Accelerating Packet Processing in Container Overlay Networks via Packet-level Parallelism

Jiaxin Lei
Binghamton University
Binghamton, NY, USA
jlei23@binghamton.edu

Manish Munikar
The University of Texas at Arlington
Arlington, TX, USA
manish.munikar@mavs.uta.edu

Hui Lu
Binghamton University
Binghamton, NY, USA
huilu@binghamton.edu

Rao Jia
The University of Texas at Arlington
Arlington, TX, USA
jia.rao@uta.edu

***Abstract*—Overlay networks serve as the *de facto* network virtualization technique for providing connectivity among distributed containers. Despite the flexibility in building customized private container networks, overlay networks incur significant performance loss compared to physical networks (*i.e.*, the native). The culprit lies in the inclusion of multiple network processing stages in overlay networks, which prolongs the network processing path and overloads CPU cores. In this paper, we propose MFLOW, a novel packet steering approach to parallelize the in-kernel data path of network flows. MFLOW exploits *packet-level* parallelism in the kernel network stack by splitting the packets of the same flow into multiple micro-flows, which can be processed in parallel on multiple cores. MFLOW devises new, generic mechanisms for flow splitting while preserving in-order packet delivery with little overhead. Our evaluation with both micro-benchmarks and real-world applications demonstrates the effectiveness of MFLOW, with significantly improved performance – *e.g.*, by 81% in TCP throughput and 139% in UDP compared to vanilla overlay networks. MFLOW even achieved higher TCP throughput than the native (*e.g.*, 29.8 vs. 26.6 Gbps).**

***Index Terms*—Packet Processing, Kernel Network Stack, Container Overlay Networks**

I. INTRODUCTION

Due to high portability, high density, low performance overhead, and low operational cost, containers have quickly gained popularity and become adopted by high performance computing systems (HPC) [1]–[9]. Unlike VMs, containers achieve *lightweight* virtualization by running directly on the host operating systems (OS) – *i.e.*, *no* guest OSes and virtual hardware emulation involved – while isolation between containers remains enforced through kernel-level features such as namespaces [10], cgroups [11], and seccomp [12].

However, containers are *no longer lightweight* with regard to peripheral components, especially for networking. Recent studies [13]–[15] revealed that compared to the native (*i.e.*, no virtualization), containers achieved $\sim$50% less network throughput and suffered much higher packet-level processing latency. The culprit of the poor container network performance lies in the complexity of constructing network connections: Containers rely on *overlay networks* – the de facto network virtualization technique in containers – allowing each container to have its own network namespace and private IP address while being independent of the host network. The construction of overlay networks requires a set of software network devices, such as `VxLAN` [16] for packet encapsulation/decapsulation, `veth` for virtual network interfaces of containers, and virtual bridges (*e.g.*, Linux bridge or Open vSwitch [17]) to connect them. The involvement of multiple software network devices prolongs the data path of container network packets, inevitably incurring additional overhead and delays to packet processing with high CPU usage [13], [15].

Worse, since the Linux kernel typically squeezes all the processing stages of a single flow on a single CPU core [13], the computation of packet processing can easily overload the core, thus throttling the network throughput of the flow. This negatively impacts the performance and scalability of many HPC workloads, such as live HD streaming, distributed machine learning tasks, and big data processing tasks – typically generating *long-lived, high-throughput* flows, known as "elephant" flows. For example, due to such a CPU bottleneck, distributed machine learning tasks stopped scaling after only consuming 25 Gbps out of a 100 Gbps network link [18].

This paper investigates how and to which degree in-kernel packet processing can be optimized to accelerate container overlay networks. Ideally, the above-mentioned CPU bottleneck can be addressed/mitigated if we can effectively convert any elephant flow into multiple mouse flows, each containing a small portion of the flow's packets and being processed upon a separate core. Several instant benefits are: (1) Each mouse flow contains fewer packets, thus avoiding overloading a single core (even for a heavyweight network device); (2) Packets of different mouse flows can be processed in parallel, thus accelerating packet processing speed; (3) It can more efficiently leverage a multi-core system to mix and balance elephant and mouse flows – *i.e.*, an elephant flow is just equivalent to a bunch of mice flows.

To seek the feasibility of this idea, we design and develop **MFLOW** – a novel approach to parallelize *in-kernel* data path of (elephant) flows. MFLOW exploits fine-grained, *packet-level* parallelism based on an often overlooked fact: While existing in-kernel packet processing requires all packets of a single flow to be processed in a pipelined manner (in sequence), in-order packet processing does *not* need to be strictly guaranteed at all times along the *stateless* network path, but instead only when necessary (for the *stateful* path), *e.g.*, before packets enter the transport layer (*i.e.*, TCP) or are sent to user-space applications. Upon this observation, MFLOW achieves packet-level parallelism by splitting the packets of the same flow

into multiple small batches, called *micro-flows*, which can be processed *in parallel* on multiple cores. MFLOW devises generic packet steering mechanisms for *in-kernel flow splitting* that can be enabled at any point of the stateless network path.

One key challenge to MFLOW lies in that as each CPU core may have different processing capability and/or be interrupted by concurrent kernel tasks, packets of different micro-flows may not preserve their arrival order after parallel processing – *out-of-order* packet delivery causes incorrectness (in TCP) or poor user experiences (in UDP). This is precisely why the existing in-kernel network stack processes packets in order, thus only needing to reorder a small number of packets that are delayed during transmission. Although MFLOW can leverage the kernel's packet reordering mechanism to ensure all packets are still in order after parallel processing, the packet-level reordering incurs significant overhead. MFLOW addresses this issue in two ways: (1) by choosing a suitable batch size for micro-flows, the number of out-of-order packets can be dramatically reduced; (2) instead of reordering packets at a per-packet level, MFLOW devises a batch-based flow reassembling mechanism incurring *little* overhead.

We know of no other kernel techniques supporting packet-level parallelism for accelerating container overlay networks. We have implemented a prototype of MFLOW in the Linux network stack (with kernel version 5.7). To summarize, in this paper, we have made the following contributions:

- We perform a detailed investigation of the performance of container overlay networks and identify the main performance bottleneck for elephant flows to be the lack of sufficient network processing parallelism.
- We design and implement MFLOW, which explores packet-level packet processing parallelism in commodity OS kernel for fast overlay networks. Unlike existing approaches that only parallelize packet processing at a coarse-grained flow/device level, MFLOW allows a flow to be parallelized at any stateless stage along the network processing pipeline.
- Our evaluation of MFLOW using both micro-benchmarks and real-world applications shows that MFLOW can significantly improve network throughput (*e.g.*, by 81% in TCP and 139% in UDP compared to the vanilla overlay networks) and application-level performance (*e.g.*, by up to 7.5x for web serving). MFLOW even achieves higher TCP throughput under container overlay networks than the native (*e.g.*, 29.8 vs. 26.6 Gbps) due to packet-level processing parallelism.

Road map: Section II discusses the background and motivates MFLOW with performance and CPU utilization comparisons among state-of-the-art overlay network techniques. Section III presents the design details of MFLOW while Section IV releases its implementation. Section V shows the experimental results in comparison with state-of-the-art. Section VI reviews related works and Section VII concludes the paper with a brief discussion of future work.

II. Background and Motivation

A. Background

Packet processing: In-kernel packet processing, as illustrated in Figure 1, involves a complicated pipeline that traverses the physical network interface controller (`pNIC`), the kernel space, and the user space. We use packet reception as an example to demonstrate the process: When a packet arrives at the `pNIC`, in step ❶, it is copied (via DMA) to the kernel *ring buffer*, and the `pNIC` triggers a hardware interrupt (IRQ). The kernel is then invoked by the IRQ and starts the packet receiving process. The in-kernel receiving procedure further involves two parts: the top half and the bottom half.

