Low-Overhead Intra-Host Container Communication With Hardware Offloading

Qiang Su, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, *Senior Member, IEEE*, Dongsu Han, *Member, IEEE*, Chun Jason Xue, *Senior Member, IEEE*, and Hong Xu, *Senior Member, IEEE*

***Abstract*— Containers are widely embraced for their deployment and performance benefits over virtual machines. Yet, for many data-intensive applications in containerized clouds, bulky data transfers may impose performance issues. In particular, communication across co-located containers on the same host incurs large overheads in memory copy and the kernel's TCP stack. Existing solutions such as shared-memory networking and RDMA have their own limitations, including insufficient memory isolation and limited scalability. This paper presents PipeDevice, a new system for low overhead intra-host container communication. PipeDevice follows a hardware-software co-design approach — it offloads data forwarding entirely onto hardware, which accesses application data in hugepages on the host, thereby eliminating CPU overhead from memory copy and TCP processing. PipeDevice preserves memory isolation and scales well to connections, making it deployable in public clouds. Isolation is achieved by allocating dedicated memory to each connection from hugepages. To achieve high scalability, PipeDevice stores the connection states entirely in host DRAM and manages them in software. Evaluation with a prototype implementation on commodity FPGA shows that for delivering 80 Gbps across containers PipeDevice saves 63.2% CPU compared to kernel TCP stack, and 40.5% over FreeFlow. PipeDevice provides salient benefits to applications. For example, we port baidu-allreduce to PipeDevice and obtain ∼2.2× gains in allreduce throughput.**

***Index Terms*— Container communication, hardware-software co-design, network stack, offloading.**

I. Introduction

CONTAINERS have become prevalent in public clouds due to the performance, portability, and deployment benefits compared to virtual machines [2], [26]. They support a wide variety of workloads, from microservices and serverless computing to data analytics and machine learning. Containerized applications often entail extensive bulky data transfers to exchange intermediate results of data processing among peers. Examples include the shuffle stage in MapReduce jobs [30], [42] and the model update process with parameter server and allreduce in distributed machine learning [9], [37].

With these applications on the rise, it is increasingly common for bulky transfers to occur in the intra-host scenario, and we expect the trend to continue in the near future. For example, in constructing a so-called service mesh, a sidecar proxy container is deployed in each (micro-)service instance (with one or more containers) to route all traffic from this instance to other services [23], [24]. Multiple services may also co-locate on the same server and communicate through common network stacks like TCP. Spark and other data-intensive frameworks deploy the mappers and reducers in containers that may also co-locate at the same server to exploit locality, and they may communicate through network stacks such as TCP and RDMA [32]. Therefore, many cloud applications generate massive intra-host traffic. Further, cloud operators and container orchestrators often consolidate a tenant's containers onto as few servers as possible [36], [83] to improve efficiency. In addition, with the server machines becoming more resourceful in terms of CPU cores and memory [3], hundreds of containers can reside on the same server.

Extensive prior work exists on reducing the overhead of bulky transfers with TCP. They generally fall into two categories, software and hardware approaches. The software approach uses shared memory to reduce the memory copy overhead in the data path [60], [81]. Due to the memory isolation requirement in public clouds and the high memory overhead for a multitude of connections (§II-C), directly sharing memory between two containers is considered infeasible. A better way is to share memory between a container and the OS kernel [81]. The copy between user and kernel spaces is removed, and isolation is preserved since data is still copied across container boundaries (by OS kernel). The downside is that the copy still burns precious CPU cycles which could be used to support more application workloads.

Another approach to low-overhead intra-host container communication is the hardware-based RDMA. By offloading the entire network stack and memory copy to hardware, RDMA achieves high throughput and low latency with no CPU overhead. However, it is widely recognized that RDMA has

Received 6 November 2023; revised 31 July 2024 and 12 November 2024; accepted 15 December 2024; approved by IEEE TRANSACTIONS ON NETWORKING Editor S. K. Das. Date of publication 22 January 2025; date of current version 18 June 2025. This work was supported in part by the Research Grants Council of Hong Kong under Grant 11209520; in part by The Chinese University of Hong Kong (CUHK) under Grant 4937007, Grant 4937008, Grant 5501329, and Grant 5501517; and in part by the Gift Fund from Microsoft under Grant 6906276. A conference version [DOI: 10.1145/3555050.3569118] has appeared in ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), December, 2022. *(Corresponding authors: Zhixiong Niu; Hong Xu.)*

Qiang Su and Hong Xu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: hongxu@cuhk.edu.hk).

Zhixiong Niu, Ran Shu, Peng Cheng, and Yongqiang Xiong are with Microsoft Research Asia, Beijing 100080, China (e-mail: zhniu@microsoft.com).

Dongsu Han is with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea.

Chun Jason Xue is with the Department of Computer Science, Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi, United Arab Emirates.

Digital Object Identifier 10.1109/TON.2024.3520210

poor scalability [52], [53], [79], [80]. The root cause is the contention of limited on-board resources for connection state management [53], [80]. In fact, we show that the performance of commodity RDMA NICs (RNICs) drops by ∼50% with 4096 connections (§ II-D). As the number of containers and connections increases, cache miss becomes unavoidable and performance degrades. Although many prior works [52], [79], [79] try to improve scalability by balancing the tradeoff between efficiency and on-board state, they do not eliminate the main culprit. In addition, these solutions are for message-based RDMA semantics, and still suffer from inefficiency on common stream-based applications [16], [65], [76].

The fundamental question is, how can we build a customized transport for intra-host container communication, that achieves memory isolation, zero CPU overhead, and connection scalability simultaneously?

We propose a new hardware-software co-design approach to tackle this challenge. We rely on hardware offloading in order to achieve isolation and eliminate CPU overhead. For scalability, we keep connection states in host DRAM and manage them entirely in software so there is no contention of limited hardware resources. The cost is prolonged latency in connection management at the OS kernel now, which is negligible for bulky transfers.

Specifically, we build a new system called PipeDevice following this approach. PipeDevice exploits commodity hardware accelerators (*e.g.,* FPGA, SmartNIC) to forward data across co-located containers, effectively creating a device that facilitates a communication *pipe* for them. Each socket is allocated dedicated memory out of an memory region in the OS kernel, and application data in the socket buffer is directly accessed by the hardware's DMA engine and copied to the destination memory address. This eliminates the overheads of (1) copy between user and kernel spaces and (2) TCP stack processing. To ensure memory isolation, the memory regions are managed solely by the OS kernel who maintains the socket buffer states. The OS kernel also manages the connection states, so data copy is streamlined and performed in stateless manners by hardware. PipeDevice also features BSD socket-like APIs for memory and communication to port applications easily.

PipeDevice can be realized over different hardware other than FPGA, with a different hardware design to interact with the DMA engine. The software design of PipeDevice that manages connection and queue management remains largely the same, and by exposing a unified set of APIs applications do not need to change their code. Note PipeDevice aims to exploit hardware that has already been deployed in data centers, such as FPGA accelerators in Azure [41] and SoC SmartNICs [25], instead of requiring new devices. In this paper, we implement PipeDevice upon FPGA to illustrate its benefits.

We make the following contributions in this work:

- We present a comprehensive measurement study on the overhead of container communication for long TCP connections (§II-B), as well as the RNIC's scalability limitation in intra-host communication (§II-D).
- We build PipeDevice for low-overhead intra-host container communication (§III-§IV). Our design represents a general hardware-software co-design approach that addresses the fundamental scalability issue of hardware-only solutions like RDMA without performance penalty for bulky transfers. We implement a prototype using commodity Intel Arria 10 FPGA [18] on Linux kernel 4.9, and present tips of implementing PipeDevice to enforce such hardware-software co-design approach on various hardware. We show that our design is feasible using a very small amount of on-board resources (1.63% ALMs and 6.63% BRAMs).
- We conduct a comprehensive testbed evaluation for PipeDevice (§V). It saves 3.93 CPU cores compared to kernel TCP when delivering 80 Gbps throughput and scales to 400 connections without throughput degradation. As concrete usecases, we port baidu-allreduce [5] to PipeDevice with 113 lines of code change, and observe an end-to-end throughput gain of ∼2.2×. We also build a network service chain using PipeDevice's new APIs. Its end-to-end completion time of serving 10K requests is reduced by over 15% with 47% less CPU overhead compared to FreeFlow.

II. Motivation

We start by presenting the background of container communication and analyzing the overhead of today's intra-host container communication. We then discuss the limitations of existing solutions and present our design choice.

A. Container Communication

Containers are essentially processes with namespace isolation; their applications use standard network libraries such as BSD sockets to invoke the default TCP stack, as shown in Figure 1. Modern container frameworks, such as Docker, containerd, and Kubernetes, support multiple networking modes. Out of these, the `host` and `macvlan` modes are rarely used in clouds due to poor isolation and portability [56], [85]. So we consider the bridge and overlay modes, in which a software bridge (or vswitch) is used for overlay routing and control plane policy enforcement, as in Figure 1; each container interfaces with the bridge/vswitch through a unique pair of virtual NICs (vNIC, veth), and packets are sent to the network fabric via the bridge's physical NIC (pNIC).

A direct result of this architecture is the long data plane path, which is particularly expensive for intra-host communication. Data is copied across the user and kernel spaces at both the sender and receiver, and the TCP stack is also traversed twice [85]. This is greatly inefficient as transport failures (*e.g.,* message loss, re-ordering) and congestion rarely occur among intra-host peers, and shared-memory approach also demonstrates this promise [43], [60], [71]. Next, we quantitatively characterize these overheads.

B. Breakdown of Communication Overhead for Bulky Transfers

Bulky transfers are common in typical container workloads, such as data analytics and machine learning for intermediate

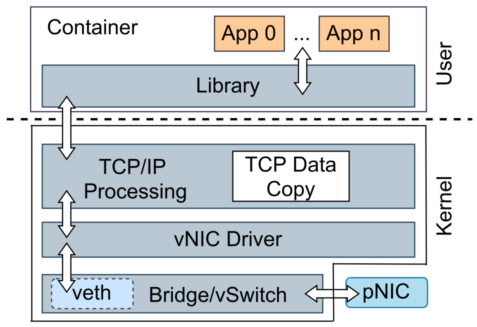


Fig. 1. Current container networking architecture in the bridge mode. TCP Data Copy denotes the data copy between the user and kernel spaces.

