PAPER

Energy-Efficient KBP: Kernel Enhancements for Low-Latency and Energy-Efficient Networking

Kei FUJIMOTO[†a)], **Ko NATORI**[†], **Masashi KANEKO**[†], ***and*** **Akinori SHIRAGA**[†], ***Members***

SUMMARY Real-time applications are becoming more and more popular, and due to the demand for more compact and portable user devices, offloading terminal processes to edge servers is being considered. Moreover, it is necessary to process packets with low latency on edge servers, which are often virtualized for operability. When trying to achieve low-latency networking, the increase in server power consumption due to performance tuning and busy polling for fast packet receiving becomes a problem. Thus, we design and implement a low-latency and energy-efficient networking system, energy-efficient kernel busy poll (EE-KBP), which meets four requirements: (A) low latency in the order of microseconds for packet forwarding in a virtual server, (B) lower power consumption than existing solutions, (C) no need for application modification, and (D) no need for software redevelopment with each kernel security update. EE-KBP sets a polling thread in a Linux kernel that receives packets with low latency in polling mode while packets are arriving, and when no packets are arriving, it sleeps and lowers the CPU operating frequency. Evaluations indicate that EE-KBP achieves microsecond-order low-latency networking under most traffic conditions, and 1.4× to 3.1× higher throughput with lower power consumption than NAPI used in a Linux kernel.

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

1. Introduction

Real-time applications, such as virtual reality (VR), online gaming, remote controlled drones, and self-driving cars, are becoming more and more popular. These real-time applications require low latency in the order of milliseconds (ms) to microseconds (μs) [1]–[4]. Due to these high real-time requirements, each applications is often installed and executed on the same terminal device. However, it is not desirable to process heavy workloads on the terminal device due to the requirements for smaller size and longer battery life. To meet these requirements, the European Telecommunications Standards Institute (ETSI) has proposed multi-access edge computing (MEC) [5], which enables a terminal device to offload these workloads to an edge cloud located near the terminal device. From the perspective of capital expenditure and ease of operation, those edge clouds are often implemented by general-purpose servers and use virtual servers such as virtual machines (VMs) and containers. However, packet forwarding from a network interface card (NIC) to an application in a virtual server on a general-purpose server brings performance problems of longer latency and lower throughput due to virtualization overhead [6]–[9]. Indeed, according to our experimental results, packet-forwarding delays in a VM can easily exceed 1 ms [10]. Thus, packet-forwarding delays in a virtual server needs to be reduced at least to the order of μs.

Power consumption of edge cloud servers also needs to be considered. To achieve low latency, performance tunings for servers are required, such as fixing the central processing unit (CPU) frequency at maximum and disabling the idle function of a CPU (C-state) [10], [11]. These tunings lower the latency at the cost of higher server power consumption. There is a trade-off between low latency and power saving. Even if the increase in power consumption per server seems small, such as a few watts, the result will be a huge increase in power consumption given the sheer number of servers. Thus, the power consumption of the server should be as low as possible.

When designing a low-latency networking solution for a virtual server, four requirements should be considered. (A): Low latency in the order of μs for packet forwarding in a virtual server. (B): Lower power consumption than existing solutions. (C): No need for application modification. From an application developer's viewpoint, it is desirable for application developers to use a low-latency networking solution without application modification. Application developers can easily use networking functions via the POSIX sockets application program interface (API) [12] since Linux kernel provides a protocol stack that includes internet protocol (IP), user datagram protocol (UDP), and other popular protocols. If a low-latency networking solution requires special network functions, application developers will need to acquire complicated network expertise, and this will increase the cost of application development. (D): No need for software redevelopment for each kernel security update. Since kernel security updates occur frequently, the cost of redeveloping software for each kernel security update would be enormous. From a service provider's viewpoint, it is desirable for service providers to apply kernel security updates without redeveloping software of a low-latency networking solution.

We thus designed and implemented a low-latency and energy-efficient networking system, energy-efficient kernel busy poll (EE-KBP), which meets requirements (A), (B), (C), and (D). This EE-KBP has polling threads that constantly check for packets arriving in a kernel and immediately transfers them to a kernel network protocol stack to meet requirement (A). This is the same as KBP [10]. The

Manuscript received November 25, 2021.
Manuscript revised January 20, 2022.
Manuscript publicized March 14, 2022.

[†]The authors are with the NTT Network Innovation Center, NTT Corporation, Musashino-shi, 180-8585 Japan.
a) E-mail: kei.fujimoto.rg@hco.ntt.co.jp

DOI: 10.1587/transcom.2021EBP3194

base system, KBP, cannot meet requirement (B) since polling threads constantly check the arrival of packets, regardless of whether packets arrive or not, and this consumes more power. Thus, EE-KBP extends KBP to achieve lower power consumption by controlling to sleep polling threads. Sleeping polling threads is challenging since the overhead of recovering from sleep increases tail delays of packet forwarding. EE-KBP minimizes this negative effect by waking up the sleeping polling thread when a hardware interrupt request (hardIRQ) is triggered by a packet arrival. In addition, to increase the power-saving effect of putting polling threads to sleep, EE-KBP dynamically controls the CPU frequency of CPU cores used by polling threads. These extensions enable requirements (A) and (B) to be met simultaneously. Since EE-KBP uses an existing kernel protocol stack and does not change it, applications can use the POSIX sockets API, and this meets requirement (C). Furthermore, since the changes caused by these extensions are small, they can be applied to the existing kernel by a kernel livepatch framework [13], and this meets requirement (D).

The rest of this paper is organized as follows:

- Analysis of how to combine low latency and power saving: we analyze packet forwarding methods that simultaneously achieve low latency and power saving (see Sect. 3);
- Design and implementation of EE-KBP: we design and implement a low-latency and energy-efficient networking system (see Sect. 4);
- Demonstration of the benefits of EE-KBP: our experimental results indicate that EE-KBP can improve energy efficiency, reduce packet-forwarding delay on the μs-scale, and achieve high throughput in a VM configuration (see Sect. 5).

2. Related Work

Linux kernel uses an interrupt-mode packet-receiving method. When a NIC receives a packet, the NIC triggers a hardIRQ to notify a kernel of the packet arrival. After this, a software interrupt request (softIRQ) is scheduled for subsequent protocol processing. Since hardIRQs have high priority, when a hardIRQ occurs, a process running on a CPU core is stopped, and work in progress is evacuated to the memory area, so this overhead is significant. When the packet arrival frequency is high, many hardIRQs are generated, and the overhead caused by the hardIRQs results in performance degradation. To overcome this problem, New API (NAPI) [14], adopted from a newer version of kernel 2.4.20, uses a hybrid method of an interrupt mode and a polling mode. When the packet arrival frequency is high, NAPI suppresses hardIRQs and a kernel thread polls a receiving buffer. However, subsequent protocol processing is performed in a softIRQ context. NAPI cannot avoid softIRQ competition and ms-scale delays [10], thus NAPI cannot meet requirement (A).

The Data Plane Development Kit (DPDK) [15] provides a polling-mode packet-receiving framework. Polling threads in user space constantly check for packets arriving and immediately transfer them to a user program. Since the DPDK bypasses a kernel protocol stack and does not generate any interrupts, it can achieve low latency. However, since polling threads in user space constantly check for packet arrivals regardless of whether packets arrive or not, this method consumes more power and cannot meet requirement (B). In addition, to use the DPDK, a function of polling and network protocol functions, such as layer 2 (L2) / layer 3 (L3) / layer 4 (L4), needs to be integrated to a user program. Thus, the DPDK cannot meet requirement (C).

l3fwd-power [16] is a DPDK sample application for low power consumption. *l3fwd-power* provides multiple application modes. In *APP_MODE_INTERRUPT*, *l3fwd-power* uses a hybrid packet-receiving method of an interrupt mode and a polling mode. When no packets arrive during 10 consecutive polling cycles, the polling interval is made sparse by putting a pause instruction in the polling loop. After that, when no packets arrive during 300 consecutive polling cycles, the polling thread is put into sleep. When a packet arrives after the polling thread has entered sleep state, *eventfd* calls up the polling thread and restarts polling. Since this sleeping logic is intended to follow slow traffic fluctuations such as tidal scale and is not suitable for high frequency sleeping, the power-saving effect is limited when the traffic fluctuation frequency is high. In addition, this *APP_MODE_INTERRUPT* does not have a function to dynamically control CPU frequency of CPU cores used by polling threads. In *APP_MODE_LEGACY*, *l3fwd-power* uses only polling mode. In this mode, polling threads sleep, and it dynamically controls the CPU frequency of CPU cores used by polling threads when no packets arrive. Since this mode does not have a function for fast restarting of polling threads, such as an interrupt mode, it can cause large delays and cannot meet requirement (A). In both *APP_MODE_INTERRUPT* and *APP_MODE_LEGACY*, to use packet-receiving functions of *l3fwd-power*, a function of polling and network protocol functions, such as L2/L3/L4, need to be integrated to a user program and this cannot meet the requirement (C).