The top half runs in the context of the IRQ, which simply marks that there is an incoming packet (in request queues) waiting for processing and notifies the bottom half (*i.e.*, by raising a software interrupt). The bottom half is then executed in the form of a software interrupt (softirq) (in step ❷). It serves as the main kernel network packet processing routine to process the packet through a set of network devices (*e.g.*, both physical and software NICs) and network protocol layers (*e.g.*, from layer 2 to layer 3/4). The Linux kernel uses a key data structure, `skb` (i.e., socket buffer), to represent each packet that can be freely manipulated and transferred across these network devices and layers. After a packet traverses all needed network devices and protocol layers along its path, it is finally delivered to the user-space application (in step ❸) — *i.e.*, the packet data/payloads (stored in the kernel ring buffer) is copied from the kernel buffer to the user-space application's buffer.

Container overlay networks: Container overlay networks hinge on a tunneling technique (*e.g.*, through `VxLAN` [16]): When a container sends a packet (with private IPs), the overlay network encapsulates the packet in a new packet with the (source and destination) host IPs as the new packet header and the original packet as payload. When a container receives a packet, the overlay network decapsulates the received packet to recover the original packet and delivers it to the target containerized application using its private IP address.

As illustrated in Figure 2, the Linux kernel constructs the container overlay network with the help of several in-kernel software network devices – *i.e.*, a `VxLAN` network device for packet encapsulation/decapsulation, a virtual Ethernet device (`veth`) for virtual network interfaces of containers, and a virtual bridge (*e.g.*, Linux bridge or Open vSwitch [17]) to connect them. Hence, before a container packet is received by the user-space application, it needs to traverse *three* software devices and goes through the network protocol stacks *twice* — one for packet decapsulation and one for sending the decapsulated packet (by `veth`) to the user-space application. Throughout the whole process, one IRQ and three softirqs — *i.e.*, by `pNIC`, `VxLAN`, and `veth` — are raised. Therefore, compared to the native, the overlay network incurs prolonged data path with extra processing overhead.

Parallel packet processing: The prolonged data path in container overlay networks slows down per-packet processing and consumes more CPU cycles. By default, as the vanilla

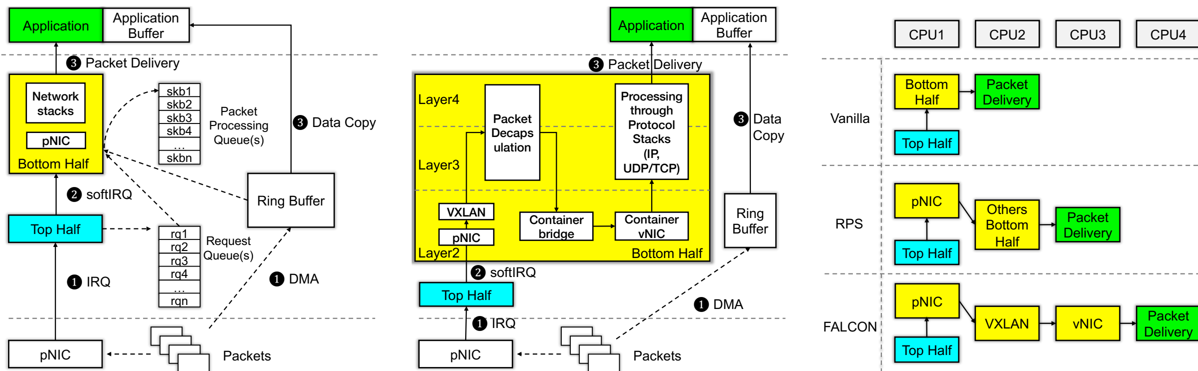


Fig. 1: In-kernel packet processing. Fig. 2: Container overlay network. Fig. 3: Parallel packet processing.

case shows in Figure 3, the Linux kernel squeezes all stages of a single flow's packet processing onto a single CPU core [1]. It is because the Linux network stack has been developed over the years and originally targeted less-powerful network devices (*e.g.*, 1/10 Gbps) where a single core was powerful enough to handle a single network flow. However, in the face of today's high-performance, high-throughout network devices (*e.g.*, 100/400 Gbps), the CPU becomes the bottleneck – *i.e.*, packet processing can easily saturate a single core, preventing a single flow from achieving higher network throughput.

To leverage a multi-core system, both hardware and software *packet steering* approaches have been proposed to parallelize packet processing:

(1) Modern physical NICs enable multiple queues and apply receive side scaling (RSS) [19] to map different flows to separate cores (via hash values). This achieves *inter-flow* parallelism as different flows are associated with distinct hash values and can be mapped to different cores. Note that, it is common that one server can have more flows than available CPU cores; multiple flows might still be mapped to the same CPU core. The hardware-based parallelism mechanism, however, does *not* parallelize a single (elephant) flow, as all packets from the same flow are assigned with the same hash value and hence processed on the same core.

(2) Receive packet steering (RPS) [20] in the Linux kernel is a software implementation of RSS, which realizes packet steering in the context of the first softirq (raised by `pNIC`'s IRQs) and again achieves *inter-flow* parallelism – *i.e.*, each flow is identified using a distinct hash value and mapped to a separate core. As the "RPS" case shows in Figure 3, for a single flow, RPS only separates the "top half" (as well as the first softirq) and the remaining "bottom half" onto two cores.

(3) Recent effort, FALCON [13], observed the lack of single-flow parallelization and enabled *device-level* and *function-level* parallelization for a single flow. As the "FALCON" case shows in Figure 3, packet processing stages associated with distinct network devices (`pNIC`, `VxLAN`, `vNIC`, etc.) can be distinguished and placed on separate cores by FALCON. However, one limitation of FALCON lies in that if a network device is heavy (*e.g.*, `VxLAN`), it can still saturate one CPU core and becomes the bottleneck. Further, the processing of a network packet in FALCON spans across multiple CPU cores, resulting in reduced data locality and extra queuing delays. Last, function-level parallelization in FALCON seems hard-coded and requires in-depth kernel code analysis.

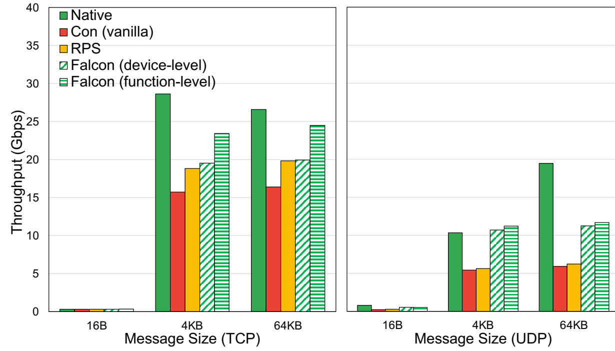
[1]The kernel thread for *packet delivery* – *i.e.*, copying data from the kernel ring buffer to the user-space buffer – is bonded with the core where the application thread runs; it can run on a separate core other than the in-kernel packet processing core(s).

B. Motivation

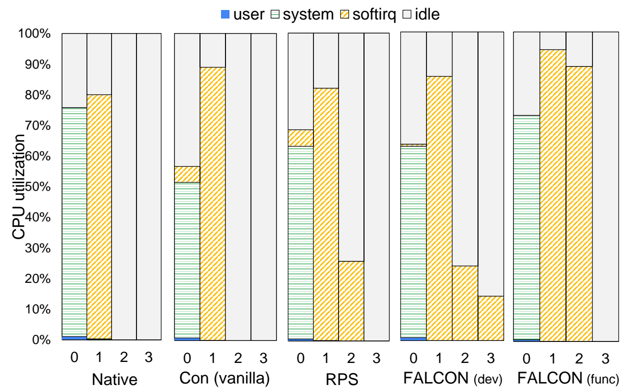
Experimental settings: To quantitatively analyze the effectiveness of existing parallel packet processing approaches, we evaluated the throughput and CPU utilization of the `VxLAN`-based overlay network using `sockperf` [21] (*i.e.*, a TCP/UDP traffic generator) between a pair of client and server machines. The machines were connected with Mellanox ConnectX-5 EN 100-Gigabit Ethernet adapters. Both the client and server had sufficient CPU and memory resources. More details of the configurations are presented in Section V.