TABLE I

CPU Time Breakdown in Intra-Host Container Communication at the Sender Side. The Throughput Shown by Iperf Is 32.00 Gbps

Component	Major functions	Function time [s]	Component time [s]
TCP/IP	`tcp_sendmsg`	1.563	2.724 (27.2%)
	`tcp_write_xmit`	0.248	
	`tcp_transmit_skb`	0.120	
	`ip_queue_xmit`	0.074	
	`ip_local_out`	0.183	
	`ip_output`	0.056	
	`Others...`	0.480	
Bridge	`br_handle_frame`	0.062	1.576 (15.8%)
	`br_nf_pre_routing`	0.302	
	`__br_forward`	0.306	
	`br_nf_forward_ip`	0.182	
	`br_dev_queue_push_xmit`	0.498	
	`Others...`	0.226	
Memory copy	-	-	4.417 (44.2%)
Dev	`veth_xmit`	0.985	1.090 (10.9%)
	`__dev_queue_xmit`	0.088	
	`Others`	0.017	
Total	-	9.807 (10)	98.1%

data exchange, and service chaining in network function virtualization [50], [57], [82] for forwarding packets between network functions.

Therefore, unlike prior works that center around short connections in host networking [60], [63], [67] or overlay processing [85], we focus on the overheads of bulky transfers in intra-host container communication.

We measure the CPU time spent on the major functions of the data path of containers using `bpftrace` [6]. We launch iperf connections that last 10 seconds. The target container is given one core with other cores disabled, while the other container is given enough cores to ensure it is not the bottleneck. The servers use Intel Xeon E5-2698 v3 CPUs at 2.30 GHz. We do not consider overlay processing here because although overlay processing at the software router also incurs overhead, it can be offloaded to programmable hardware with high efficiency [44].

Tables I and II present the results for intra-host container communication in the bridge mode. The send and receive throughput is different because of the different overhead of TCP send path and receive path. The memory copy time is extracted from either `tcp_sendmsg()` or `tcp_recvmsg()` and shown separately. The total time is slightly less than 10s since we do not include the time spent on syscalls and user-space processing. We observe that for send, the dominant overhead is memory copy which takes

TABLE II

CPU Time Breakdown in Intra-Host Container Communication at the Receiver Side. The Throughput Shown by Iperf Is 37.66 Gbps

Component	Major functions	Function time [s]	Component time [s]
TCP/IP	`tcp_recvmsg`	0.225	2.237 (22.4%)
	`tcp_rcv_established`	0.723	
	`ip_rcv`	0.209	
	`ip_rcv_finish`	0.101	
	`ip_local_deliver_finish`	0.753	
	`Others...`	0.226	
Bridge	`br_handle_frame`	0.042	1.369 (13.7%)
	`br_nf_pre_routing`	0.252	
	`__br_forward`	0.041	
	`br_nf_forward_ip`	0.151	
	`br_dev_queue_push_xmit`	0.632	
	`Others...`	0.251	
Memory copy	-	-	4.779 (47.8%)
Dev	`veth_xmit`	0.999	1.101 (11.0%)
	`__dev_queue_xmit`	0.092	
	`Others`	0.010	
Total	-	9.486 (10)	94.9%

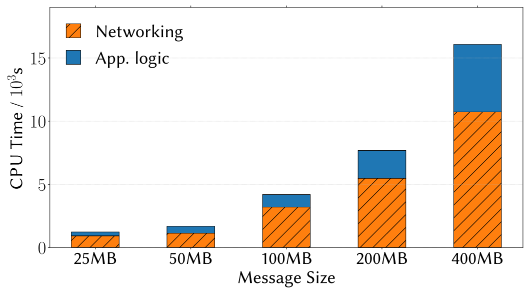


Fig. 2. CPU usage of MPI_Allreduce application.

44.2% CPU time, followed by the TCP/IP processing at 27.2%. The bridge consumes another 15.8%. Similarly, at the receiver, memory copy and TCP/IP processing together account for 70% of CPU time, as shown in Table II.

These overheads directly impact application performance. To demonstrate this, we profile allreduce, a communication primitive commonly used in distributed training for deep learning models [51], [72]. Allreduce allows workers to exchange the gradients obtained on their local batch of samples and then calculate the global average for model updates. We use the `MPI_Allreduce` implementation in Open MPI v4.1 [28] among eight single-core containers on the same server. We vary the message size and run 100 iterations for each data point. Figure 2 shows the CPU time breakdown. We count the in-kernel time spent by the program in the TCP stack as the time for communication, and the rest as the time for application logic. We can see that communication accounts for ~60% to 70% of total CPU time. This demonstrates that salient performance gains may be achieved by streamlining container communication.

C. Why Not Shared Memory?

Many systems have exploited shared-memory networking for intra-host communication [43], [60], [71]. Though it has low CPU overhead, it cannot guarantee memory isolation which is critical in public clouds [50], [84].

For example, SocksDirect [60] allocates a unique shared memory region between two co-located processes so both can directly access the data. Correspondingly, every pair of communicating containers necessitates a dedicated shared memory region [50], [84]. Furthermore, to ensure container resource isolation in public clouds, a separate memory region must be created for each container. This means that the shared-memory approach introduces additional memory overhead. For example, for a pair of single-core containers where each has 512 sockets (each socket ring buffer is 4 MB), a 2 GB memory chunk needs to be allocated. When communication occurs between any two containers simultaneously, the aggregate memory overhead will be high. Consequently, as the number of container pairs grows, the required shared memory regions also increase linearly, inevitably leading to high memory overhead.

Therefore, copying is necessary in a public cloud in order to guarantee isolation and avoid memory overhead. FreeFlow [56], [81], which also targets public clouds, is an example that corroborates this premise. In FreeFlow, each container has isolated memory regions, and shares them with a software router on the OS kernel. Although this removes overlay processing at the container's vNIC, the overheads of copying and going through the network stack remain for intra-host traffic as data is always processed by the host network stack. Note that FreeFlow's software router only undertakes the virtual networking and container memory management (Sec. III-C in [56]). This suggests the need for a hardware solution that offloads memory copy.

D. Why Not Commodity RDMA?

RDMA offloads the entire network stack onto the hardware RNIC, thus achieving high throughput and low latency without host CPU overhead. It has been shown that using it in virtualized clouds does not incur much performance penalty [48], [56]. In practice, the reliable connection (RC) transport mode is commonly used since it supports the more efficient one-sided operations with reliability.

In RC transport mode, commodity RNICs cache most connection states in the on-board SRAM for performance, including memory translation tables (MTTs), memory protection tables (MPTs), working queue elements (WQEs), and queue pair (QP) states. Each connection needs ~375B for QP states alone, while the expensive on-board cache is only a few megabytes [52], [80]. Hence commodity RNICs suffer from poor connection scalability as a result of cache contention, a phenomenon widely reported in the community [52], [53], [79], [80] based on Mellanox ConnectX-3 or ConnectX-4 RNICs. The same result is also observed in our experiment, *e.g.,* the RDMA WRITE throughput at 64 B and 1 KB drops by 51.97% and 80.38% for 400 connections on a Mellanox ConnectX-3 40 GbE RNIC (MT27520). Newer RNICs with larger on-board caches might suffer less from this problem. In spite of the existing works on inter-host networking [52], [53], [79], [80], it is still not clear whether scalability is satisfactory in the intra-host scenario. Thus, we conduct a benchmark on a Mellanox ConnectX-5 25 GbE RNIC (MT27800) and a Mellanox ConnectX-6 100 GbE RNIC (MT28908) using the same server as in §II-B. We exploit

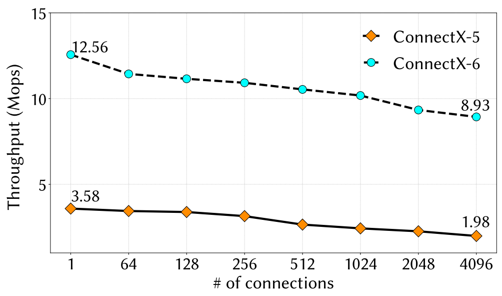


Fig. 3. Total throughput (operation-per-second, ops) of intra-host RDMA READ when the number of concurrent connections grows on Mellanox ConnectX-5 25 Gbps and ConnectX-6 100 Gbps RNICs.

`perftest` [29] and `rdma_bench` [53] to measure the throughput of 10-second flows with 1 MB-sized READ in operation-per-second (ops).

Figure 3 shows how the total throughput of RDMA READ varies as the connection number scales. We observe that READ throughput at 4096 connections of ConnectX-5 and ConnectX-6 declines by 44.69% and 28.90%, respectively. This proves that new RNICs also has the scalability issues. When the number concurrent connections grows, the RNIC has to frequently DMA the connection states from the host memory which is much more expensive than reading from the on-board cache.

To mitigate RDMA's scalability issue, prior studies [52], [79] strive to reduce the states offloaded to RNIC. However, they do not eliminate the limitation of the on-board resources. In addition, these solutions inherit the message-based semantics of RDMA, which confine the benefits for stream-based applications [45], [65], [76]. For example, the sending data stream has to be split into discrete RDMA messages, and applications have to perform other non-trivial operations, such as negotiating message sizes and initiating work queue requests.

E. Our Design Choice

We choose to only offload the performance-critical part of the data path onto hardware but rely on software to manage the connection states which are stored entirely in host DRAM. This eliminates the contention for hardware resources for the control information, while preserving the efficiency advantage of hardware offloading. The cost is prolonged latency of state management and other control path operations that happen in the OS kernel now, which is negligible for bulky transfers as will be shown in our evaluation.

As explained in §II-A, reliability and congestion issues would not happen in the intra-host case, so we choose to provide a simple communication service without these guarantees. This is more efficient in hardware resources than full-stack offloads like TOE [17] and RDMA. Specifically, we use hardware to DMA application data in hugepages and copy it to the receiving side, effectively building a communication pipe between containers without any CPU overhead. Our system is therefore dubbed PipeDevice. Because the only functionality on hardware is DMA copy, PipeDevice can be implemented upon various hardware, such as FPGA, Intel DSA [20], RNICs and SmartNICs. We currently choose FPGA for PipeDevice as it has already been massively deployed in data centers to accelerate networking workloads [41], [44]; using it for container communication does not incur extra hardware costs. We leave the implementation on other hardware for future work, and we do not focus on the raw performance given the hardware heterogeneity.

