can introduce synchronization overhead and degrade per-core performance.

Among these, the need to maintain packet order is the dominant concern. Software-level reordering across cores incurs nontrivial overhead, even outweighing potential gains from parallelism. As a result, the system favors isolating flow processing to individual cores, even at the cost of imbalanced CPU utilization under skewed traffic distributions.

2.2.2 Single-Core Hotspot Scenarios. Single-core hotspots are observed in both the fast and slow paths, mainly resulting from elephant flows. Fast path hotspots are more common due to their responsibility for handling the majority of packet processing. While the slow path usually handles only the first packets of each flow, under anomalous or adversarial conditions (e.g., SYN/FIN floods caused by application unintended behavior or attacks), high volumes of control-plane packets can also trigger CPU hotspots.

A typical case occurred in 2025 in Region A, where a fast-path instance exhibited a clear single-core hotspot, as shown in Figure 2(a). The instance was with five cores: one reserved for control tasks and four for data-plane processing. During this elephant flow event, one processing core (CPU1) reached 50% utilization, while the remaining three stayed below 15%. Given that the system is configured to raise alerts at 40% utilization (to avoid operational risk), this immediately activated a set of mitigation actions, including instance scale-out/up or sandbox isolation. The alert also triggered real-time operator intervention.

To understand the broader impact, we analyzed system-wide alert logs over a one-week period. As shown in Figure 2(b), alerts related to single-core hotspots accounted for 46.15% of the top-two severity-level incidents and occurred more than 2,500 times, comprising 12% of all alerts. These frequent and high-risk events impose significant operational overhead and require a large on-call team for real-time handling. Alleviating such hotspots, especially those caused by elephant flows, can significantly enhance service quality (e.g., lower jitter and latency), improve system stability, and reduce operational costs for cloud providers.

2.3 Intra-Server Load Balancing

A variety of mechanisms have been developed to address intra-server load imbalance. Static methods, such as RSS [22], hash packet headers (e.g., 5-tuples) to select indirection table entries, mapping flows to CPU cores while preserving flow affinity. However, such static schemes are agnostic to runtime traffic dynamics. When faced with elephant flows or traffic bursts, it can lead to CPU hotspots.

Dynamic solutions such as Shenango [25], Shinjuku [15], and Caladan [10] introduce centralized schedulers inside the VM to

enable runtime load redistribution. While these designs are effective in balancing traffic, they consume valuable CPU cycles, reducing the available compute budget for actual VNF processing. RingLeader [20] and isRSS [24] offload the scheduling logic to the NIC, reducing VM overhead and achieving better efficiency. Nonetheless, all these approaches fundamentally rely on task or flow migration, which cannot overcome a key limitation: the performance ceiling of a single CPU core. When a small number of flows dominate total throughput, migration alone cannot eliminate the bottleneck.

Another critical limitation of existing solutions is the lack of strong packet ordering guarantees. Systems like RSS++ [4] and isRSS [24] provide only best-effort ordering from the sender side and lack an in-network or end-host reordering mechanism. While Sprayer [30] makes a notable attempt by separating session-agnostic packets and distributing them across multiple cores, it compromises packet order and impacts application-level QoS, which remains a significant barrier for commercial deployment. Intel DLB [21] is a rare hardware-assisted solution that provides strict ordering semantics. However, it incurs high CPU overhead during runtime and is difficult to virtualize, limiting its applicability in multi-tenant cloud environments.

These limitations reveal a fundamental gap: existing designs are insufficient in addressing the single-core processing limit while preserving packet order. This motivates the need for a new approach that enables true per-flow parallelism with minimal overhead and strong ordering guarantees in cloud-native settings.

3 DESIGN PRINCIPLES

While the single-core hotspot appears as a local performance bottleneck, addressing it requires a holistic design of intra-server load balancing. We summarize six design principles that such systems should satisfy: (P_1) *Balanced CPU utilization*, ensuring traffic is evenly distributed across cores to maximize aggregate efficiency and avoid idle cores. (P_2) *High per-flow throughput*, tolerating elephant flows and bursty traffic without triggering single-core overloads to maintain stability and reduce operational alerts. (P_3) *Deterministic service quality*, preserving session integrity and in-order packet delivery with low latency overhead to avoid user-perceived performance degradation. (P_4) *Cost efficiency*, maximizing per-core processing capacity, avoiding resource contention, and without consuming CPU cycles allocated to VNFs. This allows more VNFs to be consolidated on the same server, reducing the cost. (P_5) *Generality and non-intrusiveness*, requiring no VNF modifications or VM cooperation to simplify deployment and maintain compatibility with existing network functions. (P_6) *Minimal offloaded resource usage*, leveraging SmartNICs or other co-processors in a lightweight manner to fit constrained compute/memory budgets shared across services.