Performance analysis: Figure 4 depicts the performance and CPU utilization comparisons between the native (*i.e.*, no containers), `VxLAN`-based container overlay network, RPS [20], and FALCON [13] using a single flow. We enabled the Linux kernel's default RPS mechanism. We downloaded FALCON's source code from its Github repository [22] and deployed its two parallelization approaches – at the device or function level.

Compared to the native, container overlay networks incurred higher performance overhead with significant performance drops – 40% for TCP and 80% for UDP under large message sizes (*e.g.*, 64 KB). The main reason is that: (1) container overlay networks entail prolonged data path with more software network devices as shown in Figure 2; (2) the Linux kernel by default places all packet processing of these devices on a single core, which easily overloads the core as indicated in Figure 4b (the container vanilla case) – softirqs of all network devices overloaded *core one* (close to 100%). Note that, Figure 4b shows average CPU utilization (*e.g.*, over 30 seconds). Although the average CPU% is under 100%, instant peak CPU% could reach 100% and throttle the performance, preventing a single flow from achieving higher throughput.



(a) Throughput under TCP/UDP.



(b) CPU utilization on separate cores (TCP with 64KB).

Fig. 4: Performance and CPU utilization comparisons between native, container, and parallel optimizations.

Compared to the vanilla overlay case, RPS slightly improved the throughput of container overlay networks – by 6% for UDP and 24% for TCP under large message sizes (*e.g.*, 64 KB). It is because, as shown in Figure 4b (the RPS case), RPS steered part of the softirqs from *core one* to *core two*, making *core one* capable of serving more packets. However, *core one* remained the bottleneck with high CPU usage, as the heavyweight network device – `VxLAN` (*i.e.*, part of the first softirq) – were still processed on core *one*.

To mitigate this, FALCON [13] distinguished different network devices and dispatched them onto separate cores, namely the *device-level* pipelining. As the example in Figure 4 shows, FALCON dispatched `VxLAN` to *core two* and placed the remaining devices on *core three*. In this way, FALCON increased the UDP throughput of container overlay networks significantly — by 80% (compared to vanilla overlay). However, it was still far below the native (only within 30%), because the device-level pipelining is still coarse-grained — *i.e.*, a heavy device/function can still saturate a single core.

Worse, the device-level pipelining merely worked for TCP with similar performance as RPS (in Figure 4a). The reason is that, under TCP, heavyweight functions – *e.g.*, per-packet `skb` allocation and generic receive offload (`GRO`) [2] – remained on *core one* and overloading it, as depicted in Figure 4b (*i.e.*, the FALCON-dev case). To overcome this, the *function-level* pipelining in FALCON can further separate these functions onto separate cores. For example, by dispatching the `GRO` function (and all the following softirqs) on *core two*, FALCON increased the throughput of TCP – by 20% (compared to RPS). Meanwhile, *core one* again was overloaded – now purely by the `skb` allocation function (*i.e.*, the FALCON-fun case in Figure 4b) that cannot be parallelized by FALCON or any existing approaches.

Summary: Overlay networks incur non-trivial performance overhead for both TCP and UDP. State-of-the-art approaches can parallelize packet processing to a certain degree but encounter new bottlenecks. Hence, the performance of container overlay networks remains significantly lower than the native.

[2] `GRO` reassembles small packets into larger ones to reduce per-packet processing overhead. We observed that the Linux kernel's `GRO` is mainly effective for TCP connections but not for UDP.

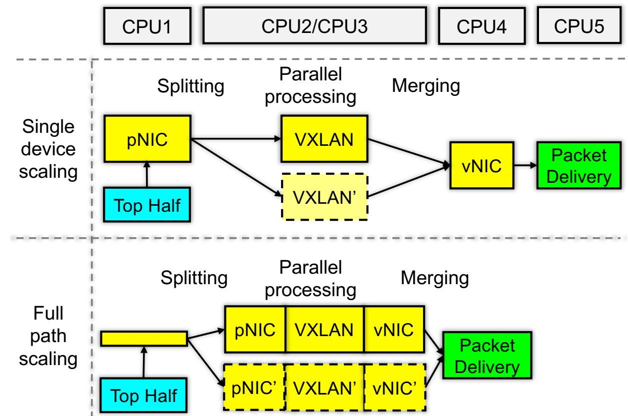


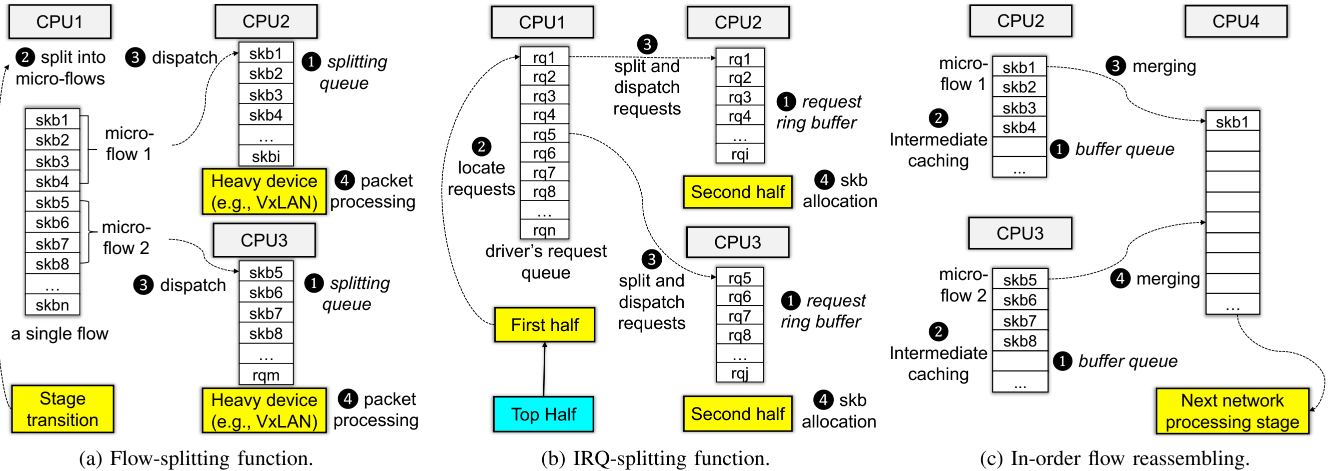
Fig. 5: MFLOW achieves single device scaling or full path scaling via exploiting packet-level parallelism.

III. DESIGN OF MFLOW

To exploit in-kernel packet processing parallelism, we design and develop **MFLOW** with the key ideas as follows: Instead of following the long-established pipelined, in-order processing, MFLOW exploits *packet-level* parallelism by splitting packets of the same flow into multiple small batches, called *micro-flows*, each being able to be processed on a separate core, called *splitting cores*. By doing this, multiple micro-flows of the same flow can be processed *in parallel* along the stateless network path and only reassembled before entering the stateful processing stage or user-space applications. As depicted in Figure 5, MFLOW can scale a heavyweight network device or even the full network path for a single flow. In the following sections, we present MFLOW's splitting mechanisms (in Section III-A) and how MFLOW efficiently preserves in-order packet delivery (in Section III-B).