Li et al. [17] formulated the relationship between the CPU utilization of the CPU core used by a polling thread and the packet-forwarding delay. When the CPU utilization is below a constant value, such as 80%, it reduces the frequency of polling by adding a short pause. Since this method is intended to follow slow traffic fluctuations such as tidal scale at low-speed link and is not suitable for high frequency sleeping, the power-saving effect is limited when the traffic fluctuation frequency is high. In addition, since this method is intended to be applied to a DPDK solution, a function of polling and network protocol functions, such as L2/L3/L4, need to be integrated to a user program and this cannot meet requirement (C).

Busy Poll Sockets (BPS) [18] provides a hybrid packet-receiving method of an interrupt mode and a polling mode, adopted from a newer version of kernel 3.11. When an

application calls a *SO_BUSY_POLL* socket option and sets the time for busy polling, busy polling is performed to receive packets for the specified period. Except for the specified period, softIRQ-based packet receiving is performed. If an application does not know when the packets will arrive, busy polling cannot be performed in accordance with the packet arrival timing. Thus, this method is not suitable for traffic where the packet arrival timing cannot be predicted. In addition, since an application needs to include a function to specify the period of time for busy polling, this cannot meet requirement (C).

AF_XDP [19] is a new socket type for raw frame processing, adopted from a newer version of kernel 4.18. Since an application can receive raw frames via an AF_XDP socket without a kernel protocol stack, AF_XDP provides a fast data path. However, since AF_XDP is an interrupt-mode packet-receiving data path and notifies an application of frame arrival in a context of *NET_RX_SOFTIRQ*, it cannot avoid softIRQ competition and ms-scale delays. Thus AF_XDP cannot meet requirement (A). In addition, to use AF_XDP, network protocol functions, such as L2/L3/L4, need to be integrated to an application and this cannot meet requirement (C).

KBP [10] enhances a kernel to provide low-latency networking by enabling polling-mode packet receiving. Since KBP launches kernel threads to perform busy polling instead of softIRQ-based packet receiving and avoids softIRQ competitions, KBP can achieve low latency in the order of μs for packet forwarding. In addition, KBP does not modify existing kernel protocol stack, so application modification is unnecessary. Furthermore, since functions of KBP are applied to existing kernels by using a kernel livepatch, software redevelopment for each kernel security update is unnecessary. However, polling threads occupy CPU cores and prevent the threads from sleeping, which incurs greater power usage and cannot meet requirement (B).

IX [20] provides a run-to-completion networking feature in a kernel for low latency. Zygos [21] provides the original scheduler, and Shenango [22] and Shinjuku [23] provide inter-processor interrupts algorithm in a kernel for low latency. These solutions require core parts of a kernel to be modified and do not meet requirement (D).

DMM lwIP [24] and Slim [25] provide low-latency packet-forwarding solutions. To add special functions to a kernel protocol stack without modifying a kernel, these solutions use the LD_PRELOAD framework. The LD_PRELOAD framework enables an application to use custom functions by hijacking *lib.c* libraries, such as *socket.c*. This framework needs to implement custom functions to an application for interworking the special functions and this cannot meet requirement (C).

3. Analysis of Low Latency and Power Saving

We discuss the factors and mechanisms that cause delays in packet forwarding using the current NAPI packet-forwarding method as an example. Figure 1 shows the architecture of

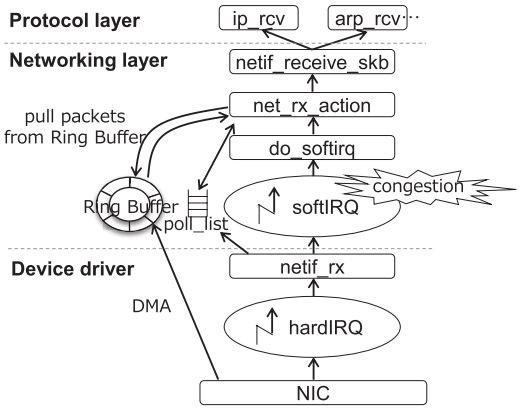


Fig. 1 NAPI architecture.

NAPI. For the receiving (Rx) side, when a NIC receives a packet, the NIC copies the packet data into the host memory space by direct memory access (DMA) without using CPU resources. To report the packet arrival, the NIC invokes a hardIRQ and registers the network device information to a *poll_list* in the hardIRQ context. After that, a softIRQ is scheduled to poll packets from the *poll_list* and performs subsequent protocol processing, such as Ethernet, IP, and UDP. Since hardIRQ has extremely high priority, and other processes cannot use its CPU core during its context, heavy processing should not be allocated in a hardIRQ context for system stability. Thus, these heavy protocol processes are performed in softIRQ context. However, this softIRQ of *NET_RX_SOFTIRQ* can be competed by other softIRQs, such as local timer interrupts and *ata_piix*, and *ksoftirqd* schedules these softIRQs, which must wait until the scheduled time. In addition, when *ksoftirqd* does not have enough CPU time, softIRQ scheduling is delayed. This softIRQ competition causes ms-scale delays as discussed in previous work [10]. This softIRQ competition can occur on a kernel in a bare-metal configuration, on a host kernel in a container configuration, and on a host kernel and a guest kernel in a VM configuration. Since this softIRQ competition cannot be avoided by any kernel tuning, architectural changes are required to obtain a low-latency data path. For the transmission (Tx) side, since the packet is transmitted without any interrupt after an application sends out a packet, a kernel does not incur any ms-scale delay.

Polling-mode packet-receiving methods such as the DPDK, BPS, and KBP are suitable for avoiding softIRQ competition. Since a polling-mode packet-receiving method can detect the arrival of packets quickly by busy polling, there is no need to report the arrival of packets by interrupts. Thus, polling-mode packet-receiving methods do not incur softIRQ competition of *NET_RX_SOFTIRQ*. However, the DPDK needs a function of polling and network protocol functions, such as L2/L3/L4, to be integrated to a user program. BPS needs a function to be integrated that specifies the period of time for busy polling. Therefore, the DPDK and BPS cannot meet requirement (C). To obtain a polling-mode packet-receiving method that does not require appli-

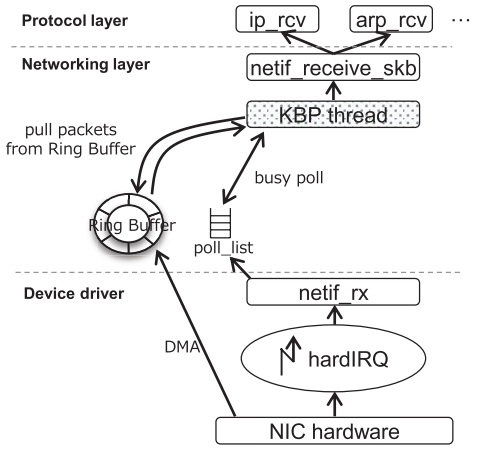


Fig. 2 KBP architecture.

cation modification, polling threads need to be integrated in a kernel. However, since there is a constraint that multiple softIRQ processes cannot be executed in a CPU core, and implementing a network protocol process, which is executed in a softIRQ context in NAPI, with a polling thread requires deadlocking of softIRQs, which is challenging. KBP, our previous work as shown in Fig. 2, has polling threads in a kernel and deadlock-control feature of softIRQs, and it does not change existing kernel packet protocol stack and the POSIX sockets API, thus enabling low-latency packet forwarding and making application modification unnecessary. Furthermore, KBP can be applied to existing NAPI by a kernel livepatch since KBP offers these polling functions with few changes to NAPI. Thus, KBP meets requirements (A), (C), and (D). However, since polling threads always perform busy polling regardless of whether packets have arrived or not, KBP wastes CPU resources and increases power consumption. Thus, KBP does not meet requirements of (B).

