IEICE TRANS. COMMUN., VOL.E105–B, NO.5 MAY 2022

PAPER

KBP: Kernel Enhancements for Low-Latency Networking for Virtual Machine and Container without Application Customization*

Kei FUJIMOTO[†a)], **Masashi KANEKO**[†], ***Members***, **Kenichi MATSUI**[††], ***Senior Member***, ***and*** **Masayuki AKUTSU**[†], ***Member***

SUMMARY Packet processing on commodity hardware is a cost-efficient and flexible alternative to specialized networking hardware. However, virtualizing dedicated networking hardware as a virtual machine (VM) or a container on a commodity server results in performance problems, such as longer latency and lower throughput. This paper focuses on obtaining a low-latency networking system in a VM and a container. We reveal mechanisms that cause millisecond-scale networking delays in a VM through a series of experiments. To eliminate such delays, we design and implement a low-latency networking system, kernel busy poll (KBP), which achieves three goals: (1) microsecond-scale tail delays and higher throughput than conventional solutions are achieved in a VM and a container; (2) application customization is not required, so applications can use the POSIX sockets application program interface; and (3) KBP software does not need to be developed for every Linux kernel security update. KBP can be applied to both a VM configuration and a container configuration. Evaluation results indicate that KBP achieves microsecond-scale tail delays in both a VM and a container. In the VM configuration, KBP reduces maximum round-trip latency by more than 98% and increases the throughput by up to three times compared with existing NAPI and Open vSwitch with the Data Plane Development Kit (OvS-DPDK). In the container configuration, KBP reduces maximum round-trip latency by 21% to 96% and increases the throughput by up to 1.28 times compared with NAPI.

key words: *low latency, high throughput, network, kernel, virtual machine, container*

1. Introduction

Packet processing on commodity hardware is a cost-efficient and flexible alternative to specialized networking hardware. Network function virtualization (NFV) [2] has been proposed by the European Telecommunications Standards Institute (ETSI) and is being put into practical use in telecom networks. In addition, multi-access edge computing (MEC) [3] has been proposed by the ETSI, which enables terminal devices to offload their workloads to edge clouds located near the terminal. From an operational flexibility perspective, virtual servers such as virtual machines (VMs) and containers are used in the edge cloud. However, virtualizing dedicated hardware as VMs or containers on commodity servers results in performance problems [4]–[7], such as longer latency and lower throughput. Since low latency is required by various real-time applications, such as online gaming, virtual reality, and autonomous vehicles [8]–[10], and a virtual server such as a VM and a container is expected to be used for operational flexibility, packet processing delay in a VM or a container needs to be reduced.

Two requirements from application-developer-friendly and service-provider-operability viewpoints should be considered when designing a low latency solution for a VM or a container. From an application-developer-friendly viewpoint, application customization should not be required. Application developers no longer need to develop networking functions in applications since a kernel provides networking functions and developers can easily use these functions via the POSIX sockets application program interface (API) [11], [12]. If a solution requires customizing applications, developers need to study and develop complicated networking functions, which impedes widespread use of the solution. In addition, from a service-provider viewpoint, software should not need to be developed for reducing networking delay to keep up with Linux kernel security updates. If software for reducing delay is needed to keep up with kernel updates, which incurs high cost, service providers must continue to develop new versions of the software for every kernel security update. As discussed in Sect. 2, existing technologies to reduce delay in a VM or a container cannot meet the requirements of making both application customization and redevelopment with kernel security update unnecessary.

We thus designed and implemented a low-latency networking system, kernel busy poll (KBP), which achieves three goals: (1) microsecond-scale tail delays and higher throughput than conventional solutions are achieved in a VM and a container; (2) application customization is unnecessary, so an application can use the POSIX sockets API; and (3) software for reducing networking delay does not need to be developed for every Linux kernel security update. KBP has a kernel thread that constantly checks for the arrival of incoming network packets without being interrupted and immediately transfers them to a kernel network protocol stack. Since these changes for existing kernels are minimal, KBP can be applied using a kernel livepatch framework [13] and does not need to be developed every time kernel security is updated, unless there is a change in *netif_rx_schedule* or

Manuscript received June 17, 2021.
Manuscript revised August 23, 2021.
Manuscript publicized October 26, 2021.

[†]The authors are with the NTT Network Service Systems Laboratories, NTT Corporation, Musashino-shi, 180-8585 Japan.
[††]The author is with the Nippon Telegraph and Telephone West Corporation, Osaka-shi, 534-0024 Japan.

[*]A part of this paper was presented at IEEE CCNC 2021 [1].

a) E-mail: kei.fujimoto.rg@hco.ntt.co.jp

DOI: 10.1587/transcom.2021EBT0004

netif_receive_skb. In addition, KBP does not change the existing POSIX sockets API, so application customization is unnecessary. KBP can be applied to both a VM configuration and a container configuration.

The rest of this paper is organized as follows:

- Understanding of millisecond-scale networking delays: mechanisms that cause millisecond-scale packet-forwarding delays in a VM are revealed (see Sect. 3.2);
- Design and implementation of KBP: we designed and implemented a low-latency networking system that avoids the causes of networking delays (see Sect. 4);
- Demonstration of the benefits of KBP: our experimental results indicate that KBP can reduce packet-forwarding delay on the microsecond-scale and improve throughput in both a VM and a container compared with conventional technologies (see Sects. 5 and 6).

2. Related Work

The current Linux kernel uses a receiver (Rx) interrupt processing scheme called NAPI [14], adopted from kernel 2.4.20. Instead of using one hardware interrupt request (hardIRQ) per packet, NAPI combines interrupts with polling so that multiple packets can be processed within a single hardIRQ. NAPI achieves higher throughput than a conventional scheme of per-packet interrupt. However, NAPI is an interrupt model, and packet processing occurs via software interrupt requests (sofIRQs). As discussed in Sect. 3.2, these softIRQs incur millisecond-scale delays.

The Data Plane Development Kit (DPDK) [15] is a user-space packet-processing framework. The DPDK enables low-latency packet processing since a DPDK Poll Mode Driver (PMD) can directly access a network interface card (NIC) over a kernel and does not generate any interrupt when packets arrive [16]. To use the DPDK, a PMD and layer 2 (L2) / layer 3 (L3) / layer 4 (L4) protocol processing functions must be integrated into an application. The DPDK does not meet the requirement of making application customization unnecessary. In addition, since a polling thread is placed in a user space, the DPDK does not have functions such as deadlock control for prohibiting multiple executions of softIRQs, which must be considered if a polling thread is placed in a kernel.

AF_XDP [17] is an express data path, and a user-space program can directly access Rx and transmitter (Tx) rings in a NIC via an AF_XDP socket. To use AF_XDP, a function of interworking the specialized AF_XDP socket interface and L2/L3/L4 protocol processing functions must be integrated into an application. Thus, AF_XDP does not meet the requirement of making application customization unnecessary either.

ptnet [18] improves the performance of existing VM network I/O mechanisms by enabling a guest to directly pass through a host physical interface via *netmap* [19]. If a guest application is a *netmap*-compatible program, the application can directly access a host NIC. However, this does not meet the requirement of making application customization unnecessary. Traditional socket applications also can use this *ptnet* network I/O mechanism by deploying the original *ptnet* driver in a guest. However, the driver generates softIRQs to notify a guest of arriving packets. As discussed in Sect. 3.2, these softIRQs incur millisecond-scale delays. Either way, *ptnet* requires the original NIC driver and hypervisor. If a security update occurs, we need to continue to develop the driver and hypervisor for every security update.

