PRISM: Streamlined Packet Processing for Containers with Flow Prioritization

Manish Munikar*, Jiaxin Lei†, Hui Lu† and Jia Rao*
*University of Texas at Arlington
†Binghamton University

***Abstract*—Advanced high-speed network cards have made packet processing in host operating systems a major performance bottleneck. The kernel network stack gives rise to various sources of overheads that limit the throughput and lengthen the per-packet processing latency. The problem is further exacerbated for short-lived, latency-sensitive network flows such as control packets, online gaming, database requests, etc. — in a highly utilized system, especially in virtualized (containerized) cloud environments, short flows can experience excessively long in-kernel queuing delays. As a consequence, recent research works propose to bypass the kernel network stack to enable lightweight, custom userspace network stacks for improved performance, but at a heavy cost of compatibility and security. In this paper, we take a different approach: We first analyze various sources of inefficiencies in the kernel network stack and propose ways to mitigate them without compromising systems compatibility, security, or flexibility. Further, we propose PRISM, a novel mechanism in the kernel network stack to differentiate incoming packets based on their performance requirements and streamline the processing stages of multi-stage packet processing pipelines (e.g., in container overlay networks). Our evaluation demonstrates that PRISM can significantly improve the latency of high-priority flows in container overly networks in the presence of heavy low-priority background traffic.**

***Index Terms*—kernel network stack, container overlay network, packet processing, performance differentiation**

I. Introduction

Network applications can have diverse performance requirements: Some want to transfer a large amount of data from one machine to another as quickly and efficiently as possible (e.g., network file systems, database replication/migration, big data analytics, etc.) while some send small requests and expect quick responses with minimal delay (e.g., RPC, key-value stores, database queries, control signals, etc.). Although in-kernel network stack has some mechanisms to control per-flow traffic and prioritize flows in the *transmission* side (e.g., *tc* [1]), there is a lack of performance differentiation mechanisms in the packet *reception* side – all packets arriving in the kernel are treated equally (i.e., processed in an FCFS manner) by the network stack. That said, the application-level performance requirements are neglected by the kernel network stack, resulting in suboptimal performance, especially when both latency-sensitive and throughput-intensive flows co-exist.

The problem worsens in the case of virtual overlay networks which are widely used in today's cloud environments. Due to its high portability, high consolidation density, low performance overhead and hence low operational cost, the container technology (e.g., Docker [2]) has become ubiquitous to encapsulate and host applications [3]. Modern distributed cloud applications are even made up of hundreds or thousands of containers (microservices) communicating with one another over a virtual private network, which is usually implemented as an overlay network (e.g., VXLAN [4] where the virtual Ethernet frame is encapsulated within an outer UDP segment) [5]–[8]. While in-kernel overlay network offers good flexibility, modularity, and generality, compared to native host networks, it incurs significant performance overhead as its packet processing pipeline involves multiple stages [9]–[11].

We propose PRISM (priority-based streamlined packet processing), a novel mechanism in the kernel network stack to enable performance differentiation during packet reception and to streamline packet processing stages for multi-stage packet processing pipelines such as container overlay networks or network function virtualization (NFV) chains. Unlike kernel bypass methods [10], [12]–[15], PRISM utilizes the existing generic in-kernel network stack — which has become mature, stable and secure with decades of development — without *reinventing the wheel*. Therefore, PRISM is fully transparent to and compatible with existing applications, and works with existing hardware — e.g., without requiring dedicated cores for polling packets from the network hardware.

More specifically, with thorough performance tracing and code analysis, we identify key optimization opportunities in the kernel network stack for batched and interleaved processing of multi-stage flows. PRISM first streamlines the packet processing stages by improving how the network devices are polled in the NAPI softirq handler. Further, PRISM differentiates the priority of a flow by detecting its priority early in the packet processing pipeline and adding a separate high-priority packet queue in network devices to enable differentiation.

Our main contributions are as follows:

- We perform a comprehensive analysis (both empirical and qualitative) to study the effect of background traffic on the performance of latency-sensitive flows in the vanilla kernel network stack.
- We propose a novel approach in the kernel network stack to streamline container overlay network packet processing and to enable performance differentiation during packet reception.
- Our design can reduce the latency (both average and tail) of high-priority flows in the presence of low-priority background traffic by more than 50%.

- We release our prototype implementation to the open-source community.

The rest of the paper is organized as follows: Section II gives a brief background about how in-kernel packet processing works and identifies sources of optimization. Section III and IV describe the design and implementation details, respectively, of PRISM. In Section V, we present our experimental results. In Section VI, we juxtapose our solution with related works, and then we discuss some remaining issues in Section VII. Finally, we conclude in Section VIII.

II. BACKGROUND AND MOTIVATION

In this section, we first describe how incoming packets are processed in the Linux kernel and then present two sources of inefficiencies when the kernel network stack processes packets in a complex pipeline that involves multiple network devices. Without loss of generality, we focus on container overlay networks as they are the most common way to construct a virtual network for containers in the cloud. Finally, we motivate some opportunities for optimizing the kernel packet processing pipeline to improve performance that are transparent to (and fully compatible with) existing applications.

A. NAPI Packet Processing

When a packet arrives in a host, the operating system (OS) can handle the packet in two fundamental ways: *interrupt* or *polling*. In *interrupt mode*, the network interface controller (NIC) raises an interrupt every time a packet arrives, causing the CPU to switch to the interrupt context and process that packet. This is favorable under low load, but becomes unreasonably expensive and inefficient under high load, especially with fast modern network cards. In *polling mode*, a CPU thread periodically checks the kernel ring buffer to see if there are any new packets (DMA-ed by the NIC). Unlike the interrupt mode, this is more efficient under high load due to reduced context-switching overheads. However, under low load, it either wastes a lot of CPU cycles or increases latency depending on the polling frequency. Since Linux kernel version 2.6, incoming network packets are processed in the kernel by the New API (NAPI) pipeline [16], which brings the best of both worlds, i.e., it enables interrupts under low load but switches to the polling mode under high load.

NAPI starts off in the interrupt mode: the NIC raises a hardware interrupt (*irq*) on the reception of the first packet. However, instead of processing the packet through the entire network stack, the hardware interrupt handler (i.e., the *top-half*) simply notifies the kernel that there are packets in the device's receive queue ready to be processed. The actual packet processing happens in a separate kernel thread as a NAPI polling loop (i.e., the *bottom-half*). This mechanism of deferring the packet processing out of the hardware interrupt context is known as software interrupt (*softirq*). The irq handler raises a softirq (`NET_RX` for packet reception) by adding the network device to the NAPI *poll list* — a per-CPU list of network devices with packets available in their receive queues — and setting a bit flag (if not already set) indicating pending softirq.[1] This will schedule the corresponding softirq handler (`net_rx_action` in the case of the `NET_RX` softirq) on that CPU. The softirq handler then starts polling and processing packets from the network devices in the NAPI poll list until either the packet queue is empty or it has used up its allotted time slice or packet quota. The time slot or packet quota prevents the softirq from monopolizing the CPU core and ensures the softirq handler behaves as a *good* kernel citizen. For each device, NAPI processes a batch of packets in the FIFO order and moves on to the next device in the poll list. The batch size is typically 64 in default Linux kernel configurations. Each packet is processed through the protocol stack and then enqueued in the application receive buffer. While the softirq handler is polling NAPI devices, hardware interrupts are disabled on that CPU. In this way, NAPI can work in either interrupt or polling mode appropriately depending on the load.