4 PRELIMINARY DESIGN

Guided by the above principles, we present ParaFlowO, an efficient intra-server load balancing system that eliminates single-core hotspots by enabling per-flow parallelism with in-order delivery. Unlike designs such as Shinjuku [15] and RSS++ [4], which rely on runtime CPU load information for task migration, ParaFlowO achieves load balancing using dynamic core affinity duration adjusted by locally collected processing delay, without requiring cooperation from the VM. Compared to Sprayer [30], which parallelizes packets but causes out-of-order, ParaFlowO incorporates a lightweight reordering module to maintain packet order, thereby preserving application-level performance and user-perceived service quality.

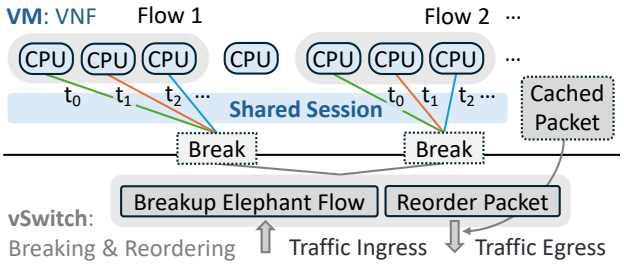


Figure 3: ParaFlowO architecture and data processing workflow.

As shown in Figure 3, ParaFlowO introduces two key modules inside the vSwitch: a *Breakup* module at ingress and a *Reorder* module at egress. The *Breakup* module detects elephant flows and splits them into flowlets based on time intervals, then dispatches them across different cores. The *Reorder* module reassembles packets in order based on sequence tags. An optional caching component allows metadata-only reordering to reduce vSwitch memory and bandwidth overhead. Below, we describe ParaFlowO's ingress and egress processes and discuss key design choices.

4.1 Ingress – Flow Breakup

The ingress pipeline consists of three major steps: switching logic, reorder queue & packet sequence number (PSN) allocation, and traffic scattering with parallel core control.

4.1.1 Basic Switching. Packets entering the vSwitch first undergo a standard switching pipeline, including functions such as basic packet validation, port lookup, and match-action rules for flow isolation. The ParaFlowO additionally checks the `para_flow` mode flag to determine whether a given flow should trigger the breakup operation for parallel processing. This mode is disabled by default. Upon detecting a potential elephant flow, either via tags from upstream detection systems (e.g., telemetry-based tagging) or vSwitch-side algorithms (e.g., recent byte-rate, packet-rate, or queue-occupancy thresholding), the system enables `para_flow` for the flow, marking it for breakup and reordering. Users and operators can also manually configure `para_flow` based on specific requirements.

4.1.2 Reorder Queue and PSN Allocation. For flows with `para_flow` enabled, ParaFlowO allocates a reorder queue based on the port ID and hash of the flow's 5-tuple, as illustrated in Figure 4. The number of available reorder queues is limited by the vSwitch's available resources. Each queue tracks a per-packet PSN, which is incremented with every arriving packet to capture intra-flow ordering. The queue ID and PSN are embedded as per-packet metadata, enabling multi-core processing while preserving the necessary information for reordering during egress.

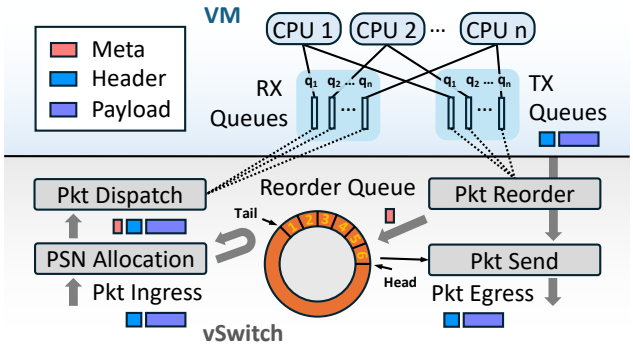


Figure 4: ParaFlowO's packet scattering and reordering pipeline. For `para_flow` mode enabled flows, the vSwitch assigns a reorder queue and PSN at ingress. Packets are dispatched to RX queues and later reassembled in order at egress based on the PSN.