III. Design

A. Highlights and Overview

The central idea of PipeDevice is to offload memory copy to hardware and bypass the intricate TCP/IP stack processing. For this, it maintains an in-kernel memory region that can be accessed by the FPGA's DMA engine and mmaps (part of) this memory to each socket in user-space for application data. Specifically, PipeDevice imposes two fundamental design questions:

1) How to make existing applications benefit from PipeDevice without much porting effort and make it easy to develop new applications?
2) How to achieve efficient interaction between software and the underlying hardware?

PipeDevice has the following highlights to answer these.

1) New APIs for Communication and Memory: One of our goals is to minimize the porting effort for applications so that they can easily benefit from PipeDevice. It is ideal to directly support the BSD socket that has the stream-based semantics, but its APIs (*e.g.,* `recv()`) do not seamlessly support the zero-copy semantic [15], [33], [38].[1] New communication APIs are needed to enable zero-copy especially for receive since otherwise the receive call would use application's own buffer which PipeDevice cannot access. PipeDevice therefore chooses to provide a set of socket-like APIs. It also provides new APIs to create and free buffers in the in-kernel memory regions for applications.

2) Memory Management: Memory isolation and efficiency are required to make PipeDevice practical in public clouds. Thus PipeDevice organizes the in-kernel memory regions as a ring buffer pool and allocates to each socket dedicated ring buffers for both `send` and `recv`. These ring buffers are dynamically allocated and reclaimed for efficiency and scalability.

3) Decoupled Forwarding Plane: Scalability is one of the primary goals in PipeDevice. As explained in §II, saving connection states in hardware with limited resources constrains scalability. This is particularly relevant in our case as the same FPGA card may be simultaneously used to accelerate other workloads in the cloud [44]. Thus, PipeDevice adopts a decoupled design: the connection states, including the socket ring buffer states, are maintained entirely in the host kernel by a driver. The host kernel interacts with FPGA using sets of per-core command queues that are also maintained in host memory, and FPGA only keeps the states (pointers) of these queues, which is scalable to hundreds of cores. This reflects

[1]Zero-copy TCP receive is now supported in Linux kernel and exposed with the `mmap()` API for programming, but it breaks the socket `recv()` semantic and needs much porting effort for socket-based applications [33].

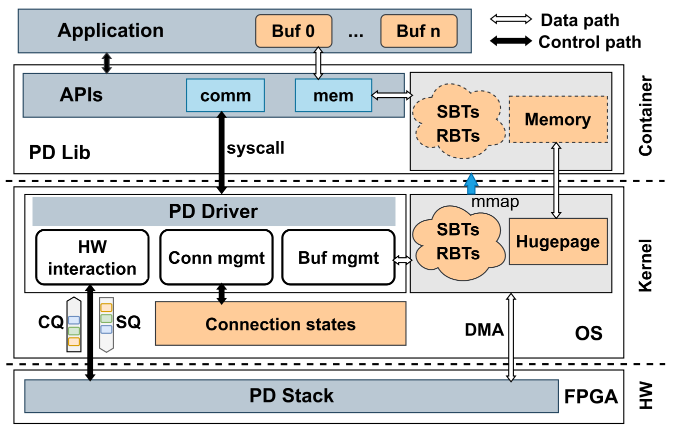


Fig. 4. PipeDevice system architecture.

our hardware-software co-design philosophy that works with simple hardware.

4) Kernel-Based Design: Containers follow a shared-kernel paradigm where the kernel isolates resources for different containers. This entails that container communication follows the same paradigm for isolation. Therefore, PipeDevice chooses a kernel-based design although it sacrifices possible latency benefits of hardware offloading that is in essence not crucial for bulky transfers.

5) Design Overview: Figure 4 depicts the architecture of PipeDevice. A PD Lib runs inside each container to serve the applications. It handles all communication requests and exposes to applications the send and receive buffers that are mapped from the in-kernel memory regions (§III-B). The API calls are directed to PD Driver which is a kernel module. PD Driver manages all the connection state information among other things. It allocates per-socket send and receive buffers on the in-kernel memory regions and tracks their states using a send buffer table (SBT) and a receive buffer table (RBT) (§III-C and §III-D). It interacts with PD Stack on FPGA by translating the API calls into fixed-sized commands and dispatches them to FPGA through a set of per-core command queues, each consisting of a submission queue (SQ) and a completion queue (CQ). Then PD Stack performs data transmission (§III-E).

B. Communication and Memory APIs

We first introduce the new APIs provided by PD Lib.

1) Communication APIs: PipeDevice exposes zero-copy BSD-like socket interfaces that largely follow the non-blocking semantics for stream-based data transfer. It has a corresponding function call for each socket function (*e.g.,* `socket()` becomes `pd_socket()`). Unlike the BSD counterpart, PipeDevice supports zero-copy in both send and receive. Specifically, `pd_send()` directly uses the mmaped buffers from the in-kernel memory regions; and `pd_recv()` simply returns a reference (pointer) to the socket's receive buffer (managed by PD Driver) which holds new data. To avoid buffer overlaps, PipeDevice introduces `pd_recv_done()` and `pd_buf_refresh()` for checking buffer states: After the application consumes the data, it calls `pd_recv_done()`

```
/* Server */
sid = pd_socket(AF_INET, SOCK_STREAM,
PROTOCOL);
ret = pd_bind(sid, &serv_addr, sizeof(
serv_addr));
ret = pd_listen(sid, BACKLOG);
acc_sid = pd_accept(sid, &cli_addr, &addr_len
);
recv_buf = NULL;
ret = pd_recv(acc_sid, DATA_LEN, &recv_buf);
ret = pd_recv_done(acc_sid, DATA_LEN);
pd_close(acc_sid);
pd_close(sid);

/* Client */
sid = pd_socket(AF_INET, SOCK_STREAM,
PROTOCOL);
ret = pd_connect(sid, &serv_addr, sizeof(
serv_addr));
send_buf = pd_malloc(DATA_LEN);
ret = pd_buf_refresh(sid, PTRS, PTRLEN);
ret = pd_send(sid, send_buf, DATA_LEN);

pd_free(send_buf);
pd_close(sid);
```

Listing. 1. An example PipeDevice application. The logic is the same with non-blocking BSD sockets.

to release the slots in the receive buffer (see §III-E). Similarly, `pd_buf_refresh()` checks if the send buffer is reusable before the application overwrites and sends it. The overheads of `pd_recv_done()` and `pd_buf_refresh()` are small since *syscall* is not the bottleneck for bulky transfers. These overheads can also be mitigated by batching, *e.g.*, multiple send buffers can be checked simultaneously by one `pd_buf_refresh()` call.

PipeDevice also provides an `epoll`-like event mechanism without any modifications to its event handling logic. `pd_epoll_wait()` and `pd_epoll_ctl()` are used for fetching and controlling the events (*e.g.*, `PD_EPOLLIN`).

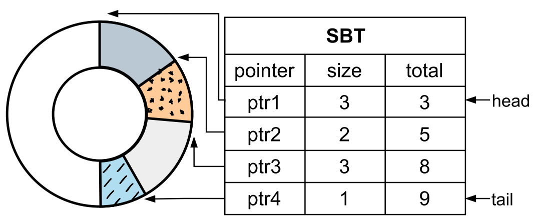
2) Memory API: PD Lib provides `pd_malloc()` and `pd_free()` for applications to dynamically create and release buffers in the in-kernel memory regions.

Listing 1 presents an example PipeDevice program with the new APIs. Note that care should be given to the zero-copy semantics of receive in PipeDevice when writing applications. PipeDevice also supports `pd_setsockopt()` and `pd_getsockopt()` for configuring socket parameters.

C. Memory Management

PD Driver in the kernel is in charge of managing memory for applications in order to achieve memory isolation.

1) Per-Socket Lock-Free Ring Buffers: PD Driver maintains the in-kernel memory regions which can be accessed by PD Stack through the DMA bus. The memory regions are organized as a ring buffer pool, and PD Driver allocates *per-socket* ring buffers to avoid locking overhead. Upon receiving the `pd_socket()` call, PD Driver applies for a send and a receive ring buffer from the pool. It also initializes a send buffer table (SBT) and a receive buffer table (RBT) to track the buffer usage. The subsequent `pd_malloc()` calls trigger PD Driver to allocate memory on the send ring buffer according to



SBT		
pointer	size	total
ptr1	3	3
ptr2	2	5
ptr3	3	8
ptr4	1	9

Fig. 5. A send ring buffer and its SBT with four allocated blocks. Once a new block is applied for, a new entry is inserted to the SBT: `pointer` denotes the header pointer of a block, `size` is the block size, and `total` means the total size of the used blocks in the ring buffer.

SBT. Figure 5 depicts a send ring buffer and its SBT. Note that the RBT for the receive ring buffer has the identical structure to SBT.

In addition, all the socket buffer states for sender and receiver are maintained by PD Driver. This ensures streamlined data transfers and zero state synchronization overhead. For example, when a `pd_send()` is called, it is able to directly find a free slot on the destination's receive buffer to send to based on the receiver's RBT.

D. Connection Management

This section describes the connection establishment and teardown processes undertaken by PD Driver.

1) Connection Establishment: PD Driver runs a kernel thread to maintain a *connection table* that contains the socket structures of endpoints and corresponding connection states. For each `pd_socket()` call, PD Driver allocates a socket structure (pointers to the buffers and SBT/RBT), sets the socket state to `ACTIVE`, and returns a distinct socket ID `sid`. As shown in Listing 1, a server application invokes `pd_bind()` after socket creation to bind the socket structure to an IP address and port, and the binding is maintained in PD Driver. When the server application is ready to accept connect requests, it calls `pd_listen()`; PD Driver changes the socket state to `LISTEN` and establishes a backlog queue which maintains a list of connect requests from clients. Once a client application invokes `pd_connect()` with the correct server address and port, PD Driver checks the `LISTEN` state and enqueues the request into the backlog of the socket.

Every time the server application calls `pd_accept()`, PD Driver dequeues a connect request from the backlog and creates a new socket structure for it. Then PD Driver inserts a new entry to the *connection table* with the socket structures, sets the connection state to `CONNECTED`, and returns the socket ID of the new socket to the server application.