In the native host network, only one network device is involved (the physical NIC) and a packet goes through the network stack just once. However, the situation becomes more complicated for virtual overlay networks such as container overlay networks. Virtual machines (VMs) and containers provide "virtually" isolated environments with their own independent virtual network interfaces. The virtual interfaces and physical interfaces within a host are connected via a software switch such as Linux bridge or Open vSwitch [18]. These software switches allow communication between VMs/containers running on the same virtual network but different physical hosts. Fig. 1 shows how various physical and virtual network devices are typically organized to construct a container network using the VXLAN overlay network.

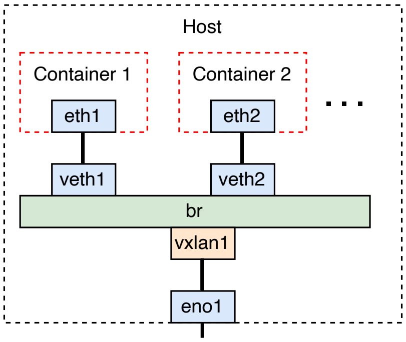


Fig. 1. A typical architecture of network devices (physical and virtual) in a container overlay network.

Packets destined for one of the virtual interfaces usually arrive at the physical interface as encapsulated packets (e.g., VXLAN). Let us consider the journey of an encapsulated packet: It is first processed during the polling of the physical interface where the packet is identified as an encapsulated packet. The outer headers are stripped off and the inner decapsulated packet is enqueued to the receive queue of the

[1] If the bottom-half is deferred to a separate CPU, e.g., by receive packet steering (RPS) [17], then an inter-processor interrupt (IPI) is also sent.

virtual bridge. Then, another softirq is raised by adding the bridge to the NAPI poll list (if not already). The processing of that packet is then delayed until the virtual bridge is polled by the softirq handler, which usually happens after a batch of packets are polled and processed from the physical interface's packet queue. When it is the bridge's turn to be polled, the inner packet goes through the network protocol layers to identify the destination interface, which is the virtual interface of the container. The packet is then again enqueued to the virtual interface receive queue and yet another softirq is raised. Again, the packet will wait until the NAPI polling loop reaches the virtual interface. The packet then goes through the entire protocol stack again to process the inner headers and finally reaches the application receive buffer. While NAPI addresses the tradeoff between interrupt and polling and adaptively switches the two in response to the incoming traffic load, the batched packet processing in each NAPI poll and the queuing delays of packets at multiple devices negatively affect packet latency, especially for virtual networks with multiple devices and a convoluted pipeline.

1) Batched Packet Processing: At each network device, NAPI polling processes packets in a batch (e.g., 64 packets by default) before moving onto the next device in the poll list. This design amortizes the overhead to switch softirq contexts when changing network devices among a large number of packets, thereby reducing the per-packet processing cost. However, the first packet completed in a batch must wait for the remaining packets to be processed before its processing on the next stage can begin. For latency-sensitive applications, the delay — which in the worst case equals to the time to process 64 packets in a batch multiplied by the number of devices — could be significant. On the other hand, only processing one packet under each softirq would avoid the queuing delay but impose the softirq context switching cost to each packet. Whether the queuing delay or the softirq switching cost plays a more important role in the overall latency depends on the workload.

2) Interleaved NAPI Polling: In addition to the queuing delay due to batched packet processing, our performance tracing and analysis found that the multi-stage polling in the NAPI poll list can also lead to interleaved processing across different batches. For example, a typical container network involves three devices, i.e., the physical Ethernet interface (`eth`), the virtual bridge (`br`), and the container's virtual Ethernet interface (`veth`). The optimal polling order for the container overlay network should be {`eth`, `br`, `veth`, ...} as this allows one batch of packets to be fully processed through all stages before the next batch is polled. However, the traced NAPI poll order using eBPF [19] shows that the stages of different batches can be interleaved and the kernel can start to process the second batch before the first batch is delivered to the application receive buffer (as shown in Fig. 6a and discussed later). In particular, the third stage (`veth`) of the first batch is always delayed by the first stage (`eth`) of the next batch. As such, latency is further delayed by the interleaving processing scheme across different batches.

Upon detailed code review, we found this interleaved polling order to be the result of a scalability optimization in the NAPI device polling design, which is illustrated in Fig. 4a and presented as the pseudocode in Fig. 2. NAPI maintains two poll lists for each CPU: (a) a global poll list where new devices are added (❶), and (b) a local poll list where the actual packet processing happens. At the beginning of each softirq, the entire global poll list is moved to the local poll list (❷). The softirq continuously polls the devices in the local poll list until either the local poll list is empty or the `NAPI_BUDGET` (300 packets by default) is consumed. The use of two poll lists allows new devices to be added to the global list and packets to be processed on the local list without any locking, thereby improving scalability. In each iteration of the packet processing loop (line 12-20 in Fig. 2 and illustrated in Fig. 4a), a device is dequeued from the head of the local poll list (❸). Once a batch of packets is processed in the current device (❹), if it still has more packets in its receive queue, the device is added back to the tail of the local poll list (❺) so that it will get a chance to get processed in subsequent iterations of the device loop.

```
 1  // device added here when softirq raised
 2  POLL_LIST := per-CPU global NAPI poll list
 3  // max packets to process in one softirq
 4  NAPI_BUDGET := 300
 5
 6  function net_rx_action():
 7    poll_list := empty local list
 8    move POLL_LIST to the tail of poll_list
 9    // at this point, the global POLL_LIST is empty
10    processed := 0
11    while true:
12      if poll_list is empty:
13        break
14      device := poll_list.pop(index=0)
15      processed += napi_poll(device, batch_size=64)
16      if device.packet_queue is not empty:
17        POLL_LIST.append(device)
18      if processed >= NAPI_BUDGET:
19        break
20    move POLL_LIST to the tail of poll_list
21    move poll_list to the tail of POLL_LIST
22    if POLL_LIST is not empty:
23      raise_softirq()
```

Fig. 2. NAPI polling logic in vanilla kernel (pseudocode).

The packet processing loop is a simple queuing system, as shown in Fig. 4b. Each device has an input packet queue from where a batch of packets are dequeued and processed (❶). After a packet has been processed by various protocol layers, it is either enqueued to the user application buffer or to the input queue of the next device (❷), which causes the next device to be added to the global poll list (if not).