4.1.3 Traffic Scattering. To enable parallel processing of elephant flows, ParaFlowO distributes packets across multiple CPU cores (P_2). It is a flowlet-inspired strategy: each flow stays assigned to a specific receive (RX) queue (CPU core) for a fixed time interval T_{switch}, before transitioning to the next queue. This creates coarse-grained flowlet-like units without relying on explicit inter-packet gap detection. As shown in Figure 5, in this example, each CPU is associated with a single RX queue. For each flow, the target RX queue is determined by an RX queue ID. This ID is initialized using a hash of the 5-tuple and incremented by 1 after each switch. The ParaFlowO then dispatches packets to the designated CPU core based on this ID via the RX queue. The RX queue is calculated from the k least significant bits of the ID and using the indirection table.

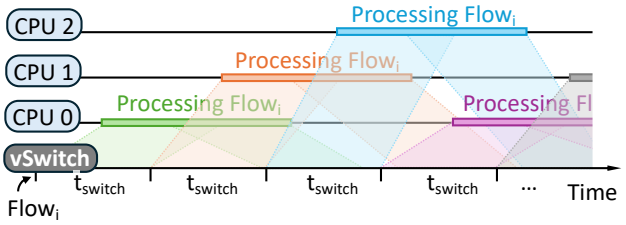


Figure 5: ParaFlowO periodically switches the target core of the flow after each time interval, enabling parallelism without overwhelming any single core.

To avoid performance degradation from excessive concurrency, ParaFlowO limits the number of active cores processing the same flow by adjusting the switching interval T_{switch} (P_4). This is achieved without coordination with the VM (P_5). ParaFlowO tracks the processing delay $T_{\text{cpu},i}$ of each core using timestamps embedded in the reordering queue, and computes the system-wide average delay T_{avg}. The baseline switching interval is then given by:

$$T_{\text{switch}} = \frac{T_{\text{avg}}}{N_{\text{para}} - 1} \tag{1}$$

This heuristic ensures (on a best-effort basis) that by the time a flow switches to the N_{para}-th core, the first core has completed processing earlier packets of this flow. This mechanism controls the maximum number of parallel CPU cores (N_{para}) allocated to a single flow, while periodic switching partially maintains the packet delivery order.

To mitigate imbalanced core utilization, ParaFlowO dynamically adjusts the switching interval of each core/queue (P_1). The switching interval is shortened for slower cores (higher $T_{\text{cpu},i}$), and lengthened for faster ones. We denote this adjusted switching interval for core i as $T^{(i)}_{\text{switch}}$. Intuitively, $T^{(i)}_{\text{switch}} \propto T_{\text{avg}}/T_{\text{cpu},i}$. To control the amplitude of adaptation and ensure system stability, we introduce a damping factor α and define the scaling coefficient $k_i = \alpha \cdot (T_{\text{avg}}/T_{\text{cpu},i})$:

$$T^{(i)}_{\text{switch}} = k_i \cdot T_{\text{switch}} = \frac{\alpha \cdot T^2_{\text{avg}}}{T_{\text{cpu},i}(N_{\text{para}} - 1)} \tag{2}$$

This adaptive mechanism allows ParaFlowO to distribute load proportionally across cores while reacting smoothly to traffic dynamics, ensuring stable and efficient intra-server load balancing.

4.2 Egress – Reordering and Caching

The egress pipeline consists of two key operations: packet reordering and optional packet caching.

4.2.1 Packet Reordering. After the VM processes a packet, Para FlowO extracts the metadata and uses the reorder queue ID and PSN to place the packet in the appropriate queue and slot (P_3), as shown in Figure 4. Once packets arrive in order or a timeout occurs, the head of the queue is released and the packet is forwarded via the standard switching pipeline. In this mechanism, VMs are not allowed to drop PSN-tagged packets directly. Instead, they mark them with a flag in the metadata. The vSwitch performs the actual drop while updating the reorder queue accordingly.