Busy Poll Sockets (BPS) [20] enhances the native kernel networking stack by enabling the socket layer code to directly poll an Rx queue of a Ethernet device and provides low-latency networking performance without application customization. However, BPS needs to modify a device driver and core parts of the kernel, such as *sk_buff* and a kernel socket. In BPS, an application thread evokes a system call to trigger a polling thread with a fixed execution time in a context of the application, not a dedicated kernel thread for polling. Therefore, application developers need to be aware of the timing of busy polling of BPS when developing applications, and they also need to tune and set the polling execution time. Other approaches to reduce delay in a virtual server are implementing a run-to-completion networking feature [21] and implementing the original scheduler and inter-processor interrupts algorithm [22]–[24]. These solutions need to modify core parts of a kernel and do not meet the requirements of not needing to develop an original kernel and not needing to continue to develop it for every kernel security update.

DMM lwIP [25] and Slim [26] provide low-latency data paths without the need to customize an application or modify a kernel. They use the LD_PRELOAD mechanism available in Linux that enables interception of any function call to any shared library to intercept the POSIX sockets API. However, the LD_PRELOAD mechanism does not work for applications that do not use *lib.c* libraries, such as *socket.c* for socket programming.

3. Analysis of Networking Delays

To design a low-latency data path, we need to know key factors of packet-forwarding delays in a server. Since a VM has a larger virtualization overhead and incurs higher latency than a container, we used a VM and analyzed the factors that cause packet-forwarding delays. As discussed in previous work [1], we measured networking delays by setting various measurement points and analyzed the causes by using the *ftrace*, kernel tracing tool. In Sect. 3.1, we discuss the measurement results regarding networking delays in a VM, and discuss millisecond-scale delay occurrence mechanisms and current countermeasures to suppress these delays in Sect. 3.2. On the basis of these mechanisms, we present KBP for a low-latency data path in Sect. 4.

3.1 Measurement Results of VM Networking Delays

For measuring networking delays inside a kernel, we used

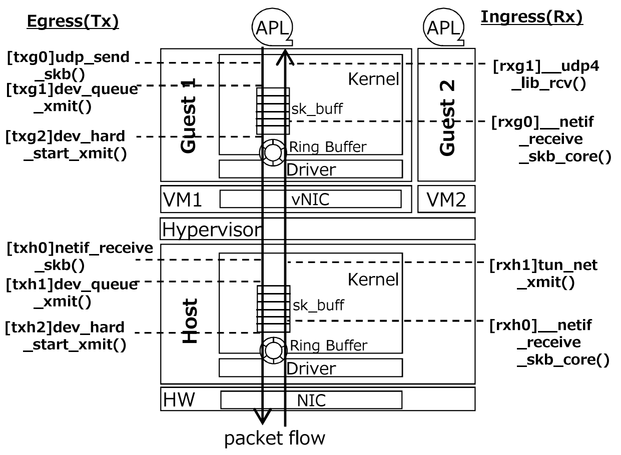

Fig. 1 Delay measurement points in virtual server.

Table 1 Experimental platform specifications.

Host server	
CPU	Intel Xeon CPU D-1587 1.70 Ghz
Memory	32 GB
NIC	Intel Ethernet Controller I350AM2 (1000Base-T)
OS	Ubuntu 18.04
kernel	4.15.0-20-generic
Hypervisor	KVM
Guest server	
vCPU	2 cores
Memory	1 GB
OS	Ubuntu 18.04
kernel	4.15.0-20-generic

Table 2 Maximum packet-forwarding delays in VM.

	interval	txg0~txg1	txg1~txg2	txg2~txh0	txh0~txh1	txh1~txh2
Tx	maximum delay	32μs	**3.6ms**	**4.7ms**	37μs	**2.5ms**

	interval	rxh0~rxh1	rxh1~rxg0	rxg0~rxg1
Rx	maximum delay	96μs	**2.6ms**	75μs

the extended Berkeley Packet Filter (eBPF), a built-in kernel feature that enables eBPF programs to be executed at hook points. We set measurement hook points as shown in Fig. 1. Table 1 lists the platform specifications for this measurement. We used a kernel-based virtual machine (KVM) hypervisor-based VM (KVM is an inbuilt hypervisor available in Linux) and *vhost-net*, a kernel-level back-end for virtio networking that reduces virtualization overhead by moving virtio-packet processing tasks out of the QEMU process. Table 2 lists the results of delay measurement at 1-Gbps one-way traffic with 1518-byte user datagram protocol (UDP) frames. Since the eBPF approach negatively affects packet forwarding performance [27], the results listed are worse than those without measuring delay. However, we can find bottlenecks where delays occur. The bottlenecks are txg1-txg2, txg2-txh0, and txh1-txh2 in the Tx traffic flow and rxh1-rxg0 in the Rx traffic flow.

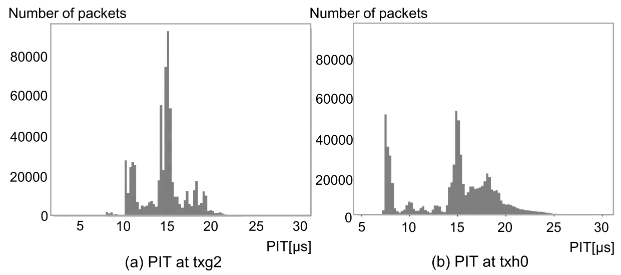

Fig. 2 Measurement results of PIT.

3.2 Mechanisms of Packet-Forwarding Delays in Server

In this section, we discuss the mechanisms that cause the millisecond-scale delays shown in Table 2. We analyzed the causes of the millisecond-scale delays by using trace tools, such as *ftrace*, and found the following are millisecond-scale delay-occurrence mechanisms:

3.2.1 Congestion in Physical NIC

Kernel and virtual NIC (vNIC) packet processing, such as *virtio-net* and *vhost-net*, causes packet jitter. We measured the packet interval time (PIT) at a 1-Gbps traffic rate with 1518-byte UDP frames. Figure 2 shows the results of PIT-measurement tests at txg2 and txh0 in the Tx traffic flow. In theory, the PIT should be 12 μs. However, after passing through a kernel and vNIC, the number of packets arriving at intervals of 7 to 8 μs increased. This means that microburst occurs after passing through a kernel and vNIC. Since the 1000BASE-T Gigabit Ethernet NIC cannot process over 1-Gbps traffic, this microburst incurs congestion in the NIC, and this congestion causes a 2.5-ms delay of interval txh1-txh2 of Tx in Table 2. A higher-throughput-performance NIC can suppress this delay.