3) Backlog Queue: While most of physical NIC drivers are fully NAPI-aware and implement their own packet queues and poll functions, most of the virtual network devices (e.g., the `veth` interface) do not have their own NAPI implementations. The common practice is to use a generic poll function (`process_backlog`) and a per-CPU backlog queue of packets that do not belong to a specific NAPI device. Kernel has a per-CPU backlog NAPI structure that is responsible for handling the packets belonging to "non-NAPI-aware" interfaces. One exception is the virtual bridge device that uses the `gro_cells` driver which has its own NAPI implementation. In our overlay network example, the first stage (`eth`) is processed by the NIC driver, the second stage (`br`) is processed by the `gro_cells` driver and the third stage (`veth`) is processed by the generic backlog handler. This will be of importance later when we explain about streamlining NAPI polling.

B. *Priority Differentiation*

Another fundamental issue with the kernel network stack is that it does not consider the performance requirements of incoming packets of network flows. Even though the kernel has a mechanism (e.g., *tc*) to prioritize the order of outgoing packets, there is no such mechanism in the reception pipeline. As mentioned in the previous sections, different applications have different network performance requirements. However, the kernel processes all incoming packets in a simple FCFS (first come, first serve) order. This can cause head-of-line (HoL) blocking when a latency-sensitive application (e.g., memcached) is running alongside a throughput-intensive application (e.g., big data analytics or video streaming). This problem is further exacerbated in the case of virtual overlay networks because packets have to wait in multiple queues.

For example, Fig. 3 shows the cumulative distribution of the latency of high-priority flows with and without low-priority background traffic, for container overlay network flows. The experimental setup is described in Section V-A. We can see that compared to the idle scenario, a loaded server increases the median overlay per-packet latency by about 400% and the tail (99th percentile) latency by about 450%.

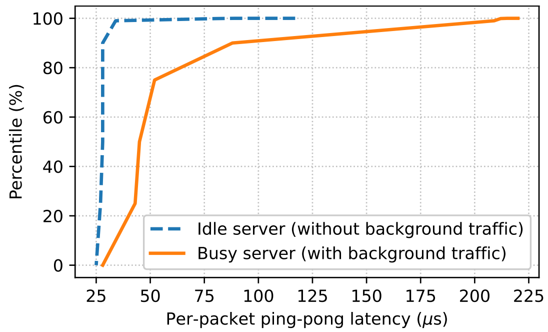


Fig. 3. Latency distribution of packets in the presence and absence of background traffic.

Our goal is to process the few and small latency-sensitive packets (end-to-end) as quickly as possible while processing the large throughput-sensitive packets in a best-effort manner. Thus, we can satisfy the latency requirements of short requests without compromising the throughput requirements of large flows. In what follows, we will describe various ways to prioritize high-priority packets in the Linux kernel.

C. *Summary*

Linux kernel network stack is designed to provide a general-purpose packet processing mechanism with good "overall" performance. It breaks the packet processing pipeline into multiple stages for flexibility and uses batching to improve the overall throughput. However, kernel network stack does not recognize the various performance requirements of individual network flows, and the interleaved and batched packet processing cause significant packet processing latency. In this work, we seek to prioritize the processing of latency-sensitive network flows, meanwhile maintaining acceptable performance of throughput-intensive network flows.

III. PRISM DESIGN

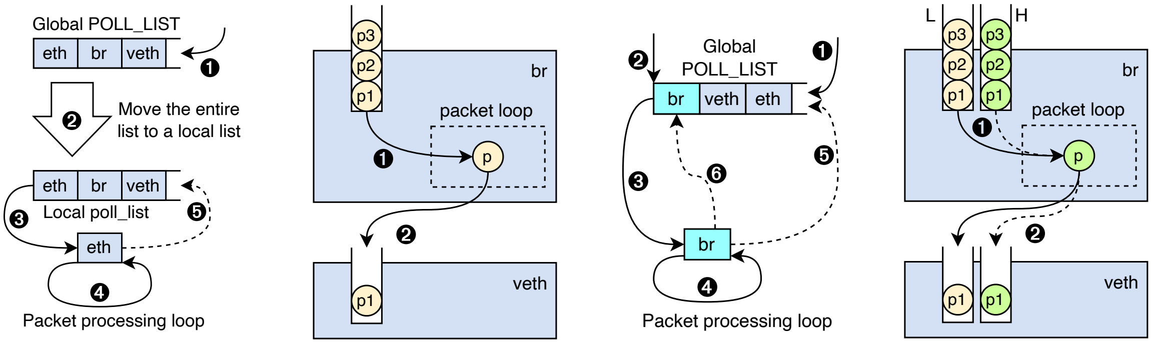
In this section, we describe the design of PRISM. First, we explain our solution to the interleaved NAPI polling for virtual overlay networks. Then, we propose a mechanism to enable priority-based packet processing in the kernel network stack.

A. *Streamlined NAPI Poll*

As discussed in Section II-A2, when the packet processing pipeline involves multiple stages or devices (e.g., in an overlay network), the order of devices is not streamlined: The second batch of packets gets interleaved between different stages of the first batch, incurring significant delays. We identify the root cause to be the *synchronization delay* between the global and local poll lists and the strict *tail-enqueuing* of pending devices in the poll lists. This design is primarily for scalability consideration and allows the existing flow parallelization techniques (e.g., RPS) to balance flows across multiple CPUs without the need for locking. However, for flows with multiple stages, a single flow could saturate a CPU, leaving little room for load balancing but causing drastic hikes in latency. Hence, to ensure streamlined packet processing and maintain the proper device order, PRISM resorts to a simplified single poll list design.

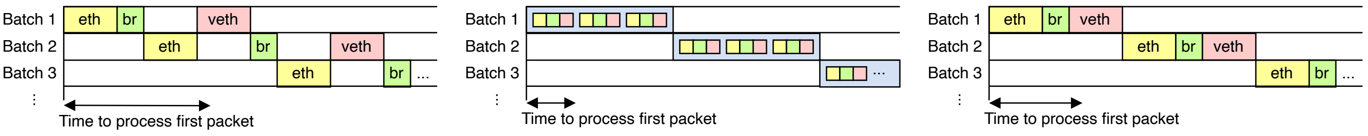
PRISM's device polling design is illustrated in Fig. 4c. Unlike the vanilla kernel, PRISM maintains only one poll list per CPU. This eliminates the synchronization step between the global and local lists. New devices are added to the *tail* of the poll list if the device has low-priority packets (❶), or to the *head* of the poll list if the device has high-priority packets (❷). In each iteration, the device at the head is dequeued from the poll list (❸) and a batch of packets are processed from its packet queue (❹). When a packet is processed in one device and requires more processing stages, it is enqueued to the packet queue of the next device, and the next device is added to the head (❻) or the tail (❺) of the poll list depending on whether the packet has high priority or low priority. The pseudocode of this device polling logic is presented, with changes highlighted, in Fig. 7.