2) Connection Teardown: When `pd_close()` is called on a socket, PD Driver releases both its own and peer's socket structures specified in the connection table, recycles their `sid`s, and deletes the corresponding connection table entry.

E. Data Transmission

On the FPGA, PD Stack undertakes data forwarding as instructed by commands from PD Driver.

1B	8B	8B	2B	2B	2B	1B	2B	6B
req type	src addr	dst addr	data len	cid	conn id	table id	entry seq	rsved

Fig. 6. The structure of a command queue entry. Here `req type` denotes the request type, *e.g.*, `PD_GENERAL_SEND`; `src addr` and `dst addr` are the source data buffer address and the destination address, and the data buffer length is `data len`; `cid` is the local container id at the host; `conn id` is the id of a connection between two endpoints, which is used for PD Driver to locate the connection so that it can get the buffer state tables of the connected endpoints; `table id` indexes either the SBT or RBT, and `entry seq` denotes a certain entry index of the SBT/RBT; `rsved` is for future extension.

1) Per-Core Command Queues: PD Driver interacts with the FPGA using per-core command queues. Though it is straightforward to establish per-connection or per-application command queues, the number of queues needed is not determinable compared to the per-core design because of the uncertain number of applications or connections. A shim layer is also required to translate the per-application or per-connection queue entries into the underlying hardware queues which are always based on cores. In addition to the inevitable locking overhead, the shim layer also needs to handle possible head-of-line blocking and out-of-order execution on the hardware queues for co-located connections or applications as a connection would use multiple hardware queues. Therefore, PD Driver chooses to use lockless per-core command queues.

Specifically, PD Driver creates a submission queue (SQ) and a completion queue (CQ) in the in-kernel memory regions for each CPU core, so that multi-core containers do not have locking overhead when accessing the command queues. The queues contain entries of a fixed size of 32 bytes. Figure 6 shows the structure of a queue entry. Note that the `cid`, `conn id`, `table id` and `entry seq` are only used by PD Driver for CQ processing, and they are off the hardware datapath. To access the entries, PD Stack maintains the queue states (*e.g.,* tail pointer) in FPGA. This is lightweight and makes PD Stack scalable to hundreds of command queues as the states consume a few MB of on-board memory.

Upon receiving an API call from the application, PD Driver translates the request into a new queue entry with the necessary information (e.g. destination address in the receive buffer for `pd_send()`) and inserts it to the SQ. PD Stack on FPGA polls each SQ, parses the new entry, and executes the command (*e.g.,* copies the data to the specified destination address by PD Driver for `pd_send()`). Then it inserts a new entry into the CQ and notifies PD Driver using an interrupt. PD Driver checks the CQ for completion notifications and performs necessary house-cleaning, such as releasing memory on the send buffer and updating the sender's SBT for `pd_send()`. To improve the interaction efficiency, batching is used when PD Driver and PD Stack fetch/insert entries from/to the command queues.

2) Data Transmission: Figure 7 illustrates the data transmission process between client container `C0` and server `C1`. The server (1) runs `pd_epoll_wait()` on socket `S1`, and PD Lib creates an epoll request to check if there is any ready data to receive (`PD_EPOLLIN` event). If not,

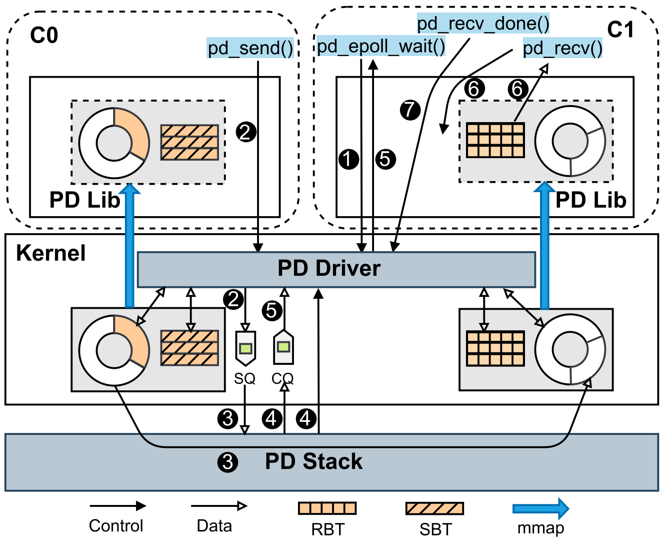


Fig. 7. An example of the data transmission process from `C0` to `C1`. The double-ended data arrows show that PD Driver maintains RBTs, SBTs, and data buffers. The ring buffers in `C0`'s and `C1`'s contexts are socket `S0`'s send buffer and `S1`'s receive buffer, respectively. Only relevant buffers and queues are shown here.

`pd_epoll_wait()` is suspended. At `C0`, when the client application invokes a `pd_send()` call, (2) PD Lib parses `pd_send()` and generates a send request to PD Driver. PD Driver determines the free slots in `S1`'s receive buffer according to its RBT, generates a new queue entry `E` with the destination memory address, and enqueues `E` to `C0`'s SQ. (3) PD Stack obtains `E` via the SQ. Then it copies data from `S0`'s send buffer to `S1`'s receive buffer according to the addresses carried in `E`. (4) PD Stack updates `E`'s `req type` to `PD_GENERAL_COMPLETED` and pushes `E` into `C0`'s CQ, and raises an interrupt to PD Driver. (5) In the interrupt context, PD Driver parses `E` from the CQ, releases memory in `S0`'s send buffer, and updates `S0`'s SBT and `S1`'s RBT. It also generates a `PD_EPOLLIN` event for `S1`, wakes up `pd_epoll_wait()` and copies the event to PD Lib to return to the server application. (6) Now the data is ready for the server application, which then calls `pd_recv()`. PD Lib returns the data chunk pointer obtained from `S1`'s RBT. (7) After the data is consumed by the server and `pd_recv_done()` is invoked, PD Driver receives the request from PD Lib, releases the corresponding memory in `S1`'s receive buffer, and updates `S1`'s RBT.

IV. Implementation

We implement a prototype of PipeDevice in Linux kernel 4.9 with 6000+ LoC. PD Lib is implemented as a dynamic shared library, and PD Driver as a kernel module. SQ and CQ are FIFO queues with a depth of 1K.

A. Hugepages

The OS kernel supports hugepages with large page sizes for better memory management efficiency (*e.g.,* mmap). We exploit hugepages for the in-kernel memory regions because they provide memory with visible physical addresses

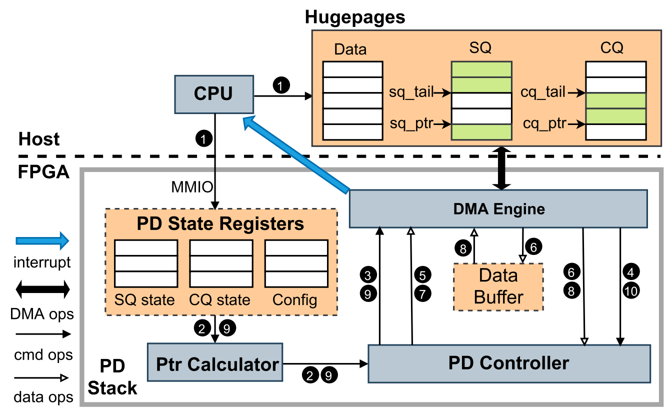


Fig. 8. The FPGA PD Stack implementation. `cmd ops` and `data ops` are for command queue and payload operations, respectively. `DMA ops` includes DMA write and read. PD Stack sets three registers to maintain the command queue states (e.g., pointers) and the FPGA configurations. The data queues in host hugepages represent send ring buffers and receive ring buffers.

which can be directly used by DMA engines and managed by PipeDevice. Without hugepages, the frequent transition from virtual and physical addresses will introduce high kernel overhead. Specifically, we use 4 GB memory in hugepages with 2 MB page size, and each ring buffer is 4 MB.

B. PD Stack

We implement PD Stack on Intel Arria 10 FPGA [18] (2×8 PCIe gen3 lanes) with 2500+ lines of Verilog. PD Stack DMAs data in the host hugepages and performs up to 4 KB reads and writes. Thus data is split into 4 KB chunks in PD Driver.

PD Stack exploits the FPGA's parallelism by running each component on an independent FPGA circuit and organizing the components in pipelines. To achieve isolated processing, each set of command queues is processed by a separate pipeline that is assigned with dedicated FPGA circuits. In addition, PD Stack processes the elements of a command queue in a FIFO order, and scans all command queues in a round-robin way. Figure 8 shows the data transmission process on PD stack, including four main steps below which run in parallel.

Read SQ: (1) The host CPU writes a command queue entry `E` to the SQ, then updates the corresponding SQ and CQ states (*e.g.,* `sq_ptr`) through MMIO; (2) Ptr Calculator computes the access offset of SQ and passes it to PD Controller; (3) PD Controller generates a DMA read request for `E`, then sends it to DMA engine; (4) DMA Engine reads `E` from SQ and sends it to PD Controller.

Read payload from send buffer: (5) PD Controller generates a DMA read request for fetching payload specified in `E` and sends it to DMA engine. (6) DMA Engine writes payload to Data Buffer, and notifies PD Controller with a read completion message.

Write payload to receive buffer: (7) PD Controller generates a DMA write request for payload and delivers it to DMA Engine; (8) DMA Engine gets the payload from Data Buffer and writes it to the receive buffer, then sends a write completion message to PD Controller. Now the payload is transmitted.

TABLE III
PipeDevice's Resource Usage on FPGA

Resource	PD Controller	DMA Engine	PD State Registers	Ptr Controller	Data Buffer	Total
ALM	4285.0	1954.2	614.2	94.2	0	6947.6
BRAM	21	0	1	0	104	180

Write CQ: (9) PD Controller updates `E`'s `req type` to `PD_GENERAL_COMPLETED`, gets CQ's free slots through Ptr Calculator similar to (2), and generates a DMA write request; (10) DMA Engine writes the updated `E` to CQ and generates a write completion message to PD Controller.

After above steps, PD Controller creates a new request and sends it to DMA Engine so that an MSI interrupt is invoked by DMA Engine to notify the CPU for kernel processing.