3.2.2 CPU Frequency Variation

To reduce power consumption and heat output, the central processing unit (CPU) clock speed changes dynamically. A governor defines the power characteristics of the system CPU. For example, when *cpufreq_ondemand* is set, the minimum clock frequency is used when the system is idle. A lower frequency clock causes packet congestion in kernel-packet processing. This is one cause of delays of intervals txg1-txg2 and txg2-txh0 of Tx and rxh1-rxg0 of Rx in Table 2. Such delays can be suppressed by using a high-performance governor, such as *cpufreq_performance*.

3.2.3 Instantaneous Stop of vCPU

The KVM virtual CPU (vCPU) emulation thread cannot completely monopolize a physical CPU core since this core is interrupted by hardIRQs, such as local timer interrupts, and interfered with by high-priority kernel threads such as migration threads. This causes the vCPU to instantaneously stop, and the guest kernel cannot process packets during these periods. This is one cause of delays of intervals txg1-txg2 of Tx and rxh1-rxg0 of Rx in Table 2. These delays can be suppressed by setting a real-time scheduling policy for the KVM vCPU thread, such as using the *chrt* command.

3.2.4 Competition between Vhost-Net and Other Kernel Threads

When a CPU core used by *vhost-net* is interfered with by other kernel threads, *vhost-net* cannot process packets during this period and packet delay occurs. This is another cause for delays of intervals txg1-txg2 of Tx and rxh1-rxg0 of Rx in Table 2. These delays can be suppressed by setting a dedicated CPU core for *vhost-net*, such as using *taskset* and *isolcpus*.

3.2.5 softIRQ Competition

To notify the guest kernel of arriving packets, *vhost-net* notifies a KVM kernel thread of the arriving packet and triggers a guest interrupt using *eventfd*. After this, a softIRQ occurs in the guest kernel for packet processing. When other softIRQs, such as local timer interrupts and *ata_piix*, compete with the softIRQ, *ksoftirqd* schedules these softIRQs, which must wait until the scheduled time. Furthermore, when the *ksoftirqd* is not given enough CPU time, softIRQ scheduling is delayed. This waiting causes delays of interval rxh1-rxg0 of Rx in Table 2. This competition is unavoidable with any tuning. This softIRQ competition can occur not only in the guest, but also in the host. So this softIRQ can also occur in the host of a container-based server.

4. KBP Architecture

Since the delay factor of the softIRQ competition mentioned in Sect. 3.2.5 cannot be avoided when using any tuning or high-throughput performance NIC, we designed KBP [1] to avoid the softIRQ competition. Figure 3 shows the high-level architecture of KBP compared with NAPI. To avoid the softIRQ competition and achieve the goal of (1) microsecond-scale tail delays and high throughput, KBP has a kernel thread, KBP thread, which constantly checks for the arrival of incoming network packets without being interrupted and immediately transfers them to a kernel network protocol stack (see Sect. 4.1). To achieve the goal of (2) making application customization unnecessary, KBP does not change the existing POSIX sockets API (see Sect. 4.2). In addition, to achieve the goal of (3) making it unnecessary to develop software for every Linux kernel security update, KBP can be deployed to existing kernels by using a kernel livepatch framework (see Sect. 4.3). No existing technology has all these features: having a busy-poll thread in a kernel, not modifying the existing kernel protocol stack, and being deployed by a kernel livepatch. To achieve these three requirements, a low-latency and high-throughput feature such as busy-poll thread needs to be developed in a kernel without modifying the existing kernel protocol stack. However, in a kernel, there is a restriction that when a softIRQ is being executed on a CPU core, another softIRQ cannot be executed on the same CPU core. Busy polling in a kernel is technically challenging to perform under this restriction. In addition, if we develop a busy-poll thread in a kernel without any effort, we will need to develop our own kernel, and we need to keep developing the kernel for every kernel security update. To avoid this, KBP was designed to reduce the number of changes to the existing NAPI as much as possible, and thus only two changes to NAPI were needed to create a polling thread in a kernel. Thus, KBP can be applied to a kernel by using a framework of the kernel livepatch. KBP can be applied to both a guest kernel and a host kernel for VM architecture as shown in Fig. 4 and to a host kernel for container architecture as shown in Fig. 5. KBP has the disadvantages that it limits flexibility of CPU core usage for applications, VMs, and containers and possibly consumes more power than NAPI since KBP occupies a CPU core for busy polling.

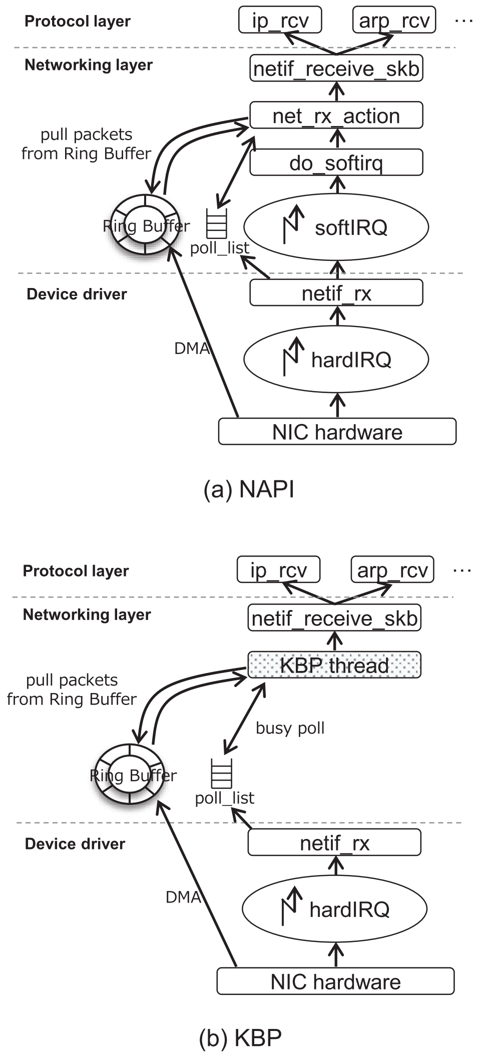

Fig. 3 High-level architecture.

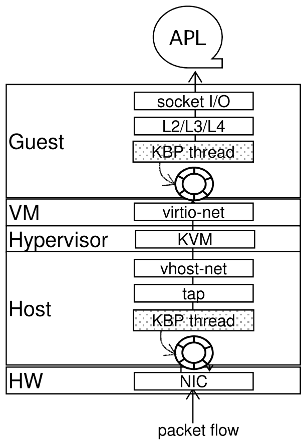

Fig. 4 KBP architecture for VM.

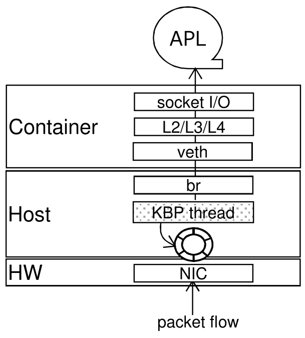

Fig. 5 KBP architecture for container.

4.1 Busy Poll Feature

To suppress softIRQs for packet processing that cause delays as mentioned in Sect. 3.2.5, KBP uses a polling model instead of an interrupt model. The following lists the changes made to NAPI.