PRISM's packet processing loop, illustrated in Fig. 4d, is slightly more complicated than the vanilla kernel. To enable priority differentiation, in PRISM, each network device has *two* input packet queues: a *high-priority* queue and a *low-priority* queue, denoted as H and L, respectively. When the device is polled, if the high-priority queue is empty, a batch of packets



(a) Vanilla device loop (b) Vanilla packet loop (c) PRISM device loop (d) PRISM packet loop

Fig. 4. Vanilla vs. PRISM design for NAPI processing.



(a) Vanilla (b) PRISM-sync (c) PRISM-batch

Fig. 5. NAPI processing sequence in Vanilla vs. PRISM.

are dequeued and processed from the low-priority queue, just like vanilla. However, if the high-priority queue is not empty, a batch of packets are dequeued and processed from that queue (❶). When a high-priority packet has finished its processing in the current device, it is enqueued to the high-priority queue of the next stage device (❷) and that device is added (or moved) to the head of the poll list. If in the last stage, the packet data is copied to the application buffer. The packet processing logic is presented using pseudocode in Fig. 7 (line 22–38).

Consider the same example as in Section II-A2. Fig. 6b illustrates the sequence of high-priority packet processing for container overlay networks in PRISM. We have a container network involving three stages (network devices): `eth`, `br`, and `veth`. When the packets first arrive at the NIC, the poll list will have {`eth`} and the softirq handler will dequeue `eth` and start processing a batch of packets from its packet queue. Once a packet is processed in `eth`, the second stage (`br`) will be added to the head of the poll list. Since `eth` will still have more packets in its packet queue, it will also add itself to the tail of the poll list. In the second iteration of the loop, the poll list will have {`br`, `eth`} and the first device (`br`) will be dequeued and its first batch of packets will be processed. This will have the third stage (`veth`) to be added to the head of the poll list causing it to have {`veth`, `eth`}. Now, in the next iteration, the first item (`veth`) is dequeued and its first batch of packets is processed. Now the first batch of packets have completed processing and are delivered to the user application. In the next iteration, the poll list will only have {`eth`} and repeat the same sequence as that in the first iteration. As such, the order of devices being polled always follows the sequence in which packets should be processed.

Iter.	Device	Poll list
0	—	[eth]
1	eth	[br, eth]
2	br	[eth, veth]
3	eth	[veth]
4	veth	[br, eth]
5	br	[eth, veth]
6	eth	[veth, br, eth]
	. . .	. . .

(a) Vanilla

Iter.	Device	Poll list
0	—	[eth]
1	eth	[br, eth]
2	br	[*veth*, eth]
3	veth	[eth]
4	eth	[br, eth]
5	br	[*veth*, eth]
6	veth	[eth]
	. . .	. . .

(b) PRISM

Fig. 6. NAPI device processing order in Vanilla vs. PRISM.

B. Flow Prioritization

Section II-A1 shows how the batching mechanism acts as a tradeoff between throughput and latency. On the one hand, as we increase the batch size, the softirq overhead (which is usually constant) is shared among all the packets in the batch thereby minimizing the per-packet processing cost and maximizing the effective use of the CPU. Moreover, batching also helps to improve the L1 instruction cache locality and its hit rate, further reducing per-packet processing time. On the other hand, if the packet processing pipeline has multiple stages of batching (e.g., in container overlay networks), each packet has to wait until all the other packets in the batch have been processed to advance to the next stage. This can significantly increase the per-packet queuing delay and hence worsening latency. Meanwhile, decreasing the batch size reduces per-packet latency but hurts overall throughput. Our aim in this work is to minimize the latency for high-priority packets. PRISM devises two operation modes to strike a good tradeoff between latency and throughput.

```
// device added here when softirq raised
POLL_LIST := per-CPU global NAPI poll list
// max num of packets to process in one softirq
NAPI_BUDGET := 300

function net_rx_action():
  processed := 0
  while true:
    device := POLL_LIST.pop(index=0)
    if device == NULL:
      break
    processed += napi_poll(device, batch_size=64)
    if device.high_packet_queue is not empty:
      POLL_LIST.insert(device, index=0)
    else if device.low_packet_queue is not empty:
      POLL_LIST.append(device)
    if processed >= NAPI_BUDGET:
      break
  if POLL_LIST is not empty:
    raise_softirq()

function napi_poll(device, batch_size):
  processed := 0
  if device.high_packet_queue is empty:
    while processed < batch_size:
      if device.low_packet_queue is empty:
        break
      pkt := device.low_packet_queue.dequeue()
      process_packet(pkt)
      processed += 1
  else:
    while processed < batch_size:
      if device.high_packet_queue is empty:
        break
      pkt := device.high_packet_queue.dequeue()
      process_packet(pkt)
      processed += 1
  return processed
```

Fig. 7. PRISM NAPI processing logic (pseudocode).

1) PRISM-sync: In this mode, all the stages of high-priority packets are processed synchronously in a *run-to-completion* manner within a single softirq. A packet is processed through all the stages and delivered to the user application before the next packet starts processing, which effectively disables batching. This approach stems from the fact that under a moderate to highly loaded system, a packet spends a significant fraction of its life waiting for its turn in various packet queues. In this mode, for high-priority flows, there is only one device in the poll list: the first physical device (`eth` in our example). The subsequent stage devices are never added to the poll list because the packet never goes into their packet queues.[2] All the stages of its processing pipeline are executed sequentially one after another synchronously.

This mode minimizes the time the packet lives in the kernel network stack and delivers it to the user-level application buffer as soon as possible. Fig. 5b illustrates the sequence of packet processing in PRISM-sync mode. Note that even though there are three packet processing stages, each batch belongs to the same device, and the time to process one packet is much smaller compared to that in vanilla Linux. This mode is suitable if the high-priority flows that have a stringent requirement on latency and are tolerant of throughput loss.

2) PRISM-batch: As discussed in Section II-A1, the batch-based NAPI polling achieves low per-packet processing cost but involves queuing at multiple devices and asynchronous invocation of multiple softirqs. Though batch-based packet polling and processing is much desired (to achieve overall high performance), it turns out to be challenging to enforce flow prioritization in such a setting: Since we cannot use interrupts inside a softirq context, prioritizing packets can be done by periodically checking (or polling) for packets in the high-priority queue of all network devices. This periodic checking can be done in different granularity. On one extreme, this can be done after processing each packet. However, if the proportion of high-priority packets is low, it wastes a lot of CPU cycles and reduces the efficiency of the system. On the other extreme, it can be done for each device when it is polled by the NAPI device polling loop, in which case, it is essentially the same as low-priority packet processing. In PRISM-batch mode, we seek a “sweet spot” between these two extremes via batch-level preemption. PRISM-batch ensures that packets of all priority are processed in batches. However, we make sure that the high-priority batches are processed (through all stages) before low-priority batches. Batch-level preemption is enabled by: 1) allowing devices to be added to the *head* of the poll list, and 2) maintaining two packet queues in each network device and processing low-priority queues only when high-priority queues are empty. Thus, the worst-case preemption latency for high-priority flows equals to the processing time of one stage of one batch of low-priority packets.