A. Flow Splitting

MFLOW does not re-design existing well-tested, mature kernel network stack, but instead realizes novel packet steer-



(a) Flow-splitting function. (b) IRQ-splitting function. (c) In-order flow reassembling.

Fig. 6: Design of MFLOW: (a) Flow-splitting function; (b) IRQ-splitting function; and (c) In-order flow reassembling.

ing mechanisms to exploit packet-level parallelism. MFLOW devises two generic *mechanisms* for in-kernel flow splitting – *i.e.*, depending on whether the per-packet `skb` data structure is created or not. These splitting mechanisms enable MFLOW to either split a flow at a very early stage (*i.e.*, right after the first IRQ) or at any point along the stateless network processing path (*i.e.*, layer 2/3 and UDP layer).

Splitting mechanism along stateless network path: MFLOW splits a single flow by leveraging in-kernel *stage transition functions*. Specifically, during packet processing, a network packet – represented in the form of a *skb* data structure – is transferred from one processing stage (*i.e.*, a network device) to another via a stage transition function (*e.g.*, `netif_rx`). The stage transition function enqueues the packet (*i.e.*, `skb`) into the queue of the device to be processed next on the same core. In this way, stage transition functions multiplex multiple stages of the flow in a pipelined manner on the same core – *i.e.*, once scheduled, each stage can process a batch of packets; stages are processed in an interleaved manner.

MFLOW re-purposes the stage transition functions into a *flow-splitting function* for heavyweight network devices (in Figure 6a): During network device initialization, for any network device (*e.g.*, `VxLAN`) that needs the packet-level parallelism, MFLOW creates per-core, per-device *splitting queues* (❶). During packet processing, before any identified (elephant) flow enters the heavyweight network device, MFLOW divides the packets of the flow into multiple small batches (❷). Each batch is called a *micro-flow* and covers a portion of the consecutive packets in the original flow. Then, MFLOW can select a distinct *splitting core* for a micro-flow and enqueues the packets of the micro-flow into its target core's splitting queue (❸). Meanwhile, a softirq is raised on the target splitting core via inter-processor interrupt (IPI). In this way, the bottom half of the network device will be executed later on all the involved splitting cores in parallel (❹).

This *flow-splitting function* works upon the per-packet `skb` data structure and can parallelize the processing of any stateless heavyweight network devices (or functions, *e.g.*, `GRO`). However, similar to the "FALCON-func" case in Figure 4, after MFLOW scales the heavyweight `VxLAN` device in container overlay networks via the flow-splitting function, we observed that the construction of the `skb` data structure (in the first stage of packet processing) became a heavy process – *i.e.*, it overloaded a single core. To scale these heavyweight functions, we need a flow splitting mechanism that works at the earliest point of the network stack:

Splitting mechanism for the first stage: Splitting the packets of a flow before `skb` allocation is challenging due to two factors: (1) It requires the support of the physical network device driver to locate raw packets. (2) As there is *no* `skb`, it needs a lightweight data structure to represent each raw packet, thus being able to dispatch them onto separate cores. To overcome these, MFLOW devises an *IRQ-splitting function* to split/parallelize packet processing at the first stage:

As depicted in Figure 6b, during the initialization of a flow that needs first stage parallelization, MFLOW creates per-core *request ring buffers* on the splitting cores that will parallelize the first stage processing (❶). Then, the IRQ-splitting function divides the first stage – *i.e.*, the softirq context of the `pNIC` – into two halves. The first half (1) locates the incoming packet *requests* from the driver's request queue (❷) – *e.g.*, each request represents an incoming packet and contains information for the `skb` creation; (2) dispatches the requests onto target cores (❸) – similar to the above micro-flow based dispatching [3]; and (3) raises softirqs on target splitting cores (via IPIs). Finally, the second half will be invoked on the splitting cores to finish the remaining part of the original first stage – *e.g.*, `skb` allocation (❹). With this, MFLOW can split and scale heavyweight functions at the earliest network software point by taking advantage of multiple cores. Note that, the design of the IRQ-splitting function relies *little* on

[3]Note that the IRQ-splitting function dispatches packet requests rather than `skbs`; it relies on the data structure of packet requests, created by device drivers, to represent each raw packet, hence being lightweight.

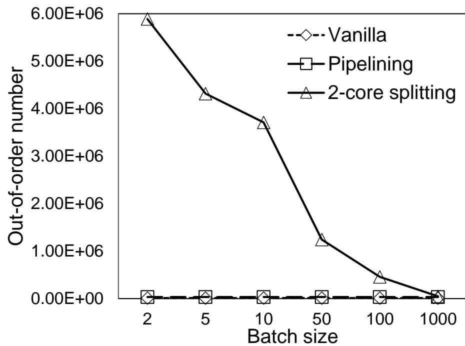


Fig. 7: Number of out-of-order packet delivery vs. batch size of micro-flows (TCP with 64KB packets).

a specific network device driver – *i.e.*, it only needs to know the driver's request queue and the way to locate its requests – making it portable to different network devices.

Parameters for packet-level parallelism: The degree of packet-level parallelism in MFLOW is mainly determined by: (1) the number of outstanding packets; (2) the batch size of micro-flows; and (3) the number of splitting cores. We discuss the implications of each parameter as follows:

For both TCP and UDP workloads, a number of *outstanding* packets could arrive at the receiver side approximately at the same time, especially for elephant flows. For example, given a TCP connection under the throughput of $\sim$ 30 Gbps, the sender (*e.g.*, iperf3 [23]) can issue $\sim$2,000 outstanding packets (with the size of MTU being 1,500 bytes) without receiving an ACK from the receiver. As there is no acknowledgment mechanism in UDP, a sender theoretically can issue as many outstanding packets as possible to the receiver.[4] As the outstanding packets arrive at the receiver approximately at the same time, dispatching them onto multiple cores enables packet-level parallelism. Therefore, the "heavier" a flow is, the more outstanding packets it produces and the higher the packet-level parallelism degree can be exploited.

Simply dispatching the outstanding packets of the same flow onto multiple cores may cause out-of-order delivery as different cores may not have a uniform processing speed. Though MFLOW's flow reassembling mechanism (detailed in Section III-B) eventually preserves packet orders, more out-of-order delivery means additional effort for order preservation.

We observed that, in Figure 7, the number of out-of-order delivery after splitting reduced significantly as the batch size of micro-flows increased. When the batch size was set to 256 or above, *little* overhead was incurred for packet order preservation in MFLOW. Having a large batch size also preserves optimizations in packet processing. For example, GRO reassembles small packets into larger ones, thus reducing the number of packets to be processed. GRO can merge more consecutive small packets given a larger batch size. Batch size also has implications on load distribution: If all micro-flows have the same batch size and MFLOW evenly distributes them on multiple splitting cores, CPU utilization of each core would be similar (as packets go through similar processing).

Ideally, MFLOW can leverage as many cores as possible to exploit packet-level parallelism. However, in practice, the performance benefit may diminish as the core number increases due to multiple factors, such as the number of outstanding packets, batch size, queuing delay, and reassembling overhead. Our evaluation (in Section V) shows that using two cores for parallel packet processing greatly accelerates container overlay networks performance – *e.g.*, even higher than the vanilla native case. Further, as the original packet processing bottleneck has been mitigated by MFLOW, a new bottleneck arises due to data copying from the kernel to the user-space application. We will discuss this issue in detail in Section V.

B. *Flow Reassembling*

A key design goal of MFLOW is *not* to involve out-of-order packet delivery due to MFLOW's splitting mechanisms and parallel processing. We note that splitting a single flow into micro-flows ensures that packets in each micro-flow naturally preserve their arrival orders. However, since each core may have different processing capability and/or be interrupted by other concurrent kernel tasks, packets of different micro-flows may not preserve their arrival orders after parallel processing.