C. FPGA Resource

We quantify the FPGA on-board resource usage, including the use of ALM (Adaptive Logic Module), BRAM (Block RAM), and DSP (Digital Signal Processing) blocks. Specifically, PD Stack uses a very small fraction of FPGA resources: only 6947.6 (1.63%) ALMs, 180 (6.63%) BRAMs, and no DSP block. Table III presents the detailed on-board resource usage. PD State Registers are lightweight as they only maintain 2 pointers for each command queue (*e.g.,* `sq_tail` and `sq_ptr` for a SQ in Figure 8), making PD Stack scalable to hundreds of per-core command queues using only 1 BRAM (tens of MB).

D. How to Implement PipeDevice Over Other Hardware?

PipeDevice's PD Stack can be implemented on various hardware with DMA engines, including SmartNIC [25], Intel IOAT [11], *etc.* Similar to the FPGA implementation, the on-board resources are not the bottleneck. We provide some tips to enforce PD Stack on the SoC SmartNIC(*e.g.,* Mellanox BlueField-2 DPU [25]) and the Intel IOAT [11] (*e.g.,* Intel C610 series chipset [19]).

SoC SmartNIC. Unlike FPGA, SoC SmartNIC has multiple on-board CPU cores (*e.g.,* BlueField-2 DPU has 8 ARMv8 cores). Thus, Ptr Calculator, PD Controller, and DMA engine may run as processes on on-board CPU cores. SQ/CQ states and the data buffer are allocated from the on-board memory, which is a constant overhead.

Intel IOAT. Intel IOAT is integrated into the host CPU chipset and driven by the host CPU cores. PD Stack can be implemented by extending the IOAT copy interface with the command queue processing. Here the processes run on host CPU cores, and the memory is allocated from host memory.

V. Evaluation

We now present the evaluation of PipeDevice by answering the following questions.

1) How much resource saving does PipeDevice offer? (§V-A)
2) Does PipeDevice provide high throughput and good connection scalability? (§V-B and §V-C)

3) What about other microbenmarks, such as latency and fairness? (§V-D)
4) Is it easy to port applications to PipeDevice, and how much performance gain at the application level? (§V-E)

Methodology: The server we use has an Intel Xeon E5-2698 v3 CPU with 2 NUMA nodes each with 16 cores, 256 GB DRAM, an Intel Arria 10 FPGA, a Mellanox ConnectX-5 25 GbE NIC, and a Mellanox ConnectX-6 100 GbE NIC. We use Ubuntu 16.04 with Linux kernel 4.9. Hyper-threading and turbo boost are disabled.

We use Docker to create containers that are connected to a Linux bridge and uses the default kernel TCP stack. This is the baseline of our evaluation, denoted as Baseline. We compare PipeDevice against Intel IOAT [11] which is a memory copy engine on Intel CPU chipset, and software approaches including Cilium [7], UNIX domain socket (UDS) [35], io_uring [22], Slim [85], and state-of-the-art FreeFlow [56] which has both RDMA- and TCP-based implementations [12], denoted as FreeFlow RDMA and FreeFlow TCP, respectively. We mainly look at FreeFlow TCP considering that FreeFlow RDMA inherits RDMA's scalability issue in public clouds. We implement PipeDevice with IOAT by emulating the key logic of PD Lib and PD Driver on it, denoted as PipeDevice-IOAT. The performance of raw IOAT is obtained by Intel SPDK [21]. Unless otherwise stated, PipeDevice denotes its FPGA implementation, and uses 64 KB as the message size for DMA operations; flows in all schemes last 60 seconds to represent bulky transfers. Throughput is measured using `iperf` and `perftest`, and CPU usage is measured by `sysstat`. The results are averaged over five runs. To avoid startup interference, we run enough warm-up rounds before collecting the results.

A. Resource Savings

We look at PipeDevice's benefits on resource efficiency, including CPU and memory usage.

1) CPU Usage: We quantify PipeDevice's CPU savings over Baseline, Cilium, UDS, io_uring, Slim and FreeFlow TCP. The CPU usage is obtained as the total number of consumed CPU cores for both the sender and receiver. FreeFlow uses an extra core dedicated for the centralized controller (FreeFlow router). Baseline, Cilium, UDS, io_uring, Slim and FreeFlow TCP use 16 TCP streams to saturate cores, while PipeDevice uses a single stream.[2]

Figure 9 presents the CPU usage with varying throughput. We do not include the extra core used by FreeFlow; thus its result should be taken as an optimistic underestimation. We observe that PipeDevice significantly reduces the CPU usage while achieving the same throughput as other schemes. In achieving 80 Gbps, it saves 3.93, 1.56, 1.09, 1.19, 1.54 and 2.28 cores compared to Baseline, FreeFlow TCP, Cilium, Slim and io_uring. This confirms PipeDevice's benefit in removing the overheads of memory copy and TCP/IP stack processing. Among the compared approaches, Cilium and UDS bypasses the kernel TCP/IP stack, io_uring reduces the syscall overhead, and Slim and FreeFlow TCP mitigates the data path overhead

[2]More steams deliver the same total send throughput in PipeDevice.

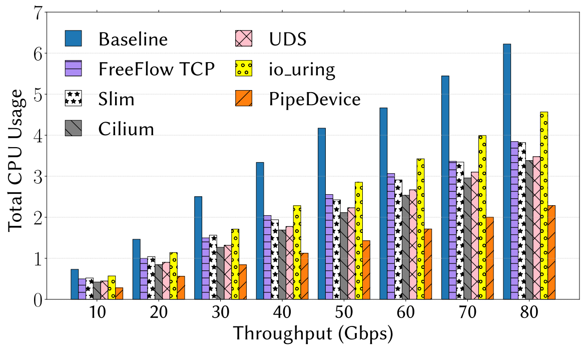


Fig. 9. Total CPU usage for throughput targets (Not count the core for FreeFlow router).

on overlay processing, so they all save CPU over Baseline. Our current prototype relies on PCIe gen3 lanes, and we expect PipeDevice to achieve greater CPU saving with newer hardware (more in §VI).

2) Memory Consumption: PipeDevice dynamically manages memory for containers (§III-C). On our server PipeDevice can easily support 32 single-core containers using 2 M hugepages in total (4 GB memory), and each container has 512 active sockets (each socket ring buffer is4 MB). On the contrary, as explained in §II-C, 32 GB memory is necessary for 16 pairs of containers, *i.e.,* 8 $\times$ the footprint of PipeDevice.

B. Throughput

We now show how much throughput PipeDevice can reach by measuring the send throughput with a pair of 2-core containers, based on our FPGA implementation and IOAT emulation, denoted as PipeDevice-FPGA and PipeDevice-IOAT, respectively.

We do not evaluate receive throughput as PipeDevice consumes little CPU for receive. Figure 10 shows its overall throughput and the raw throughput delivered by PD Stack and IOAT without interaction with the host. For PipeDevice-FPGA, we observe that when the message size is small (<4 KB), PipeDevice's throughput increases as the message size grows, but it is much lower than PD Stack's raw throughput. This is because the kernel processing on CPU cores is the bottleneck for small messages. When messages are larger than 4 KB, PipeDevice's throughput saturates at ∼85 Gbps and is very close to PD Stack's throughput. Now PD Stack becomes the bottleneck: since it supports up to 4 KB operations (recall §IV), its raw throughput saturates with 4 KB messages. This demonstrates that our co-design approach with PD Driver and PD Lib does not incur much performance loss for bulky data transfer. Further, considering that 2 $\times$ 8 PCIe lanes promise ∼110 Gbps theoretical bandwidth in gen3 [70], there is room to optimize the PD Stack implementation on FPGA for better performance (more in §VI).

The throughput of raw IOAT and PipeDevice-IOAT starts to drop when message size is larger than 16 KB,[3] and PipeDevice-IOAT may achieve ∼62 Gbps at 16 KB. We can

[3]The reason for the throughput drop may be that IOAT splits messages into chunks with a fixed size and the increasing per-chunk hardware processing becomes the bottleneck with large message sizes.

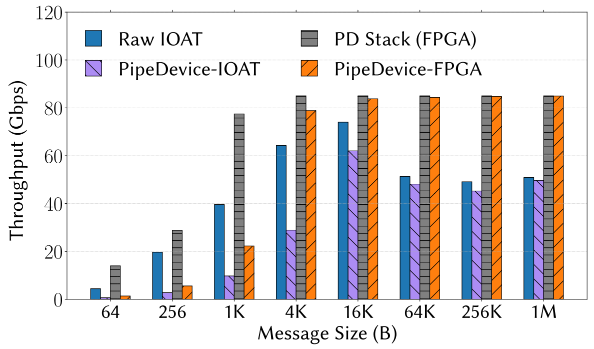


Fig. 10. Comparison of PipeDevice raw throughput with varying message sizes.

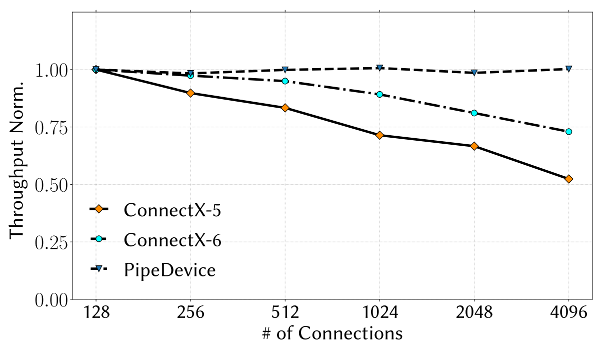


Fig. 12. Comparison of the connection scalability. The throughput is normalized by that of 128 connections.

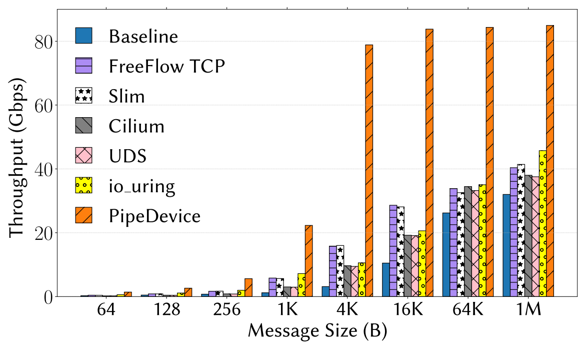


Fig. 11. The throughput comparison with varing message sizes.

also see PipeDevice's software overhead by comparing PipeDevice-IOAT and raw IOAT. Note that PipeDevice is also able to split the messages into 16 KB chunks to achieve ∼62 Gbps for larger messages (recall §IV). In addition, PipeDevice may achieve better throughput by implementing PD Stack on new hardware with better DMA bandwidth, *e.g.,* FPGAs with PCIe gen4/gen5. In the following, we mainly evaluate PipeDevice by its FPGA implementation.