- In NAPI, when NIC receives packets and delivers them to an Rx queue, it raises a hardIRQ and activates a softIRQ with type *NET_RX_SOFTIRQ* in *netif_rx_schedule*. KBP disables this *NET_RX_SOFTIRQ* and never activates a softIRQ.
- KBP has an original kernel thread, KBP thread, which constantly checks a *poll_list* for the arrival of incoming packets without being interrupted and immediately transfers the packets from a ring buffer to *netif_receive_skb*.

A KBP thread is deployed in a kernel as a kernel thread. In a kernel, there is a restriction that when a softIRQ is being executed on a CPU core, another softIRQ cannot be executed on the same CPU core. Polling performed by a KBP thread and the subsequent processes such as *netif_receive_skb* are originally performed in the context of softIRQs in the base system, NAPI. If a KBP thread keeps busy polling without deadlock control, it will fall under the restriction of prohibiting multiple executions of softIRQs, and other softIRQs will not be executed. Busy polling is technically challenging to perform in a kernel. Therefore, KBP enables a deadlock control mechanism that prohibits the execution of other softIRQs when a KBP thread receives a packet and releases the prohibition once the packet processing is finished. In addition, since a KBP thread keeps polling all the time in a kernel, application developers do not need to consider the timing of polling or tuning the polling execution time at all, making KBP an application developer-friendly solution for low latency.

For low latency and high throughput, a KBP thread occupies a CPU core and should not be preempted by other processes. This may limit flexibility of CPU core usage for applications, VMs, and containers. In addition, the busy polling prevents the CPU from sleeping to save power, possibly incurring greater power usage. There is a trade-off between low latency and power savings.

The steps of packet processing in a kernel with KBP are as follows:

1. When NIC receives a packet, it directly copies the packet to a ring buffer via direct memory access and notifies a kernel by raising a hardIRQ, which calls *netif_rx_schedule* and adds pointer information to a *poll_list*. *NET_RX_SOFTIRQ* is disabled and never activates a softIRQ.
2. The KBP thread constantly checks whether the *poll_list* is empty or not. If the *poll_list* is not empty, it disables other softIRQs and derives the pointer information from the *poll_list* and pulls packets from the ring buffer. After this, it passes them to *netif_receive_skb* for further processing. After the processing, the prohibition of other softIRQs is released.

In VM configuration, the delay factor of the softIRQ competition mentioned in Sect. 3.2.5 can occur in a guest kernel and a host kernel. Therefore, KBP should be deployed in both the guest kernel and the host kernel as shown in Fig. 4. Especially, since the *ksoftirqd* in the guest kernel is more likely to run out of CPU time due to the overhead of emulating a VM and cause delays, applying KBP to the guest kernel is more effective. For further lower latency and higher throughput, KBP should be deployed in both the guest kernel and the host kernel. In container configuration, the delay factor of the softIRQ competition can occur in a host kernel. Therefore, KBP should be deployed in the host kernel as shown in Fig. 5. This is supplemental information, but softIRQs occur in not only the host kernel but also a container when *veth* (virtual ethernet device) receives a packet. Since *veth* is not a NAPI device, the old receiving function *process_backlog* is used. These softIRQs for *veth* are immediately processed right after they are invoked and the softIRQ competition mentioned in Sect. 3.2.5 is hard to occur. As we discuss in Sect. 6, we have found that applying KBP only to the host kernel can suppress delays.

In terms of scaling out, adding more KBP threads

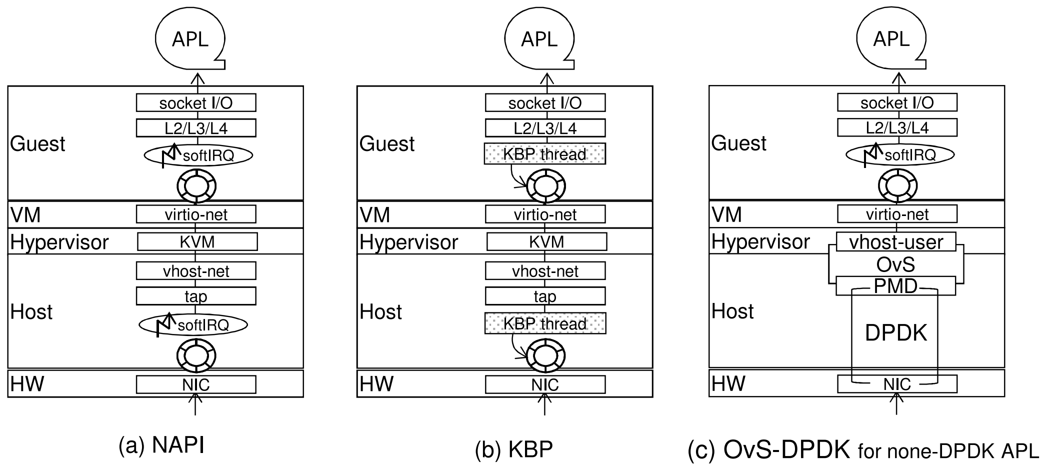

Fig. 6 High-level architecture of solutions for VM to be evaluated.

can handle more traffic flow streams. By configuring *smp_affinity* for the CPU scale-out mechanism of receive-side scaling (RSS), a service provider can manage the number of CPU cores used for packet processing and invoke the KBP threads associated with these CPU cores.

In terms of isolation, when a dedicated CPU core is allocated to a KBP thread and CPU cores used by applications, VMs, and containers are allocated separately, KBP does not degrade the performance of the applications, the VMs, or the containers.

4.2 No Modification of Kernel Protocol Stack

Since KBP does not change the existing kernel protocol stack upstream from *netif_receive_skb*, applications can use the POSIX sockets API. Service providers can use KBP without any application customization.

4.3 Deploying by Kernel Livepatch

The kernel livepatch enables critical kernel security updates or additional functions to be installed without rebooting a system by directly patching the running kernel. KBP can be applied to a kernel by using a framework of the kernel livepatch since it needs only two changes to NAPI: disabling *NET_RX_SOFTIRQ* in *netif_rx_schedule* and starting up a KBP thread. By using this framework, service providers do not need to develop their original kernel and can obtain low latency by simply applying the livepatch of KBP to the kernel. Unless *netif_rx_schedule* or *netif_receive_skb* is changed, the livepatch of KBP does not need to be modified. In addition, when a security update occurs in a kernel, it can be dealt with by reapplying the livepatch of KBP after the security update.

5. Performance Evaluation of VM

We conducted a series of experiments of VM to compare KBP with current solutions, NAPI and Open vSwitch with the DPDK (OvS-DPDK) for none-DPDK applications [28] as discussed in previous work [1]. Figure 6 shows the high-level architecture of these solutions. These solutions met the requirements of making both application customization and redevelopment with each kernel security update unnecessary. In this OvS-DPDK architecture, the DPDK was not used for a guest application to meet the requirement of making application customization unnecessary as shown in Fig. 3. We measured latency and throughput performance. Table 3 lists the platform specifications for these experiments. To use a high-throughput NIC, we used a different machine from the one used in the analysis of networking delays in Sect. 3.1 due to the space limitations of the PCI card. Note that the CPU performance used for this performance evaluation is higher than the CPU used for the analysis of networking delays in Sect. 3.1. To suppress the delay factors mentioned in Sect. 3.2.1 to Sect. 3.2.4, we used a high-throughput NIC of Intel X520 DP 10Gb, the CPU cores that process packets were set using a high performance governor, KVM vCPU thread and *vhost-net* were set as the real-time scheduling policy, and these CPU cores were isolated from user processes using *isolcpus*. In the OvS-DPDK architecture, dedicated CPU cores were allocated for the DPDK PMD threads and isolated from user processes using *isolcpus*. The server to be evaluated and a traffic generator were directly connected with an optical cable. We added a traffic load to a single

Table 3 Experimental platform specifications of VM.