To preserve the original sequences of micro-flows, MFLOW devises an efficient *batch-based* flow reassembling mechanism. As depicted in Figure 6c, for heavyweight network devices (or functions) that need packet-level parallelism, MFLOW creates per-core, per-device *buffer queues* (❶). Then, for each splitting core that finishes the processing of a packet, it enqueues the packet to its buffer queue (❷), instead of directly sending it to the next processing stage. Meanwhile, each micro-flow is associated with an identifier which is incremented based on the position of the micro-flow in the original flow [5]. In other words, the ID reflects each micro-flow's order in the original flow. MFLOW uses a global *merging counter* to keep track of the ID of the micro-flow being merged. To merge a micro-flow, MFLOW (1) locates the buffer queue that stores the packets having the ID same as the *merging counter*; (2) fetches the packets from the buffer queue; and (3) sends them to the next processing queue/stage (❸ and ❹). MFLOW keeps consuming packets from the same buffer queue until the next packet stores a different ID than the *merging counter*, indicating that MFLOW should move to consume the next micro-flow. After MFLOW increments the *merging counter*, it repeats step (1).

MFLOW's batch-based flow reassembling approach has the following advantages: (1) The per-core, per-device buffer queues (used to cache intermediate micro-flows) ensure that each core can keep processing packets without being blocked by the merging process. (2) Packets are "re-ordered" on a per-batch basis, which is extremely efficient, especially compared to the kernel's existing per-packet reordering mechanism using

[4]Practical UDP workloads implement congestion control upon the UDP protocol, which adjusts sending rate based on the observed quality of service such as packet loss, delay, jitter, etc.

[5]MFLOW stores the ID information in each packet's `skb` data structure.

an out-of-order queue. It also indicates that using a large batch size can significantly reduce merging overhead – *i.e.*, mFlow does not need to frequently switch between buffer queues to locate the next micro-flow.

Note that, although it makes intuitive sense to merge micro-flows right after a heavy device/function and before the next processing stage, we find that micro-flows can actually be merged as late as possible as long as the following packet processing is stateless (*i.e.*, no inter-packet processing dependency). For example, for UDP flows, micro-flows can be merged right before being delivered to user-space applications. The advantages for the late merging are as follows: (1) mFlow can reuse existing in-kernel backlog queues [6] as buffer queues with reduced queuing delay. (2) mFlow can parallelize the full packet processing path with fewer splitting cores (in Figure 5). (3) Packets are being processed on the same core with good data locality.

IV. Implementation

We have implemented mFlow on the Linux network stack with kernel version 5.7 (∼600 LoC of addition or modification) with the focus on the presented splitting and reassembling mechanisms as stated in Section III. mFlow is available at: `https://github.com/jlei23/mflow.git`.

Flow-splitting function: mFlow implements the flow-splitting function by re-purposing a state transition function, `netif_rx`. Originally, such a state transition function enqueues a packet (*i.e.*, its `skb`) into the current core's *backlog queue* for future processing on the *same* core. mFlow, instead, splits received packets of a flow – that requires packet-level parallelism for a heavy network device – into micro-flows (❷ in Figure 6a) and enqueues each micro-flow's packets onto one selected splitting core (❸ in Figure 6a). mFlow creates and associates the per-core, per-device splitting queues to the device's NAPI structure `napi_struct` (❶ in Figure 6a), which can be easily accessed by the network device's softirq handler once executed on the splitting cores (❹ in Figure 6a).

IRQ-splitting function: mFlow implements the IRQ-splitting function in the Mellanox NIC driver – its softirq handler (`mlx5e_napi_poll`). The IRQ-splitting function relies on two inputs from the driver code: a request queue (`mlx5e_rq`), and the way to retrieve requests (`mlx5e_poll_rx_cq`) (❷ in Figure 6b). With this, mFlow, once enabled, can retrieve any available incoming packet requests in the context of the physical NIC's softirqs and dispatch them onto selected splitting cores (❸ in Figure 6b). mFlow creates and associates the per-core request buffer to Linux kernel's per-core data structure, `softnet_data`, which can be easily accessed in a softirq context (❶ in Figure 6b). mFlow implements the second half (❹ in Figure 6b) as a regular softirq handler (scheduled by kernel's NAPI scheduler and executed on the splitting cores). The second half can be processed in parallel most of the time except when it needs to update the driver that a packet request has been consumed (*i.e.*, after its `skb` has been created) and can be released (*i.e.*, can be reused for another incoming request). To reduce any possible contention, mFlow updates the driver once in a while (*e.g.*, every 128 requests).

Flow reassembling: The implementation of batch-based flow reassembling uses two queues – the *backlog* queue for receiving packets from the previous network processing stage and the *receive* queue for delivering packets to user-space applications. Under UDP, `sk_receive_queue` serves as the backlog queue, while `reader_queue` serves as the receive queue. Under TCP, `sk_backlog` serves as the backlog queue, while `sk_receive_queue` serves as the receive queue. mFlow extends the backlog queue into per-core buffer queues (❶ in Figure 6c), with each serving one splitting core. Thus, all packets from the previous stage are first cached in the buffer queues (❷ in Figure 6c) before merging. mFlow does not create a new kernel thread for executing the merging functionality (Section III-B). Instead, it adds the merging functionality in the existing kernel thread for packet delivery, *i.e.*, `tcp_recvmsg` for TCP and `udp_recvmsg` for UDP (❸ and ❹ in Figure 6c). These threads will be woken up when new packets arrive, during which mFlow checks which micro-flow's packets should be merged.

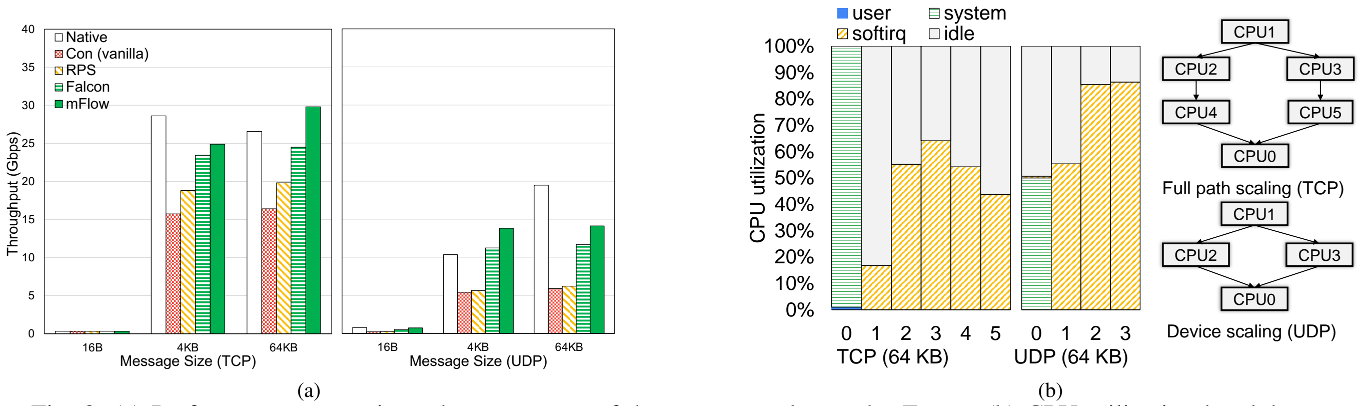
V. Evaluation

We have evaluated the effectiveness of mFlow. Results with micro-benchmarks demonstrate that: (1) mFlow significantly improves the throughput of an elephant single flow – by 81% for TCP and 139% for UDP compared to vanilla overlay networks; (2) mFlow achieves even higher throughput than the native under TCP (29.8 vs. 26.6 Gbps); (3) mFlow reduces average and tail latency for both TCP and UDP. Results with real-world applications demonstrate significant application-level performance benefits brought by mFlow– the performance of a web serving application increases by up to 7.5x, while the latency of a data caching application reduces by up to 48%, compared to vanilla overlay networks.