We also compare PipeDevice's throughput against the software approaches by measuring the send throughput on a single-core sender container. Observe from Figure 11 that PipeDevice achieves better throughput than Baseline, FreeFlow TCP, Cilium, UDS, Slim and io_uring. When the message size is small, PipeDevice's benefits come from its simpler transport. As the message size becomes larger, memory copy between kernel and user spaces becomes the bottleneck and is effectively mitigated by PipeDevice via hardware offloading. Additionally, FreeFlow TCP and Slim exhibit comparable throughput as they both mitigates the overlay processing overhead. Similarly, Cilium and UDS achieve similar throughput by circumventing the kernel TCP/IP stack.

C. Connection Scalability

We run a pair of 16-core containers and measure the overall throughput when the number of connections scales. The DMA message size is 1 KB. We compare it to the READ throughput of the Mellanox ConnectX-5 25 GbE RNIC and ConnectX-6 100 GbE RNICs, and normalize the result by that of 128 connections. Figure 12 shows the normalized results. We can see that as the number of connections increases to 4096, PipeDevice's throughput remains stable at peak while RDMA's decreases sharply due to cache contention on RNIC as discussed in §II-D.

D. Latency and Fairness

1) Latency: We examine PipeDevice's packet processing latency using the completion time of sending very short messages based on its FPGA implementation and IOAT emulation, denoted as PipeDevice-FPGA and PipeDevice-IOAT, respectively. We measure the latencies using different message sizes and repeat for 5000 runs. Figures 13 and 14 show the average and 99%-ile tail latencies, respectively. In Figure 13, we observe that PipeDevice-FPGA and PipeDevice-IOAT achieves lower average latency than Baseline, Slim and FreeFlow TCP in general. This is because PipeDevice provides a simpler transport than TCP. Additionally, the latencies of PipeDevice-FPGA and PipeDevice-IOAT increases less when the message size increases. This is because copy overhead between user and kernel spaces in Baseline and FreeFlow TCP is non-negligible as message size grows. When the message size is small ($<$ 512 B), we can see that PipeDevice-FPGA is worse than Cilium, UDS, and io_uring, the reason is that PipeDevice-FPGA introduces extra delay on PCIe transmission and kernel processing (*e.g.,* connection management). As the message size increases, io_using has higher average latency than PipeDevice because the latency on memory copy starts to becomes the bottleneck. In addition, PipeDevice-IOAT has lower latencies that PipeDevice-FPGA. This also confirms the incurred delay on PCIe transmission in PipeDevice-FPGA while IOAT is integrated on the CPU chipset. PipeDevice has higher average latency than RDMA which is not surprising. PipeDevice relies on the OS kernel to carry out control actions (memory management, connection state management) and thus bears the context switching and syscall overheads (recall §III-A), while RDMA offloads the entire stack to hardware and enjoys hardware-level latency. We also observe high tail latencies of PipeDevice from Figure 14, and the latencies is not stable compared to FreeFlow RDMA. This confirms PipeDevice's software overhead on latencies (*e.g.,* connection management). The latency is a small price we consciously choose to pay for a more practical design for bulky transfer.

2) Fairness: We evaluate PipeDevice's fairness in terms of per-core bandwidth sharing. Specifically, we run 16 pairs of

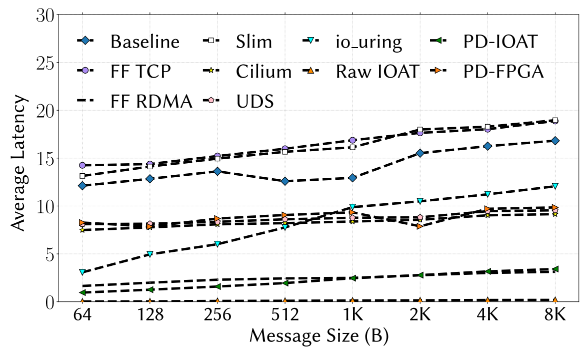


Fig. 13. Comparison of average latency (μs) as the message size increases. PD-IOAT and PD-FPGA are PipeDevice's two implementations.

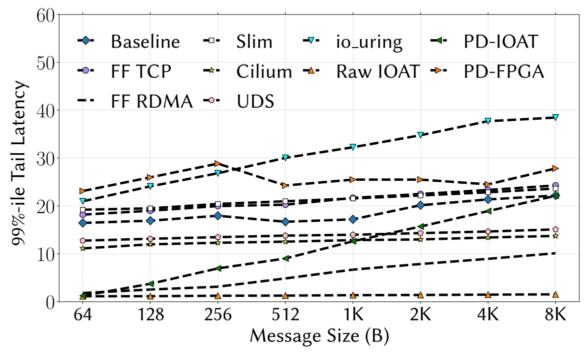


Fig. 14. Comparison of tail latency (μs) as the message size increases. PD-IOAT and PD-FPGA are PipeDevice's two implementations.

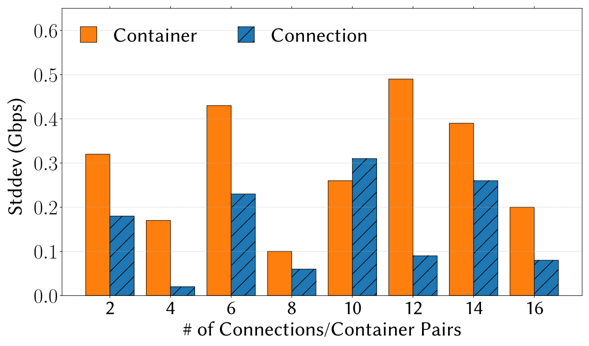


Fig. 15. Stddev of throughput with varying numbers of container pairs or connections.

single-core containers and measure the bandwidth of each pair. We also establish two 16-core containers, vary the number of connections between them, and measure the bandwidth obtained by each connection. These two measurements essentially set up one connection for each CPU core. We calculate the standard deviation for the shared throughput and show the results in Figure 15. Observe that the standard deviation for PipeDevice stands at a low value (< 0.5 Gbps). This is because PipeDevice interacts with FPGA through per-core command queues, which ensures fair sharing of FPGA's performance across multiple cores.

E. Application Usecases

We quantify the application-level performance gain of PipeDevice.

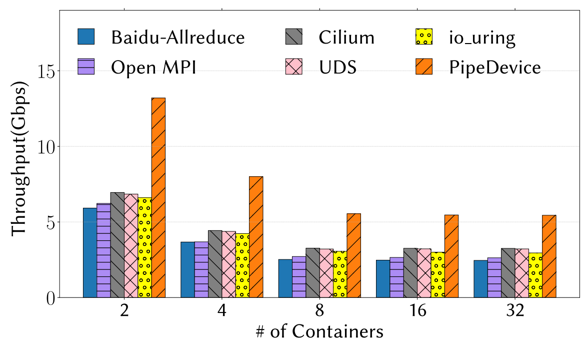


Fig. 16. Comparison of allreduce's end-to-end throughput.

1) Ring Allreduce: Allreduce is widely used in distributed training of deep learning models to aggregate gradients from different workers [13], [27]. We implement a ring allreduce library using PipeDevice by porting baidu-allreduce [5]. Specifically, we replace the MPI calls (*e.g.,* `MPI_Send`) in baidu-allreduce with PipeDevice's APIs. Out of ~650 lines of C++ code, 26 lines are modified and 87 lines are added to use PipeDevice. Note that the reduce operation is done on CPU. We also port baidu-allreduce using Cilium, UDS, and io_uring for comparison, by adding 199 and 227 and modifing 48 lines, respectively, and also compare to `MPI_Allreduce` in Open MPI v4.1 [28]. Each container is assigned one core and works as a rank of allreduce. We report the end-to-end throughput of processing 200 MB data with 1 MB message size at each container.

Figure 16 shows the results averaged over 100 runs. PipeDevice significantly improves the allreduce throughput by ~2.21×, ~2.14×, ~1.92×, ~1.97× and ~2.05× over vanilla baidu-allreduce, Open MPI, Cilium, UDS, and io_uring versions. Considering that communication takes ~60% CPU cycles for allreduce (recall §II-B), this proves the performance benefits by using the saved CPU cycles to accelerate the reduce operation. We also observe that the Cilium, UDS, and io_uring's performace gains are smaller than PipeDevice. This is because they still suffer from the memory copy overhead at 1 MB although Cilium and UDS bypass the TCP/IP stack processing and io_uring reduces the syscall overhead.

We integrate all ported baidu-allreduce versions into Gloo [13] for PyTorch 2.3.0 to examine PipeDevice's benefits for real machine learning models. Gloo is the default for CPU training in PyTorch [4]. We modify only the allreduce API invocation in Gloo to use our baidu-allreduce versions. Gloo's interfaces for PyTorch are unchanged. Specifically, we use train two typical models, VGG11 [75] and ResNet52 [47], on 32 single-core containers on the same server, with each container as a rank.[4] The dataset is from CIFAR-10 [34]. We set the batch size to 32 to saturate all cores, and measure the per-iteration latency averaged from the first 100 training iterations.

Figure 17 presents the time spent on allreduce and computation per iteration, denoted by the bottom and top sub-bars, respectively. The computation includes the forward,

[4]We choose these models as they are cost-efficient with CPU training compared to GPU [14], [61].

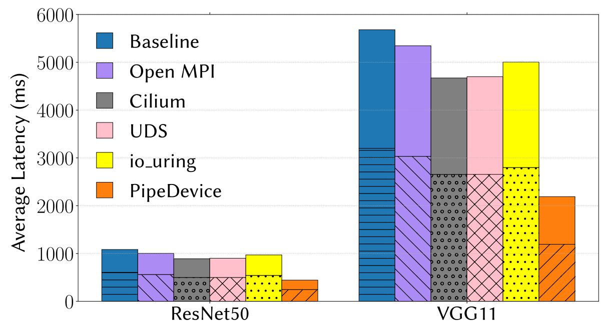


Fig. 17. Comparison of two ML models' per-iteration average processing time.