Host server	
CPU	Intel Xeon CPU E5-2660 v4 2.0 GHz
Memory	64 GB
NIC	Intel X520 DP 10 Gb DA/SFP+
OS	Ubuntu 18.04
kernel	4.15.0-20-generic
Hypervisor	KVM
Guest server	
vCPU	4 cores
Memory	4 GB
OS	Ubuntu 18.04
kernel	4.15.0-20-generic

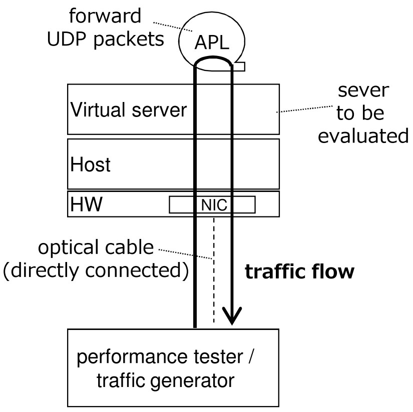

Fig. 7 Evaluation configuration.

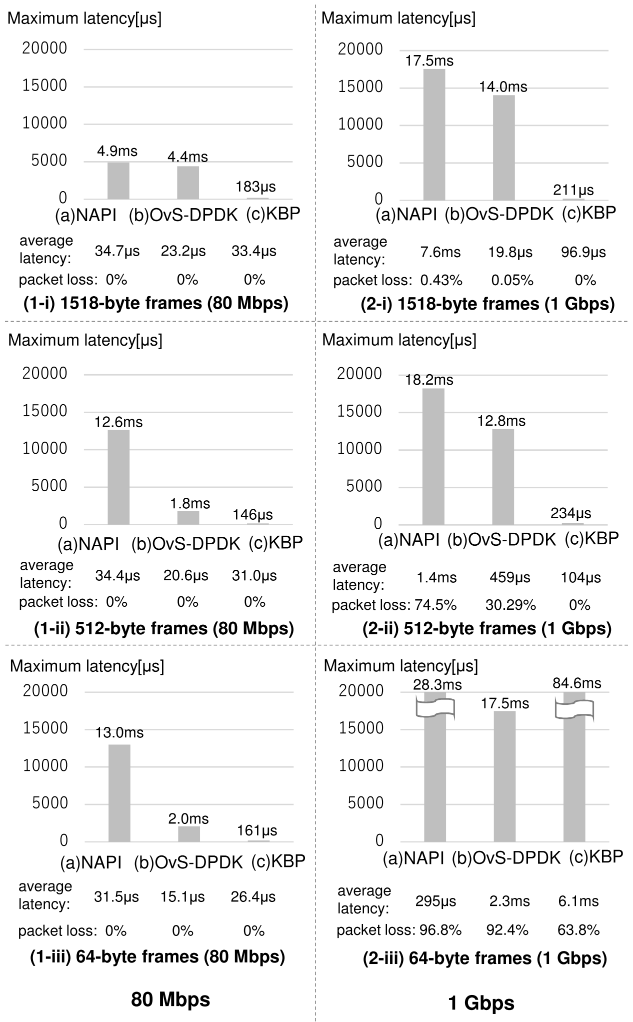

Fig. 8 Latency performance.

flow.

5.1 Latency

5.1.1 Measurement Methodology

We measured round-trip latency by using a performance tester, Spirent Test Center SPT-N4U (STC). Figure 7 shows the traffic flow. The STC sent a UDP packet to the server to be evaluated. The server received the packet and transmitted it to an application on the server. The application transmitted the packet to the STC. The STC received the packet and then calculated the round-trip latency between sending and receiving times. We added a traffic load and measured the maximum round-trip latency for 60 s and repeated the measurement 5 times.

5.1.2 Results

The left side of Fig. 8 shows the measured maximum round-trip latency at 80-Mbps traffic rate with 64-, 512-, and 1518-byte UDP frames. The 80-Mbps traffic rate was that at which no packet loss occurred in NAPI, OvS-DPDK, or KBP. KBP achieved maximum round-trip latency within 200 μs with all byte frames. NAPI and OvS-DPDK incurred millisecond-scale latencies with all byte frames. As discussed in Sect. 3.2.5, NAPI and OvS-DPDK cannot prevent softIRQ competition, which caused these latencies. On the other hand, KBP can prevent softIRQ competition and immediately transfers incoming packets to a kernel network protocol stack by busy polling, KBP improves latency performance.

The right side of Fig. 8 shows the measured maximum round-trip latency at a 1-Gbps traffic rate with 64-, 512-, and 1518-byte UDP frames. KBP achieved maximum round-trip latency within 250 μs with 512- and 1518-byte frames. However, it incurred millisecond-scale latency with 64-byte frames. At 1 Gbps with 64-byte frames, frames arrived at 0.5 μs intervals. According to the measurement experiments using an eBPF program discussed in Sect. 3.1, the average time to process a frame from rxg0 to rxg1 for Rx is 2.20 μs; the guest kernel cannot process the frames at 0.5 μs intervals, causing frame congestion. The congestion incurs delays and frame losses. Since the average time to process a frame from rxg0 to rxg1 for Rx is 2.20 μs, the frame-size limit at which KBP can suppress delay is 282 bytes at 1-Gbps traffic rate in theory. NAPI and OvS-DPDK incurred millisecond-scale latencies and packet loss with all byte frames.

KBP improved average round-trip latency compared with NAPI except for at a 1-Gbps traffic rate with 64-byte frames. In terms of average latency, OvS-DPDK outperforms KBP. KBP uses an existing framework of *vhost-net* for interworking with *virtio-net*, and this incurs higher costs than *vhost-user* of OvS-DPDK. However, OvS-DPDK incurred millisecond-scale delays. The frequency at which millisecond-scale delays occur depends on the frequency of the softIRQ competition, such as local timer interrupts and swapping. In this environment, instantaneous burst delays over a millisecond occurred once per second. OvS-DPDK is not suitable for mission-critical services such as controlling drones since control signals can be delayed. On the other hand, KBP achieved microsecond-scale tail latencies and suits those services. To elucidate stability, we added a traffic load for 12 hours, and KBP achieved maximum round-trip latency within 250 μs except for at a 1-Gbps traffic rate with 64-byte frames.