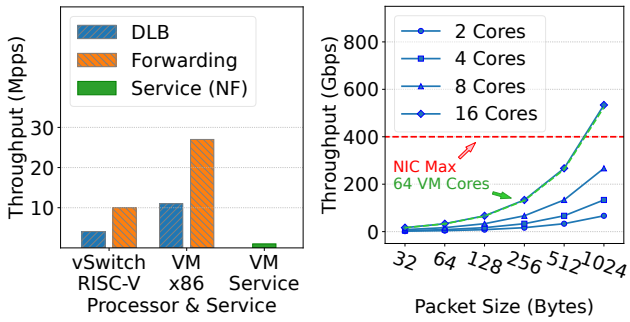
4.2.2 Packet Caching. To reduce memory and bandwidth overhead within the vSwitch, ParaFlowO supports lightweight packet caching (P_6), as shown in Figure 4. Instead of transferring full packets, ParaFlowO only fetches metadata for reordering. Once a packet is ready to be forwarded (either in order or after a timeout), the vSwitch releases it from the reorder queue, fetches the full packet from the VM, and transmits the packet.

5 PRELIMINARY EVALUATION

We perform a preliminary evaluation of ParaFlowO to validate its key design choice. Results show that reordering is not a bottleneck (Section 5.1), the scattering mechanism can balance loads and mitigate single-core hotspots (Section 5.2), and caching reduces memory overhead on vSwitch SoC (Section 5.3).

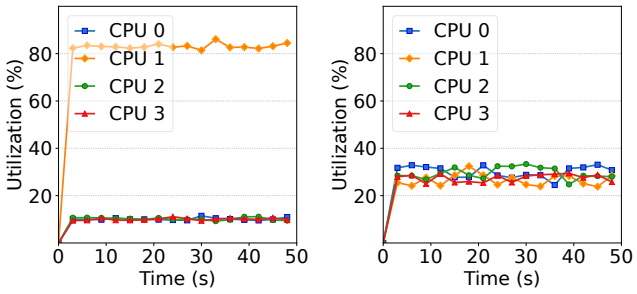
5.1 Processing Capacity and Bottleneck

To ensure that packet reordering does not become a system bottleneck, we evaluate the processing capacity of both the SmartNIC's RISC-V cores and the host x86 CPUs, and compare their throughput against typical in-production VNFs. Specifically, we aim to answer: Can the CPU cores on the vSwitch have sufficient throughput for reordering and forwarding tasks?



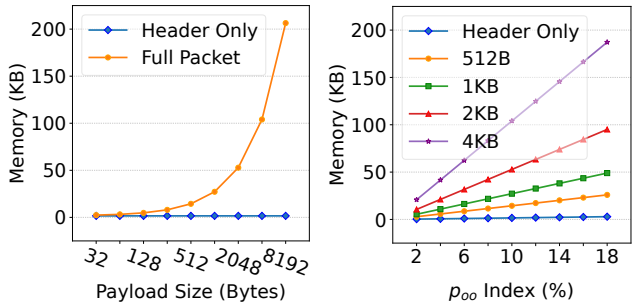
(a) Processing Capacity (b) Bottleneck Analysis

Figure 6: (a) Estimated throughput of forwarding, reordering, and service processing across RISC-V and x86 CPUs. (b) Projected SmartNIC throughput with increasing cores and packet sizes.



(a) RSS CPU Util. (b) ParaFlowO CPU Util.

Figure 7: Per-core CPU utilization under RSS and ParaFlowO. The elephant flow overwhelms a single core under RSS, while ParaFlowO distributes it across cores.



(a) Mem. vs Payload Size (b) Mem. vs P_{oo} Index

Figure 8: Memory usage comparison between fetching full-packet and metadata-only.

Figure 6(a) summarizes the per-core throughput of different operations across platforms. On the SmartNIC, a RISC-V core achieves 4.07 Mpps for reordering and 10 Mpps for simple forwarding, based on empirical calibration against x86 measurements. VM CPUs achieve 27 Mpps for forwarding and 11 Mpps for reordering per core. In contrast, in-production NFs such as firewalls or NAT gateways, process only around 1 Mpps per core on average. These results indicate that while reordering (whether implemented in software on the VM or offloaded to the SmartNIC) introduces nontrivial overhead, it still operates several times faster than typical NF workloads.