The processing sequence of our example container overlay network in PRISM-batch mode is illustrated in Fig. 5c. Note that even though packets are processed in batches, we make sure that the first batch is processed completely (through all three stages) before the next batch is fetched. The time to process one packet is greater than PRISM-sync mode, but still significantly less compared to that in the vanilla kernel. This mode is suitable for improving the latency of high-priority flows while still maintaining acceptable throughput.

IV. IMPLEMENTATION

We implemented PRISM on top of Linux kernel version 5.4 with ∼550 lines of code. Our implementation is open-sourced.[3]. This section discusses the implementation details and its limitations.

A. Packet Priority Identification

Our work focuses on the *mechanism* of prioritizing high-priority packets. The *policy* of deciding which packets to treat as high-priority is a matter of users' decisions. In fact, there are many ways to set packet priorities. One could treat small packets/flows as high-priority packets and big ones as low-priority. Similarly, we can set the priority based on specific applications. For example, one can treat memcached as a high-priority application and spark as a low-priority application.

[2] Even though the packet does not go to the packet queue, each stage is processed in the same *context* of the respective network devices.

[3] https://github.com/munikarmanish/prism

This would essentially translate to marking some port numbers as high priority. In our implementation, we employ a simple-yet-generic user-configurable priority policy. We allow users to dynamically set IP and port pairs to mark high-priority flows at runtime via the *proc* filesystem [20]. PRISM maintains a global database of high-priority IP and port numbers that are checked for each incoming packet to determine its priority. It also maintains a global binary *proc* variable that users can use to select either PRISM-sync or PRISM-batch mode of operation.

Network packets in the Linux kernel are represented by a metadata structure called the socket buffer (`sk_buff` or skb in short). While traversing through the network stack processing stages, the same skb is passed through different protocol layers and different device queues. So, to prevent having to re-compute the priority of an skb in every stage, we add a binary variable to the skb structure. Upon packet reception, when the skb is allocated for the first time in the first stage in the context of the physical device (inside the `mlx5e_napi_poll` function in the case of Mellanox ConnectX-5 EN driver), the priority of the packet is determined by comparing its IP address and port numbers against the global high-priority database.

B. Multiple Packet Queues

The Linux kernel maintains a per-CPU global data structure called `softnet_data` that contains a member variable `poll_list`. This is the global NAPI poll list as shown in Fig. 4c. Network devices are represented in the Linux kernel by the `net_device` data structure (or netdev in short), which stores all the device metadata such as ifindex, name, MAC address, etc. However, the `poll_list` does not hold pointers to the netdev structures. Instead, it contains another device-level data structure created to implement NAPI called `napi_struct`, which is a per-CPU data structure that is associated with a particular netdev. The input packet queue for NAPI is defined in the `net_device` structures (for physical devices) or some other per-CPU data structures associated with the `napi_struct` structure. In this paper, we extend these data structures to add another packet queue designated as a high-priority queue. For example, the *backlog* device — which is used as a fallback `napi_struct` for virtual devices that do not have their own NAPI implementations such as for `veth` — uses a packet queue called `input_pkt_queue` in the per-CPU `softnet_data` structure. So we simply extend the `softnet_data` structure by adding another packet queue similar to `input_pkt_queue`. This adds one `sk_buff_head`, consuming negligible 24-byte additional kernel memory per CPU.

C. PRISM *NAPI Processing*

The kernel raises the `NET_RX` softirq for packet reception, which is handled by the `net_rx_action` function. This is the function where NAPI devices are polled from the global `poll_list`. In PRISM, we modify the `net_rx_action` function to implement the logic presented in Fig. 7. Specifically, we remove the usage of the local poll list and make sure we get the next device to process directly from the global `poll_list` to enable batch-level preemption.

PRISM allows adding NAPI device to the head of the poll list. This is done by the *stage transition functions* that are responsible for sending packets to the input packet queue of another device or stage, adding the device to the poll list, and raising a softirq if needed. `gro_cells_receive` and `netif_rx` are commonly used as stage transition functions for Linux bridges and `veth` interfaces, respectively. In PRISM, we modify the stage transition functions to treat high-priority packets differently. In the PRISM-batch mode, the stage transition functions simply move a packet to the appropriate queue and add the device to the head or tail of the poll list depending on the packet's priority. In the PRISM-sync mode, instead of enqueuing the packet, we directly call the function responsible for processing the next stage of the packet, e.g., `netif_receive_skb`.

Finally, each NAPI struct has a virtual `poll` function which is responsible for processing one batch of packets. For the backlog device, this function is `process_backlog`. In PRISM, we modify these packet polling functions to give precedence to the high-priority queue. If there are packets in the high-priority queue, we *only* process one batch of high-priority packets and return. We process the low-priority queue only if the high-priority queue is empty when this function is called.

D. Implementation Limitations

One limitation of our implementation lies in that it cannot perform priority-differentiation in the first stage of in-kernel network reception (at the physical NIC driver). It is because the packet processing code in the physical device driver is vendor and model-specific, i.e., every vendor's driver processes packets in their own way. They have their own data structures to represent packets and packet queues to exploit the hardware-specific optimization features, hence usually very different from generic packet processing functions written in the core kernel network stack. Implementing our design on physical device drivers is not impossible: We need to capture a packet before the skb is allocated, determine its priority, and then process it differently. However, it would take considerable engineering effort to decode the internals of the vendor-specific code base and require modifying the implementation of their network interface drivers. We leave this as a possible future work. In contrast, in this work, we implement our design in the core kernel stack that implements the generic drivers for the virtual network devices (e.g., bridges and virtual Ethernet interfaces) and evaluate how much performance improvement we can achieve with this vendor-agnostic approach.

V. EVALUATION

In this section, we present the experimental results for PRISM. We focus on the effectiveness of PRISM in improving the responsiveness of high-priority flows in the presence of low-priority background traffic. We compare our results with the vanilla Linux kernel. We conduct the experiments within two groups. First, we use a set of microbenchmarks to evaluate PRISM. Further, we report the performance results of PRISM using real-world application benchmarks.