Experimental configurations. The experiments were performed on two PowerEdge R740XD servers, each with 2×16-core Intel Xeon Gold 5218 processors (2.30 GHz) and 384 GB memory. The two machines were connected directly by Mellanox ConnectX-5 EN 100-Gigabit Ethernet. We used Ubuntu 20.04 (with the kernel version 5.7) as the host OSes and the Docker overlay network mode (with Docker version 19.03) as the container overlay network. Docker overlay network uses Linux's builtin VxLAN. We evaluated the following cases: (1) *native*: the physical host network (i.e., no containers); (2) *vanilla overlay*: containers with the default docker overlay network (VxLAN); (3) *RPS*: containers with Linux RPS [20] enabled; (4) Falcon: containers with FALCON [22] enabled – the state-of-the-art in-kernel parallelization optimization for container networks; and (5) mFlow.

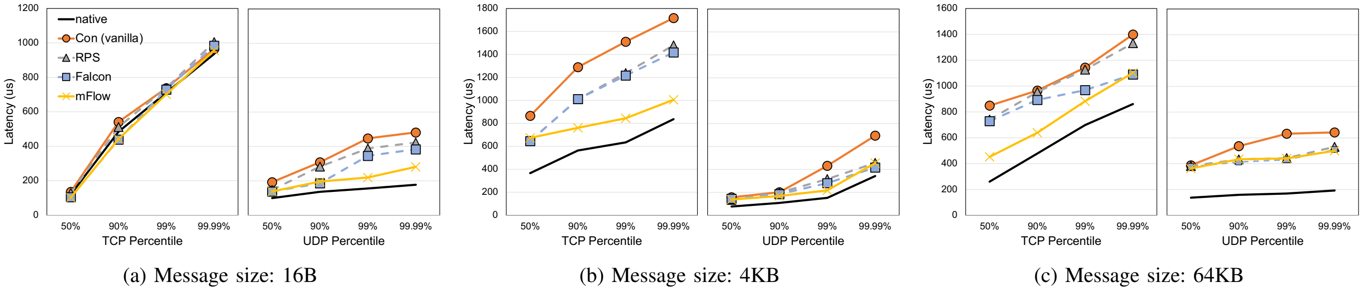
For mFlow, unless otherwise specified, we set the batch size to 256 and the number of splitting cores to 2, evenly

[6] In delivering packets to a user application, the kernel uses a backlog queue to store packets temporarily while the receive queue is being used by the application's receiving thread.



(a) (b)

Fig. 8: (a) Performance comparisons between state-of-the-art approaches and MFLOW. (b) CPU utilization breakdown.



(a) Message size: 16B (b) Message size: 4KB (c) Message size: 64KB

Fig. 9: Latency comparisons between state-of-the-art approaches and MFLOW under different message sizes.

distributed micro-flows to the splitting cores and enabled full path scaling for TCP and device scaling for UDP (in Figure 5). For all tests, CPU and memory resources were sufficient. All experiments were run multiple times to mitigate variation.

A. Micro-benchmarks

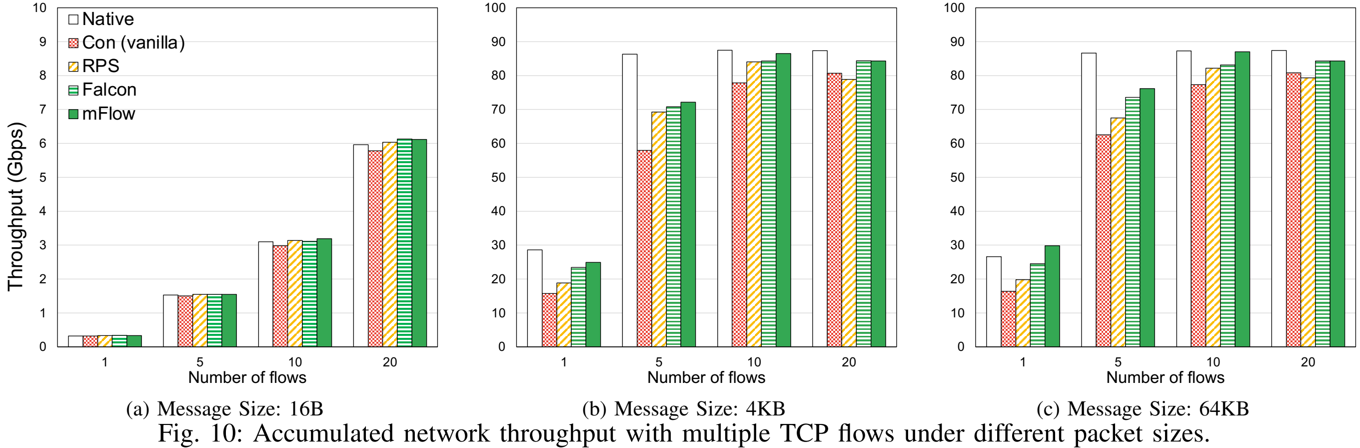
Single-flow throughput: To measure the throughput of a single flow, we used `sockperf` [21] to generate traffic with various message sizes. Note that when a message is larger than MTU (1,500 bytes), it will be fragmented into multiple packets during transmission. For TCP, we used a pair of `sockperf` client and server. However, the client under UDP was often bottlenecked (*i.e.*, overloading a CPU core). Hence, similar to FALCON [13], we used *three* `sockperf` clients to send traffic to one `sockperf` server to stress the network stack on the receiver side to its limit for a UDP flow.

In Figure 8a, MFLOW improved the throughput of a single flow significantly, especially with large message sizes (*e.g.*, 64 KB), by 81% for TCP and 139% for UDP, compared to vanilla overlay. Under TCP, MFLOW even achieved higher throughput than the native – 29.8 Gbps vs. 26.6 Gbps. It is because although the native network was much simpler than overlay network, a single core (for `skb_allocation`) was overloaded at the high throughput. In contrast, MFLOW leveraged multiple cores to process a single flow in parallel. For UDP (under 64 KB), MFLOW achieved lower throughput than the native. The reason is that, under UDP, the clients were throttled after they overloaded client-side CPU cores.

Compared to FALCON, MFLOW achieved 22% more throughput under TCP and 21% more under UDP (with 64 KB). It indicates that exploiting *packet-level* parallelism can keep pushing the in-kernel network stack to achieve higher network performance. For UDP under small message/packet size (16B), MFLOW achieved even higher performance than FALCON – more than 40%. For TCP with a small packet size (16B), both FALCON and MFLOW did not help much (similar to the vanilla overlay). This is because the TCP client became the bottleneck. This also indicates that further optimization focus should be placed on the sender side.

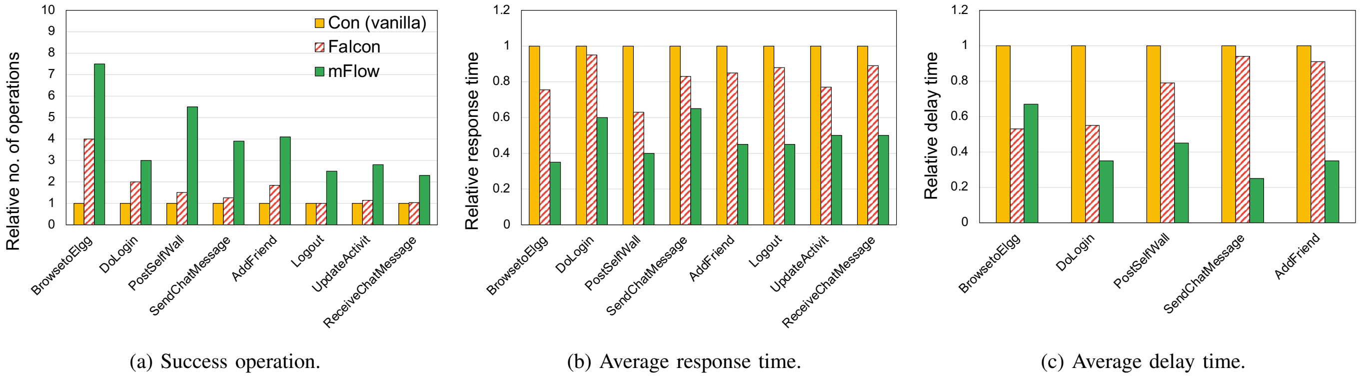
Single-flow splitting and CPU utilization: Figure 8b shows how MFLOW splits the TCP and UDP flows and the breakdown of average CPU utilization on each core (with 64 KB).