To understand system scalability, Figure 6(b) estimates the aggregate reordering throughput using 2, 4, 8, and 16 RISC-V cores under varying packet sizes (32–1024 bytes). Even under conservative assumptions, the total throughput exceeds 400 Gbps when using 16 cores and packet sizes larger than 800 bytes. This is already comparable to the line rate of high-end SmartNICs/DPUs like the BlueField-3 [23]. The figure also shows that 16 RISC-V cores can deliver reordering throughput on par with a 64-core VM running typical NFs.

These results show that the SmartNIC's RISC-V/SoC can conduct high-throughput reordering at line rate. Offloading reordering to the SmartNIC is unlikely to become the system bottleneck.

5.2 Balancing Load and Breaking Hotspots

A key challenge for intra-server scheduling is handling imbalanced traffic, especially when a single elephant flow is hashed to a core and saturates it, creating persistent hotspots.

We simulate a 4-core system over 50 seconds, consisting of one elephant flow (0.77 Mpps ±10%) and 100 small flows (2–6 Kpps). Each CPU core has a slightly different service rate centered around 1 Mpps. Under RSS, as shown in Figure 7(a), flow-to-core mappings are static. The elephant flow is pinned to a single core, quickly creating a hotspot (~80% utilization) while other cores remain underutilized (~10% utilization). As shown in Figure 7(b), ParaFlowO eliminates single-core hotspots and achieves more balanced per-core utilization (all cores are ~30% utilization).

These results prove that ParaFlowO can effectively address single-core hotspots. Moreover, this is achieved without centralized coordination and without compromising the flow state, as introduced in Section 4.1.

5.3 Memory Occupancy

SmartNICs' RISC-V cores typically have limited SRAM (128 to 512 KB), optimized for low-latency control-plane tasks. ParaFlowO reduces memory usage by only fetching *metadata* for reordering.

Let M denote the metadata size per packet, and P denote the total packet size. Given an out-of-order window size N and average out-of-order rate p_{oo}, the memory usage under full-packet buffering is $M_{\text{full}} = N \cdot p_{\text{oo}} \cdot P$. In ParaFlowO, only metadata is stored on the SmartNIC, where $M_{\text{meta}} = N \cdot p_{\text{oo}} \cdot M$. The resulting memory saving ratio becomes:

$$\text{Saving Ratio} = 1 - \frac{M_{\text{meta}}}{M_{\text{full}}} = 1 - \frac{M}{P} \tag{3}$$

This formula shows that the saving ratio grows with increasing packet size. For example, with $M = 64$ B, $N = 256$, $p_{\text{oo}} = 10\%$, and $P = 500$ B, full-packet fetching requires 12.5 KB, while metadata-only fetching consumes just 1.6 KB, achieving a 87.2% reduction. For 1500 B packets, the saving ratio reaches over 95.7%. Figure 8(a) illustrates how full-packet fetching scales linearly with payload

size, rapidly reaching about 200 KB. In contrast, metadata-only fetching remains constant regardless of packet size. Figure 8(b) further shows that while memory usage grows with the out-of-order rate p_{oo} in both designs, metadata-only fetching consumes orders of magnitude less memory.

These results confirm that separating metadata from payloads enables ParaFlowO to support efficient in-order delivery while staying within the strict memory constraints of SmartNICs.

6 DISCUSSION AND FUTURE WORK

Flow Identification. ParaFlowO needs heavy-flow identification to avoid spraying small flows. This problem is relatively mature, with many applicable approaches, including sketch-based methods, threshold-based counters, and ML-based detectors [2, 5, 34, 46]. Future work will explore algorithms that best match ParaFlowO's design principles.

Dispatch Policy. While it is possible to distinguish between co nnection-related (SYN/FIN/RST) and regular packets and scatter only the latter [30], in practice, session packets are redirected to the slow path. Furthermore, in some cases, connection packets themselves can become heavy hitters. Even for regular packets, VNFs maintain per-flow state (e.g., for metering or charging), and indiscriminate scattering introduces contention. Hence, ParaFlowO directly targets elephant flows to do the scattering.