Since KBP can meet requirements (A), (C), and (D), if KBP can be improved to meet requirement (B), then all requirements can be satisfied. The basic idea for power saving is to put the polling threads to sleep while no packets arrive. Packet arrival timing is often unpredictable for real-time applications, such as VR, online gaming, remote controlled drones, and self-driving cars. For example, in VR, it is difficult to predict when a user will move his or her gaze, so it is difficult to predict when the gaze information from the head-mounted display will arrive at the server. Thus, it is difficult to control the timing of sleeping polling threads by using a timer. It is desirable that a solution can be also applied to traffic where the arrival timing of packets is unpredictable. Thus, in this paper, we attempt to find a way to wake up sleeping polling threads quickly after a packet arrives, as we discuss in Sect. 4. In addition, simply putting the polling thread to sleep will only have a limited power-saving effect. Even if polling threads stop busy polling and call CPU *pause* instruction, the CPU core continues to read and execute the instruction from a program counter and registers the next instruction to prevent memory order violation from occurring. This means that the power supply to the CPU core will not stop and the CPU core continues to operate and consumes CPU cycles. Since fewer CPU cycles are consumed by pause instruction than by busy polling, the power consumption of the CPU core can be reduced by stopping busy polling. However, the power-saving effect is limited since the CPU core does not stop operating. As methods to increase the power-saving effect, power management technologies, dynamic voltage and frequency scaling (DVFS) and low power idle (LPI), are implemented in CPUs. DVFS dynamically controls power supply voltage and operating frequency of CPU cores in response to changes in load and temperature. LPI controls the sleep state of the CPU core when the core is inactive and is often called C-state. LPI enables some circuits in cores to be powered off while there is no CPU load. DVFS and LPI are implemented and used in almost all modern CPUs. Since most of these functions including LPI are operated by hardware control of CPUs, these power-saving effects can be obtained by just reducing the CPU load by putting the polling threads to sleep. When the polling threads go to sleep and the CPU load drops, the CPU autonomously deepens the sleep state of CPU cores by LPI under hardware control. However, these functions can cause processing delays when recovering from an idle state, increasing networking latency. Since LPI can be enabled/disabled by turning C-state ON/OFF, in the performance evaluation in Sect. 5, we use an Intel Xeon processor to evaluate the power-saving effect and networking latency by controlling polling threads to sleep for each case of C-state ON/OFF. In addition, a CPU operating frequency can be controlled by using CPU-frequency governor [26] from a kernel. If a CPU operating frequency can be controlled in accordance with the arrival timing of packets as well as a sleep control of polling threads, further power-saving effects can be expected. Thus, we attempt to control a CPU operating frequency with a sleep control of polling threads, as we discuss in Sect. 4.

4. Architecture of Energy-Efficient KBP

We designed and implemented a low-latency and energy-efficient networking system, energy-efficient KBP (EE-KBP), which meets all requirements: (A) low latency in the order of μs for packet forwarding in a virtual server, (B) lower power consumption than existing solutions, (C) no need for application modification, and (D) no need for software redevelopment for each kernel security update. Since KBP is a low-latency networking system that can satisfy requirements (A), (C), and (D), we adopted KBP as the base system and attempt to lower power consumption to meet requirement (B). As discussed in Sect. 3, the polling threads of KBP have higher power consumption since busy polling is performed regardless of whether packets arrive or not. Thus, the basic idea for power saving is to put the polling threads to sleep while no packets arrive. Figure 3 shows the high-level architecture of EE-KBP. To reduce power consumption of polling threads, EE-KBP has a sleep-control function for an EE-KBP thread and a CPU-frequency control function for a

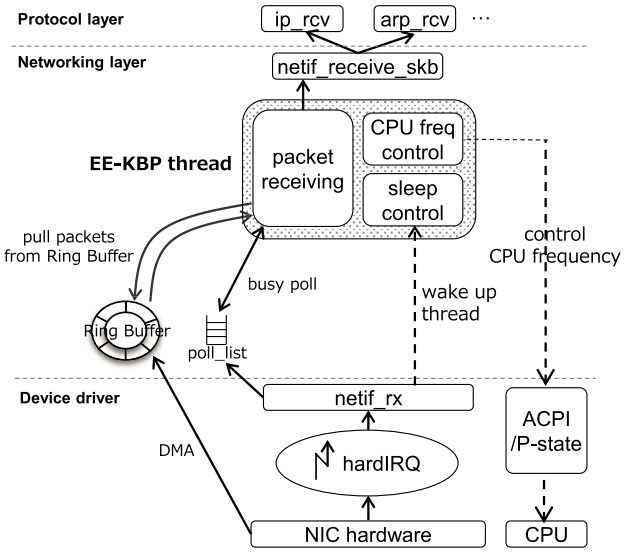


Fig. 3 Energy-efficient KBP architecture.

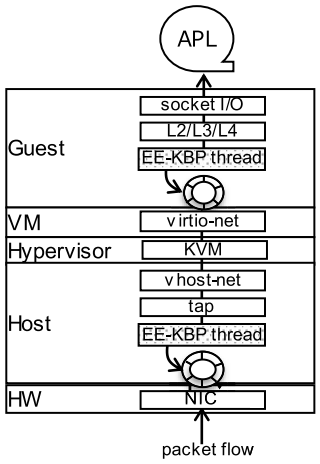


Fig. 4 Energy-efficient KBP architecture for VM.

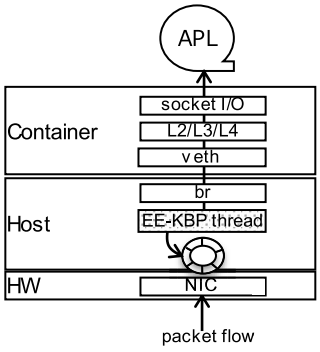


Fig. 5 Energy-efficient KBP architecture for container.

CPU core that is used by a polling thread. EE-KBP has four novel points. The first is EE-KBP wakes up a polling thread and resumes polling fast by using the context of hardIRQ when a packet arrives while the polling thread is asleep in order not to compromise low latency (see Sect. 4.1). The second is EE-KBP dynamically controls a CPU operating frequency from a kernel to increase the power saving effect of a sleeping polling thread (see Sect. 4.2). The third is EE-KBP has a conflict control logic to execute these functions in a kernel since a kernel has a restriction that multiple softIRQ processes cannot be executed simultaneously on the same CPU core (see Sect. 4.1). The fourth is that these functions can be applied to the existing kernel by a kernel livepatch since these changes to the existing kernel have been made as small as possible (see Sect. 4.3).

This EE-KBP can be deployed to a host kernel for a bare-metal server. In addition, EE-KBP can be deployed to a host kernel and a guest kernel for a VM configuration as shown in Fig. 4 and to a host kernel for container configuration as shown in Fig. 5. Since a guest kernel in a VM configuration can cause a CPU time shortage of *ksoftirqd* due to emulation overheads of a VM, which can cause ms-scale latency, EE-KBP to a guest kernel is highly effective in reducing latency. According to our previous work [10], since a host kernel of a container configuration in *Kubernetes* [27] cluster can also cause a CPU time shortage of *ksoftirqd* due to a *Kubernetes* scheduler and other related threads, which can cause ms-scale latency, EE-KBP to a host kernel is effective in reducing latency.

To achieve low latency using EE-KBP, an EE-KBP thread should have a dedicated CPU core, and no other processes should coexist on the CPU core where the EE-KBP thread runs. Since modern CPUs have many CPU cores, this limitation is not a major disadvantage, but EE-KBP is expected to be difficult to apply to a single board computer with a small number of CPU cores, such as Raspberry Pi.

4.1 Hybrid Algorithm of Sleep and Busy Poll

If busy polling by polling threads can sleep while no packets are arriving, the increase in power consumption by polling threads can be lowered. However, overhead of sleeping polling threads can increase delay time of packets receiving. We attempt to control the polling threads to sleep without compromising low latency as much as possible. First, we need to design when to put a polling thread to sleep and when to wake it up. Figure 6 shows a simplified traffic model that describes the timing at which packets arrive. Ideally, a polling thread should be woken up when a packet arrives and sleep when a packet stops arriving. However, as discussed in Sec. 3, since packet arrival timing is often unpredictable for real-time applications, it is difficult to predict the packet arrival timing and wake up a polling thread in advance with a timer. In addition, when packets are arriving in succession, it is difficult to predict when the packet arrivals will stop, so it is also difficult to predict when to sleep the polling thread. For the method of waking up a polling thread, EE-KBP wakes up the polling thread as fast as possible after the packet arrives at the NIC. HardIRQ has extremely high priority, and when the hardIRQ is triggered, a process running on a CPU core must suspend and give up CPU time to the

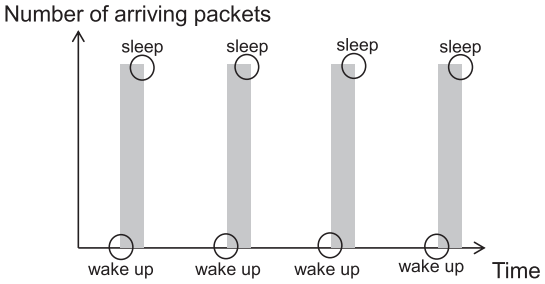


Fig. 6 Wake up and sleep timing for polling threads.

hardIRQ. When a NIC receives a packet, the NIC notifies a kernel that a packet has arrived by generating a hardIRQ. Thus, using this hardIRQ context enables the polling thread to be woken up fast. As for when to put the polling thread to sleep, *poll_list* can be used to judge whether to sleep or not. When a NIC receives a packet, the NIC registers its network device to a *poll_list* and a kernel can recognize whether there is a packet to receive or not by checking the *poll_list*. As long as the network device is registered in the *poll_list*, it means that unprocessed packets are being buffered, so the polling thread should continue polling. When the network device does not exist in the *poll_list*, it means that there is no packet to be processed by the polling thread, so the polling thread can stop polling and sleep. Thus, when the network device is not registered in a *poll_list*, the EE-KBP thread stops polling and sleeps.

Algorithm 1 shows a hybrid algorithm of sleep and busy poll for EE-KBP. To disable a softIRQ for networking of NAPI to avoid softIRQ competition, which causes delays, EE-KBP removes a function of raising a softIRQ of *NET_RX_SOFTIRQ* in *netif_rx*. Instead, EE-KBP added a function to wake up an EE-KBP thread in a context of hardIRQ for networking in *netif_rx*. An EE-KBP thread is a kernel thread that performs busy polling to receive packets and sleeps when there is no registered network device in *poll_list*. By combining modified *netif_rx* and the EE-KBP thread, the EE-KBP thread performs busy polling to receive packets with low latency while there are unreceived packets, the EE-KBP thread sleeps when *poll_list* becomes empty to reduce power consumption, and when a packet arrives while the EE-KBP thread is asleep, a handler of a hardIRQ of packet arrival wakes up the EE-KBP thread to resume busy polling quickly. Q (quota) is the value of the number of packets to be received in one polling, and the default value of NAPI of existing kernel is 64. This value is used in the performance evaluation as will be discussed in Sect. 5.

There is a restriction that multiple softIRQ processes cannot be executed simultaneously on the same CPU core. In NAPI of existing kernel, network protocol processes are processed in a softIRQ context. Since EE-KBP uses an existing network protocol stack in a kernel to meet requirements (C) and (D), EE-KBP does not make any changes to the functions after *netif_receive_skb*. Thus, we need to consider this restriction when the EE-KBP thread pulls a packet and proceeds to network protocol processing. To control this restriction of prohibiting execution of multiple softIRQs, EE-KBP has a conflict control logic. After the EE-KBP thread pulls a packet, the EE-KBP thread prohibit other softIRQs and proceeds to subsequent network protocol processing. After protocol processing finishes, EE-KBP releases this prohibition.

Algorithm 1 Hybrid algorithm of sleep and busy poll

***netif_rx* for EE-KBP**

```
1:          // This function is invoked by hardIRQ from a network device
2:  Disable hardIRQ of this network device
3:  Add the device to poll_list
4:  /* Raise softIRQ of NET_RX_SOFTIRQ */                    // removed
5:  if EE-KBP thread is not running then                     // new
6:      Wake up EE-KBP thread                                // new
7:  end if                                                   // new
```

EE-KBP thread (new)

```
1:  quota Q = 64
2:  Maximize the CPU operating frequency for this EE-KBP thread
3:  while poll_list is not empty do
4:      Extract a network device from poll_list
5:      Using this device, poll up to Q packets or until no more packets
        in ring buffer of the device and process the packets calling
        netif_receive_skb
6:      if Ring buffer is empty then
7:          Remove the network device from poll_list
8:          Enable hardIRQ of the network device
9:      else
10:         Place the network device at the end of poll_list
11:     end if
12: end while
13: Minimize the CPU operating frequency for this EE-KBP thread
14: Sleep this thread
```

4.2 CPU-Frequency Control Feature

To further reduce power consumption of a CPU core while an EE-KBP thread is sleeping, EE-KBP dynamically controls the CPU operating frequency of the CPU core. As discussed in Sect. 3, even if a polling thread stops busy polling and sleeps, the CPU core continues to read and execute an instruction from a program counter, and registers the next instruction to prevent memory order violation. This means the CPU core continues to consume power while the polling thread is sleeping. Since the operating frequency of a CPU core can be changed dynamically from a kernel, it is expected to reduce this power wastage of the CPU core by lowering the CPU operating frequency while the polling thread is sleeping. EE-KBP has a function of CPU-frequency control and sets the CPU operating frequency of the CPU core low when the EE-KBP thread sleeps, and restores the CPU operating frequency when it wakes up from sleep, as shown in algorithm 1. EE-KBP uses the CPU-frequency governor to change a CPU operating frequency. There are multiple types of governor policy, such as "performance," "powersave," "userspace," and "ondemand." We use the CPU-frequency governor "userspace" since it enables a specific frequency to

be set for a CPU core. By using CPU-frequency governor "userspace," the EE-KBP thread sets the maximum CPU frequency at the timing triggered by a hardIRQ of networking, and sets the minimum CPU frequency just before the EE-KBP thread goes to sleep. The time lag between changing a CPU operating frequency and the change being reflected to a CPU core needs to be short, on the order of μs, but we have determined that it is short enough based on performance evaluation results as will be discussed in Sect. 5.

As for power saving functions by hardware control that many CPUs have, such as DVFS and LPI (C-state), their operation cannot be controlled from a kernel, but these functions can be activated by just putting a polling thread to sleep, as discussed in Sect. 3. Thus, EE-KBP does not have a feature to control these CPU-hardware functions, but EE-KBP changes the utilization rate of a CPU core by controlling the sleep state of a polling thread, which indirectly increases the effectiveness of these power-saving functions. The effect of these functions and long delay time due to overhead will be discussed in the performance evaluation in Sect. 5.

4.3 Applying by Kernel Livepatch

The features of EE-KBP can be enabled by making a few changes from NAPI of an existing kernel. As shown in algorithm 1, EE-KBP only removes the function of raising a softIRQ of *NET_RX_SOFTIRQ* and adds the function of waking up the EE-KBP thread in *netif_rx* of NAPI. The EE-KBP thread is woken up by *netif_rx*, and after pulling packets, it passes the process to *netif_receive_skb*. Thus, unless either *netif_rx* or *netif_receive_skb* is changed, EE-KBP can be run on any version of kernel after version 2.4.20, when NAPI was implemented. A kernel has a mechanism called kernel livepatch, which is used to apply security patches to a kernel or change some of behaviors of a kernel. Since EE-KBP needs a few changes from NAPI, EE-KBP can be applied by a kernel livepatch. If a security update occurs to a kernel, EE-KBP can be enabled by simply applying the kernel livepatch to a new version of kernel again. Thus, unless either *netif_rx* or *netif_receive_skb* is changed, service providers do not need to modify software of the kernel livepatch. In addition, since kernel livepatch can be applied without rebooting a system, service providers can apply EE-KBP without service interruption.

4.4 Scaling Out

If the amount of incoming traffic is too large for a single EE-KBP thread to handle, multiple EE-KBP threads can be deployed in a kernel. Almost all modern NICs have a feature of receive-side scaling (RSS) and use multiple CPU cores for packet receiving. When a NIC receives a packet, it chooses to which CPU core to invoke a hardIRQ. Scaling out can be obtained by launching EE-KBP threads on multiple CPU cores and linking those CPU cores to the CPU cores to which hardIRQs are generated by RSS.

5. Performance Evaluation

To evaluate the effect of power saving and low latency of EE-KBP, we conducted performance evaluations comparing EE-KBP with existing NAPI and KBP adopted as a base machine of EE-KBP. We measured power consumption, packet-forwarding latency, and throughput when traffic load was added to the target solutions. As discussed in Sect. 4, EE-KBP can be deployed to a host kernel for a bare-metal server, to a host kernel and a guest kernel for a VM configuration, and to a host kernel for container configuration. In a VM configuration, a host and a guest each have a polling thread, and since the number of polling threads to be slept by EE-KBP is larger than in a bare-metal server or a container configuration, the power saving effect can be largely measured. In addition, in a VM configuration, since packet forwarding delays are likely to be long due to an emulation overhead to create a VM, the low-latency effect with EE-KBP can be largely measured. For these reasons, we used a VM configuration for performance evaluations. To evaluate the effect of the sleep control feature of EE-KBP with hardware-controlled power-saving features of CPU, we evaluated each case when C-state was enabled and disabled. In addition, to evaluate the effect of the CPU-frequency control feature of EE-KBP, we evaluated whether each case when the CPU-frequency control feature was enabled and disabled. Thus, we conducted comparative evaluations of solutions and settings for (a) NAPI, (b) KBP, (c) EE-KBP when C-state was disabled and the CPU-frequency control feature was disabled, (d) EE-KBP with C-state was disabled and the CPU-frequency control feature was enabled, (e) E-KBP when C-state was enabled and the CPU-frequency control feature was disabled, and (f) EE-KBP when C-state was enabled and the CPU-frequency control feature was enabled.

Table 1 shows experimental platform specifications. For low latency, CPU cores used for packet processing were set to a high performance governor, and virtual-CPU emulation threads of kernel-based virtual machine (KVM) and *vhost-net* were allocated dedicated CPU cores and isolated from other processes using *isolcpus*. For power saving, C-state was enabled for (a) NAPI and (b) KBP. For the CPU-frequency control feature of EE-KBP in (d) and (f), a "userspace" governor was used, and the maximum frequency that could be set for the Intel Xeon processor used was 2.8 GHz, and the minimum frequency was 1.2 GHz.

We added a round-trip traffic load as shown in Fig. 7. Since transmission control protocol (TCP) has retransmission control and congestion control, and the effect of the sleep-control feature of EE-KBP is difficult to evaluate, we used UDP traffic. To evaluate the power-saving effect and the increase of delay time due to the overhead of waking up from sleep of EE-KBP by varying the time when polling threads were asleep, six conditions of traffic with different traffic rates and frame sizes were used as shown in Table 2. In these traffic conditions, one packet arrives at the packet-arrival interval described in Table 2. When a packet arrives,

Table 1 Experimental platform specifications.

Host server	
Machine	Dell PowerEdge R730
CPU	Intel Xeon CPU E5-2660 v4 2.0 GHz, 14 cores
Memory	64 GB
NIC	Intel X520 DP 10Gb DA/SFP+
kernel / OS	4.15.0-20-generic / Ubuntu 18.04
Hypervisor	KVM
Guest server	
vCPU	4 cores
Memory	4 GB
kernel / OS	4.15.0-20-generic / Ubuntu 18.04

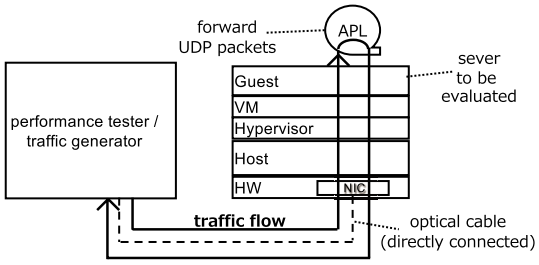


Fig. 7 Evaluation configuration.

Table 2 Traffic conditions.

Traffic rate	Frame size	Packet arrival interval
80 Mbps	1518 byte	151.8 μs
	512 byte	51.2 μs
	64 byte	6.4 μs
1 Gbps	1518 byte	12.1 μs
	512 byte	4.1 μs
	64 byte	0.5 μs

a polling thread is woken up by a hardIRQ of the packet arrival and immediately sleeps after that, and this process is repeated as shown in Fig. 6. It is not possible to pull packets continuously by busy poll. These traffic conditions are disadvantageous for EE-KBP. However, these are suitable traffic conditions to evaluate overhead caused by polling thread sleep of EE-KBP. 80 Mbps is the traffic rate at which no packet loss occurs in any of the solutions being compared.

5.1 Power Consumption

There are three methods to measure power consumption of a server. The first is to physically measure power consumption of an entire server by inserting a power meter into a power supply line of the server. This method does not depend on the accuracy of a sensor of a server and can measure power consumption with high accuracy. However, it requires onsite work to install a power meter and check the measurement values of a power meter. Since it was difficult to perform onsite work for a long time this time, we used this method only to check the measurement accuracy of a method described below. The second is to use Intel's running average power limit (RAPL) interface. Intel's CPU provides an RAPL interface that enables power consumption information to be obtained. It is reported that RAPL provides high accuracy power consumption data of CPU [28] and dynamic random-access memory (DRAM). However, power consumption of an entire server cannot be measured by using an RAPL interface. Since performance and power consumption of CPU cores are closely related to the cooling fan speed of a server chassis, it is necessary to evaluate power consumption of an entire server that includes power consumption of cooling fans. The third is to use an intelligent platform management interface (IPMI). IPMI enables a server to be controlled and monitored remotely. By using this IPMI, we can measure power consumption of an entire server. However, it was reported that the accuracy of power consumption measurement using an IPMI is low in some cases, depending on the accuracy of a server onboard sensors [29]. Thus, we compared results of power measurement between using an IPMI and a power meter to verify the accuracy of an onboard sensor on a server (Dell PowerEdge R730). The result of the measurement by the IPMI was 0.29% smaller than the result of the measurement by the power meter and the difference between the two was stable, so we concluded that the accuracy of the IPMI power measurements for this server was high enough. We measured power consumption of each solution and setting when traffic with the conditions described in Table 2 was added. To add these conditions of traffic, we used a Spirent Test Center SPT-N4U (STC) as a performance tester. We added a traffic load and measured power consumption for 60 seconds and repeated the measurement 5 times.

Figure 8 shows power-consumption measurement results of each solution and setting over time. We started applying traffic 10 seconds after we started the test, and then kept applying traffic for 60 seconds. First, we analyze the increase in power consumption due to busy polling of a polling thread by comparing the results of (a) NAPI and (b) KBP. Comparing the results of power consumption with no traffic applied, (a) NAPI was 170 watt (W) and (2) KBP was 181 W, a difference of 11 W. In a VM configuration with KBP, there was a polling thread on each of a host and a guest, which means that the incremental power consumption of busy polling by the two polling threads was 11 W. Thus, the power consumption due to useless busy polling when no packets were coming wasted 5.5 W per polling thread. As for the analysis of the effect of EE-KBP to put a polling thread to sleep, the power consumption of (e) and (f) with C-state enabled was 170 W while no traffic was added, which was the same as (a) NAPI. These results mean that EE-KBP can reduce power consumption to the same level as NAPI while no packets are arriving by putting polling threads to sleep. On the other hand, the results of (c) and (d), which are the results when the C-state is disabled, show power consumption increasing over about 25 W even while no traffic is added. Since C-state cannot be changed for each CPU core, for the entire CPU, the hardware-controlled power-saving functions of all 14 CPU cores were disabled, resulting in such a large increase in power consumption. If service providers need saving power, it is desirable to set C-state to enabled.

Next, we analyzed the power-saving effect of EE-KBP while traffic was added. Figures 9 and 10 classify the re-

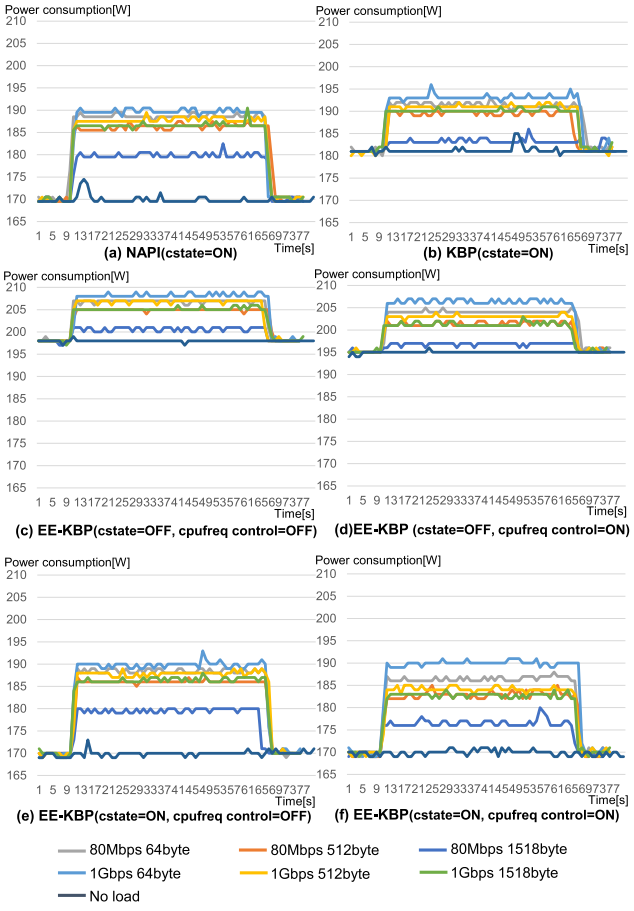


Fig. 8 Results of time-lapse power-consumption measurement.

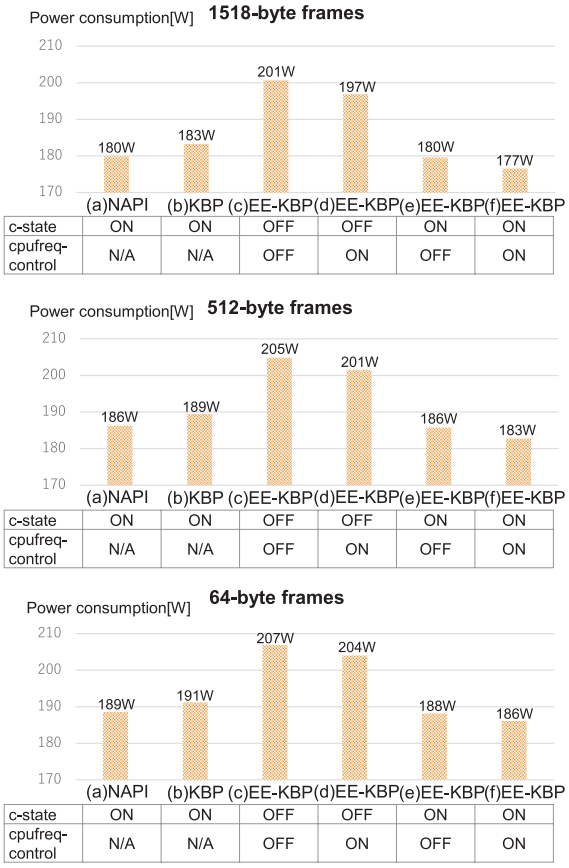


Fig. 9 Results of power consumption measurement with 80 Mbps traffic.

sults of power consumption for each solution and setting by frame size and traffic rate. Figures 9 and 10 show the results with 80 Mbps and 1 Gbps traffic loads, respectively. Comparing the results of (b) KBP and (e) EE-KBP without CPU-frequency control, EE-KBP reduced the power consumption by 3 W in all traffic conditions compared with KBP by putting polling threads to sleep. These results of (e) EE-KBP were almost the same as the results of (a) NAPI. This means that EE-KBP solves the problem of increased power consumption due to busy polling of polling threads in KBP. Furthermore, comparing the results of (b) KBP and (f) EE-KBP with CPU-frequency control, EE-KBP with CPU-frequency control consumed about 5 to 7 W less power than KBP, except for the 1-Gbps 64-byte traffic condition. The CPU-frequency control function of EE-KBP reduces power consumption by an additional 2 to 4 W compared with the results in (e) EE-KBP without CPU-frequency controlling. By controlling a CPU frequency as well as putting polling threads to sleep, EE-KBP could further reduce power consumption more than NAPI. In the 1-Gbps 64-byte traffic condition, EE-KBP with CPU-frequency control could not further reduce power consumption more than EE-KBP without CPU-frequency control. In the 1-Gbps 64-byte traffic condition, packet-arrival interval is 0.5 μs, which is quite short. It is thought that changing CPU operating frequency by EE-KBP was not reflected in time during this short interval. In 1-Gbps 512-byte traffic condition, since EE-KBP with CPU-frequency control could further reduced power consumption more than EE-KBP without CPU-frequency control, it was shown that a CPU operating frequency can be changed on the order of μs by the function of EE-KBP. These results are in a VM configuration in which there are two polling threads, which are in a host and a guest. When providing real-time services, a large number of users is expected to be accommodated in the server, and the number of polling threads is also expected to increase, so the power-saving effect of EE-KBP increases with each additional polling thread compared with existing KBP. In addition, in this test traffic, packets arrived continuously and the sleep time was short, so the power-saving effect of EE-KBP was limited. If the traffic has a long sleep time, such as intermittent packets incoming, the power-saving effect of EE-KBP will be even higher.

5.2 Latency

To evaluate the increase of packet-forwarding latency in EE-KBP due to overhead of sleep control and CPU-frequency control, we measured maximum round-trip latency of each solution and setting when traffic with the conditions described in Table 2 was added. To add these conditions of traffic and measure latency, we used the STC as a perfor-

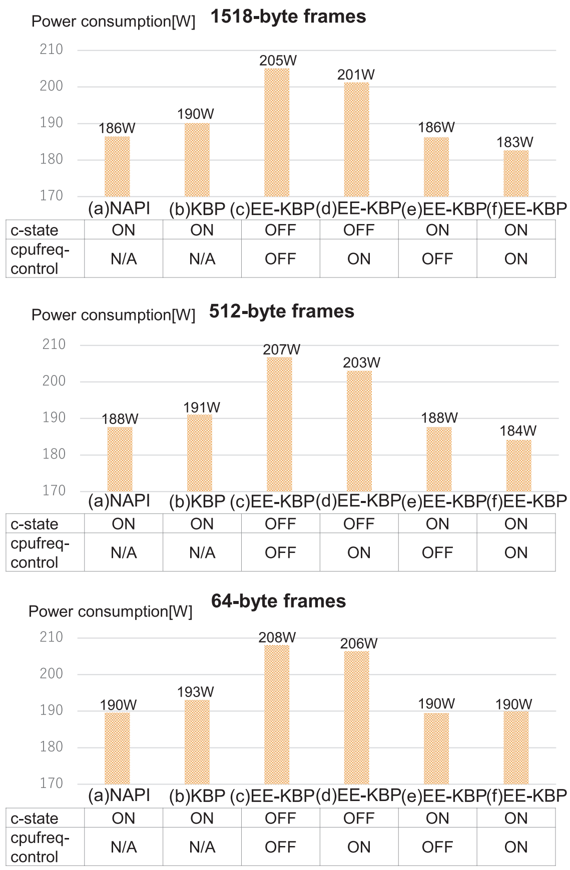


Fig. 10 Results of power consumption measurement with 1 Gbps traffic.

mance tester. We added a traffic load and measured latency for 60 seconds and repeated the measurement 5 times.

Figures 11 and 12 show the latency-measurement with 80 Mbps and 1 Gbps traffic loads, respectively. Comparing the results of (b) KBP and (e) EE-KBP without CPU-frequency control, although the maximum round-trip latency was increased by up to 87 μs compared with KBP, EE-KBP achieved low latency in the order of μs. For 1-Gbps traffic, the increase in delay time of EE-KBP compared with KBP was small, and for 80-Mbps traffic, the increase in delay time tended to be larger. Since 80-Mbps traffic has a longer packet interval and polling threads can sleep for a longer period of time, it is assumed that polling threads need longer time to recover due to deeply sleeping by a hardware power-saving feature of a CPU. Although latency slightly increased due to the overhead of putting polling threads to sleep, EE-KBP without CPU-frequency control can achieve power savings without compromising low latency. With the 1-Gbps 64-byte traffic condition, all solutions and settings incurred ms-scale delays and packet losses. This is due to the limit of a processing performance of a kernel protocol stack in a guest, which is a performance limit of the KBP architecture without any changes to the kernel protocol stack, as discussed in our previous work [10].

Next, we analyze the overhead caused by the CPU-frequency control function of EE-KBP. Comparing the results of (e) EE-KBP without CPU-frequency control and (f) EE-KBP with CPU-frequency control, (f) EE-KBP with CPU-frequency control achieved low latency in the order of μs except for 64-byte frames at 80-Mbps and at 1-Gbps, although the maximum round-trip latency was increased by up to 260 μs compared with (e) EE-KBP without CPU-frequency control. (f) EE-KBP with CPU-frequency control incurred ms-scale delays with 64-byte frames at 80-Mbps traffic rate. Even though the packet arrival interval was shorter for 512-byte frames at 1-Gbps than for 64-byte frames at 80-Mbps, ms-scale delays occurred for 64-byte frames at 80-Mbps. These delays correlate with the number of hardIRQ times that were triggered by a NIC. Figure 13 shows the number of hardIRQs when traffic with each condition described in Table 2 was added for 60 seconds. The number of hardIRQs in 64-byte frames at 80-Mbps traffic rate was the highest. If the number of hardIRQs of packet arrival is large, a packet-receiving process must be interrupted

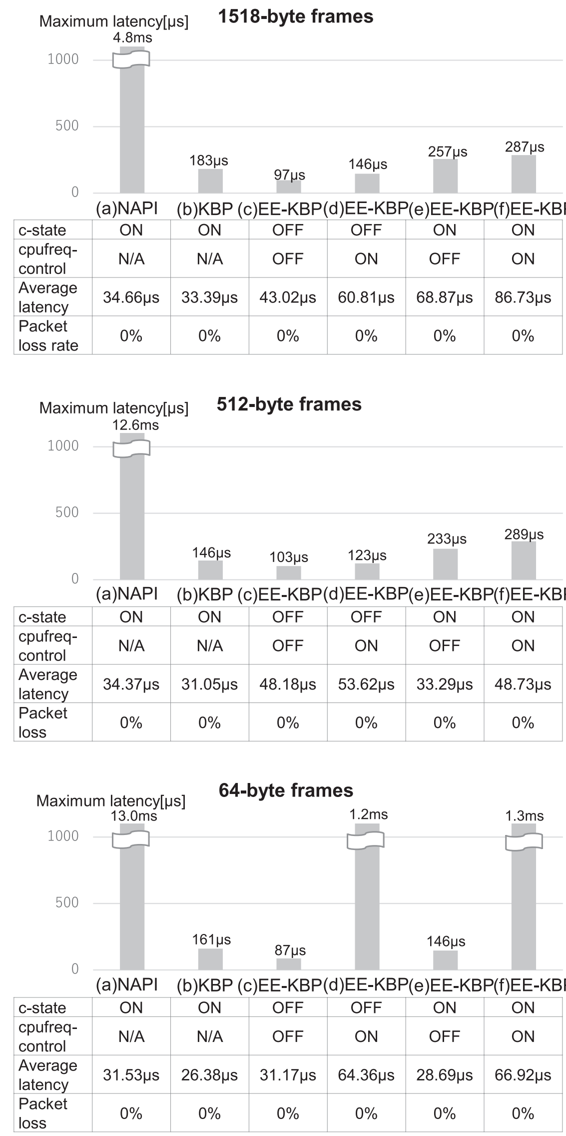
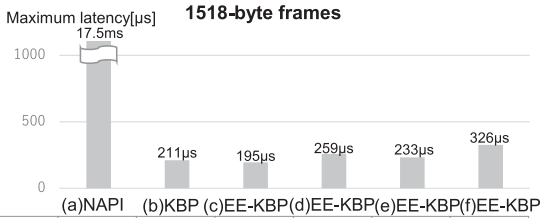
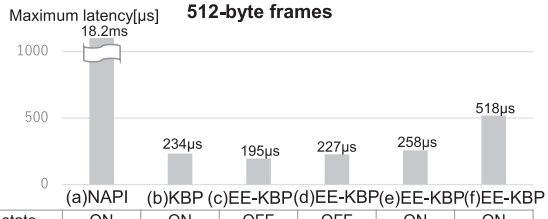


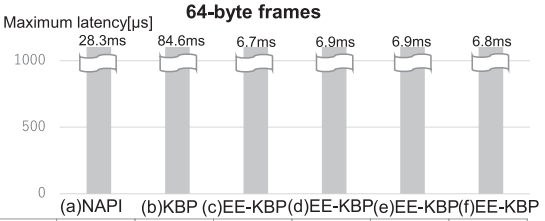
Fig. 11 Results of latency measurement with 80 Mbps traffic.



c-state	ON	ON	OFF	OFF	ON	ON
cpufreq-control	N/A	N/A	OFF	ON	OFF	ON
Average latency	7.6ms	96.9μs	99.5μs	104.9μs	97.4μs	105.0μs
Packet loss rate	0.43%	0%	0%	0%	0%	0%



c-state	ON	ON	OFF	OFF	ON	ON
cpufreq-control	N/A	N/A	OFF	ON	OFF	ON
Average latency	1.4ms	103.6μs	107.4μs	102.3μs	106.3μs	99.4μs
Packet loss	74.46%	0%	0%	0%	0%	0%



c-state	ON	ON	OFF	OFF	ON	ON
cpufreq-control	N/A	N/A	OFF	ON	OFF	ON
Average latency	294.7μs	6.1ms	6.3ms	6.4ms	4.7ms	4.7ms
Packet loss	96.75%	63.83%	67.48%	67.69%	66.34%	64.21%

Fig. 12 Results of latency measurement with 1 Gbps traffic.

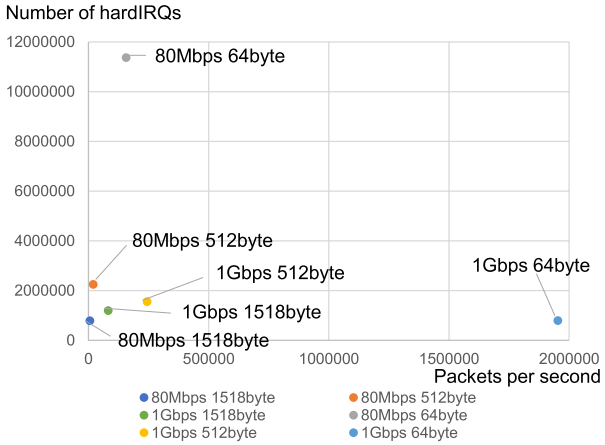


Fig. 13 Trend in number of hardIRQs.

at each hardIRQ, the data being processed must be stored in memory space, and CPU time must be given over to the hardIRQ, resulting in a large overhead. This caused a shortage of CPU time for a packet-receiving function that ran on the CPU core where the EE-KBP thread operated, and since the CPU-frequency control function was also attempted on the same CPU core, the CPU time was further shortened, resulting in ms-scale packet-forwarding delays. As discussed in Sect. 4.1, when there is no more packet data to be received in a *poll_list*, an EE-KBP thread permits hardIRQ and goes into sleep. If a packet arrival frequency is relatively sparse, a hardIRQ will be triggered every time a packet arrives, so the number of hardIRQs will become high. If a packet arrival frequency is higher than the speed of packet receiving and processing by a kernel protocol stack, subsequent packets will be stored in a ring buffer and the NET_DEV will remain in the *poll_list*, so the time that hardIRQ is prohibited and the thread continues polling will be longer. When multiple packets are stored in the ring buffer, the EE-KBP thread receives Q packets at once to batch them, so the ring buffer will be empty again. EE-KBP with C-state was disabled achieved smaller maximum round-trip latency than EE-KBP with C-state was enabled by comparing the results of (c), (d), (e), and (f). However, since disabling C-state causes a significant increase in power consumption as discussed in Sect. 5.1, a use case of disabling C-state is considered to be rare.

EE-KBP without CPU-frequency control achieved low latency in the order of μs with small degradation of latency compared with KBP and the same level of power saving effect as NAPI used in existing kernel by overcoming the problem of KBP, which is increased power consumption by continuous busy polling. By enabling the CPU-frequency control function of EE-KBP, EE-KBP further reduced power consumption to a lower level than that of NAPI and achieved low latency in the order of μs under many traffic conditions. However, in the case of a traffic condition in which EE-KBP invoked a lot of hardIRQs for networking, measured maximum round-trip latency of EE-KBP with CPU-frequency control was 1.3 ms, we note that this value was smaller than NAPI. Although the number of hardIRQs in EE-KBP is difficult to estimate since the packet processing time by a kernel protocol stack varies depending on CPU performance, packet length, and protocol used, which affect the speed at which packets are pulled from a ring buffer, EE-KBP with CPU-frequency control may be used for a service that has traffic consisting of packets with long or short packet-arrival intervals.

5.3 Throughput

We conducted a throughput measurement evaluation to measure the maximum throughput performance without packet loss as specified in the RFC2544 [30]. We used the Pktgen-DPDK [31] as a traffic generator and traffic of 64-, 512-, and 1518-byte frames was added. We repeatedly added traffic from the Pktgen-DPDK by gradually increasing the traffic rate, measured the maximum throughput without packet loss. We repeated the measurement five times.

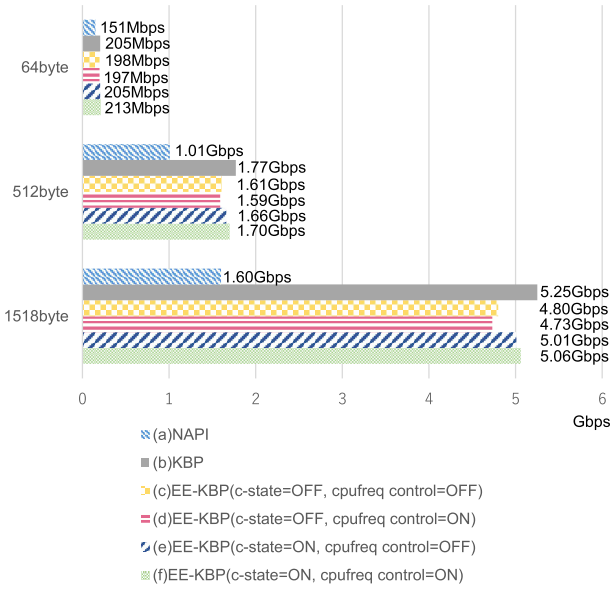


Fig. 14 Results of throughput measurement.

Figure 14 shows the results of throughput measurements. Comparing the results of (b) KBP, (e) EE-KBP without CPU-frequency control, and (f) EE-KBP with CPU-frequency control, EE-KBP had only a maximum throughput reduction of 4.6% compared with KBP. In throughput testing, there is not much opportunity for the EE-KBP polling threads to sleep since a large number of test packets are added, so the behavior is almost the same as KBP where the polling threads keep busy polling. The throughput performance degradation to KBP was small since there was little opportunity for overhead due to sleep control of polling threads by EE-KBP. Compared with NAPI, EE-KBP achieved a throughput improvement of 1.4× to 3.1× higher. The results of (c) and (d) with C-state disabled showed lower throughput than the results of (e) and (f) with C-state enabled. This is due to a specification of the Intel Xeon processor used, which has a maximum CPU frequency of 2.4 GHz when C-state is disabled, and 2.8 GHz when C-state is enabled.

5.4 Discussion

On the basis of the results in Sects. 5.1 to 5.3, we discuss the characteristics and desirable tuning of EE-KBP with respect to performance. LPI (C-state) and CPU-frequency control are independent features, and the evaluation results did not reveal any special synergy between these functions. Thus, a service provider who uses EE-KBP should enable/disable these functions in accordance with the guidelines described below. As for EE-KBP with C-state, the power saving effect with C-state enabled was large, the maximum delay time with C-state enabled worsened only by a few tens to 200 μs, and the throughput was better with C-state enabled than with C-state disabled. EE-KBP with C-state enabled was superior to NAPI in terms of power saving, low latency, and throughput under all traffic conditions. Based on these results, when a server needs to save power, C-state should be enabled regardless of traffic conditions. As for EE-KBP with CPU-frequency control, the power saving effect and the throughput were higher with CPU-frequency control than without it. However, in the case of traffic with packet arrival intervals where many hardIRQs were generated in EE-KBP, ms-order delays occurred on very rare occasions with CPU-frequency control. For use cases where a very rare ms-order delay is acceptable, CPU-frequency control should be enabled, since higher power savings can be achieved. For use cases where a very rare ms-order delay is unacceptable, CPU-frequency control should be disabled.

Since ms-order delays only rarely occur with CPU-frequency control, a service provider may check whether the traffic of its service meets this specific condition. However, it is highly challenging for a service provider to add traffic and count the number of hardIRQs to see if they meet specific conditions. In addition, the number of hardIRQs in EE-KBP is difficult to estimate since the packet processing time by a kernel protocol stack varies depending on CPU performance, packet length, and protocol used, which affect the speed at which packets are pulled from a ring buffer. Thus, we believe that a feature to suppress the number of hardIRQs in accordance with the traffic conditions is necessary to prevent ms-order delays. This is our future work, as will be discussed in Sect. 6.

6. Conclusion and Further Study

We designed and implemented a low-latency and energy-efficient networking system, energy-efficient kernel busy poll (EE-KBP), which meets four requirements: (A) low latency in the order of μs for packet forwarding in a virtual server, (B) lower power consumption than existing solutions, (C) no need for application modification, and (D) no need for software redevelopment for each kernel security update. EE-KBP achieves low-latency networking by performing busy polling in a kernel, and saves power by putting polling threads to sleep while no packets are coming in, and by dynamically controlling the operating frequency of CPU cores for the polling threads. EE-KBP reduced power consumption by 1.5 W per polling thread in a VM configuration by controlling polling threads to sleep, with only an 87 μs increase in latency compared with our conventional KBP, which keeps busy polling in a kernel. EE-KBP achieved low-latency networking on the order of μs, while consuming power as low as NAPI adopted in a current kernel. Furthermore, by dynamically controlling a frequency of a CPU core used by the EE-KBP's polling thread, EE-KBP reduced power consumption by 2.5 to 3.5 W per polling thread compared with KBP, and this was lower power consumption than NAPI. However, due to overhead of the CPU-frequency control function, EE-KBP caused a round-trip delay time of up to 1.3 ms under certain traffic conditions, but it achieved much lower latency than NAPI. EE-KBP achieved 1.4× to 3.1× higher throughput than NAPI, and the performance degradation from KBP

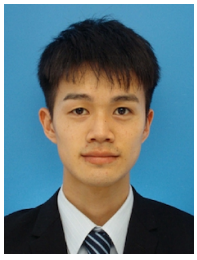
was limited to 4.6% at most.

EE-KBP achieved low latency networking in the order of μs under most traffic conditions and lower power consumption than NAPI, but the dynamic CPU-frequency control function caused a slight delay in the order of ms for some traffic conditions. For future work, we plan to study tuning the CPU-frequency control function by hardIRQ count in accordance with traffic conditions. In addition, since we only measured the effectiveness of EE-KBP in a VM configuration, we plan to evaluate its effectiveness in a bare-metal configuration and a container configuration.

References

[1] 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.
[2] 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.
[3] 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.
[4] R. Gupta, D. Reebadiya, and S. Tanwar, "6G-enabled edge intelligence for ultra-reliable low latency applications: Vision and mission," Comp. Stand. Inter., vol.77, p.103521, 2021.
[5] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, and A. Neal, "Mobile edge computing introductory technical white paper," 2014.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] K. Fujimoto, M. Kaneko, K. Matsui, and M. Akutsu, "KBP: Kernel enhancements for low-latency networking for virtual machine and container without application customization," IEICE Trans. Commun., vol.E105-B, no.5, pp.522–532, May 2022.
[11] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda, "CARB: A C-state power management arbiter for latency-critical workloads," IEEE Comput. Arch. Lett., vol.16, no.1, pp.6–9, 2017.
[12] IEEE Std, "Standard for information technology — Portable operating system interface (POSIX)," 1003.1, 2001.
[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] Intel Corp., "L3 forwarding with power management sample application," https://doc.dpdk.org/guides/sample_app_ug/l3_forward_power_man.html
[17] X. Li, W. Cheng, T. Zhang, F. Ren, and B. Yang, "Towards power efficient high performance packet I/O," IEEE Trans. Parallel Distrib. Syst., vol.31, no.4, pp.981–996, 2020.
[18] J. Cummings and E. Tamir, "Open source kernel enhancements for low latency sockets using busy poll," Intel White Paper, 2013.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] Huaweii, "DMM lwIP," https://github.com/Huawei/DMM 2018.
[25] 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.
[26] D. Brodowski, "Linux CPUFreq governor," https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[27] Kubernetes, https://kubernetes.io
[28] K.N. Khan, M. Hirki, T. Niemi, J.K. Nurminen, and Z. Ou, "RAPL in action: Experiences in using RAPL for power measurements," ACM Trans. Model. Perform. Eval. Comput. Syst., vol.3, no.2, pp.1–26, March 2018.
[29] R. Kavanagh, D. Armstrong, and K. Djemame, "Accuracy of energy model calibration with IPMI," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp.648–655, 2016.
[30] S. Bradner and J. McQuaid, "RFC 2544: Benchmarking methodology for network interconnect devices," IETF, 1999.
[31] Intel Corp., "Pktgen-DPDK," https://github.com/pktgen/Pktgen-DPDK

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. He is a senior research engineer at NTT Network Innovation Center and his current research fields are low-latency networking and power-aware computing. He is a member of IEICE. He received Young Researcher's Award from IEICE Technical Committee on Network Systems in 2020 and Highly Commended Paper Award from IEEE ITNAC in 2021.



Ko Natori received his B.E. and M.E. degree from Keio University in 2019 and 2021. He is currently studying networking system software in NTT Network Innovation Center. His research interests include low-latency networking, system software, and operating systems. He is a member of IEICE. He received Highly Commended Paper Award from IEEE ITNAC in 2021.

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. He received Highly Commended Paper Award from IEEE ITNAC in 2021.

Akinori Shiraga received the B.E. and M.E. degrees in applied physics from the University of Tokyo, Tokyo, in 2000 and 2002, respectively. In 2002, he joined NTT Information Sharing Platform Laboratories. He had engaged in research on web authentication technologies and network functions virtualization and also in development of session control server in NGN system. From 2017 to 2020, he engaged in the development of software-defined network services at NTT Communications Corporation. He is currently studying on photonic disaggregated computer and its control technologies at NTT Network Innovation Center. He received Highly Commended Paper Award from IEEE ITNAC in 2021.