A. Experimental Setup

The experiments were performed on two Dell PowerEdge R640 machines each equipped with 20-core (40 hyperthreads) Intel Xeon Silver 4114 processors (2.2 GHz), 128 GB memory. They were directly connected (point-to-point) with Mellanox ConnectX-5 EN 100-Gigabit Ethernet. Hyperthreading and Turbo Boost were both enabled, and the CPU frequency was set to the maximum. The maximum processor C-state was set to 1 (lowest possible) to minimize the effect of processor wakeup cycles. Both machines ran Ubuntu 18.04 with Linux kernel version 5.4. For container support, we used Docker version 20.10.7 with the *overlay* mode. Docker overlay network uses Linux's builtin VXLAN to encapsulate container network packets. Network optimization features such as TSO, GRO and RSS were all enabled in all tests. In almost all experiments, the low-priority background traffic was generated with *sockperf* [21] in the UDP or TCP throughput mode and consumed 60–70% of the available CPU time on the server machine. In addition, all the network processing was directed to a single core to stress test that core.

B. Microbenchmarks

1) Streamlined processing: In this test, we compared the latency and throughput offered by Vanilla, PRISM-sync and PRISM-batch approaches in the absence of low-priority background traffic. Fig 8 shows the per-packet latency and overall throughput observed in these three approaches. In these experiments, the server was configured to dedicate one core for packet processing and an another core for user application (containerized *sockperf* UDP server). This dedicated core was used to simulate a busy server where single core is responsible for processing packets from multiple flows. The client machine was used to generate a constant load of 300 Kpps (containerized sockperf client). Compared to the Vanilla mode, the PRISM-sync mode reduced per-packet latency (both median and tail) by about 50%. Meanwhile, the PRISM-batch lies in between. Sockperf measures latency from the client application as the round-trip time divided by two.

We also compared the maximum throughput (in terms of packet rate using small packets) supported by one packet processing core on the server machine. In Fig. 8, the throughput of Vanilla and PRISM-batch are close to each other (∼400 Kpps). However, PRISM-sync mode only supports a per-core throughput of about 300 Kpps. The reason for the drop in throughput is because PRISM-sync mode lacks batching benefits, as described in Section III-B1 — it is equivalent to a packet processing system with the batch size being one.

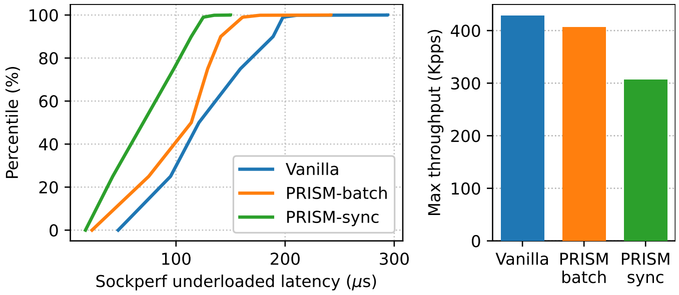


Fig. 8. Performance comparison of Vanilla, PRISM-batch, and PRISM-sync

These results show a clear trade-off between latency and throughput in a constrained system. PRISM-sync mode provides the best per-packet latency, but compromises throughput. On the other hand, Vanilla mode provides the maximum relative throughput but the worst per-packet latency. The PRISM-batch mode provides a middle-ground that can be used if the high-priority flows can afford to compromise a little latency for better throughput.

2) Priority differentiation: In this test, we compared the latency of a high-priority flow when it was running concurrently with other low-priority background traffic. In the server machine, the packet processing was done on a single core to simulate different types of flows being processed concurrently. Two containerized sockperf servers ran on separate cores, one for low-priority background traffic and the other for high-priority traffic. On the client machine, containerized sockperf clients were used to generate 1) a low-priority constant background traffic of ∼300 Kpps, which consumed about 60–70% of the receiver's packet processing core, and 2) a high-priority constant traffic of 1000 pps. We again compared the latency for various modes.

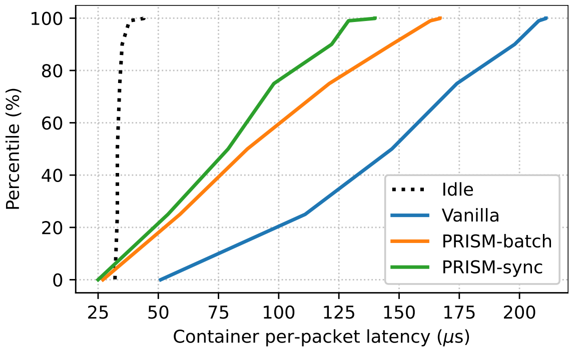


Fig. 9. Per-packet latency of high-priority container traffic in the presence of low-priority background traffic.

In Fig. 9, the dashed black line represents the per-packet latency of the high-priority flow in the *idle* case, i.e., the requests were sent to an idle server. The solid colored lines represent the per-packet latency of the high-priority flows in the *busy* case, i.e., in the presence of a constant low-priority background traffic. The three colors represent the different modes of NAPI processing. From this figure, we can see that compared to the idle case, the latency of high-priority flow degrades significantly when it is competing with other flows for CPU. Again, compared to Vanilla, PRISM-sync mode can reduce both the average and tail latency by 50%. PRISM-batch mode, on the other hand, reduces average latency better (closer to PRISM-sync) than tail latency.

Fig. 10 shows the result of a similar experiment, but on the *host* network. We can see that in this case, PRISM cannot improve the latency of high-priority flows, compared to Vanilla, when competing with low-priority background traffic. This is because the host network stack does not contain any virtual devices, and thus the host packet processing pipeline only consists of a single stage. As explained in Section IV-D, our prototype of PRISM cannot enable priority-based packet processing in the physical NIC driver (i.e., the first packet processing stage). Therefore, PRISM is most effective for a multi-stage packet processing pipeline involving multiple virtual devices (e.g., container overlay networks).

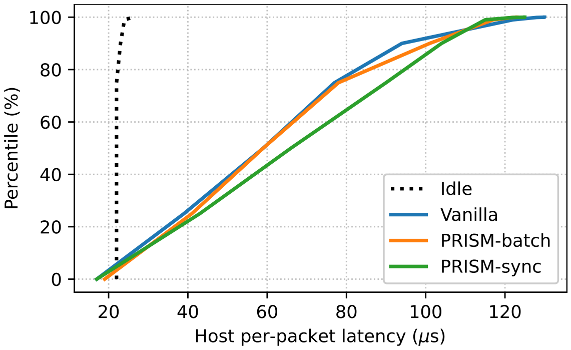


Fig. 10. Per-packet latency of high-priority host traffic in the presence of low-priority background traffic.

Moreover, Fig. 11 shows how the per-packet latency of the foreground (high-priority) traffic is affected by increasing the background load. The shaded regions represent the *minimum* and the *tail* (99th percentile) latency, whereas the solid colored line denote the average latency. The dashed line is the CPU utilization of the background packet processing. As the background load increases, the latency increases suddenly at the beginning. This is because the load is not enough to keep the CPU from going into power-saving mode, and more packets trigger the CPU sleep-wakeup cycle, which significantly affects the latency. Yet, as the background load increases to about 80–90% CPU usage, the latency decreases steadily. Once the CPU is overloaded, the latency explodes to 1–2 ms (not shown in the figure). We also see that PRISM's tail latency is close to Vanilla's average latency, and PRISM's average latency is close to Vanilla's minimum latency. Therefore, PRISM can improve the latency in all background loads.