ML-Based Balancing. ParaFlowO realizes balancing and contention control through a fixed policy, which may exhibit suboptimal efficiency in highly dynamic workloads (e.g., TCP Tahoe or Reno). Machine learning techniques, particularly reinforcement learning, offer dynamic policies, have shown promise in load balancing tasks [11, 39, 41, 44], and are proving deployable on network devices [42, 43, 45]. We plan to explore ML-based scheduling as a future enhancement.

Deployment Targets and Scenarios. ParaFlowO is designed to be general-purpose. We plan to test the design across a range of platforms, such as custom NICs with FPGA fast paths and SoC slow paths [3, 40], commodity SmartNICs/DPUs [1, 23], and software-based implementations [27]. These platforms already support high-throughput packet processing and lightweight datapath extensions, and have been used by prior systems with similar designs (e.g., Triton [19], Bifrost [8], and CStar Gateway [18]). Further exploration of detailed performance characteristics, as well as broader deployment scenarios, remains part of future work.

Application Generality. While ParaFlowO's scattering strategy is initially tailored for middlebox-like VNFs, its principles extend to end-host applications. Our operational experience indicates that TCP congestion control mechanisms show increased tolerance to out-of-order delivery in high-version kernels (e.g., Linux 5.x+), and mild packet reordering does not degrade application performance. In addition, ParaFlowO does not enforce strict equivalence to serial execution during packet processing. Instead, its core guarantee is user-visible per-flow in-order delivery: packets may be processed by multiple cores internally, but are released in order at egress through the reordering mechanism. This guarantee is sufficient for many production VNFs whose correctness depends primarily on in-order packet delivery rather than strict single-core execution. For VNFs or traffic classes that require strict per-flow serial-processing semantics (e.g., due to shared state updates), ParaFlowO can be conservatively configured by reducing the degree of parallelism, increasing the switching interval, or disabling `para_flow` entirely. In the extreme case, the system degenerates to the original flow-affinity model, preserving full serial-processing behavior. Exploring these diverse deployment scenarios is a direction for future work.

Tail Latency. By scattering heavy flows and balancing CPU loads, ParaFlowO can reduce tail latency. Our operational experience suggests that a subset of VNFs' applications (e.g., financial services [13, 14]) are sensitive to tail latency. Future work will further explore and evaluate ParaFlowO's design in reducing application latency and jitter.

7 CONCLUSION

This paper presents ParaFlowO, a practical intra-server load balancing solution that mitigates single-core hotspots in NFVs. ParaFlowO enables parallel processing of elephant flows by scattering flowlets across multiple cores while preserving packet order through a low-cost reordering mechanism. The design is transparent to VNFs, making it practical to deploy with good generalizability. Our preliminary evaluation demonstrates the feasibility of the design, validates the effectiveness of design choices, and suggests that ParaFlowO can support high-throughput, order-preserving processing with modest resource requirements. Future work will explore more accurate elephant flow detection, smarter scheduling, and integration across diverse hardware platforms and latency-sensitive workloads.

REFERENCES

[1] 2025. AMD Pensando DPU Technology: Front-end networking powering modern data centers. https://www.amd.com/en/products/data-processing-units/pensando.html. Accessed: 2025-07-08.

[2] Amer AlGhadhban and Basem Shihada. 2018. FLight: A Fast and Lightweight Elephant-Flow Detection Mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 1537–1538. https://doi.org/10.1109/ICDCS.2018.00161

[3] Amazon Web Services, Inc. 2024. AWS Nitro System. https://aws.amazon.com/ec2/nitro/. Accessed: 2025-07-08.

[4] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3359989.3365412

[5] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2017. Optimal elephant flow detection. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057216

[6] Yunfeng Bi, Hongjun Ni, Zhijun Tang, Dong Wang, Pan Zhang, Tao Zhu, and Mrittika Ganguly. 2023. *Intel Dynamic Load Balancer (Intel DLB) - Accelerating Elephant Flow.* Technical Report. Intel Corporation. https://builders.intel.com/docs/networkbuilders/intel-dynamic-load-balancer-intel-dlb-accelerating-elephant-flow-technology-guide-1677672283.pdf Network Builders University Technical Guide.