5.2 Throughput

5.2.1 Measurement Methodology

We adopted the RFC2544 [29] benchmark to measure throughput at zero frame loss. We used the Pktgen-DPDK [30] as a traffic generator. Figure 7 shows the traffic flow. The Pktgen-DPDK sent a UDP packet to the server to be evaluated. The server received the packet and transmitted it to an application on the server. The application transmitted the packet to the Pktgen-DPDK, and the Pktgen-DPDK received the packet. We repeatedly added a traffic load for 60 s while gradually increasing the load amount little by little and calculated the maximum throughput at zero frame loss. We repeated the measurement 10 times.

5.2.2 Results

Figure 9 shows the results of throughput measurement. KBP achieved up to 1.36× higher throughput than NAPI and 2.77× higher throughput than OvS-DPDK with 64-byte UDP frames, up to 1.75× higher throughput than NAPI and 6.48× higher throughput than OvS-DPDK with 512-byte UDP frames, and up to 3.28× higher throughput than NAPI and 3.39× higher throughput than OvS-DPDK with 1518-byte UDP frames. As discussed in Sect. 3.2.5, NAPI and OvS-DPDK cannot prevent softIRQ competition, so retrieving incoming packets from *poll_list* is delayed, which causes buffer overflow and packet loss. Since this RFC2544 benchmark is an evaluation method that measures the maximum throughput without packet loss, throughput performance is measured to be low if packet loss is likely to occur. In this OvS-DPDK architecture, since packets are transferred fast in the host by the DPDK, buffer overflow is more likely to occur at the bottleneck point in the guest than NAPI. On the other hand, KBP can prevent softIRQ competition and immediately transfers incoming packets to a kernel network protocol stack by busy polling, so KBP improves throughput performance.

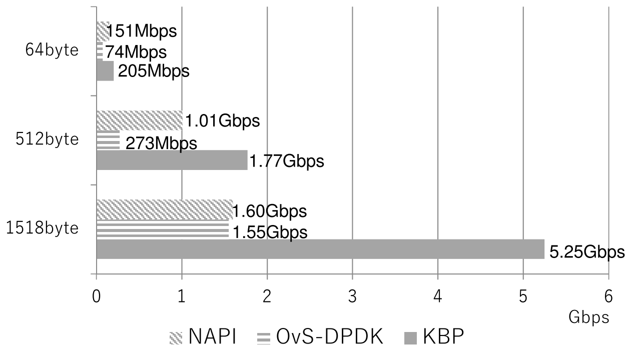

Fig. 9 Throughput performance.

6. Performance Evaluation of Container

Similar to the performance evaluation of VM, we conducted a series of experiments of containers to compare KBP with current solutions. Figure 10 shows the high-level architecture of these solutions. (d) NAPI with *docker* container [31] was a simple container architecture. (e) KBP with *docker* container was an architecture in which KBP was applied to the host of a *docker* container. (d) NAPI with *docker* container and (e) KBP with *docker* container used the standard *docker* network model, the host and the container were connected by a bridge device and network address translation (NAT) rules were applied. (f) NAPI in *Kubernetes* [32] cluster was a simple *Kubernetes* Pod architecture. (g) KBP in *Kubernetes* cluster was an architecture in which KBP was applied to the host of a *Kubernetes* pod. These solutions met the requirements of making both application customization and redeveloping with each kernel security update unnecessary. We measured latency and throughput performance. Table 4 lists the platform specifications for these experiments. To use a high-throughput NIC, we used a different machine from the one used in the analysis of networking delays in Sect. 3.1 due to the space limitations of the PCI card. Note that the CPU performance used for this performance evaluation is higher than the CPU used for the analysis of networking delays in Sect. 3.1. *Flannel* [33] was installed as the container network interface (CNI) in (f) NAPI in Kubernetes cluster and (g) KBP in *Kubernetes* cluster. However, in this performance test, *flannel* was not used in the test route since the server to be evaluated communicated with nodes outside the Pod, so the effect of performance degradation due to *flannel* software processing did not need to be considered. To suppress the delay factors mentioned in Sect. 3.2, the CPU cores that process packets were set using a high performance governor, and these CPU cores were isolated from user processes using *isolcpus*. The server to be evaluated and a traffic generator were directly connected with an optical cable. We added a traffic load to a single flow.

6.1 Latency

6.1.1 Measurement Methodology

We measured round-trip latency in the same way as the test of VM as discussed in Sect. 5.1.1. We added a traffic load and measured the maximum round-trip latency for 60 s and repeated the measurement 5 times.

6.1.2 Results

The left side of Fig. 11 shows the measured maximum round-trip latency at a 100-Mbps traffic rate with 64-, 512-, and 1518-byte UDP frames. (e) KBP with *docker* container achieved maximum round-trip latency within 200 μs with all byte frames. (d) NAPI with *docker* container incurred

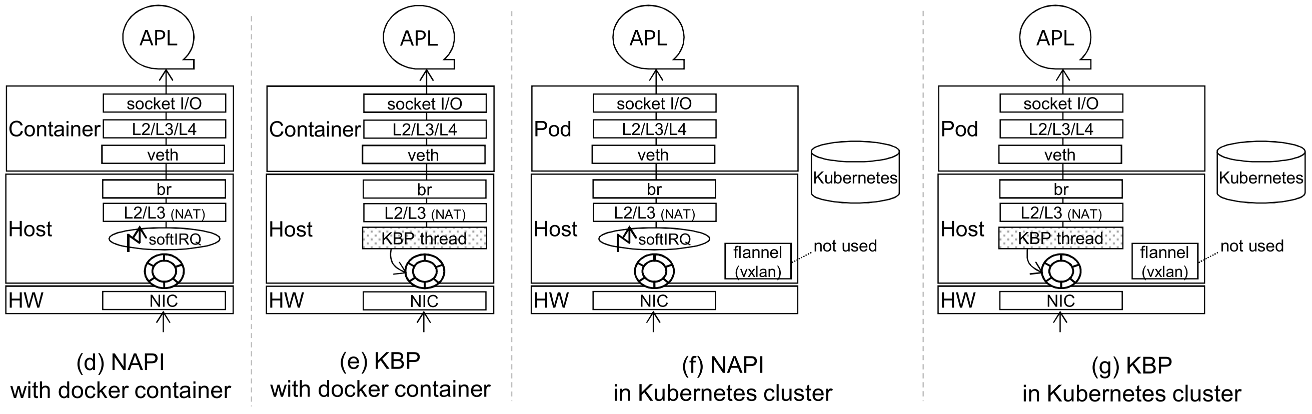
APL
Container
socket I/O
L2/L3/L4
veth
Host
br
L2/L3 (NAT)
softIRQ
KBP thread
HW
NIC
Pod
Kubernetes
flannel (vxlan)
not used
(d) NAPI with docker container
(e) KBP with docker container
(f) NAPI in Kubernetes cluster
(g) KBP in Kubernetes cluster

Fig. 10 High-level architecture of solutions for container to be evaluated.

Table 4 Experimental platform specifications of container.