C. Application benchmarks

1) Memcached: We also tested the effect of low-priority background traffic on the response times of memcached requests (high-priority). Memcached is a distributed in-memory key-value store that is widely used for caching web objects [22]. We used the *memaslap* [23] benchmark as the high-priority flow and sockperf UDP throughput traffic as the low-priority background traffic. The result is shown in Fig. 12. Here, for both the idle and busy server, we ran the memaslap benchmark between the two containers. We compare the result between Vanilla and PRISM-sync mode. In the idle case, there is no significant difference between Vanilla and PRISM-sync. However, in the busy case, the throughput of memcached drops significantly (by 80%) and the average memcached latency increases by more than 5×. The throughput loss is partially due to less CPU time available for memcached in the presence of background traffic. Compared to Vanilla, PRISM has almost 2× the throughput and the minimum, average and tail latencies are reduced by ∼66%, ∼47% and ∼27% respectively, similar to our microbenchmark results. In PRISM mode, the memcached latency on a busy server is closer to that on an idle server. The throughput, on the other hand, is still significantly lower than the idle case.

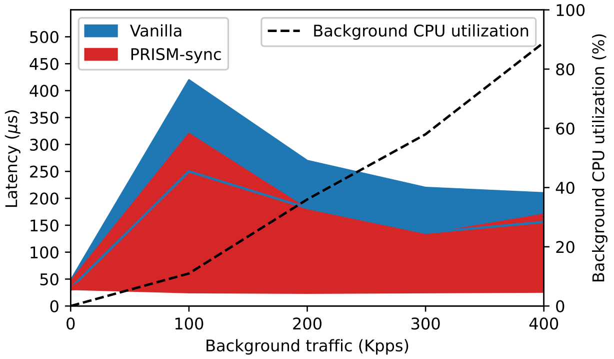


Fig. 11. Effect of changing background load on high-priority latency.

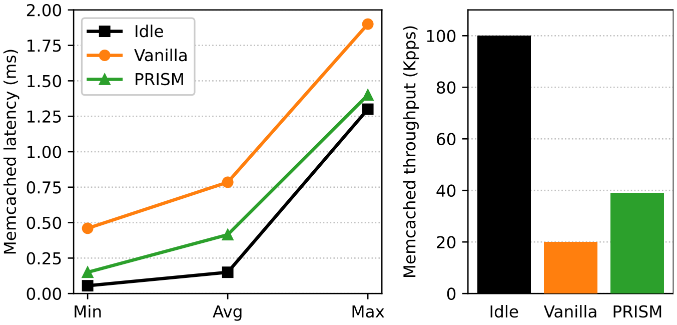


Fig. 12. Memcached performance in the presence of low-priority background traffic.

2) Web Server: We evaluated the performance of web-serving traffic (high-priority) in the presence of low-priority background traffic. We ran the Nginx [24] server in a container on the receiving host to serve a small static HTML file of less than 1 KB. Meanwhile, on a different container running on the other host, we used the *wrk2* [25] HTTP benchmarking tool with a single connection to generate high-priority web requests and measured the performance of the web traffic. Low-priority background traffic was generated using the sockperf TCP throughput test with a constant rate of 20 Kpps with 64 KB packets (which is fragmented into MTU-sized packets by the egress kernel stack).

Fig. 13 shows latency (average and tail) and throughput of the web traffic under various configurations. In the figure,

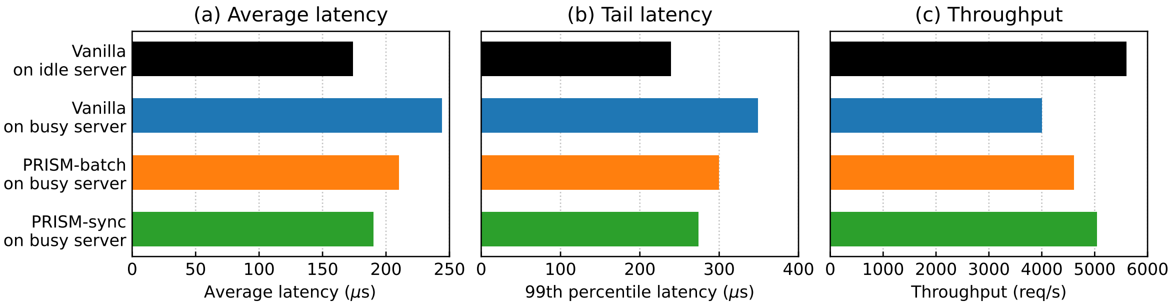


Fig. 13. Web server latency and throughput in the presence of low-priority background traffic.

a *busy* server implies the presence of the low-priority background traffic. When busy, compared to Vanilla, PRISM-batch reduces both average and tail latency by ∼14%, and improves throughput by ∼15%. Furthermore, PRISM-sync improves latency and throughput by ∼22% and ∼25% respectively. This throughput improvement for PRISM-sync contradicts the microbenchmark results. We believe in this case, the network is still dominated by low-priority background traffic which is batched to improve the overall throughput.

VI. Related Work

1) Kernel stack optimizations: The inefficiency of the in-kernel network stack has been a well-known problem, especially in light of modern fast network cards. It is reported that the Linux kernel network stack has latency and throughput many times worse than raw hardware [26]. As such, many researchers have proposed various ways to optimize different aspects of the in-kernel network stack such as reducing data copies [27]–[29], interrupt coalescing [30], improving cache locality [27], [28], [31], system call batching [32], flow steering [17] and load balancing [31].

2) OS bypass: Due to the inherent overhead of OS kernel network stack and the challenges in kernel development, many researchers have tried to completely or partially bypass the kernel network stack and propose custom network stacks. One class of work uses modern NIC capabilities and userspace network library such as DPDK [12] to couple packet processing directly with the userspace applications [13]–[15]. Another group of work propose unikernel-based library operating systems specifically designed for fast network processing [33]. Moreover, hardware manufacturers have been trying to mitigate the kernel overhead by offloading (various parts of) network processing to hardware [34]. Recently, there is a growing trend in the Linux networking community to accelerate packet processing using eBPF [19] and XDP [35].

3) Container network acceleration: Recently, a number of work has been done to specifically analyze and accelerate container overlay networks. For example, Slim [10] tries to bypass the overhead of extra virtual devices in container overlay networks while still maintaining similar API to user applications — by providing a shim library that performs IP/port manipulation behind the scene. Similarly, FALCON [11] accelerates a single overlay flow by pipelining the processing stages on to different cores.