For TCP, we tested MFLOW's full path scaling scenario – *i.e.*, splitting occurred in the first stage and merging occurred before packets entered the stateful TCP transport layer. Core *one* was used for dispatching raw packet requests to two separate cores – splitting core *two* and *three*. We noticed that, if all network processings were placed on one splitting core, the splitting core was easily overloaded (as MFLOW increased TCP throughput significantly). Hence, to scale the performance of a single TCP flow, we further split and pipelined the processings on two cores for each parallel branch – *i.e.*, we used core *two* only for `skb` allocation and dispatched the remaining processings on core *four*. The same configuration was applied to core *three* and *five*. With this, MFLOW achieved extremely high TCP throughput for container overlay network as shown in Figure 8a. Now, we observe that core *zero* – upon which a single kernel thread copies data from the kernel ring buffer to the user-space application – was fully utilized and became the new bottleneck.



(a) Message Size: 16B (b) Message Size: 4KB (c) Message Size: 64KB

Fig. 10: Accumulated network throughput with multiple TCP flows under different packet sizes.



(a) Success operation. (b) Average response time. (c) Average delay time.

Fig. 11: MFLOW improves the throughput of a web serving application with reduced response time.

For UDP, we tested MFLOW's single device scaling scenario – *i.e.*, splitting occurred before the heavyweight VxLAN device and merging occurred before packets were copied to applications. As shown in Figure 8b, we placed all network devices after VxLAN on the same core as they consumed way less CPU utilization. Core *one* was used for the first stage and dispatching packets in the form of skbs to two separate cores – splitting core *two* and *three*. With this configuration, MFLOW achieved higher UDP throughput than FALCON for container overlay network (Figure 8a). We noticed that none of these cores were fully utilized. Instead, the three clients overloaded their sender-side cores and were the bottleneck.

Single-flow latency: Figure 9 depicts the per-packet latency of a single TCP or UDP flow with various message sizes. We measured the latency in the "overloaded" scenario (using sockperf), in which each case was driven to its maximum throughput before packet drops occurred. We observe that, under all cases, MFLOW reduced per-packet processing latency compared to vanilla overlay, RPS, and FALCON. For example with 64 KB, compared to vanilla overlay, MFLOW reduced the median latency by ∼46% and 99th percentile latency by ∼21% for TCP. It is because MFLOW's packet-level parallelism reduces the latency resulting from the pipelined processing (i.e., the processing of the following packet depends on the completion of its previous packet). We observe that there remained a gap in latency between MFLOW and the native due to prolonged data path in container overlay networks.

Multi-flow testing: We further conducted *multi-flow* tests – *i.e.*, multiple flows co-existed within the same host machine. Since for UDP, clients were the main bottlenecks preventing MFLOW from saturating available network bandwidth, we showed the multi-flow TCP case in Figure 10. The message sizes were set to 16 B, 4 KB, and 64 KB, and the number of flows varied from 1 to 20. In all tests, we used 5 dedicated cores for application threads and 10 dedicated cores for all in-kernel packet processing to have a more controlled experimental environment for the ease of result analysis.

In Figure 10, with the small message/packet size (*i.e.*, 16 B), all test cases scaled linearly, as the client side became the bottleneck. With the larger message/packet sizes (*i.e.*, 4 KB and 64 KB), MFLOW consistently outperformed vanilla overlay – *e.g.*, by 24% with 5 concurrent flows (under 4 KB). This benefit shrank as more flows were added – *e.g.*, by 11% with 10 flows and by 5% under 20 flows. It is because as the flow number increased, there was little CPU resource to scale up MFLOW. This can be further verified with the comparison between FALCON and MFLOW – MFLOW outperformed FALCON by 5% with 10 concurrent flows (with 64 KB) while they achieved the same performance with 20 flows, where CPU was the bottleneck.

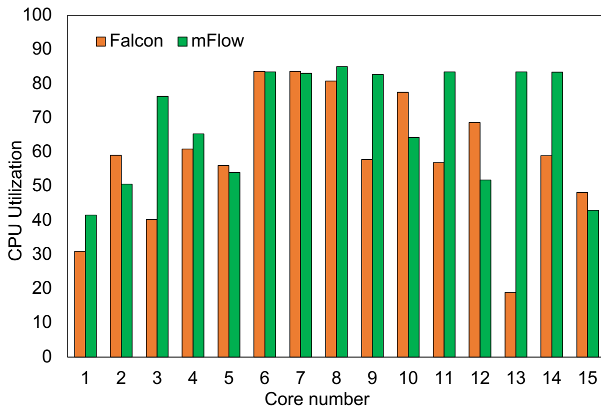


Fig. 12: MFLOW uses CPU cores in a more balanced manner.

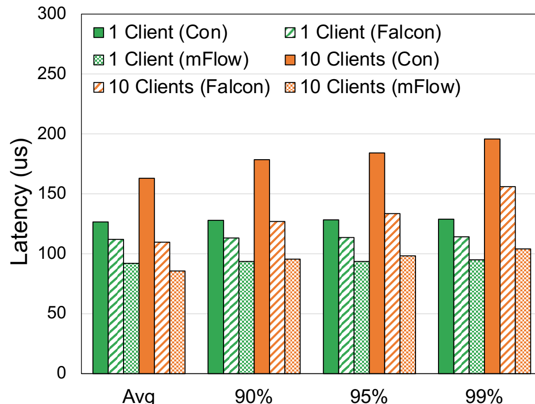


Fig. 13: MFLOW reduces the average and tail latency of a data caching application (Memcached).

MFLOW overhead: Figure 12 shows the average CPU load distribution among all used cores for the multiple TCP flow case (with 10 flows under 64 KB). More fine-grained flow steering in MFLOW does incur additional overhead – compared to FALCON, MFLOW consumed 15% more CPU utilization (among 10 cores for packet processing) in exchange for 5% performance gains. However, this is the worst-case scenario. We observed less than 5% additional overhead with 5 flows and the same CPU utilization with 20 flows (the system was overloaded). On the other hand, the advantage of MFLOW lies in that, in Figure 12, MFLOW can leverage CPU cores in a more balanced manner with even load distribution. In contrast, CPU utilization variation under FALCON was larger than MFLOW – *i.e.*, the standard deviation of CPU utilization among 10 cores was 20.5 (FALCON) vs. 11.6 (MFLOW).

B. Applications

In this section, we use two representative real-world applications, web serving and data caching, to evaluate MFLOW.

Web serving: We measured the performance of Cloudsuite's Web Serving benchmark [24] under vanilla overlay, FALCON, and MFLOW. Cloudsuite's Web Serving – the benchmark to evaluate page load throughput and access latency – contains four components: an *nginx* web server, a *mysql* database, a *memcached* server, and clients. The web server runs the Elgg [25] social network and connects to the cache and database servers. The clients send different types of request workloads, including login, chat, update, etc., to the web server. In our experiments, all of the services were performed inside containers that were connected via the Docker overlay network upon the 100 Gbps NIC.