Host server	
CPU	Intel Xeon CPU E5-2660 v4 2.0 GHz
Memory	64 GB
NIC	Intel X520 DP 10 Gb DA/SFP+
OS	Ubuntu 18.04
kernel	4.15.0-20-generic
Container / Pod	
Docker	19.03.8
kernel	the same as host (4.15.0-20-generic)
Pod	one-container-per-Pod mode
Kubernetes	1.18.4
CNI	flannel (vxlan)

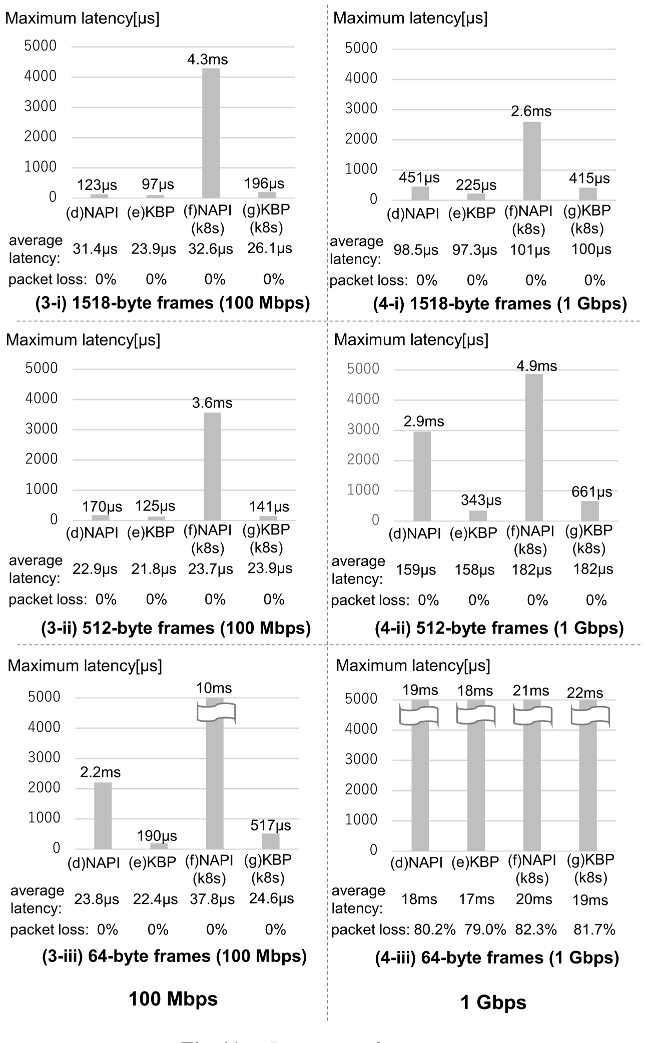

Fig. 11 Latency performance.

millisecond-scale latencies with 64-byte frames and achieved maximum latency within 200 μs with 512- and 1518-byte frames. In the 64-byte frames test, KBP was effective at reducing tail latency, but in the 512- and 1518-byte frames tests, its effectiveness was limited. Since container does not need to emulate VM and the virtualization overhead is small, softIRQ competitions and lack of CPU time for *ksoftirqd* are hard to generate. Thus, the effect of KBP was limited in the 512- and 1518-byte frames tests. (f) NAPI in *Kubernetes* cluster incurred millisecond-scale latencies with all byte frames. (g) KBP in *Kubernetes* cluster achieved maximum round-trip latency within 550 μs with all byte frames. *Kubernetes* schedulers (*kube-scheduler*) and other related threads are running in Pods of *Kubernetes* cluster and these threads deprive *ksoftirqd* of CPU time. Due to lack of CPU time for *ksoftirqd* and softIRQ competitions, (f) NAPI in *Kubernetes* cluster incurred millisecond-scale latency. On the other hand, KBP can prevent softIRQ competition by busy polling, so KBP improved latency performance.

The right side of Fig. 11 shows the measured maximum round-trip latency at a 1-Gbps traffic rate with 64-, 512-, and 1518-byte UDP frames. (e) KBP with *docker* container achieved maximum round-trip latency within 350 μs with 512- and 1518-byte frames. (d) NAPI with *docker* container incurred millisecond-scale latencies with 64- and 512-byte frames. (f) NAPI in *Kubernetes* cluster incurred millisecond-scale latencies with all byte frames. (g) KBP in *Kubernetes* cluster achieved maximum round-trip latency within 700 μs with 512- and 1518-byte frames. Since the packet arrival interval was short at a 1-Gbps traffic rate and the frequency of softIRQ competitions of NAPI was high, NAPI incurred millisecond-scale latency. On the other hand, KBP can prevent softIRQ competition by busy polling, so

KBP improved latency performance. However, (e) KBP with *docker* container and (g) KBP in *Kubernetes* cluster incurred millisecond-scale latency with 64-byte frames. As discussed in Sect. 5.1.2, at 1 Gbps with 64-byte frames, frames arrived at 0.5 μs intervals and the protocol stack in the container could not process the frames at 0.5 μs intervals, causing frame congestion. This was beyond the performance limits of (e) KBP with *docker* container and (g) KBP in *Kubernetes* cluster architecture.

6.2 Throughput

6.2.1 Measurement Methodology

We adopted the RFC2544 benchmark to measure throughput at zero frame loss in the same way as in the test of VM as discussed in Sect. 5.2.1. We repeated the measurement 10 times.

6.2.2 Results

Figure 12 shows the results of throughput measurement. (e) KBP with *docker* container achieved up to 1.25× higher throughput with 1518-byte UDP frames, up to 1.05× higher throughput with 512-byte UDP frames, and up to 1.07× higher throughput with 1518-byte UDP frames than (d) NAPI with *docker* container. (g) KBP in *Kubernetes* cluster achieved almost the same throughput with 1518-byte UDP frames as (f) NAPI in *Kubernetes* cluster but up to 1.28× higher throughput with 512-byte UDP frames and up to 1.10× higher throughput with 1518-byte UDP frames. Compared with the effect in the VM configuration as discussed in Sect. 5.2.2, the effect of KBP in the container configuration was limited. One reason for this is the network-processing overhead such as NAT in the host. This is necessary to connect the container network. Another reason is the overhead of the container network layer. Even if KBP is used to suppress the softIRQ competition, these overheads can become a bottleneck, so throughput performance cannot be improved very much.

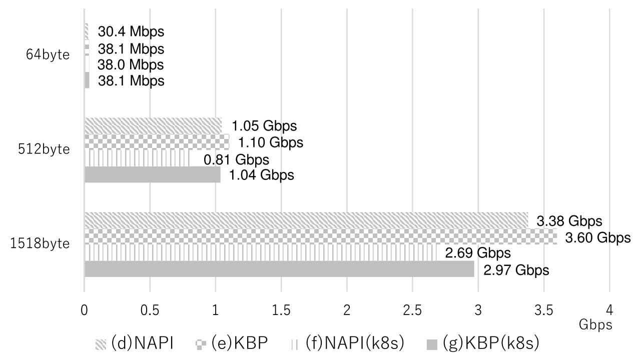

Fig. 12 Throughput performance.