[7] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[8] Zihao Fan, Xing Li, Ye Yang, Bo Jiang, Bowen Yang, Yilong Lv, Yuke Hong, Yinian Zhou, Junnan Cai, Jiayue Xu, Yunrui Hu, Zhao Gao, Ke Sun, Yimin Liu, Xiangdong Zhang, Enge Song, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Haonan Li, Mingxin Li, Changgang Zheng, Yang Song, Jun Liang, Biao Lyu, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2026. Bifrost: Alibaba's Next-Generation VPC Network with High-Performance Multipath Reliable Transport. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association,

Renton, WA, 571–590. https://www.usenix.org/conference/nsdi26/presentation/fan

[9] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[10] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[11] Emre Gures, Ibraheem Shayea, Mustafa Ergen, Marwan Hadri Azmi, and Ayman A. El-Saleh. 2022. Machine Learning-Based Load Balancing Algorithms in Future Heterogeneous Networks: A Survey. *IEEE Access* 10 (2022), 37689–37717. https://doi.org/10.1109/ACCESS.2022.3161511

[12] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (2015), 90–97. https://doi.org/10.1109/MCOM.2015.7045396

[13] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. 2023. LOBIN: In-Network Machine Learning for Limit Order Books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*. 159–166. https://doi.org/10.1109/HPSR57248.2023.10147958

[14] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. 2024. Accelerating Machine Learning for Trading Using Programmable Switches. In *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2024)*. IOS Press, 3429–3436. https://doi.org/10.3233/FAIA240894

[15] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[16] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.* 37, 2 (March 2007), 51–62. https://doi.org/10.1145/1232919.1232925

[17] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2391229.2391238

[18] Haonan Li, Tian Pan, Jin Ke, Baohai Hu, Changgang Zheng, Enge Song, Zhi Xu, Ye Yang, Bowen Yang, Donglin Lai, Yisong Qiao, Bengbeng Xue, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Zihao Fan, Mingxin Li, Yang Song, Jun Liang, Xionglie Wei, Biao Lyu, Rong Wen, Zhigang Zong, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2026. CStar Gateway: Augmenting Public Cloud Infrastructure for Heterogeneous Network Function Virtualization. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 1501–1515. https://www.usenix.org/conference/nsdi26/presentation/li-haonan

[19] Xing Li, Xiaochong Jiang, Ye Yang, Lilong Chen, Yi Wang, Chao Wang, Chao Xu, Yilong Lv, Bowen Yang, Taotao Wu, Haifeng Gao, Zikang Chen, Yisong Qiao, Hongwei Ding, Yijian Dong, Hang Yang, Jianming Song, Jianyuan Lu, Pengyu Zhang, Chengkun Wei, Zihui Zhang, Wenzhi Chen, Qinming He, and Shunmin Zhu. 2024. Triton: A Flexible Hardware Offloading Architecture for Accelerating Apsara vSwitch in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 750–763. https://doi.org/10.1145/3651890.3672224

[20] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. 2023. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1293–1308. https://www.usenix.org/conference/nsdi23/presentation/lin

[21] Jiaqi Lou, Srikar Vanavasam, Yifan Yuan, Ren Wang, and Nam Sung Kim. 2025. Dynamic Load Balancer in Intel Xeon Scalable Processor: Performance Analyses, Enhancements, and Guidelines. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 664–678. https://doi.org/10.1145/3695053.3731026

[22] Microsoft Learn. 2024. Introduction to Receive Side Scaling. Microsoft Windows Hardware Documentation. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling Accessed: 2025-07-08.

[23] NVIDIA. 2021. NVIDIA BlueField-3 DPU Programmable Data Center Infrastructure on a Chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. Accessed: 2025-06-21.

[24] Andreas Oeldemann, Franz Biersack, Thomas Wild, and Andreas Herkersdorf. 2020. Inter-Server RSS: Extending Receive Side Scaling for Inter-Server Workload Distribution. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 46–53. https://doi.org/10.1109/PDP50117.2020.00014

[25] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[26] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3452296.3472889

[27] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The design and implementation of open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 117–130.

[28] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[29] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: core-aware thread management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 145–160.

[30] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. 2018. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 127–133. https://doi.org/10.1145/3286062.3286081

[31] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 24.

[32] Amazon Web Services. 2020. *5G Network Evolution with AWS*. Amazon Web Services, Inc. Accessed: 2025-07-08.