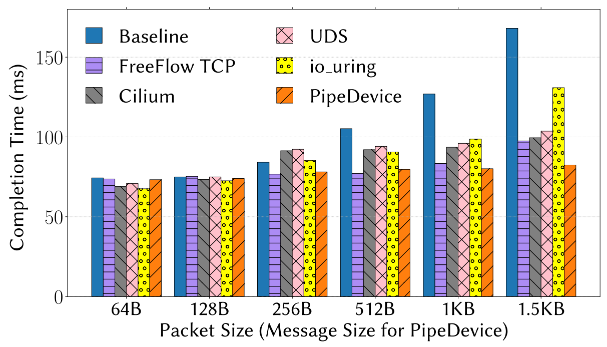


Fig. 18. Comparison of the completion time of the network service chain.

backward, and optimizer steps. We can observe that models using PipeDevice exhibit the shortest per-iteration latency. For VGG11, PipeDevice reduces the per-iteration latency by 61.45%, 59.03%, 53.13%, 53.51%, 56.24% compared to vanilla baidu-allreduce, Open MPI, Cilium, UDS, and io_uring versions, respectively. For ResNet52, it cuts latency from 1084 ms to 444 ms compared to vanilla baidu-allreduce, a 59.04% reduction. According to the time breakdown, we infer this benefits is due to (1) allreduce performance improvement with PipeDevice, and (2) saved CPU cycles from hardware offloading being delivered to the computation steps.

2) Network Service Chain: Virtual network functions are increasingly deployed using containers in practice for improved performance and efficiency [8], [82], and we use it as another usecase here. We implement a typical network service chain consisting of a firewall followed by a load balancer to determine the worker node, and finally a decryption NF to decrypt the request using AES. The ingress traffic is delivered from a 40 GbE NIC. We run 3 single-core containers for each NF, which sets up two threads, one for data transmission and the other for NF processing logic. We implement the baseline chain with 1526 lines of C, and the PipeDevice version only needs 82 lines of code change. In addition, we port the network service chain using Cilium, UDS, and io_uring with adding 199 and 242 and modifing 184 lines, respectively. We measure the completion time and total CPU usage of servicing 10K requests. The results are averaged over 100 runs.

Figure 18 presents the completion time with varying message sizes. When the packet size is smaller than 512B, PipeDevice does not improve performance much, and shows similar results with Cilium, UDS, and io_uring. This is

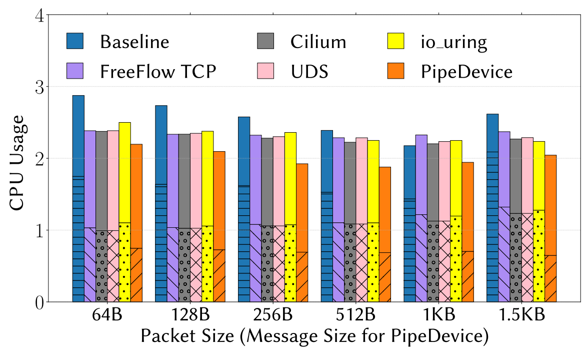


Fig. 19. Comparison of the overall CPU usage of the network service chain. For each bar, the bottom and top parts show the communication and computation cpu usage, respectively.

because the bottleneck is NF packet processing. PipeDevice's small gain is due to its simpler transport. However, as packet size increases, the bottleneck shifts to data copy, and PipeDevice's gain becomes more salient as it offloads copy to FPGA. The performance gains of Cilium, UDS, and io_uring also increases in this case. Figure 19 shows the CPU usage breakdown. We can see the largest gain is at 1.5 KB packets for which PipeDevice uses ~51% less CPU than FreeFlow TCP, which consumes 3.37 cores for 97.51 ms completion time, while PipeDevice only takes 2.04 cores for 82.42 ms (15.57% reduction).

VI. Discussion

We discuss some immediate concerns about PipeDevice here.

Can PipeDevice support overlay networking? To enhance existing container with PipeDevice's design, overlay networking should be integrated to capture network dynamics such as changes in routing rules and IP address mapping. To achieve this, a software overlay router can be used for necessary overlay processing, which is similar to the FreeFlow router in FreeFlow [56] and Slim's SlimRouter [85], whose policies can be configured by the network control plane. Specifically, this overlay router can be implemented in PD Lib to intercept API calls and apply corresponding policies. This certainly adds overheads which are also unavoidable in the current architecture. In case overlay processing is offloaded to hardware (*e.g.,* FPGA) as some providers have already deployed [44], PipeDevice can integrate with the offloading logic on FPGA. Essentially, overlay processing focuses on enforcing control plane policies, which does not impact the data path that PipeDevice focuses on.

Are there reliability and congestion issues? Reliability and congestion issues in the inter-host case are common because traffic goes through many switches, and their packet buffers may be full and unavailable for new packets. However, all packets are buffered locally in the intra-host scenario without reliability problems. Congestion may happen when the receive buffer is full in PipeDevice, which is the same as the BSD sockets that always return a buffer unavailability flag to applications. The usage of UNIX domain socket in Kubernetes for intra-host container communication also confirms this rationale [35]. Additionally, typical shared-memory approaches such as SocksDirect [60] take the same approach of not considering congestion and provide only a simple transport for intra-host communication.

How to preserve container location transparency for tenants? In container service frameworks, the orchestrator (*e.g.,* Kubernetes) takes charge of managing both intra- and inter-host communication, while the application programs can only be crafted using PipeDevice's APIs. For example, once containers are deployed, the orchestrator can maintain a container address map in each container, which is then utilized by the overlay router in PD Lib (as discussed before) to determine whether the destination application container is on the same server, if so, PD Driver sends data directly to the co-located application container; otherwise, it calls PD Stack to send data to the co-located sidecar proxy container, which then conducts inter-host communication through the TCP/IP stack. Therefore, with the container orchestrator, PipeDevice can maintain the transparency of container locations to tenants.

Does PipeDevice support various network policies? The current design of PipeDevice can support access control and rate limiting policies as all network requests are manipulated in PD Driver before being forwarded to FPGA. In addition to pure offloading, PipeDevice lends itself to implementing a set of network policies to operators. Access control policies could be parsed and injected to PD Driver directly, which inspect the metadata in each request and only process the authorized flows. Rate limiting policies could also be enforced by adjusting the enqueueing behaviors of PD Driver and the polling behaviors of FPGA (with regards to SQ). Better yet, one could consider integrating PipeDevice with the overlay processing in FPGA [44] to facilitate the control plane policy enforcement as discussed before.

Can we acheive higher throughput? As explained before in §V-B, PipeDevice currently delivers ∼85 Gbps bandwidth as a result of our implementation overhead and the PCIe limitation of our FPGA. As PCIe gen4 becomes mature with higher lane bandwidth, soon we expect to see FPGAs (2 × 8 PCIe gen4 lanes) offering ∼220 Gbps theoretical bandwidth [70] and beyond which is sufficient in public clouds.

Does PipeDevice introduce additional offloading cost? PipeDevice leverages existing data center hardware, like FPGA [41] in Azure, for efficient intra-host container communication without the need for new device deployments. As detailed in §IV, PD Stack primarily involves hardware DMA copy operations, with a one-time programming overhead. Therefore, the hardware offloading of PipeDevice doesn't involve high implementation hurdles.

Future work: We are considering several directions for future work. First, PipeDevice is designed for applications with long connections, like data analytics and ML training, and achieves better performace than traditional TCP/IP stack at all packet sizes due to simpler kernel processing and hardware-based data transmission (recall §V-B). However, for other applications involving extensive short connections, such as nginx, the dominant overhead becomes frequent connection setups and releases, which is beyond PipeDevice's scope and current design. Therefore, expanding PipeDevice for short-connection applications is future work. This will involve analyzing the overhead and behaviors of short connections systematically, like TCB contentions, and optimizing performance and connection scalability through hardware-software co-design. Second, we seek to eliminate the semantic gap between PipeDevice's zero-copy receive and the BSD receive which needs to be taken care of by applications now (recall §III-A). Third, charging policies should be designed to promote fair utilization of PD Stack. For example, tenants may be charged according to how many data transmission requests are processed. Fourth, with the growing demand for machine learning workloads, leveraging spare GPU resources for PD Stack offloading is an interesting prospect. For instance, exploiting CUDA kernels for DMA access on host hugepages and harnessing GPU parallelism can potentially enhance container communication performance. Fifth, expanding beyond a single hardware configuration, exploiting multiple hardware concurrently to achieve higher aggregate throughput is also worth exploring. This can involve assigning different hardware components with distinct command queues. Lastly, given that PipeDevice integrates memory and connection management within the OS kernel, PipeDevice may preserve the same security domain with the kernel. Therefore, there is potential for incorporating existing security policies [68] into PipeDevice.

VII. Related Work

We discuss related work to PipeDevice in this section.

A. Container Networking

Containerization in clouds drives innovation in container networking. Other than [56], [81] discussed in details already, there are a few efforts here. Iron [55] improves performance isolation by accurately counting CPU cycles spent on the RX path and integrating this to the Linux scheduler. Slim [85] and Falcon [59] aim to reduce the overhead of overlay processing: Slim bypasses the virtual bridge and virtual network device in overlay networks to achieve near-native performance; Falcon provides a fine-grained low-cost flow parallelization using softirq pipelining. BASTION [69] offers a container-aware communication sandbox for securing container networking. They do not consider the overheads of bulky transfer in the intra-host scenario and do not exploit hardware to reduce these overheads.

B. SR-IOV

SR-IOV compliant hardware [62], [64] efficiently shares PCIe devices across VMs. The host connects to a physical function (PF) of the device while each VM connects to a distinct virtual function (VF) which represents a virtual device. PipeDevice is orthogonal to SR-IOV because it removes the overhead of network stacks SR-IOV does not; while with or without SR-IOV applications in a VM still need to use the network stack. Thus, SR-IOV support and its newer scalable design such as Intel's Scalable VT-d [1] for the offloading hardware (*e.g.,* FPGA, RNIC) can facilitate PipeDevice's deployment inside VMs.