While most of the existing work has focused on the overall network performance such as overall latency and throughput, to the best of our knowledge, PRISM is the first study focusing on priority differentiation in the kernel network stack, and is orthogonal to all the existing work.

VII. Discussion

1) Priority-differentiation in the driver: The end goal of this project is to enable *end-to-end* priority-based packet processing, from the NIC all the way to user-level applications. In our proof-of-concept implementation, we have managed to enable PRISM on the generic driver used by the virtual network devices such as bridge and virtual Ethernet (`veth`) interfaces. However, reaping the full potential of this design also requires implementing PRISM both 1) when the packet is first received by the physical NIC, and 2) when the packet has finished the protocol processing and is ready to be delivered to the user-level applications. Enabling PRISM in the physical NIC is challenging as it requires modifying the packet queuing logic in the *vendor-specific* device drivers. While this has to be done separately for each driver, the idea should be simple enough and be portable to almost all network drivers. We leave it as one of our ongoing work. Due to this limitation, PRISM cannot differentiate priority in host network.

2) Kernel-user interface: Synchronizing the kernel-user interface is another challenge. Even when the packet processing is prioritized in the kernel stack for a specific flow, the packets may still have to wait an indefinite amount of time waiting for the user-level applications to wake up. This wakeup time is worse when a user application is running on a different core as it also adds inter-processor communication (IPC) overheads. We believe that our solution can benefit more if user applications, just like network packets, are also prioritized and can be preempted. This requires significant study and we leave it as a future work.

3) Multiple priority levels: Our current design only considers the simple case of *two* levels of priorities based on the two classes of network traffic: 1) high-volume throughput-intensive flows, and 2) low-volume latency-sensitive flows. While this classification is sufficient for many scenarios, there may be more complex scenario where a more fine-grained priority control is desired. We believe that it should be feasible to extend PRISM to multiple priority levels, and we leave it as a future study.

4) Effect on other applications: PRISM modifies the NAPI design of the in-kernel network stack — it only changes the order how incoming packets are processed in the kernel. However, the total amount of packet processing work, which depends on the total number of packets, remains unchanged. Additionally, the packet processing logic executes under the softirq context, which has a strictly higher priority than other user and/or kernel threads. Therefore, as long as the CPU has packets to be processed, whether in Vanilla or PRISM mode, packet processing always has a higher priority — there is a possibility of starvation for other threads that wait to run on that CPU. That said, PRISM does not affect other applications in a different way compared to Vanilla.

VIII. CONCLUSION

The paper has showed that the competition with throughput-intensive background traffic negatively affects the per-packet latency of short-lived, latency-sensitive flows. This problem becomes more prominent for container overlay network which involves multiple stages in its packet processing pipeline. We have demonstrated that it is largely due to the head-of-line blocking and the inefficient interleaved packet processing in the kernel network stack. We have presented PRISM, a new design to enable priority-based streamlined packet processing for container overlay networks. PRISM not only streamlines the interleaved packet processing pipelines, but also minimizes the head-of-line blocking by allowing the high-priority flows to preempt the low-priority flows. The evaluation of our prototype on both microbenchmarks and real-world applications shows that PRISM can significantly reduce the packet processing latency of high-priority flows by more than 50% and increase the throughput by up to 100% in the presence of low-priority background traffic.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported by NSF under Award 1909877 and 1909486.

REFERENCES

[1] "tc - show / manipulate traffic control settings." [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[3] K. Finley, "Amazon embraces docker, following Google and Microsoft's lead." [Online]. Available: https://www.wired.com/2014/11/following-google-microsoft-amazon-embraces-next-big-thing-cloud-computing

[4] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," Internet Requests for Comments, Aug. 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7348

[5] "Calico." [Online]. Available: https://github.com/projectcalico/calico

[6] "Weave." [Online]. Available: https://github.com/weaveworks/weave

[7] "Flannel." [Online]. Available: https://github.com/flannel-io/flannel

[8] "Docker overlay network." [Online]. Available: https://docs.docker.com/network/overlay

[9] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for container overlay networks," in *USENIX HotCloud '19*, Renton, WA, Jul. 2019.

[10] 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 *USENIX NSDI '19*, Boston, MA, Feb. 2019, pp. 331–344.

[11] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *EuroSys '21*, Online Event, UK, 2021, pp. 261–276.

[12] Linux Foundation, "Data plane development kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org

[13] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *ACM SOSP '17*, Shanghai, China, 2017, p. 325–341.

[14] 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 *USENIX OSDI '14*, Broomfield, CO, Oct. 2014, pp. 49–65.

[15] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *USENIX ATC '12*, Boston, MA, Jun. 2012, pp. 101–112.

[16] "Driver porting: Network drivers." [Online]. Available: https://lwn.net/Articles/30107

[17] "Scaling in the Linux networking stack." [Online]. Available: https://static.lwn.net/kerneldoc/networking/scaling.html

[18] "Open vSwitch." [Online]. Available: https://github.com/openvswitch/ovs

[19] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Computing Surveys*, vol. 53, no. 1, Feb. 2020.

[20] "proc(5) - linux man page." [Online]. Available: https://linux.die.net/man/5/proc

[21] "Sockperf." [Online]. Available: https://github.com/Mellanox/sockperf

[22] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[23] "Memaslap." [Online]. Available: http://docs.libmemcached.org/bin/memaslap.html

[24] "Nginx open-source web server." [Online]. Available: https://nginx.org

[25] "wrk2." [Online]. Available: https://github.com/giltene/wrk2

[26] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *USENIX OSDI '14*, Broomfield, CO, Oct. 2014, pp. 1–16.

[27] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein, "Network stack as a service in the cloud," in *ACM HotNets '17*, Palo Alto, CA, USA, Nov. 2017, p. 65–71.

[28] N. Binkert, L. Hsu, A. Saidi, R. Dreslinski, A. Schultz, and S. Reinhardt, "Performance analysis of system overheads in tcp/ip workloads," in *PACT '05*, 2005, pp. 218–228.

[29] 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*, Houston, Texas, USA, 2013, p. 461–472.

[30] "Performance tuning for mellanox adapters." [Online]. Available: https://community.mellanox.com/s/article/performance-tuning-for-mellanox-adapters

[31] L. Cheng and C.-L. Wang, "VBalance: Using interrupt load balance to improve I/O performance for SMP virtual machines," in *ACM SoCC '12*, San Jose, CA, 2012.

[32] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with Exception-Less system calls," in *USENIX OSDI '10*, Vancouver, BC, Oct. 2010.

[33] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with ClickOS," in *ACM HotSDN '13*, Hong Kong, China, 2013, p. 67–72.

[34] "Mellanox ASAP^2: Accelerated switching and packet processing." [Online]. Available: https://www.mellanox.com/files/doc-2020/sb-asap2.pdf

[35] "XDP: express data path." [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP