Intelligent Packet Processing for Performant Containers in IoT

Wonmi Choi*, Yeonho Yoo*, *Member, IEEE,* Kyungwoon Lee, Zhixiong Niu, Peng Cheng, Yongqiang Xiong, Gyeongsik Yang, *Member, IEEE,* and Chuck Yoo, *Member, IEEE*

***Abstract*—This paper explores the computing and communication overhead of network processing in IoT devices, focusing on containers, a major building block for edge computing. Our experiments reveal that containers on IoT devices suffer ∼2.6× higher CPU usage for SoftIRQ processing, ∼59% less network throughput, and 2× higher per-packet latency on average than native processes. While several existing studies enhance networking performance, they often sacrifice interoperability by requiring special hardware or modifying networking semantics or APIs. Thus, we design and implement a kernel networking accelerator, called SCON, that maintains interoperability, crucial for IoT devices. SCON addresses major bottlenecks in container networking through system-level profiling. We evaluate SCON with three types of IoT devices. On the Raspberry Pi 4, SCON reduces the latencies of major IoT application protocols (e.g., HTTP and MQTT) by ∼10×, achieving a similar level of latency to the native process. Further analysis shows that SCON reduces CPU usage for SoftIRQ processing by ∼26%. We also report similar improvements on the other two IoT devices. Our conclusion is that SCON is unique in significantly reducing the computing and communication overhead of container networking in IoT devices while maintaining interoperability. Furthermore, it works consistently across different types of devices, whether wired or wireless, and regardless of heavy or sporadic traffic.**

***Index Terms*—Container Virtualization, Device Virtualization, Efficient Communications and Networking, Resource-Constrained Networks, Real-Time Systems.**

I. Introduction

THE Internet of things (IoT) is a substantial paradigm that interconnects physical objects, devices, and systems through the Internet [1], enabling data collection from diverse sources, analysis, and autonomous decision-making. Specifically, the collected data in the IoT concept has enabled the emergence of smart industries, such as smart factories, smart homes, and smart farms, by creating intelligent and responsive environments that enhance operational efficiency and user experience [2], [3]. For instance, in a smart factory, a large volume of data is collected from various sensors to have direct real-time feedback control for efficient management of the manufacturing process [4]. This control mechanism relies on the use of IoT with low latency and high-performance communication [5], [6], [7], [8]. Similar requirements apply to other industries where efficient IoT environments of communication, control, and computing are crucial.

In order for IoT devices[1] to efficiently run various customer applications, containers have become the most popular environment of choice [11]. Containers provide a lightweight way to virtualize IoT devices with resource constraints because the containers run as user-level processes on the host operating system and share its system binaries, unlike other virtualization schemes (e.g., virtual machines) [12], [13]. For containers running in IoT devices, one of the most crucial performance aspects is networking because the significant role of IoT devices is to receive and send data to and from edge clouds continuously [14], [15], [16], [17]. For example, IoT devices with camera sensors in autonomous driving continuously send road videos to the edge cloud for real-time analysis of the road's surroundings [14]. Also, IoT devices are used to repeatedly measure patients' vibration data and send the data to edge servers for real-time emergency monitoring [15], [16]. All these examples necessitate high network performance of containers along with low latency and less CPU usage.

However, on IoT devices, existing container technologies face significant challenges in networking performance. Our motivating experiments, conducted on representative IoT devices using Linux and Docker—the de facto container runtime—highlight the challenges (see §III in detail). First, in a container, the CPU usage for packet processing (SoftIRQ) is ∼2.6× higher than that of a non-virtualized process (native). Second, the network throughput of a container is ∼59% worse than that of the native. In addition, per-packet latency is also 2× higher compared to the native on average. Given that IoT devices have limited power and CPU resources [11], [13], [18], these high CPU usage and poor throughput and latency are significant challenges.

*Wonmi Choi and Yeonho Yoo are co-first authors.

This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2021R1A6A1A13044830), by the NRF grant funded by the Korea government (MSIT) (NRF-2023R1A2C3004145, RS-2024-00336564), by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation grant funded by the Korea government (MSIT) (RS-2020-II201819), and by the Google Cloud Research Credits program. (*Corresponding authors*: Gyeongsik Yang and Chuck Yoo.)

W. Choi, Y. Yoo, G. Yang, and C. Yoo are with the Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea (e-mail: ymcui@os.korea.ac.kr, yhyoo@os.korea.ac.kr, g_yang@korea.ac.kr, chuckyoo@os.korea.ac.kr)

K. Lee is with the School of Electronics Engineering, Kyungpook National University, Daegu 41566, Republic of Korea (e-mail: kwlee87@knu.ac.kr)

Z. Niu, P. Cheng, and Y. Xiong are with the Microsoft Research, Beijing 100080, China (e-mail: zhniu@microsoft.com, pengc@microsoft.com, yqx@microsoft.com)

[1]The IoT devices that this paper focuses on are any devices that can run a standalone operating system and multiple Docker containers, such as the Raspberry Pi 4, as covered in many other previous IoT studies [8], [9], [10].

Previous research has proposed various techniques to enhance the networking stack of containers [19], [20], [21], [22], [23], but these techniques have limitations when applied to IoT devices. First, most existing approaches have hardware dependency or significant semantic changes, which makes them not easily applicable to IoT devices. For example, DPDK [22] processes packets at the user level and bypasses the kernel. However, it requires a specialized network interface card (NIC) that is not commonly supported in IoT devices [24]. Also, AF_GRAFT [19] changes the packet processing semantics in the kernel, so the APIs for the networking from the applications are changed. This means that all the applications should be re-compiled and tested to use the changed APIs. For some cases, it can be acceptable to get accelerated networking performance, but not desirable for IoT devices where interoperability (ease of use) is more valuable than performance boost.

Second, previous studies have yet to address the specific needs of IoT traffic. Some IoT traffic consists of small messages, typically less than 200 B and ranging from 32 B to 256 B [25], [26]. Our motivation experiments with Slim [21], the latest open-source research, show that it cannot enhance the throughput of small IoT messages sufficiently. This implies that existing studies may not be designed to take care of the IoT traffic (§III). Considering the growing importance of edge computing with IoT, container networking needs attention on small IoT messages.

Our study presents a kernel accelerator, called SCON, for IoT devices. The design goal of SCON is not to have any hardware dependency nor modify the semantics changes in APIs. The design of SCON starts with profiling the root causes of the severe bottleneck in processing messages in containers. By profiling the bottlenecks of the networking stack per function, this study reveals that the primary bottlenecks of containers are located in the Internet protocol (IP) layer, where additional Netfilter, header processing, and routing lookup operations are performed for containers (§IV).

SCON addresses the uncovered root causes by introducing *SCON express forward*, which intelligently memorizes the decisions on packet processing for recently seen connections in the kernel (§V-A). The SCON design has four components—Contable, Confilter flag, SCON composer, and SCON flusher (§V-B). Note that SCON maintains the original kernel semantics and socket semantics, so there is no need for any legacy application to be re-compiled or re-tested. Also, the SCON design is entirely kernel-based without any hardware dependency.

We implement SCON in the recent Linux (version 5.10) and conduct experiments with Raspberry Pi 4, a representative IoT device [27], [28]. We use Docker as the container runtime. The major contributions of this paper are as follows:

- Through in-depth system-level profiling, we identify the challenges of IoT containers: high CPU usage and poor networking performance (e.g., throughput) and we find that the root cause is in the packet processing sequence (e.g., Netfilter, header processing, and routing lookup).
- We design and implement SCON, a kernel accelerator to overcome these three challenges. Unlike existing studies, SCON does not require additional hardware support and maintains interoperability with legacy applications and devices.
- Through micro-benchmark evaluations, we show that SCON improves CPU usage and network performance for messages over TCP by ∼26% and ∼45%, respectively, and messages over UDP by an average of 33% and 31%, respectively, compared to the containers.
- We also demonstrate that SCON is scalable with multiple containers, specifically, as the number of containers increases from one to four, the throughput across message sizes increases by 4.2× on average.
- Through evaluation on real-world application protocols (e.g., HTTP and MQTT), SCON demonstrates effectiveness at both streaming and sporadic IoT traffic by improving latency by 48% on average and achieving a similar level of latency to the native process.
- For three different types of IoT devices and wireless connection, SCON consistently enhances CPU usage and network performance compared to the containers, demonstrating the interoperability of SCON.

For the rest of this paper, §II describes the background. §III explains the motivation of this study, and §IV details the root cause analysis of the results and our approach. §V presents the SCON design. §VI shows the implementation of SCON and its evaluations. §VII summarizes related works of SCON. §VIII presents the discussion and future work, and finally, §IX concludes this study.

II. Background

This section provides the background on container networking in IoT devices and its packet processing sequence.

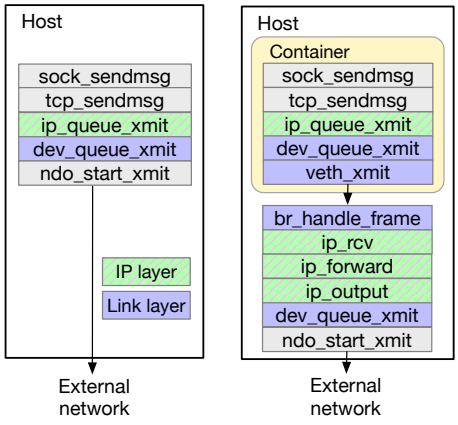
A. Container Networking in IoT Devices

Containers have their unique IP addresses to ensure isolation from other containers.[2] The network switches or routers connecting the hosts recognize only the host server's IP address. Thus, to deliver the packets generated from containers across hosts, existing container platforms utilize one of two techniques: network address translation (NAT) or overlay.

NAT. Each host manages a NAT table that maps container addresses to host addresses. Based on the mappings, NAT modifies the packet headers by replacing the container IP with the host IP. So, packets traversing the external network carry the host addresses, and switches and routers can recognize and correctly deliver them between hosts.

Overlay. Overlay utilizes additional headers. Container packets from the source host are encapsulated with additional headers, such as virtual extensible local area networks (VxLAN) or IP-in-IP, and then decapsulated at the destination host. The external network delivers the packets based on the host addresses in the additional header. Although overlay successfully delivers the container packets across hosts, it is known that overlay incurs bigger overheads in CPU cycle and

[2]Containers can share an IP address of a host (known as host mode), but their network packets are not isolated. So, in real-world scenarios, most container platforms assign unique addresses to containers.



(a) Native Linux. (b) Container.

Fig. 1. Packet processing sequence comparison. Containers have a longer sequence than native Linux.

latency than NAT [29]. This is because the overlay requires the overhead for packet en/decapsulation processes. Also, when the packet size becomes bigger, the packet can be fragmented when the encapsulated packet size exceeds the maximum transmission unit (MTU), which is a non-trivial operation for IoT devices. Thus, for IoT devices with a limited amount of computing resources, NAT is the preferred choice [30], [31], [32].

Various container platforms, such as Docker [33], LXC [34], and OpenVZ [35], utilize a bridge network. There are a few other techniques that deploy alternatives to the bridge, such as Open vSwitch [36], but their role is identical to that of a bridge. Since the NAT operation with a bridge network is the de facto standard in container networking with IoT devices, our focus is on the ***NAT with the bridge***.

B. Packet Processing Sequence

Fig. 1 shows the summary of packet processing sequences of native Linux and containers. We explain transmission control protocol (TCP) packet processing as an example due to its prevalent use in IoT devices and its comprehensive packet processing capabilities, such as connection establishment, reliable packet delivery, and congestion control mechanisms [37], [38], [39]. While user datagram protocol (UDP) can also be used in IoT applications, it is much simpler than TCP, so we use TCP to explain the background of this paper. Note that the processing in the physical network interface, link, and IP layers are identical between TCP and UDP.

We first look into the functions of the native Linux in Fig. 1a. A packet from the user application passes through several network layers in the following order: socket layer (`sock_sendmesg`), TCP layer (`tcp_sendmsg`), IP layer (`ip_queue_xmit`), link layer (`dev_queue_xmit`) to add packer header for data transmissions. Then, the packet is transmitted to the network interface layer (`ndo_start_xmit`) and finally transmitted to the external network.

Next, in Fig. 1b, for the containers, the functions in the TCP and IP layers are the same as the native Linux. However, in the link layer of containers, `dev_queue_xmit` transmits the packet to *veth* (`veth_xmit`) instead of the actual network device of the host. *veth* is a virtual network interface assigned to each container by default, which provides an isolated network address space. Each *veth* has unique media access control (MAC) and IP addresses so the containers can be distinguished.

The packet passing through the *veth* is delivered to the bridge (`br_handle_frame`), which connects *veth* of containers and the host kernel. The bridge then forwards the packet to corresponding physical network interfaces that lead to another IP layer processing: we call it the second IP layer. In the second IP layer, `ip_rcv`, `ip_forward`, and `ip_output` functions are called in order. This call chain involves routing lookups that determine the proper destination IP of the packet and Netfilter operations that decide whether to forward the packet. Especially, NAT is conducted by the Netfilter hook in the `ip_output` in the IP layer that changes the source IP address of the packet from the IP address of the container to that of the host. After NAT from the second IP layer, the packet is delivered to the actual network interface of the host server (`dev_queue_xmit`), and the packet is finally transmitted (`ndo_start_xmit`). Please refer to the Appendix for a more detailed packet processing sequence.

Through the analysis of the packet processing sequence, we find that the networking processing sequence of containers depicted in Fig. 1b has additional function calls in the bridge (e.g., `br_handle_frame`) and IP (e.g., `ip_rcv` and `ip_output`) compared to the native Linux in Fig. 1a. In the next section, we demonstrate how these additional calls impact networking performance.

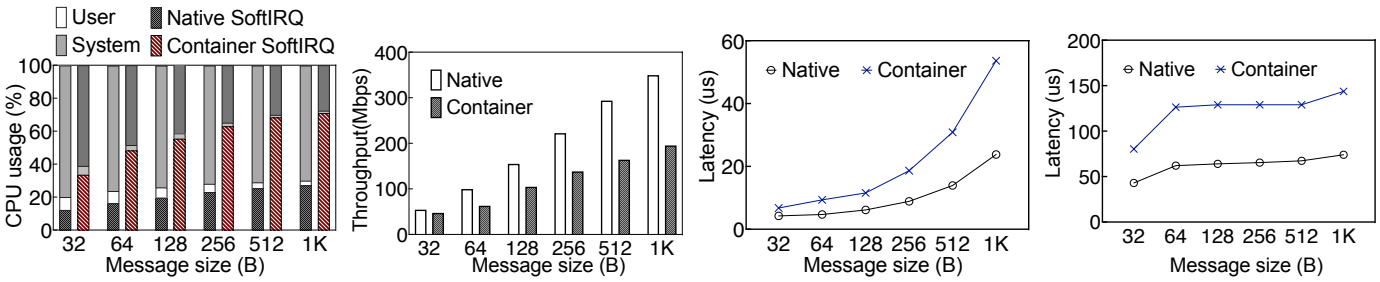
III. Motivation

This section presents the motivating experiments that demonstrate the three significant challenges of IoT containers: 1) inefficient CPU usage in IoT devices, 2) poor throughput and latency in IoT devices, and 3) inadequacy of existing acceleration techniques.

Experiment setting. We show the challenges of IoT containers through various experiments. For experiments, we use a Raspberry Pi 4 as our IoT device as it is a widely-used IoT device [13], [27], [28], [40]. It is equipped with an ARM Cortex-A72 64-bit quad core@1.5GHz CPU, 8GB of RAM, and a 1000 Mbps Ethernet chip. It runs Linux version 5.10 and constructs containers by Docker runtime version 20.10.

The Raspberry Pi is connected to an edge cloud server (server machine) by 1000 Mbps Ethernet. The server machine is equipped with an Intel i7-3770K octa-core@3.5GHz CPU with 64GB memory and a 256GB SSD. The server runs Linux version 5.4. We generate TCP traffic from a container on Raspberry Pi to the server machine, using iperf [41], the most popular network benchmark on IoT system evaluation [18], [42], [43]. Unless otherwise specified, we use the experiment settings described here throughout this paper.

For comparison, we also experiment with the "native" on an identical IoT device with the same settings (e.g., iperf). We present experiment results using one CPU core for the container, which is a typical specification of resource-constrained



(a) CPU usage. (b) Throughput. (c) Per packet latency (average). (d) Per packet latency (99% tail).

Fig. 2. Challenges of SCON: inefficient and poor CPU usage, network throughput, and latency of containers on IoT devices.

IoT environments [44]. We have also conducted experiments with a higher number of cores (up to four cores), which show similar trends to the following results; so, we omit them.

Challenge 1: Inefficient CPU usage in IoT devices. We first evaluate the CPU usage (Fig. 2a) in IoT devices. We increase the size of the message over TCP from 32 B to 1 KB (x-axis) to cover the full range of typical IoT traffic. For example, message sizes from 32 B to 256 B are commonly generated by smart sensors like bulb and temperature sensors [25], [26]. Message sizes from 512 B to 1 KB are seen in smart cameras and smart speakers [45]. We measure the CPU usage of User, System, and SoftIRQ by mpstat [46] and compare the measurement results between the container and the native process.

In Fig. 2a, we observe that both the native and the container in IoT fully utilize the CPU, but the detailed usage is different between the two. Specifically, the CPU usage of the container for SoftIRQ processing is significantly higher compared to the native. For instance, when processing 1 KB messages, the container networking incurs $\sim$2.6$\times$ higher CPU usage than the native for SoftIRQ processing. These results negatively impact the network throughput because fewer CPU resources become available for processing at the system level that handles system calls and at the user level that generates packets. This inefficient CPU usage poses a significant challenge for IoT devices, especially because many are battery-powered and, therefore, highly sensitive to CPU usage.

Challenge 2: Poor throughput and latency in IoT devices. Next, we evaluate network throughput and latency for IoT devices. Fig. 2b displays network throughput increase with message size. For 1 KB messages, in particular, the network throughput of the container is reduced by $\sim$59% compared to the native. On average, the container has 44% lower throughput than the native.

We also evaluate per packet transmission latencies in terms of average and 99% tail values measured by Netperf [47]. The results show that both the average and the 99% tail latency values increase. Specifically, in Fig. 2c and Fig. 2d, the container increases the average latency by 2$\times$ and the 99% tail latency also by 2$\times$ on average. Although the difference is not large in terms of the absolute value (40 μs), this difference is still problematic for IoT devices used in remote health monitoring, and factory control, which require real-time communication in microseconds because such difference may damage finely calibrated machinery [48]. Also, industrial IoT systems require latencies of microsecond level (less than 1 ms) and jitter under 100 µs [49]. Thus, even microsecond delays can impact synchronization and precision control, potentially leading to defects or failures in real-time IoT systems.

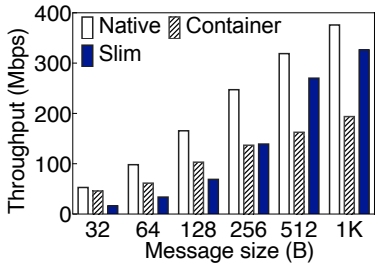


Fig. 3. Comparison of traditional container environment and networking acceleration technique (Slim) on an IoT device.

Challenge 3: Inadequacy of existing acceleration techniques. Because the containers draw significant attention from the datacenter perspective, various networking acceleration studies have been carried out. However, the studies are inadequate for IoT devices due to the following reasons.

First, existing studies lack interoperability for heterogeneous IoT devices. In order to accelerate networking, several studies changed kernel semantics in packet processing (e.g., kernel bypassing and user-level processing) [19], [22], and it requires modifications to the socket APIs of all legacy applications. This implies that considerable development efforts are required to adapt these techniques to various heterogeneous IoT applications.

Other studies have proposed the use of additional hardware (e.g., RDMA NIC or DPDK NIC) to offload network processing [20], [22], [50]. However, to our knowledge, it is unclear whether such NICs are readily available in IoT devices. Therefore, although these techniques can improve the networking performance, they may not provide interoperability for heterogeneous IoT devices that lack the necessary hardware support.

Second, such studies do not sufficiently accelerate IoT traffic. We experiment with the performance of the open-source acceleration technique, Slim [21], the latest and representative method that bypasses parts of the networking stack without requiring specialized hardware. On IoT devices, we evaluate Slim using iperf, which is the only benchmark we found to be compatible with Slim on IoT devices. Note that Slim includes a custom library for running legacy Linux applications. However, due to changes in packet processing semantics (host bypassing), this library does not fully support native system

calls (e.g., `accept`). We tried to run IoT traffic applications on Slim but it requires modifications. So, we present the iperf results here.

We compare the throughput between the native, Slim, and the container. First, for large messages of 512 B and 1 KB, Fig. 3 shows that Slim improves network throughput by 66% than the container on average. When compared with the native, Slim shows only 14% lower throughput on average. However, for small messages ranging from 32 B to 256 B, Slim shows quite poor throughput values compared to the native, with 59% decrease on average. Even compared to the container, Slim performs poorly by 34% on average. The results indicate that existing techniques such as Slim are designed for the acceleration of large traffic [21], but small IoT messages do not gain the improvements. This highlights the need for a new accelerator to cover the diverse IoT messages in container networking.

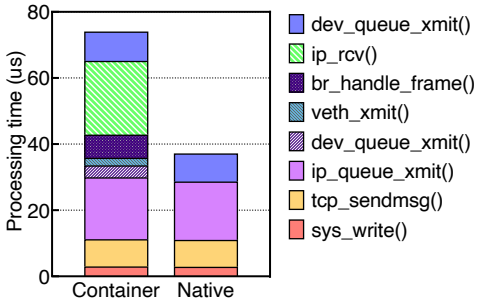
IV. Root Cause Analysis and Our Approach

To design a new accelerator for IoT devices and traffic, we investigate the underlying causes of inefficient and poor CPU usage, network throughput, and latency. We analyze the IoT container networking stack using the ftrace [51]. Specifically, we measure the time it takes to process each function symbol in the networking stack of IoT containers when sending 64 B messages over TCP.

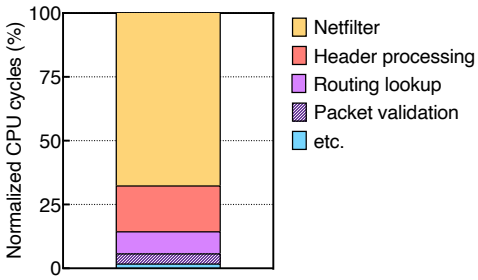
Fig. 4a depicts the measurement results for the key functions involved in packet processing in container networking, as discussed in §II. The results indicate that the time required for processing a single message in the container is twice as high as the native. We find this is because the container networking stack requires "additional" processing, such as bridge layer (`br_handle_frame`), the second IP layer (`ip_rcv`), and veth/eth (`dev_queue_xmit` and `veth_xmit`), which are discussed in §II. Note that other parts in the container networking stack, such as TCP and IP layers, consume similar amounts of CPU time to the native environment.

The results show that the IP layer in the additional processing accounts for the largest portion (63%). So, we conduct further analysis on the packet processing in the IP layer. We measure the CPU cycles spent for every symbol and categorize them into the following operations: Netfilter, header processing, routing lookup, and packet validation. Each category implies the following operations:

- Netfilter: the packet filtering offered by the Linux kernel. Netfilter also includes NAT processing that changes the packets' source IP address of containers to the host server in order to enable external networking. The default configuration is used in our experiments.
- Header processing: recompute the time-to-live (TTL) field and checksum values and re-allocate the packet header to expand the size of the header for NAT.
- Routing lookup: determine the corresponding network interface of the packet to be transmitted.
- Packet validation: check whether the packet is local or remote in the Linux bridge.



(a) Profiling results on the network processing.



(b) Overhead breakdown of the additional IP layer.

Fig. 4. Profiling results on the network processing in Container environment.

Fig. 4b shows that the sum of Netfilter, header processing, and routing lookup occupies 94% of the CPU cycles in the additional packet processing. Our deep analysis reveals that these three operations consume such a large amount of CPU cycles due to the following reasons. First, Netfilter performs NAT table lookup for the external networking between containers. Also, Netfilter implements filtering policies for firewalls or security reasons, which includes iptables lookup according to the policies. We find that Netfilter requires expensive spin-lock operations, resulting in high CPU overhead. Furthermore, the additional IP header processing is carried out for the second IP layer processing. Also, routing requires a lookup over data structures (trie) for every received packet to find the proper destination.

Netfilter, header processing, and routing lookup are conducted repeatedly for every packet, even though a container transmits packets with the same source and destination pair. So, when IoT devices transmit the burst traffic, it significantly drains the CPU, and results in throughput and latency degradation. So, in this study, we design a kernel accelerator on the three high-overhead operations: 1) Netfilter, 2) header processing, and 3) routing lookup. The goals of the SCON are summarized as follows:

- Reduce repetitious resource consumption while improving the networking throughput and latency for IoT network traffic.
- Design a networking acceleration technique without any hardware offloading supports for IoT devices.
- Avoid kernel semantics changes or API changes resulting in application re-implementation.

V. SCON Design

This section presents a detailed design of SCON that offers high network performance with efficient CPU usage on IoT

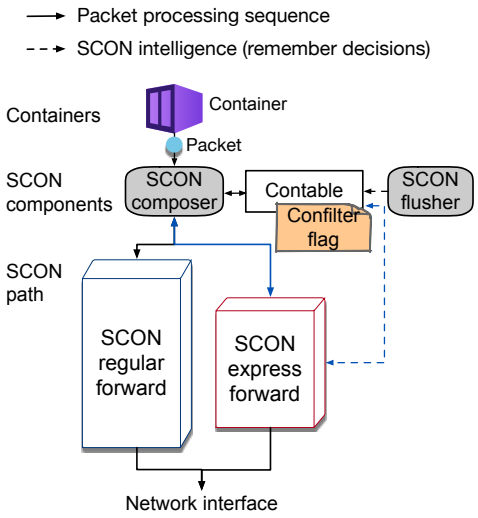


Fig. 5. SCON design overview.

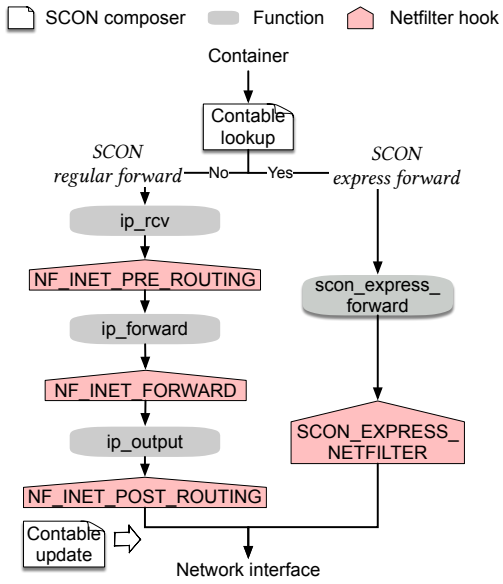


Fig. 6. SCON path details.

devices. Fig. 5 shows the design of SCON. In order to eliminate the high-overhead operations observed in §IV, we devise "SCON path" that consists of *SCON regular forward* and *SCON express forward*, and they add "intelligence" to the container packet processing in order to avoid repetitious operations in Netfilter, header processing, and routing lookups. The intelligence of SCON is to remember the decisions on packet processing (e.g., routing table lookup results) in *SCON regular forward* for recently seen connections.

Specifically, when the packet from a container is delivered to the Linux bridge, SCON distinguishes which routine is suitable between *SCON regular forward* and *SCON express forward* to process the packet. If the packet belongs to a newly generated connection between source and destination containers (identified by 5-tuple information including the source and destination IP and port addresses with the L4 protocol identifier), it goes to the *SCON regular forward*. If the packet belongs to a network connection that SCON has already seen recently, it goes to the *SCON express forward*.

However, identifying the 5-tuple information of every incoming packet without hardware offloading and API modification can cause significant overhead in the Linux bridge as the Linux bridge handles packets from multiple containers simultaneously. In order to reduce the cost of packet identification, we introduce the following SCON components: 1) *Contable* and *Confilter flag* as per-container structures for packet processing and 2) *SCON composer* and *SCON flusher* for allocating and managing *Contable*. In the following subsections, we explain the SCON path and SCON components one by one.

A. SCON Path

We first explain the *SCON regular forward* of SCON path where a packet follows the call chain of the original Linux networking stack (Fig. 6). While traversing the call chain, SCON detects the packet and collects the packet processing results from Netfilter, routing lookup, and header processing, as listed below.

- Netfilter: the results of `NF_INET_PRE_ROUTING`, `NF_INET_FORWARD`, and `NF_INET_POST_ROUTING`.
- Routing lookup: IP and MAC addresses of the next-hop.
- Header processing: MTU size, TTL value, and L3 protocol ID.

After the packet has passed through the call chain, SCON "memorizes" the collected packet processing results in a store called Contable (to be explained in §V-B1) along with the identifier of the network connection (5-tuple values) to which the packet belongs.

Next, the *SCON express forward* avoids repetitious packet processing by utilizing the Contable configured in the previous *SCON regular forward*. The *SCON express forward* consists of two parts: 1) `scon_express_forward` and 2) `SCON_EXPRESS_NETFILTER`. Compared to its native counterpart, `scon_express_forward` offers lightweight alternatives for packet header processing and routing lookup, and `SCON_EXPRESS_NETFILTER` for Netfilter processing.

`scon_express_forward` retrieves the results stored in Contable and fills the packet with the necessary header information (e.g., the result of routing lookup, TTL value, and MTU size). It also performs the essential checksum operation and sends it to the next `SCON_EXPRESS_NETFILTER`.

In `SCON_EXPRESS_NETFILTER`, SCON reduces the overheads of Netfilter processing (e.g., repetitious spin-locks for structure lookup, §IV) by memorizing the Netfilter decision stored in the Contable. The decisions from Netfilter are the following two actions: 1) success and 2) discard. Success means that the packets are ready to go out immediately (including source NAT processing), so the `SCON_EXPRESS_NETFILTER` sends the packets to the network interface. Discard means that the packet should be dropped due to a security breach or firewall policies set by system administrators.

To ensure the SCON path operates as intended, Contable structures should be allocated and created at the proper time to store connection information intelligently. Also, it is necessary

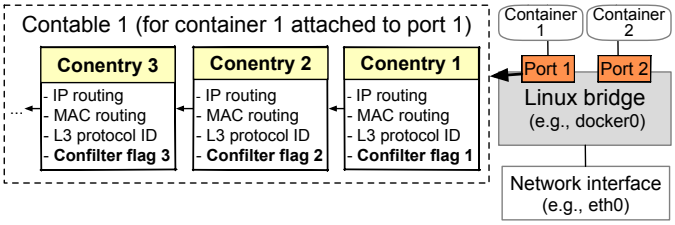


Fig. 7. Structure of Contable and Conentry.

to update the memorized connection information of Conentry (§V-B4). Specifically, the Conentry of Contable needs to be updated in two situations. First, if the Conentry is not accessed within a specific time interval, it should be flushed. Second, when an existing connection is refreshed (e.g., when a TCP connection is closed and then restarted), the Conentry must also be flushed and updated accordingly. SCON components fulfill these roles, which will be explained in the following subsection.

B. SCON Components

Here, we describe four components of SCON: 1) Contable, 2) Confilter flag, 3) SCON composer, and 4) SCON flusher.

1) Contable: The Contable structure is shown in Fig. 7. Contable memorizes how the packet of the connection has been processed by SCON regular forward. It also identifies which SCON routine each packet should take. SCON maintains a separate Contable per container, which consists of multiple Conentry structures managed as a linked list.

A Conentry is built and used for each connection. SCON identifies both the Conentry and connection by the *5-tuple*. Each Conentry maintains 1) routing information (i.e., *IP routing, MAC routing, L3 protocol ID*) and 2) *Confilter flag*. The *IP routing* stores the destination network interface (i.e., struct `net_device`) of the packets, the TTL value, and the MTU size of the corresponding interface collected from the *SCON regular forward*. *MAC routing* stores the L2 information of the next-hop server that receives the packets sent from the host server. The *L3 protocol ID* is the IP protocol number required for the IP header. *Confilter flag* aggregates the Netfilter operations and is a part of packet header processing, which will be explained further in the next subsection.

2) Confilter flag: The Confilter flag of the Contable embeds processing results from Netfilter and header processing. Netfilter plays a role for users or system administrators to define rules for malicious or filterable network connections. Also, for normal network connections, Netfilter performs source NAT for external communication. The result from Netfilter is either 1) success or 2) discard, as explained in §V-A.

During header processing, the packet header checksum is calculated to detect any errors in the packet bits. Note that these errors exist on a per-packet basis and do not persist over the entire connection to which the packet belongs. The result of the header processing can be classified into 1) pass (normal) or 2) error (dropped).

When the packet (connection) is "success" from Netfilter and is "pass" from header processing, SCON sets the Confilter flag as "success." In this case, the *SCON regular forward* directly sends the packets to the network interface. If the packet is discarded from Netfilter (regardless of the results of header processing), SCON sets the Confilter flag as "discard" and the *SCON regular forward* drops the packets. Lastly, when the packet is considered "error" from header processing, the packet coming at the next time goes through the *SCON regular forward* so that the information of the Contable can be updated with the packets. The Confilter flag of success, discard, and go-to *SCON regular forward* is defined by integers (i.e., 1, -1, and 0).

3) SCON composer: SCON composer is to construct the Contable structure. When a packet is transmitted from a container, the SCON composer looks up the Contable to check if any Conentry matches the packet's 5-tuple. If there is no matching Conentry, it indicates that the packet belongs to a new network connection that has not been seen before. Therefore, the SCON composer allocates a new Conentry, adds it to the Contable, links it to the previous Conentry, and initializes its entries. After the packet goes through the SCON regular path, the SCON composer updates the Conentry with the memorized processing information of the packet.

4) SCON flusher: SCON flusher is responsible for removing Contable or Conentry structures that are no longer valid. There are several situations in which these structures become invalid. The first situation is when the corresponding container is stopped or removed, in which case the Contable allocated for the container will no longer be used. We design each Contable to be dependent on the container's interface (i.e., bridge port), as the container's interface is removed by the kernel when that container is stopped or removed. So, the Contable is removed together when the container goes down.

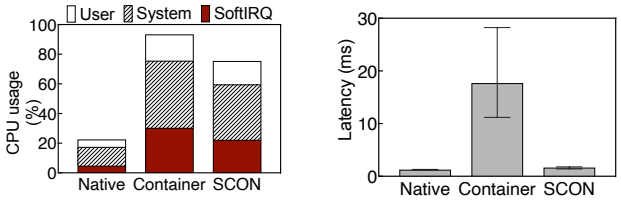
The second situation arises when the container stops transmitting packets in the connection. To detect this, a timer periodically checks the frequency of Conentry hits. If a Conentry is not hit within the timer interval (e.g., 10 s), the SCON flusher is activated to free the Conentry.

SCON flusher is also designed to take care of situations where an existing connection is refreshed. When a TCP connection is closed and then restarted, Conentry must be flushed and updated accordingly. To detect this situation, the SYN flag of the TCP packets is monitored. Note that this monitoring is particularly important for IoT devices because TCP-based protocols such as MQTT/HTTP heavily use short-lived messages for energy efficiency [52], [53], [54], [55].

VI. Implementation and Evaluation

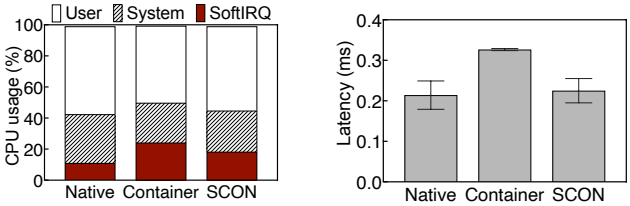
We implement all the components of SCON on Debian GNU/Linux 11 (bullseye) using Linux kernel version 5.10. The implementation comprises 2K lines of C code. Additionally, we provide a configuration option that disables the SCON. We release the implementation of SCON on GitHub repository[3]. The experimental settings are identical to those explained in §III. We compare the native Linux process, container, and SCON. Although we try to compare SCON with other existing studies, they assume either special hardware support or their

[3]https://github.com/wonmi123/SCON.git



(a) CPU usage. (b) Average latency.

Fig. 8. [Real-world application protocols] HTTP scenario.



(a) CPU usage. (b) Average latency.

Fig. 9. [Real-world application protocols] MQTT scenario.

own API/semantics, making direct comparisons with SCON unfeasible (details in §VII).

We conduct three sets of experiments: 1) real-world application protocols, 2) micro-benchmarks, and 3) interoperability on other devices and wireless networking. As real-world application protocols, we evaluate the effectiveness of SCON with 1) HTTP scenario and 2) MQTT scenario because most IoT traffic is generated using HTTP or MQTT protocols [52], [53], [54], [55]. For both scenarios, we measure the amount of CPU cycles and job completion time of transmitting the IoT traffic.

As micro-benchmarks, we first assess CPU usage and network throughput by gradually increasing the message sizes using TCP protocol from 32 B to 1 KB. This message size range includes almost all IoT packets as explained in §III. Second, we analyze the processing time per packet and function symbol using detailed profiling similar to Fig. 4. Third, we measure the flow completion time (FCT) for IoT traffic and present the 99% tail values from the results. Fourth, we report the performance of SCON with the UDP protocol. Lastly, we examine the scalability of SCON's throughput values as the number of containers on an IoT device increases. Each experiment is conducted at least five times and we report the average values.

We also test the performance of SCON on different IoT devices that have distinct CPU architectures and processing speeds. In addition, we evaluate SCON in a wireless network setting. The detailed device specifications and experimental settings are explained together with the results in §VI-C.

A. Real-world Application Protocols

1) HTTP: We first test the HTTP performance of SCON by running one Nginx [56] (version 1.23.2) on a Raspberry Pi to create HTTP traffic. On the opposite side, we run wrk2 [57] on the Intel server and mpstat [46] to measure network performance and CPU usage.

In particular, for each connection from the IoT device, wrk2 spawns one thread and requests a 64 B HTML file. In smart factories, smart cities, disaster management, and healthcare, the number of messages per second increases exponentially due to the interconnection of hundreds to thousands of IoT devices [58], [59]. Also, IoT devices typically use HTTP or MQTT protocols to collect and send massive amounts of sensor data keeping real-time delivery and constant monitoring [60]. Thus, we set the input as 2K requests/s to reflect these scenarios, which is close to the upper limit of the container's capabilities in our setting. Also, we measure the CPU usage through mpstat and HTTP latency from wrk2. Note that the latency measured here is the time for sending a request and receiving a corresponding reply. Each experiment lasts 10 s.

CPU usage. Fig. 8a shows the CPU usage breakdown. Although the identical 64 B file is used for experiments, native, container, and SCON show quite different CPU usage. In comparison to the container, SCON reduces 19% of CPU usage. In particular, SCON reduces 26% of CPU cycles spent in SoftIRQ processing. As a result, the gap in total CPU usage between native and container to achieve the required throughput declines by 25% using SCON.

HTTP latency. Fig. 8b shows the average HTTP latency reported by wrk2, which includes the times for an HTTP request and reply. The bars represent average values, and the whiskers represent ranges. The container exhibits much more pronounced HTTP latency than the native, which is 14× longer. On the other hand, SCON reduces the latency by 10× compared to the container. Specifically, the latency difference between the native and container is 1325%, while the one of native and SCON is only 31%, which is 42× improvement.

2) MQTT: Next, we evaluate MQTT [61], a widely used data streaming protocol in IoT. MQTT provides "MQTT broker" that intervenes between the data publishers (e.g., IoT devices) and subscribers (e.g., edge cloud servers). MQTT broker then receives various data from publishers and delivers it to proper subscribers.

For the experiments, we use the latest MQTT protocol of version 5.0 [62]. Also, for the MQTT broker component, we use Mosquitto application version 2.0.15 [63]. For the MQTT publisher and subscriber, we use MQTTLoader version 0.8.6 [64]. Specifically, an MQTT publisher runs on a Raspberry Pi, while a broker and a subscriber run on an Intel server. We generate messages of 64 B data each from an MQTT publisher (Raspberry Pi) using MQTTLoader. To reflect scenarios where IoT devices process highly frequent requests, we experiment by sending 1,000,000 messages within 60 seconds. The broker receives the 1,000,000 messages and then delivers them to the subscriber (Intel server).

We repeat the experiments at least five times and present the average values. We measure CPU usage and MQTT latency. MQTT latency includes the processing time from message generation by the publisher to message reception by the subscriber. The CPU usage is measured by mpstat, and the latency by MQTTLoader.

CPU usage. Fig. 9a shows the CPU usage of MQTT scenario. When comparing the native and container envi-

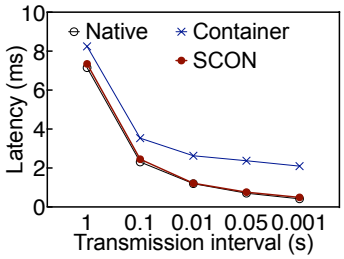
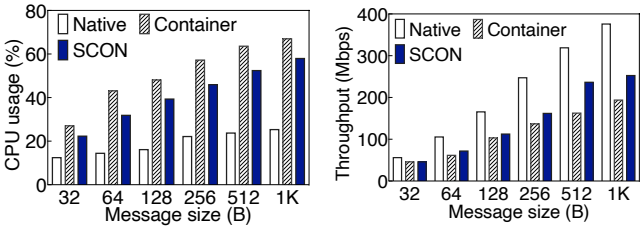
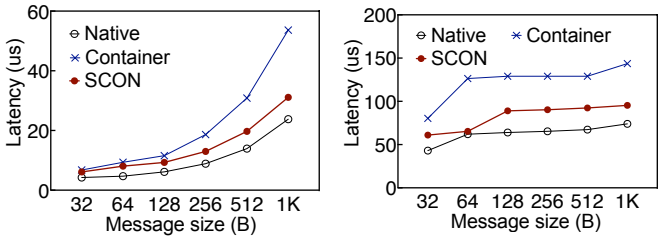


Fig. 10. [Real-world application protocols] MQTT latency with various message transmission intervals.



(a) CPU usage. (b) Network throughput.

Fig. 11. [Micro-benchmark] CPU usage for SoftIRQ and network throughput in TCP.



(a) Average latency. (b) 99% tail latency.

Fig. 12. [Micro-benchmark] Per packet latency.

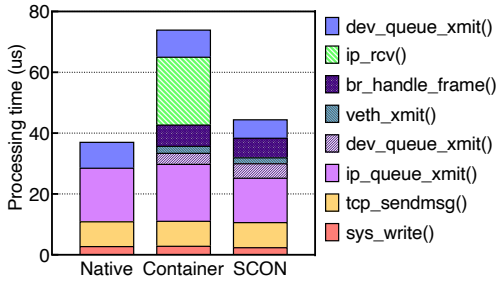


Fig. 13. [Micro-benchmark] Processing time per packet and function symbol.

ronments, the container environment consumes much more CPU cycles for packet processing (SoftIRQ), which is 2.2× higher. Due to this high processing in SoftIRQ, the User time of the container environment experiences a decrease in CPU cycles (0.87×), resulting in poor performance as shown in Fig. 9b. On the other hand, SCON reduces the SoftIRQ consumption of containers by 24%, which leads to improved MQTT performance.

MQTT latency. Fig. 9b shows MQTT latency. We depict the average value by bars and ranges by whiskers. When comparing the native and container environments, the latency of the container environment increases by 1.5×. On the other hand, when comparing the container environment with SCON, SCON reduces the latency by 31%. Moreover, when comparing the native environment with SCON, the delay of SCON is only 5% higher than the native environment (at least 2%). The results show that SCON significantly improves MQTT latency to the level similar to the native process. This means that SCON greatly reduces the networking latency of IoT application protocols by lowering CPU consumptions.

3) Transmission intervals on MQTT: Then, we evaluate how SCON can handle various transmission request rates by measuring the latency of MQTT. We vary the MQTT message interval from 1 s to 0.001 s, so the number of requests generated for each second ranges from 1 to 1000. All other configurations are similar to those in §VI-A2.

Fig. 10 shows that, compared to the container, SCON reduces the latency by 48% on average. In addition, SCON exhibits only 7% longer latencies on average compared to the native. The results show that SCON effectively improves MQTT latency across a range of message transmission intervals.

B. Micro-benchmarks

1) CPU usage: We generate TCP traffic using iperf [41] and measure CPU usage (%) using mpstat. Each bar of Fig. 11a shows the average CPU cycles for SoftIRQ processing (y-axis) per message size (x-axis).

For CPU usage, we focus on the SoftIRQ part because it corresponds to the packet processing overhead that SCON aims to improve. As stated in §III, all experiments are conducted on one CPU core, which is fully utilized in all cases (100%). By subtracting the SoftIRQ usage presented in Fig. 11a from the total CPU usage of 100% (one core), we obtain the CPU usage mainly utilized by the User, i.e., the IoT application. Thus, minimizing SoftIRQ usage is desirable as it frees up more CPU cycles for IoT applications.

Fig. 11a shows the CPU usage for SoftIRQ increases as the message size increases for all cases. This is because as the message size increases, the number of messages generated at the user level and the number of system calls processed at the system levels decrease. Fig. 11a also shows that SCON reduces the CPU cycle spent in SoftIRQ by 19% on average compared to the container. In particular, the maximum CPU usage reduction is 26% at 64 B messages.

2) Network throughput: We measure the network throughput (Mbps) and depict the results in Fig. 11b. On average, SCON demonstrates 20% higher network throughput compared to the container. When comparing the throughput against the native process, the container shows 40% lower throughput on average, while SCON shows 29% lower throughput than the native. Thus, SCON reduces the throughput difference by 28% compared to the container. For 512 B messages, SCON improves throughput by ∼45% and reduces the throughput difference with the native process by ∼47%.

These results show that SCON effectively reduces CPU usage and improves container networking throughput when processing highly frequent messages.

3) Per packet latency: We measure the per packet latency using Netperf [47] by changing the message sizes from 32 B to 1 KB. Fig. 12a and Fig. 12b show the average and 99% tail latency for sending a message using TCP from the IoT client to the Linux edge server for 10 s, respectively. SCON reduces the average latency by 25% on average compared to the container and declines the gap between the native and containers by 49% on average. For 99% tail latency, SCON reduces 32% of container latency on average and minimizes the gap between native processes and containers by 66% on average.

4) Processing time per function symbol: Here, we further investigate the impact of SCON by conducting the same profiling shown in §IV. We measure the time it takes to process each function in the networking stack of containers in IoT when sending messages of size 64 B. Fig. 13 illustrates the per packet execution time on average in native, container, and SCON. Each bar consists of the functions we profiled, such as `ip_rcv`, `br_handle_frame`, and `veth_xmit`.

SCON reduces the total time to process a single message by 43% compared to the container. While the CPU usage of container increases 99% than native, the difference between SCON and native is 20%, which is 5× improvement. More specifically, SCON saves 22 μs spent in `ip_rcv`, keeping the time spent in other symbols similar to the container. This indicates that *SCON express forward* successfully reduces the overhead caused by additional and repetitious packet processing in the existing container networking stack.

5) Flow completion time: We measure FCT, which is the time required to complete a flow of IoT packets. We generate IoT traffic on Raspberry Pi by traffic generator [65]. The traffic generator requires a flow size distribution that characterizes a packet trace by 1) message size and 2) frequency. The message size refers to the size of data (payloads) that a single flow delivers, and the frequency indicates how often each message size appears in the trace, calculated as the number of flows of that size divided by the total number of flows. We use the flow size distribution of IoT sensor devices [66], such as LiFX lightbulbs, Belkin Motion sensors, and Amazon Echo devices, where each flow sends 100 to 1400 bytes of data. In each trial, the traffic generator sends 100 IoT traffic flows to an Intel server acting as an edge cloud server. Once the traffic generator finishes sending a flow, we measure the time to complete the flow (FCT). From the 100 FCTs, we calculate the 99% tail value. We conduct the measurements at least five times and present the average.

The measurement results are shown in Fig. 14. When comparing SCON and container, SCON reduces the 99% tail FCT by 20%. Furthermore, SCON decreases the gap between the native and container from 12 ms to 2.9 ms, which means 4.1× small.

6) SCON on UDP: In the IoT domain, UDP is also employed as a lightweight protocol to facilitate low-power communication (e.g., Constrained application protocol (CoAP)) [67]. Because SCON is an optimization technique on the IP layer, SCON is compatible with UDP without any modifications. Here, we measure the CPU usage and network

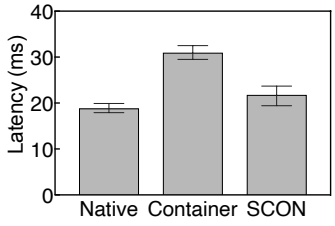
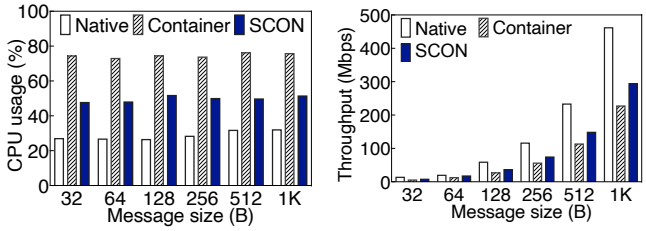


Fig. 14. [Micro-benchmark] 99% tail FCT of IoT traffic.



(a) CPU usage. (b) Network throughput.

Fig. 15. [Micro-benchmark] CPU usage for SoftIRQ and network throughput in UDP.

throughput with similar settings to those described in the TCP evaluation, but generating UDP traffic from iperf.

We first explain the CPU usage for SoftIRQ results in Fig. 15a. As the message size (x-axis) increases, SCON reduces the CPU cycles spent in SoftIRQ by an average of 33%. Also, when the message size is increased from 32 B to 1 KB, the container experiences only 1.7% increases in CPU cycle consumption, which can be considered negligible.

Next, Fig. 15b shows the network throughput of UDP. SCON demonstrates 31% higher throughput on average compared to the container. Also, SCON narrows the throughput gap between the native and the container processes by an average of 36%.

These results reveal that SCON effectively improves both CPU usage and throughput for UDP traffic as well as TCP traffic.

7) Scalability of SCON: Considering that multiple containers run simultaneously [68], [69], we measure SCON's network throughput as the number of containers on a device increases. Since the IoT device in our experiments has four CPU cores, we increase the number of containers from one to four by pinning one core per container using cgroups [70]. Similar to the experiments in §VI-B, we run iperf on the containers and send messages to the edge cloud server machine.

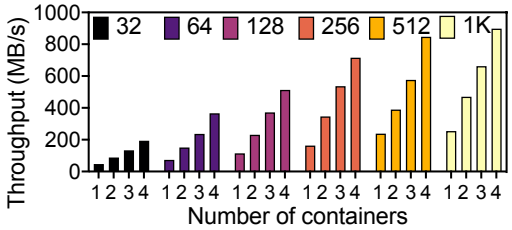
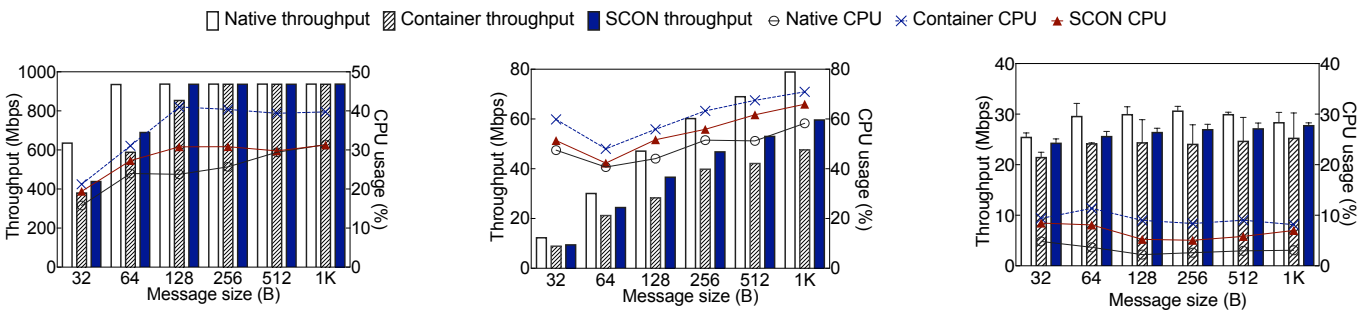


Fig. 16. Network throughput of SCON with increasing number of containers.



(a) ASUS PN42 (b) Raspberry Pi 3B+ (c) Wireless networking (Raspberry Pi 4)

Fig. 17. Network throughput and CPU usage for SoftIRQ for other devices and wireless networking.

TABLE I
SPECIFICATION OF THE DEVICES.

Specification		ASUS PN42 [71]	Raspberry Pi 3 Model B+ [72]
Hardware	CPU	quad-core Intel N100 @ 3.4 GHz	quad-core ARM Cortex-A53 @ 1.4 GHz
	Storage	128 GB	128 GB
	NIC	1000 Mbps	300 Mbps
Software	OS	Ubuntu: 20.04, Linux: 5.10	Ubuntu: 20.04, Linux: 5.10

Fig. 16 shows the measurement results. For different message sizes ranging from 32 B to 1 KB, we measure the networking throughput as the number of containers increases from one to four. Each bar represents the sum of the individual container's throughput, so the bar for three containers is the sum of the throughput of three containers. For all message sizes, the networking throughput scales linearly with the number of containers. When the number of containers increases from one to four, the throughput across message sizes increases by an average of 4.2×. These results demonstrate that SCON is scalable with the number of containers.

C. Interoperability on Other Devices and Wireless Networking

1) Different IoT devices: In addition to the experiments with the Raspberry Pi 4 up to this section, we also evaluate SCON with different types of devices. Table I summarizes the specifications of the two devices. First, the ASUS PN42 is chosen because it has much faster (2.3× faster in clock speed) CPU cores and different architecture (Intel) compared to the Raspberry Pi 4, which has ARM cores. Second, we choose the Raspberry Pi 3B+ as it has similar ARM cores to the Raspberry Pi 4, but with 70% lower network bandwidth capacity and 7% slower clock speeds. Except for the IoT devices, the other setup remains the same as in §VI-B.

ASUS PN42. Fig. 17a shows the network throughput (bars, left y-axis) and CPU usage for SoftIRQ processing (lines, right y-axis) on the ASUS PN42 device. As the message size increases, SCON demonstrates ∼17% better throughput (in 64 B) and ∼25% reduced SoftIRQ CPU usage (128 B) compared to the container. On average, these improvements are 7% and 17%, respectively, compared to the container. Also, the native and SCON saturate the network capacity of 1000 Mbps with message sizes starting from 128 B, and the container does so from 256 B. In the experiment with the Raspberry Pi 4, the throughput does not saturate up to 1000 Mbps (shown in Fig. 11b). This is because the ASUS PN42 device has faster CPU cores than the Raspberry Pi 4, which accelerates packet processing and saturates the entire network capacity.

Raspberry Pi 3B+. Fig. 17b shows the results on the Rasberry Pi 3B+. SCON increases network throughput by 20% on average and reduces the CPU cycle spent in SoftIRQ by 10% on average compared to the container. Also, we observe that the throughput of Raspberry Pi 3B+ does not saturate its network capacity (300 Mbps) in contrast to the results of ASUS PN42 (in Fig. 17a). This is because the Raspberry Pi 3B+ has low-speed CPU cores, requiring more time to process each message.

In summary, for both different devices, SCON improves CPU usage and throughput compared to the container.

2) Wireless network setting: We now evaluate SCON in a wireless network setting. To eliminate interference from the devices, we set up a WiFi router with 802.11ax LAN to connect the Intel server and a Raspberry Pi 4. We also evaluate other devices, such as ASUS PN42 and Raspberry Pi 3+. Since they show similar tendencies to the Raspberry Pi 4, we omit their detailed results.

Fig. 17c shows the network throughput and SoftIRQ CPU usage, which is similar to Figs. 17a and 17b. We first explain the results of CPU usage (lines). In a wireless networking setting, SCON effectively reduces SoftIRQ CPU usage by an average of 28% compared to the container. Also, regardless of the message sizes, all results remain under 11% of SoftIRQ CPU usage. This is because the WiFi provides low networking bandwidth (30 Mbps at maximum), so the networking capacity becomes a bottleneck before CPU resources.

Second, for the network throughput (bars of Fig. 17c), SCON improves network throughput by an average of 10% than the container. In addition, we observe that, from 128 B, the throughput of SCON does not vary significantly (∼5%) over message sizes. This is because, starting from 128 B messages, the WiFi bandwidth becomes saturated quickly, so no significant throughput difference exists. In summary, even in wireless networking, SCON enhances networking throughput while consuming less CPU.

VII. RELATED WORK

We summarize the related studies in Table II and compare them with this study in terms of interoperability.

First, many studies have been built on the concept of bypassing kernel processing, either partially or entirely, through

TABLE II
RELATED WORK COMPARISON.

	Bypass kernel (or in-part)	Additional hardware for IoT devices	Modification of kernel semantics	Application modification required
DPDK [22]	O (Host)	O	X	O
Freeflow [20]	O (Host)	O	O	X
Falcon [23]	O (Host)	O	X	X
Socket-grafting [19]	O (Container)	X	O	O
Slim [21]	O (Container)	X	O	△
SCON (this study)	**X**	**X**	**X**	**X**

hardware or by shifting the processing to the user space. One example is Intel DPDK [22] that bypasses the kernel networking stack and delivers packets directly to the user space, facilitating high throughput and low latency for packet processing. However, it is quite well-known that its use is limited to NICs that support DPDK functionalities and APIs and that user applications need to be modified to use DPDK APIs to communicate with NICs. Furthermore, DPDK also requires pre-allocated large memory to handle heavy traffic, making it questionable for IoT devices with limited memory size.

Also, FreeFlow [20] bypasses host parts of the kernel networking by leveraging additional hardware (i.e., RDMA [50]) and shared memory between containers. Although FreeFlow showed great performance compared to the existing overlay networking, it requires NICs that support RDMA. To our knowledge, it is not clear when such NIC is available in IoT devices. Even when such NIC becomes available, because the containers share memory space to improve throughput, FreeFlow brings the security issue that may compromise isolation between containers. So, although FreeFlow has advantages over native container networking, it may not have "interoperability" for heterogeneous IoT devices that lack the hardware support for RDMA.

FALCON [23] addresses the issue of a bottleneck in container networking caused by SoftIRQ processing. They proposed a pipelined processing of a SoftIRQ using separate cores, resulting in improved network throughput. However, it should be noted that FALCON achieved this performance improvement by utilizing additional CPU cores. Since CPU cores are typically limited in IoT devices, FALCON may not be suitable for IoT environments and is more appropriate for resource-rich datacenters. Moreover, FALCON requires Mellanox ConnectX-5 NICs with advanced offload features, which are not supported by most IoT hardware.

Socket-grafting [19] proposed a new socket layer protocol called AF_GRAFT that bypasses the container networking stack by connecting the container and host socket layer. However, utilizing AF_GRAFT requires modification of the socket APIs of all legacy applications. This means that considerable development efforts are necessary to apply AF_GRAFT to various heterogeneous IoT applications.

Lastly, Slim [21] improved the network throughput by bypassing the container networking stack and using the host networking stack directly without any NAT or overlay. However, since most packets in Slim contain only host IP addresses, the host kernel cannot distinguish between packets originating from the host or containers. This makes network diagnosis challenging. In addition, Slim implemented custom APIs to reroute system calls to its framework for interoperability. However, legacy applications are still incompatible due to semantic differences in packet processing. Specifically, Slim lacks the complete support for the `accept` call, which is necessary for the MQTT library in IoT applications, resulting in segmentation faults.

In contrast, SCON improves CPU efficiency, network throughput, and latency for small IoT packets without changing any system calls or requiring additional hardware features. Consequently, SCON keeps the interoperability with legacy IoT devices and applications.

VIII. DISCUSSION AND FUTURE WORK

Scope of this study. In this paper, we focus on IoT devices that can run a standalone Linux kernel. Our evaluation is based on 1) micro-benchmarks that try to saturate the link (stress test) and 2) IoT application protocols with a range of message transmission intervals in order to cover the sporadic pattern as well as bursty one. For future work, we will characterize the details of IoT traffic patterns and evaluate SCON with the patterns.

Portability of SCON. The current version of SCON is implemented in the kernel. One might be curious about its portability. In the future, we plan to explore the option of re-implementing SCON using loadable modules to enhance its portability.

Timer interval and overhead of SCON flusher. SCON flushes the Conentry if it is not referenced within a specific timer interval (e.g., 10 s). First, to decide a reasonable timer interval, we investigate other papers that deal with IoT traffic characteristics [5], [66], [73], [74]. However, we observe that, to our knowledge, no study presents a specific duration that can determine the idleness or termination of IoT traffic. Thus, we determine the value empirically for our experiments. In the future, we plan to test various flush intervals by collecting IoT traffic traces from real-world IoT sensors (e.g., smart bulbs and smart cameras). Second, to identify the overhead of the SCON flusher, we measure the time taken to reallocate Conentry for newly-arrived packets after the Conentry is flushed. The time taken is 0.03 ms on average, and this delay can be hidden by pre-allocating Conentry structures in advance. We leave this optimization as future work.

SCON in changing communication environments. The communication environment can change over time, for example, from less congested to more congested. Then, congestion control or reliable delivery of transport protocols might kick in. However, *SCON express forward* can still work due to the following reasons. First, *SCON express forward* remembers only the decisions that occur in the IP and bridge layers, such as the addresses of the container and cloud server, the interface to deliver the packet, etc. Since changes in the environment, like network congestion, are dealt with in the upper layer (e.g., transport layer), SCON is not affected. Second, the addresses of the container and cloud server remain consistent over their connection regardless of environmental changes, so the Contable remains valid.

Comparison with existing studies. We investigate other existing methods as much as possible, as listed in Table 1. However, we find that they cannot be evaluated directly with SCON due to the following reasons. First, several studies [22], [20] require additional new hardware features (e.g., DPDK) that are not available on IoT devices, which makes them unsuitable for IoT device networking. Similarly, one study [23] relies on Mellanox ConnectX-5 NICs with advanced offload features, which most IoT hardware does not support. So, it is not feasible to make a direct comparison with SCON. Second, several studies [19], [21] require API changes for all legacy IoT applications. For example, Slim [21] implements its custom APIs to reroute system calls, and Socket-grafting [19] entirely replaces the socket layer and its APIs, making them not interoperable with the existing IoT applications. In our efforts for fair comparisons, we try to port Slim to run IoT applications but fail, in particular for the MQTT protocol (Mosquitto broker). The reason is Slim modifies the semantics of the accept system call so that it does not work with applications that assume the semantics. In short, although we try to compare SCON with other existing studies, they assume either special hardware support or their own API/semantics. A key design goal of SCON is to make it interoperable with the existing IoT applications yet to accelerate the networking performance of IoT devices.

Consideration on overlay. The current design of SCON is built with the default container networking on NAT that is the de-facto option for IoT devices [11]. However, one may wonder about the feasibility of using overlay techniques, which are commonly utilized in cloud datacenters. Overlays are rarely used in IoT due to their heavy overhead, so this study does not focus on them. However, given that various cloud orchestration platforms, such as OpenStack, maintain network connections through overlays, it might be useful to use overlays in IoT to further enable compatibility between IoT devices and cloud networks. So, we leave improving overlay overheads in resource-constrained IoT devices as the future work.

Extension to secure containers. The SCON design aims to enhance networking performance while minimizing resource consumption. However, it is well-known that containers are less isolated compared to other virtualization technologies, like virtual machines [75], [76]. Numerous studies have been conducted to improve the security of containers in common cloud datacenters [77], [76], [78]. However, to the best of our knowledge, little research has addressed both performance and security issues in IoT devices. In the future, we plan to extend SCON to incorporate both performance and security.

IX. Conclusion

This study presents SCON, a high-performance container networking accelerator for resource-restricted IoT devices. We identify the overheads in container networking of IoT devices. SCON intelligently memorizes decisions on packet processing without hardware dependency. Also, SCON maintains the native kernel and socket semantics, eliminating the need for re-compilation or re-verification of legacy applications.

We implement SCON on a recent version of the Linux kernel, and the evaluation results reveal that SCON effectively reduces CPU usage for networking processing (SoftIRQ) by 26% and improves network throughput by 18% for TCP packets. For UDP packets, SCON also reduces CPU usage for Softirq by 33% and improves network throughput by 32%. Moreover, SCON significantly reduces the latency of real-world applications like HTTP and MQTT, with $\sim$10$\times$ improvement compared to containers, reaching a level similar to the native process.

Appendix

Here, we explain a detailed background to help understand this study.

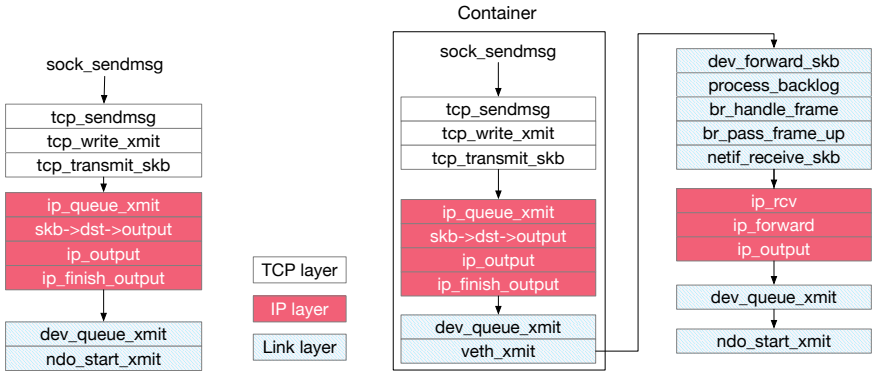
A. Container Virtualization

Virtualization creates multiple VM instances to run user applications on a single physical machine. While sharing the underlying hardware resources, such as CPU, memory, and storage, virtualization provides isolation between the multiple instances to avoid interference between them. Container virtualization has become widely used as it has much lower overheads than virtual machines. Unlike virtual machines that include the kernel in the virtualized instance, containers share the host server's kernel but isolate the application processes. Since containers are created above the host server, they introduce an additional routine to provide isolation. We explain the sequence of packet processing toward the network interface to deliver IoT traffic between containers.

B. Details in Packet Processing Sequence

We explain the necessary background on the packet processing sequence in §II. Here, we provide more detailed explanations.

1) Native Linux: In Linux, user applications create the data (payload) to send, and the kernel crafts packets with the data and handles network protocols. Fig. A.1a shows the packet processing sequences for TCP packets of the native Linux. Firstly, the packet from the user application passes through `sock_sendmsg`, the first function of the transport layer of the Linux networking stack. `tcp_sendmsg` then allocates memory spaces for the data structures of the metadata and



(a) Native Linux. (b) Container.

Fig. A.1. Detailed packet processing sequence comparison. Containers have a longer sequence than native Linux.

payload of the packet (e.g., `sk_buff`) and establishes a connection to the remote destination host.

`tcp_write_xmit` checks whether the packet needs to be fragmented before transmission. `tcp_transmit_skb` invokes `ip_queue_xmit` and `ip_output` for routing lookup and Netfilter processing. When the next-hop MAC address is identified (via `ip_finish_output` and `ip_finish_output2`), the packet is transferred to the link layer (`dev_queue_xmit`) and further to the corresponding network interface layer (`ndo_start_xmit`). Then, the packet is transmitted by the actual network device, such as a network interface card.

2) Container: Fig. A.1b summarizes how packets are processed when containers are used. The functions in the TCP and IP layers are the same as in native Linux. In the link layer, however, `ndo_start_xmit` calls `veth_xmit` that transmits packets to *veth* instead of the actual network device of the host. *veth* is a virtual network interface assigned to each container by default, which provides an isolated network address space. Each *veth* has unique MAC and IP addresses so the containers can be distinguished.

After passing through the *veth*, the packet is delivered to the bridge that offers connectivity. The role of the bridge is to identify the MAC address of the *veth* interfaces of the containers to forward packets to appropriate containers. First, the bridge performs an ARP (Address Resolution Protocol) table lookup to identify the MAC address of the packet. Next, the bridge passes packets to the Netfilter process to determine which *veth* interface a packet goes out on by `br_handle_frame` and `br_pass_frame_up` functions. Finally, `br_pass_frame_up` calls `netif_receive_skb` that leads to another IP processing.

After the operations from the bridge are finished, `netif_receive_skb` triggers the second IP layer processing to invoke `ip_rcv`, `ip_forward`, and `ip_output`. This involves routing lookups that determine the proper destination IP of packets and Netfilter operations that decide whether to forward packets are performed. In particular, NAT is conducted by the Netfilter hook, which is called by `ip_output` in the IP layer to change the source IP address of packets from the IP address of the container to that of the host. The container networking has to perform network address translation (NAT) because the IP address of the containers is unique only within a host, so they cannot be recognized on the external network. So, through NAT, container networking makes the IP address translatable to the external network. After NAT from the second IP layer, the packet is delivered to the actual network interface of the host server (`dev_queue_xmit`), and the packet is finally transmitted (`ndo_start_xmit`).

REFERENCES

[1] H. Tran-Dang, N. Krommenacker, P. Charpentier, and D.-S. Kim, "Toward the internet of things for physical internet: Perspectives and challenges," *IEEE internet of things journal*, vol. 7, no. 6, pp. 4711–4736, 2020.

[2] T. Chakraborty, H. Shi, Z. Kapetanovic, B. Priyantha, D. Vasisht, B. Vu, P. Pandit, P. Pillai, Y. Chabria, A. Nelson, M. Daum, and R. Chandra, "Whisper: IoT in the TV white space spectrum," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 401–418.

[3] J. Xu, H. Cao, Z. Yang, L. Shangguan, J. Zhang, X. He, and Y. Liu, "SwarmMap: Scaling up real-time collaborative visual SLAM at the edge," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 977–993.

[4] D. A. Chekired, L. Khoukhi, and H. T. Mouftah, "Industrial IoT data scheduling based on hierarchical fog computing: A key for enabling smart factory," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4590–4602, 2018.

[5] D. Yu, W. Li, H. Xu, and L. Zhang, "Low reliable and low latency communications for mission critical distributed industrial internet of things," *IEEE Communications Letters*, vol. 25, no. 1, pp. 313–317, 2020.

[6] L. Hobert, A. Festag, I. Llatser, L. Altomare, F. Visintainer, and A. Kovacs, "Enhancements of V2X communication in support of cooperative autonomous driving," *IEEE communications magazine*, vol. 53, no. 12, pp. 64–70, 2015.

[7] I. Bedhief, L. Foschini, P. Bellavista, M. Kassar, and T. Aguili, "Toward self-adaptive software defined fog networking architecture for IIoT and industry 4.0," in *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 2019, pp. 1–5.

[8] Y. Yoo, Z. Niu, C. Yoo, P. Cheng, and Y. Xiong, "SegaNet: An advanced IoT cloud gateway for performant and priority-oriented message delivery," in *Proceedings of the 7th Asia-Pacific Workshop on Networking*, 2023, pp. 54–60.

[9] R. Morabito, "Virtualization on Internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[10] W. A. Jabbar, C. W. Wei, N. A. A. M. Azmi, and N. A. Haironnazli, "An IoT raspberry Pi-based parking management system for smart campus," *Internet of Things*, vol. 14, p. 100387, 2021.

[11] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 35–50.

[12] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[13] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–37, 2019.

[14] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1270–1278.

[15] S. Kumar, M. P. Andersen, H.-S. Kim, and D. E. Culler, "Performant TCP for low-power wireless networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 911–932.

[16] D. Jung, Z. Zhang, and M. Winslett, "Vibration analysis for IoT enabled predictive maintenance," in *2017 ieee 33rd international conference on data engineering (icde)*. IEEE, 2017, pp. 1271–1282.

[17] P. Bahl, A. Adya, J. Padhye, and A. Wolman, "Reconsidering wireless systems with multiple radios," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 5, pp. 39–46, 2004.

[18] G. Chen, Y. Wang, H. Li, and W. Dong, "TinyNET: a lightweight, modular, and unified network architecture for the internet of things," in *Proceedings of the ACM SIGCOMM 2019 conference posters and demos*, 2019, pp. 9–11.

[19] R. Nakamura, Y. Sekiya, and H. Tazaki, "Grafting sockets for fast container networking," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, 2018, pp. 15–27.

[20] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "FreeFlow: Software-based virtual rdma networking for containerized clouds." in *NSDI*, 2019, pp. 113–126.

[21] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 331–344.

[22] "DPDK (data plane development kit." https://www.dpdk.org/, [Accessed: Mar. 05. 2023.].

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

[24] "DPDK - supported hardware." https://core.dpdk.org/supported/, [Accessed: Mar. 05. 2023.].

[25] W. Svensson, "An evaluation of how edge computing is enabling the opportunities for industry 4.0," 2020.

[26] M. Wallschläger, A. Gulenko, F. Schmidt, A. Acker, and O. Kao, "Anomaly detection for black box services in edge clouds using packet size distribution," in *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. IEEE, 2018, pp. 1–6.

[27] "Raspberry Pi for industry," https://www.raspberrypi.com/for-industry, [Accessed: Feb. 23. 2023.].

[28] T. Gizinski and X. Cao, "Design, implementation and performance of an edge computing prototype using Raspberry Pis," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2022, pp. 0592–0601.

[29] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[30] A. W. Services, "Explore AWS IoT core services in hands-on tutorial," https://docs.aws.amazon.com/iot/latest/developerguide/iot-gs-first-thing.html, [Accessed: Mar. 17. 2023.].

[31] Microsoft, "Azure IoT edge documentation," https://docs.microsoft.com/azure/iot-edge, [Accessed: Oct. 17. 2023.].

[32] C. Devnet, "Cisco edge device manager - application management," https://developer.cisco.com/docs/iotod/application-management, [Accessed: Oct. 23. 2023.].

[33] D. docs, "Docker - networking overview," https://docs.docker.com/network, [Accessed: Nov. 1. 2023.].

[34] C. LXD, "Run system containers with LXD," https://ubuntu.com/lxd, [Accessed: Nov. 1. 2023.].

[35] openVZ, "Open source container-based virtualization for Linux," https://openvz.org, [Accessed: Nov. 1. 2023.].

[36] L. foundation, "Open vSwitch - production quality, multilayer open virtual switch," https://www.openvswitch.org, [Accessed: Feb. 13. 2023.].

[37] AWS, "What is MQTT?" https://aws.amazon.com/what-is/mqtt/?nc1=h_ls, [Accessed: Nov. 11. 2023.].

[38] IBM, "MQTT messaging," https://www.ibm.com/docs/en/maximo-monitor/continuous-delivery?topic=concepts-mqtt-messaging, [Accessed: Sep. 2. 2023.].

[39] Microsoft, "Communicate with an IoT hub using the MQTT protocol," https://learn.microsoft.com/en-us/azure/iot/iot-mqtt-connect-to-iot-hub, [Accessed: Jun. 27. 2023.].

[40] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. D. Lara, W. Shi, and C. Stewart, "A survey on edge performance benchmarking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–33, 2021.

[41] "iPerf - the ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, [Accessed: Feb. 23. 2023.].

[42] H.-W. Cho and K. G. Shin, "BlueFi: Bluetooth over WiFi," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 475–487.

[43] A. Balasingam, K. Gopalakrishnan, R. Mittal, V. Arun, A. Saeed, M. Alizadeh, H. Balakrishnan, and H. Balakrishnan, "Throughput-fairness tradeoffs in mobility platforms," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 363–375.

[44] T. Lu, W. Xia, X. Zou, and Q. Xia, "Adaptively compressing IoT data on the resource-constrained edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[45] A. J. Pinheiro, J. d. M. Bezerra, C. A. Burgardt, and D. R. Campelo, "Identifying IoT devices and events based on packet length from encrypted traffic," *Computer Communications*, vol. 144, pp. 8–17, 2019.

[46] "mpstat," https://www.intel.com/content/www/us/en/robotics/real-time-systems.html, [Accessed: May. 16. 2023.].

[47] "Netperf," https://github.com/HewlettPackard/netperf, [Accessed: Mar. 05. 2023.].

[48] Intel, "Real-time systems overview and examples," https://man7.org/linux/man-pages/man1/mpstat.1.html, [Accessed: Jul. 12. 2024.].

[49] M. Fletcher, E. Paulz, D. Ridge, and A. J. Michaels, "Low-latency wireless network extension for industrial Internet of things," *Sensors*, vol. 24, no. 7, p. 2113, 2024.

[50] mellanox, "Rdma aware networks programming user manual," https://indico.cern.ch/event/218156/attachments/351725/490089/RDMA_Aware_Programming_user_manual.pdf, [Accessed: Mar. 05. 2023.].

[51] "Debugging the kernel using Ftrace - part 1," https://lwn.net/Articles/365835/, [Accessed: Jan. 07. 2023.].

[52] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi, "Scalable kernel TCP design and implementation for short-lived connections," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 339–352, 2016.

[53] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, pp. 11–20, 2020.

[54] L. Belli, "Big stream cloud architecture for the Internet of things," in *Proceedings of the 2015 on MobiSys PhD Forum*, 2015, pp. 5–6.

[55] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 15–20, 2011.

[56] "Nginx," https://www.nginx.com, [Accessed: Mar. 05. 2023.].

[57] "wrk2 - a HTTP benchmarking tool based mostly on wrk," https://github.com/giltene/wrk2, [Accessed: Mar. 05. 2023.].

[58] F. Tusa and S. Clayman, "The impact of encoding and transport for massive real-time iot data on edge resource consumption," *Journal of Grid Computing*, vol. 19, no. 3, p. 32, 2021.

[59] K. Waehner, "Apache kafka and mqtt – smart city and 5g," https://www.kai-waehner.de/blog/2021/03/29/apache-kafka-mqtt-part-5-of-5-smart-city-government-citizen-telco-5g, [Accessed: Mar. 18. 2023.].

[60] C. Gündoğan, P. Kietzmann, M. S. Lenders, H. Petersen, M. Frey, T. C. Schmidt, F. Shzu-Juraschek, and M. Wählisch, "The impact of networking protocols on massive m2m communication in the industrial iot," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4814–4828, 2021.

[61] "MQTT: The standard for IoT messaging," https://mqtt.org/, [Accessed: Mar. 05. 2023.].

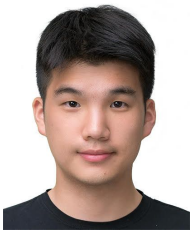
[62] "MQTT version 5.0," https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, [Accessed: Jul. 16. 2024.].

[63] E. Foundation, "Eclipse Mosquitto - an open source MQTT broker," https://mosquitto.org/, [Accessed: Mar. 05. 2023.].

[64] "MQTTLoader," https://github.com/dist-sys/mqttloader, [Accessed: Mar. 05. 2023.].

[65] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 537–549. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/bai
[66] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, 2018.
[67] W. Dong, B. Li, H. Li, H. Wu, K. Gong, W. Zhang, and Y. Gao, "LinkLab 2.0: A multi-tenant programmable IoT testbed for experimentation with edge-cloud integration," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1683–1699.
[68] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating performance of containerized IoT services for clustered devices at the network edge," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 1019–1030, 2017.
[69] "What is Azure IoT Edge," https://learn.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-1.5, [Accessed: Jul. 16. 2024.].
[70] "cgroups - Linux control groups," https://man7.org/linux/man-pages/man7/cgroups.7.html, [Accessed: Jun. 12. 2024.].
[71] "Asus PN42," https://www.asus.com, [Accessed: Jun. 22. 2024.].
[72] "Raspberry Pi," https://www.raspberrypi.org, [Accessed: Jun. 22. 2024.].
[73] H. Nguyen-An, T. Silverston, T. Yamazaki, and T. Miyoshi, "Generating iot traffic: A case study on anomaly detection," in *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*. IEEE, 2020, pp. 1–6.
[74] ——, "Iot traffic: Modeling and measurement experiments," *IoT*, vol. 2, no. 1, pp. 140–162, 2021.
[75] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE access*, vol. 7, pp. 52 976–52 996, 2019.
[76] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling MEC services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 209–214.
[77] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications." in *NSDI*, vol. 20, 2020, pp. 419–434.
[78] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The true cost of containing: A gVisor case study." in *HotCloud*, 2019.

Wonmi Choi received the B.S. degree in Computer Science from Korea University, Seoul, Republic of Korea, in 2021. She is currently pursuing her Ph.D. degree with Korea University, Seoul, Republic of Korea. Her research interests include container virtualization, container orchestration and kernel networking stack.



Yeonho Yoo [M] received his B.S. degree in computer science from Kookmin University, Seoul, Republic of Korea, in 2017, and his M.S. and Ph.D. degrees in computer science from Korea University, Seoul, Republic of Korea, in 2021 and 2024, respectively. He worked as a research intern at Microsoft Research Asia in 2023. He is currently a postdoctoral researcher at Korea University. His current research interests include network virtualization, SDN, datacenter systems, and AI systems.

Kyungwoon Lee received the B.E. degree from the School of Electronics Engineering, Kyungpook National University, Daegu, South Korea, and the M.S. and Ph.D. degrees in computer science from Korea University, Seoul, South Korea. From 2020 to 2022, she was with the Department of Computer science and Engineering as a research professor. She is currently working as an assistant professor in the School of Electronics Engineering, Kyungpook National University. Her research interests include resource scheduling in cloud computing, container and server virtualization, and TCP/IP kernel networking stack.

Zhixiong Niu is a Senior Researcher at Microsoft Research Asia. He received his Ph.D. from Department of Computer Science, City University of Hong Kong in 2019. Before that, he received his B.E. in Network Engineering at Dalian Maritime University (DMU) in 2012 and M.Sc. in Computer Science at the University of Hong Kong (HKU) in 2014. His research is primarily concentrated on systems and networking.

Peng Cheng received the Ph.D. degree in computer science and technology from Tsinghua University in 2015 and B.S. degrees in Software Engineering from Beihang University in 2010. He was a visiting Ph.D. student at UCLA from 2013 to 2014. He is a Senior Principal Research Manager at Microsoft Research Asia. His research interests are in the broad areas of systems and networking.

Yongqiang Xiong received the B.S, M.S. and Ph.D. degrees from Tsinghua University, Beijing, China, in 1996, 1998 and 2001, respectively, all in computer science. He is currently a Senior Principal Researcher and Research Manager with Microsoft Research Asia and leads the Networking Infrastructure Group. His research interests include system and networking, as well as network security.

Gyeongsik Yang [M] received his B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, Republic of Korea, in 2015, 2017, and 2019, respectively. He worked as a research intern at Microsoft Research Asia and as a research professor at Korea University. He is currently an assistant professor in the Department of Computer Science and Engineering at Korea University. His research interests include operating systems, AI systems, datacenter systems, network virtualization, and SDN.

Chuck Yoo [M] received the B.S. and M.S. degrees in electronic engineering from Seoul National University, and M.S. and Ph.D. degrees in computer science from the University of Michigan, Ann Arbor. He worked as a researcher at Sun Microsystems. Since 1995, he has been at the College of Informatics at Korea University, where he is currently a professor. His research interests include server/network virtualization and operating systems.