7. Conclusion and Further Study

To design a low-latency networking system, we measured networking delays inside a kernel and analyzed their causes. The analysis revealed that the main cause of the delays was the softIRQ competition, and this cannot be avoided by using any tuning or high-throughput performance NIC. We thus designed and implemented a low-latency networking system, kernel busy poll (KBP), which can avoid the softIRQ competition and achieves microsecond-scale tail delays and high throughput in a VM and a container. KBP does not require any application customizations, and a service provider does not need to develop KBP software for every kernel security update. Evaluation results indicate that KBP achieves microsecond-scale tail delays in both a VM and a container. In the VM configuration, KBP reduces maximum round-trip latency by 98.8% compared with NAPI and 98.5% compared with OvS-DPDK at a 1-Gbps traffic rate with 1518-byte UDP frames, and it achieves up to 3.28× higher throughput than NAPI and 3.39× higher throughput than OvS-DPDK with 1518-byte UDP frames. In the container configuration, KBP reduces maximum round-trip latency by 21.1% to 96.1% but improves the throughput by only up to 1.28× compared with NAPI. KBP achieves a microsecond-scale latency and does not require application customization, so it can be used for various real-time applications, such as online gaming, virtual reality, and autonomous vehicles. Since KBP occupies a CPU core and consumes more power than NAPI, it may be difficult to use in a mobile device with a battery, but it is expected to be used in a server in a data center.

For future work, since KBP has the disadvantage that the busy-poll thread occupies a CPU core and consumes more power than NAPI, we plan to study a method to save power while maintaining low latency.

References

[1] K. Fujimoto, K. Matsui, and M. Akutsu, "KBP: Kernel enhancements for low-latency networking without application customization in virtual server," IEEE CCNC, 2021.
[2] ETSI, "Network Function Virtualization: An Introduction, Benefits, Enablers, Challenges, & Call for Action," 2012.
[3] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, and A. Neal, "Mobile edge computing introductory technical white paper," 2014.
[4] D.B. Oljira, A. Brunstrom, J. Taheri, and K.J. Grinnemo, "Analysis of network latency in virtualized environments," IEEE Global Communications Conference (GLOBECOM), pp.1–6, 2016.
[5] P. Apparao, S. Makineni, and D. Newell, "Characterization of network processing overheads in Xen," 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC), 2006.
[6] G. Aceto, V. Persico, A. Pescapé, and G. Ventre, "SOMETIME: Software defined network-based available bandwidth measurement in MONROE," Proc. 1st Network Traffic Measurement and Analysis Conference, 2017.
[7] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, pp.189–197, 2018.
[8] S. Harcsik, A. Petlund, C. Griwodz, and P. Halvorsen, "Latency evaluation of networking mechanisms for game traffic," Proc. 6th ACM SIGCOMM workshop on Network and system support for

games - NetGames'07, pp.129–134, 2007.
[9] M.S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, "Toward low-latency and ultra-reliable virtual reality," IEEE Network, vol.32, no.2, pp.78–84, 2018.
[10] M.A. Lema, A. Laya, T. Mahmoodi, M. Cuevas, J. Sachs, J. Markendahl, and M. Dohler, "Business case and technology analysis for 5G low latency applications," IEEE Access, vol.5, pp.5917–5935, 2017.
[11] IEEE Std, "Standard for Information Technology–Portable Operating System Interface (POSIX)," 1003.1, 2001.
[12] W.R. Stevens, B. Fenner, and A.M. Rudoff, UNIX Network Programming, Addison-Wesley, 2004.
[13] The Linux Kernel Organization, "Kernel livepatch." https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt
[14] J.H. Salim, "When NAPI comes to town," Linux Conference, 2005.
[15] Intel Corp., "Dpdk: Data plane development kit," http://dpdk.org/, 2014.
[16] R. Kawashima, S. Muramatsu, H. Nakayama, T. Hayashi, and H. Matsuo, "A host-based performance comparison of 40G NFV environments focusing on packet processing architectures and virtual switches," Fifth European Workshop on Software-Defined Networks (EWSDN), pp.19–24, IEEE, 2016.
[17] T. Høiland-Jørgensen, J.D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," Proc. 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), pp.54–66, 2018.
[18] V. Maffione, L. Rizzo, and G. Lettieri, "Flexible virtual machine networking using netmap passthrough," IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), pp.1–6, 2016.
[19] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," Proc. IEEE International Conference on Computer Communications (INFOCOM), pp.2471–2479, 2012.
[20] J. Cummings and E. Tamir, "Open source kernel enhancements for low latency sockets using busy poll," Intel White Paper, 2013.
[21] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp.49–65, 2014.
[22] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," 26th ACM Symposium on Operating Systems Principles (SOSP), pp.325–341, 2017.
[23] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019.
[24] K. Kaffes, T. Chong, J.T. Humphries, D. Mazières, C. Kozyrakis, A. Belay, and D. Mazì Eres, "Shinjuku: Preemptive scheduling for μ second-scale tail latency," 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019.
[25] Huaweii, "Dmm lwip," https://github.com/huawei/dmm, 2018.
[26] D. Zhuo, K. Zhang, Y. Zhu, H. Harry Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network slim: OS kernel support for a low-overhead container overlay network," 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp.331–344, 2019.
[27] W. Tu, J. Stringer, Y. Sun, and Y.H. Wei, "Bringing the power of eBPF to open vSwitch," Linux Plumbers Conference, 2018.
[28] Intel Corp., "Open vswitch with dpdk," https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview
[29] S. Bradner and J. McQuaid, "RFC 2544: Benchmarking methodology for network interconnect devices," IETF, 1999.
[30] Intel Corp., "Pktgen-dpdk," https://github.com/pktgen/pktgen-dpdk
[31] Docker Inc., "Docker engine," https://www.docker.com
[32] Kubernetes, https://kubernetes.io
[33] flannel, https://coreos.com/flannel/docs/latest

Kei Fujimoto received his B.E. degree in electrical and electronic engineering and M.S. degree in informatics from Kyoto University in 2008 and 2010, respectively. Since joining NTT Network Service System Laboratories in 2010, he had engaged in development of a transfer system for ISDN services and research of network-system reliability and network API. From 2016 to 2018, he engaged in creation of new services related to big data in NTT West Corporation. His current research fields are low-latency networking and power-aware computing. He is a member of IEICE.

Masashi Kaneko received an M.E. from the University of Electro-Communications, Tokyo, in 2004. He joined NTT Network Service Systems Laboratories the same year and studied network server platform technologies including web-telecom service convergence, and a sharding method of telecom systems. From 2015 to 2017, he engaged in the development of commercial NFV/software-defined wide area network services at NTT Communications Corporation. He is currently studying photonic disaggregated computers.

Kenichi Matsui received his B.E. and M.S. in information engineering/sciences from Tohoku University in 1995 and 1997, respectively. Since joining NTT laboratories in 1997, he had engaged in research on traffic engineering, network security and authentication, cloud computing. He is currently working in NTT West R&D center where his focus is on communication services. He is a senior member of the IEICE, and member of IPSJ and the IEEE.

Masayuki Akutsu received his M.S. from University of Electro-Communications, Tokyo, in 1994. Since joining NTT laboratories in 1994, he had engaged in research on PSTN software, VoIP system, network API, NFV/SDN server platform and network security. From 2003 to 2005, he engaged in the development of commercial VoIP services in NTT East Corporation. He is currently working on network R&D strategies in NTT Network Service Systems Laboratories.