[33] Amazon Web Services. 2022. *Amazon EC2 Overview and Networking Introduction for Telecom Companies - Implementation Guide*. Amazon Web Services, Inc. Accessed: 2025-07-08.

[34] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer L. Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *SOSR 2017 - Proceedings of the 2017 Symposium on SDN Research (SOSR 2017 - Proceedings of the 2017 Symposium on SDN Research)*. Association for Computing Machinery, Inc, 164–176. https://doi.org/10.1145/3050220.3063772 Publisher Copyright: © 2017 ACM.; 2017 Symposium on SDN Research, SOSR 2017 ; Conference date: 03-04-2017 Through 04-04-2017.

[35] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, Jun He, Jinkui Mao, Zhengjie Luo, Chenhao Jia, Jiao Zhang, Tao Huang, Biao Lyu, and Shunmin Zhu. 2022. MIMIC: SmartNIC-aided Flow Backpressure for CPU Overloading Protection in Multi-Tenant Clouds. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940340

[36] The DPDK Community. 2025. Data Plane Development Kit Documentation. https://doc.dpdk.org. Accessed: 2025-07-08.

[37] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, Zhentao Zhang, Zikang Chen, Zeke Wang, Zihui Zhang, Shunmin Zhu, and Wenzhi Chen. 2023. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 769–782. https://doi.org/10.1145/3603269.3604859

[38] Tingting Xu, Bengbeng Xue, Yang Song, Xiaomin Wu, Xiaoxin Peng, Yilong Lyu, Xiaoliang Wang, Chen Tian, Baoliu Ye, Camtu Nguyen, Biao Lyu, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2024. CyberStar: Simple, Elastic and Cost-Effective Network Functions Management in Cloud Network at Scale. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 227–246. https://www.usenix.org/conference/atc24/presentation/xu-tingting

[39] Yue Xu, Wenjun Xu, Zhi Wang, Jiaru Lin, and Shuguang Cui. 2019. Load Balancing for Ultradense Networks: A Deep Reinforcement Learning-Based Approach. *IEEE Internet of Things Journal* 6, 6 (2019), 9399–9412. https://doi.org/10.1109/JIOT.2019.2935010

[40] Hang (Hangxian) Yang and Xinlu Yao. 2022. A Detailed Explanation about Alibaba Cloud CIPU. Alibaba Cloud Community Blog. https://www.alibabacloud.com/blog/599183 Accessed: 2025-07-08.

[41] Zhiyuan Yao, Zihan Ding, and Thomas Clausen. 2022. Multi-Agent Reinforcement Learning for Network Load Balancing in Data Center. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 3594–3603. https://doi.org/10.1145/3511808.3557133

[42] Kaiyi Zhang, Changgang Zheng, Nancy Samaan, Ahmed Karmouch, and Noa Zilberman. 2025. MUTA: Enabling Multi-Task Neural Network Inference in Programmable Data-Planes. In *2025 IEEE 26th International Conference on High Performance Switching and Routing (HPSR)*. 1–6. https://doi.org/10.1109/HPSR64165.2025.11038854

[43] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. In-Network Machine Learning Using Programmable Network Devices: A Survey. *Commun. Surveys Tuts.* 26, 2 (April 2024), 1171–1200. https://doi.org/10.1109/COMST.2023.3344351

[44] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. 2023. QCMP: Load Balancing via In-Network Reinforcement Learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing* (New York, NY, USA) *(FIRA '23)*. Association for Computing Machinery, New York, NY, USA, 35–40. https://doi.org/10.1145/3607504.3609291

[45] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Trans. Netw.* 32, 3 (Feb. 2024), 2555–2570. https://doi.org/10.1109/TNET.2024.3364757

[46] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2024. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *SIGCOMM Comput. Commun. Rev.* 54, 1 (Aug. 2024), 2–21. https://doi.org/10.1145/3687230.3687232

[47] Yan Zou, Tian Pan, Lu Lu, Zhiqiang Li, Kehan Yao, Tao Huang, and Yunjie Liu. 2023. P4RSS: Load-Aware Intra-Server Load Balancing with Programmable Switching ASICs. In *ICC 2023 - IEEE International Conference on Communications*. 1893–1898. https://doi.org/10.1109/ICC45041.2023.10279596