Figure 11a depicts the "success operation" rate when we ran the benchmark with 200 users. We observe that MFLOW improved the successful individual operation rate by 2.3x – 7.5x compared to the vanilla overlay network. For the same metric, MFLOW outperformed FALCON by 1.5x – 3.6x. Figure 11b and Figure 11c present the average response time and delay time for different operations. The response time denotes the time to complete one request while the delay time represents the difference between the target and actual processing time. Compared to the vanilla overlay network, MFLOW reduced the average response time by 35% – 65% while the average delay time by up to 75%. Compared to FALCON, MFLOW reduced the average response time by 22% – 54% and the average delay time by 36% – 73%.

Data caching: We further measured the average and tail latency using Cloudsuite's data caching benchmark. It uses the *Memcached* data caching server, simulating the behavior of a Twitter caching server with a Twitter dataset. In our experiments, the Memcached server was configured with 4GB memory, 4 threads, and 550 bytes object size. The Memcached server and clients were running under the same Docker overlay network. As illustrated in Figure 13, compared to the vanilla overlay network, MFLOW reduced the tail latency (99th percentile latency) by 26% when we used one client. When the number of clients increased to ten, MFLOW's benefit became more significant – reducing the average and tail latency by 48% and 47% (99th percentile). It is because, as the number of clients (and the request rate) increased, the in-kernel network stack was more stressed. MFLOW improved its efficiency by using multiple cores for parallel packet processing. In addition, compared to FALCON, MFLOW reduced the average latency by 22% and tail latency (99th percentile) by 33%, demonstrating a higher degree of packet processing parallelism.

VI. RELATED WORK

There is a large body of work aimed at optimizing the in-kernel network stack for efficient packet processing. Focus has been on eliminating redundant data copy [26]–[29], improving interrupt locality [26], [28], [30], [31] and load balance [30], and alleviating packet processing overhead via interrupt coalescing [32] and system call batching [33]. However, some work reported that both latency and throughput are still many times worse than the hardware can achieve [28], [34]. Some other papers proposed lightweight and customized network stacks [34]–[39] to improve the network performance. However, such designs require changes to the application-kernel interface, not compatible with legacy applications. Alternatively, research has shifted to bypass the OS kernel and implements the network stack entirely in user space [37], [40], [41]. Benefits of user-space approaches include a reduced

number of context switches and direct hardware access that eliminates much of the indirection and overhead in the kernel. Intel's Data Plane Development Kit (DPDK) [40] is one such example of user-space libraries. In contrast, MFLOW does not re-design in-kernel network stacks but instead focuses on exploiting network processing parallelization at the packet level for container overlay networks. Hence, MFLOW preserves the current design of overlay networks and retains all existing network management tools.

To address the inefficiencies of container overlay networks, recent work seeks to either eliminate packet transformation from the network stack or parallelize packet processing. For example, Slim [15] can bypass the virtual bridge and the virtual network device in containers, achieving near-native performance. However, Slim does not apply to connection-less protocols, such as UDP, and limits the scalability of host network management as each Slim overlay network connection needs a unique file descriptor and port created in the host network. FALCON [13] parallelizes packet processing in container overlay networks by pipelining software interrupts associated with different network devices of a single flow on multiple cores — achieving device-level parallelism. In contrast, MFLOW investigates unexploited packet-level parallelism in the kernel network stack.

VII. CONCLUSIONS

We have presented MFLOW, a novel in-kernel packet steering approach to accelerate container overlay networks by exploiting packet-level parallelism. MFLOW splits the packets of a single flow into multiple micro-flows and processes them in parallel by taking advantage of a multi-core system while efficiently preserving in-order packet delivery. Our evaluation with both micro-benchmarks and applications demonstrates the effectiveness of MFLOW. Meanwhile, the results have revealed new bottlenecks that prevent a single flow from further scaling: One lies in clients/senders and the other is the receiver-side single data-copying thread. We seek to address these bottlenecks in our future work.

REFERENCES

[1] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe, "Container orchestration on hpc systems through kubernetes," *J. Cloud Comput.*, 2021.

[2] C. Cérin, N. Greneche, and T. Menouer, "Towards pervasive containerization of hpc job schedulers," in *SBAC-PAD*, 2020.

[3] G. Li, J. Woo, and S. B. Lim, "Hpc cloud architecture to reduce hpc workflow complexity in containerized environments," *Applied Sciences*, vol. 11, no. 3, p. 923, 2021.

[4] Y. Zhou, B. Subramaniam, K. Keahey, and J. Lange, "Comparison of virtualization and containerization techniques for high performance computing," in *ACM/IEEE Supercomputing*, 2015.

[5] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan, "A study on the evaluation of hpc microservices in containerized environment," *Concurrency and Computation: Practice and Experience*, 2021.

[6] S. Herbein, A. Dusia, A. Landwehr, S. McDaniel, J. Monsalve, Y. Yang, S. R. Seelam, and M. Taufer, "Resource management for running hpc applications in container clouds," in *HPC*, 2016.

[7] A. Ruhela, M. Vaughn, S. L. Harrell, G. J. Zynda, J. Fonner, R. T. Evans, and T. Minyard, "Containerization on petascale hpc clusters," in *HPC*, 2020.

[8] A. Torrez, T. Randles, and R. Priedhorsky, "Hpc container runtimes have minimal or no performance impact," in *ANOPIE-HPC*, 2019.

[9] M. Höb and D. Kranzlmüller, "Enabling easey deployment of containerized applications for future hpc systems," in *International Conference on Computational Science*, 2020.

[10] "Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces," https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[11] "Linux control groups," "https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt".

[12] "Seccomp security profiles for docker," "https://docs.docker.com/engine/security/seccomp/".

[13] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *EuroSys'21*, 2021.

[14] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for container overlay networks," in *HotCloud*, 2019.

[15] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a Low-Overhead container overlay network," in *NSDI'19*, 2019.

[16] *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, https://datatracker.ietf.org/doc/html/rfc7348.

[17] *Open vSwitch*, http://openvswitch.org/.

[18] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?" in *NetAI '20*, 2020.

[19] "Receive side scaling (rss)," "https://rb.gy/hmjbaj".

[20] *Receive Packet Steering*, https://lwn.net/Articles/362339/.

[21] *Sockperf*, https://github.com/Mellanox/sockperf.

[22] *Falcon github*, https://github.com/munikarmanish/falcon.

[23] *iPerf*, https://iperf.fr/.

[24] *cloudsuite*, https://cloudsuite.ch.

[25] *Elgg*, https://elgg.org.

[26] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt, "Performance analysis of system overheads in tcp/ip workloads," in *PACT' 05*, 2005.

[27] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ASPLOS'13*, 2013.

[28] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," 2014.

[29] L. Rizzo, "netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[30] L. Cheng and C.-L. Wang, "vbalance: using interrupt load balance to improve i/o performance for smp virtual machines," in *SoCC'12*, 2012.

[31] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," in *ASPLOS'12*, 2012.

[32] "Performance tuning for mellanox adapters," https://community.mellanox.com/docs/DOC-2489.

[33] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *OSDI'10*, 2010.

[34] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *OSDI'14*, 2014.

[35] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li, "Los: A high performance and compatible user-level network operating system," in *APNet'17*, 2017.

[36] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes." in *USENIX NSDI*, 2017.

[37] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems." in *USENIX NSDI*, 2014.

[38] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein, "Network stack as a service in the cloud," in *HotNets*, 2017.

[39] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *CoNEXT*, 2012.

[40] *DPDK*, http://dpdk.org/.

[41] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX NSDI'14*, 2014.