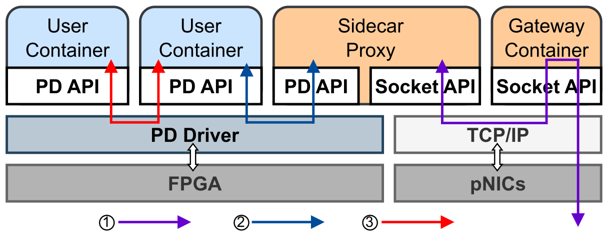


Fig. 20. Integrating with inter-host communication.

C. Host Network Stacks

High-performance network stacks are also beneficial to container networking, including kernel optimizations [40], [46], [63], [77] and user-space stacks [10], [31], [39], [49], [66]. These works mainly aim to optimize short-message communication and intra-host communication can do away with complicated stack processing. Meanwhile, user-space stacks are also not practical to deploy in public clouds for security concerns, and most packet I/O engines do not support virtualization across multiple tenants.

D. Hardware Offloading

Our community has also devoted efforts to hardware-assisted low overhead networking. Other than RDMA [52], [53], [80] discussed in §II-D, much work offloads various network workloads onto SmartNICs [17], [44], [54], [58], [67], [73], [74], including TCP connection processing, stateful operations of short connections, SDN policies, and the RPC stack. Though they do not consider long connections in the container context, they do inspire PipeDevice's design: offloading greatly helps to reduce CPU overhead by moving the expensive components of the current architecture away from general-purpose CPU.

VIII. Conclusion

This paper presents PipeDevice, a hardware-software co-design system for low-overhead intra-host container communication. PipeDevice removes the principal overheads of memory copy and TCP stack for long connections by offloading data copy and transmission to hardware. It achieves high scalability by keeping connection states entirely in host DRAM and managing them in software without causing hardware resource contention. We implement PipeDevice on FPGA and conduct extensive testbed experiments. The results show that PipeDevice can save up to 3.93 cores to deliver 80 Gbps throughput compared to kernel TCP.

At the application level, PipeDevice improves the throughput of baidu-allreduce by ∼2.2× over using TCP, and reduces the request completion time in a typical network service chain by over 15% with 47% less CPU compared to FreeFlow.

Appendix

In order to integrate PipeDevice into existing container service frameworks (*e.g.*, Istio [23]), we advocate porting it to the widely used sidecar proxy Envoy. The architecture for this integration is illustrated in Figure 20. Initially, inter-host traffic passes through the gateway container and the sidecar proxy via the kernel TCP/IP stack (①), which also manages dynamic network policies. Then, the sidecar proxy redirects this traffic to user containers through PipeDevice (②). Additionally, communication between user containers is handled by PipeDevice as well (③). In the current Envoy implementation, intra-host communication uses UNIX domain sockets [35]. Specifically, we can modify UNIX domain socket API calls to use the PD API. For example, `socket(AF_UNIX, SOCK_STREAM, 0)` is replaced with `pd_socket(AF_INET, SOCK_STREAM, 0)`. Since UNIX domain sockets do not support IP addresses, whereas PipeDevice relies on IP addressing for intra-host communication, To address this, we extend Kubernetes with a table of available internal IP addresses, assigning each user container a unique internal IP at launch.

References

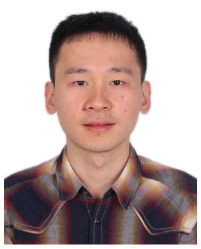
[1] *Achieving Fast, Scalable I/O for Virtualized Servers*. Accessed: Nov. 2023. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/scalable-i-o-virtualized-servers-paper.pdf
[2] *Amazon Web Service*. Accessed: Nov. 2023. [Online]. Available: https://aws.amazon.com/
[3] *AMD Zen 4 Epyc CPU*. Accessed: Nov. 2023. [Online]. Available: https://www.techradar.com/news/amd-zen-4-epyc-cpu-could-be-an-epic-128-core-256-thread-monster
[4] *Backends That Come With PyTorch*. Accessed: Nov. 2023. [Online]. Available: https://pytorch.org/docs/stable/distributed.html#backends-that-come-with-pytorch
[5] *Baidu-Allreduce*. Accessed: Nov. 2023. [Online]. Available: https://github.com/baidu-research/baidu-allreduce
[6] *Bpftrace: High-level Tracing Language for Linux Systems*. Accessed: Nov. 2023. [Online]. Available: https://bpftrace.org/
[7] *Cilium*. Accessed: Nov. 2023. [Online]. Available: https://github.com/cilium/cilium
[8] *Cloud-Native Network Functions*. Accessed: Nov. 2023. [Online]. Available: https://www.cisco.com/c/en/us/solutions/service-provider/industry/cable/cloud-native-network-functions.html
[9] *Deep Learning Containers in Google Cloud*. Accessed: Nov. 2023. [Online]. Available: https://cloud.google.com/deep-learning-containers
[10] *F-Stack: A High Performance Userspace Stack Based on FreeBSD 11.0 Stable*. Accessed: Nov. 2023. [Online]. Available: http://www.f-stack.org/
[11] *Fast Memcpy With SPDK and Intel I/OAT DMA Engine*. Accessed: Nov. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html
[12] *FreeFlow TCP*. Accessed: Nov. 2023. [Online]. Available: https://github.com/microsoft/Freeflow/tree/tcp
[13] *Gloo*. Accessed: Nov. 2023. [Online]. Available: https://github.com/facebookincubator/gloo
[14] *GPUs Vs CPUs for Deployment of Deep Learning Models*. Accessed: Nov. 2023. [Online]. Available: https://azure.microsoft.com/en-us/blog/gpus-vs-cpus-for-deployment-of-deep-learning-models/
[15] *Implement Mmap() for Zero Copy Receive*. Accessed: Nov. 2023. [Online]. Available: https://lwn.net/Articles/752207/
[16] *Implementing TCP Sockets Over RDMA*. Accessed: Nov. 2023. [Online]. Available: https://www.openfabrics.org/images/eventpresos/workshops2014/IBUG/presos/Thursday/PDF/09_Sockets-over-rdma.pdf
[17] *Information About the TCP Chimney Offload, Receive Side Scaling, and Network Direct Memory Access Features in Windows Server 2008*. Accessed: Nov. 2023. [Online]. Available: https://support.microsoft.com/en-us/help/951037/information-about-the-tcp-chimney-offload-receive-side-scaling-and-net
[18] *Intel Arria 10 Product Table*. Accessed: Nov. 2023. [Online]. Available: https://www.intel.co.id/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf
[19] *Intel C610 Series Chipset Datasheet*. Accessed: Nov. 2023. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/x99-chipset-pch-datasheet.pdf
[20] *Intel DSA Specification*. Accessed: Nov. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/develop/articles/intel-data-streaming-accelerator-architecture-specification.html

[21] *IOAT Benchmark*. Accessed: Nov. 2023. [Online]. Available: https://github.com/spdk/spdk/tree/master/examples/ioat/perf
[22] *IO_Uring*. Accessed: Nov. 2023. [Online]. Available: https://man.archlinux.org/man/io_uring.7.en
[23] *ISTIO*. Accessed: Nov. 2023. [Online]. Available: https://istio.io/latest/about/service-mesh/
[24] *Linkerd Architecture*. Accessed: Nov. 2023. [Online]. Available: https://linkerd.io/2.11/reference/architecture/
[25] *Mellanox BlueField-2 DPU*. Accessed: Nov. 2023. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf
[26] *Microsoft Azure*. Accessed: Nov. 2023. [Online]. Available: https://azure.microsoft.com/
[27] *NCCL*. Accessed: Nov. 2023. [Online]. Available: https://github.com/NVIDIA/nccl
[28] *Open MPI: Open Source High Performance Computing*. Accessed: Nov. 2023. [Online]. Available: https://www.open-mpi.org/
[29] *Perftest*. Accessed: Nov. 2023. [Online]. Available: https://github.com/linux-rdma/perftest
[30] *Run Spark Applications With Docker Using Amazon EMR 6.x*. Accessed: Nov. 2023. [Online]. Available: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-docker.html
[31] *Seastar*. Accessed: Nov. 2023. [Online]. Available: http://www.seastar-project.org/
[32] *Spark and Docker: Your Spark Development Cycle Just Got 10× Faster!* Accessed: Nov. 2023. [Online]. Available: https://towardsdatascience.com/spark-and-docker-your-spark-development-cycle-just-got-10x-faster-f41ed50c67fd
[33] *TCP MMAP() Program*. Accessed: Nov. 2023. [Online]. Available: https://lwn.net/Articles/752197/
[34] *The CIFAR-10 Dataset*. Accessed: Nov. 2023. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html
[35] *UNIX Domain Socket*. Accessed: Nov. 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/unix.7.html
[36] *What is Container Management and Why is It Important*. Accessed: Nov. 2023. [Online]. Available: https://searchitoperations.techtarget.com/definition/container-management-software
[37] *Why Use Docker Containers for Machine Learning Development?* Accessed: Nov. 2023. [Online]. Available: https://aws.amazon.com/cn/blogs/opensource/why-use-docker-containers-for-machine-learning-development/
[38] *Zero-Copy TCP Receive*. Accessed: Nov. 2023. [Online]. Available: https://lwn.net/Articles/752188/
[39] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proc. USENIX OSDI*, Oct. 2014, pp. 49–65.
[40] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal, "Towards μs tail latency and terabit Ethernet: Disaggregating the host network stack," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 767–779.
[41] A. M. Caulfield et al., "A cloud-scale acceleration architecture," in *Proc. IEEE/ACM MICRO*, Oct. 2016, pp. 1–13.
[42] Y. Cheng et al., "OPS: Optimized shuffle management system for apache spark," in *Proc. ACM ICPP*, Aug. 2020, pp. 1–11.
[43] P. Fent, A. V. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-latency communication for fast DBMS using RDMA and shared memory," in *Proc. IEEE ICDE*, Apr. 2020, pp. 1477–1488.
[44] D. Firestone et al., "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. USENIX NSDI*, Apr. 2018, pp. 51–64.
[45] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin, "Sockets direct protocol over infiniband in clusters: is it beneficial?" in *Proc. IEEE HOTI*, Mar. 2005, pp. 28–35.
[46] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A new programming interface for scalable network I/O," in *Proc. USENIX OSDI*, Oct. 2012, pp. 135–148.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, Jun. 2016, pp. 770–778.
[48] Z. He et al., "MasQ: RDMA for virtual private cloud," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 1–14.
[49] M. Honda, G. Lettieri, L. Eggert, and D. J. Santry, "PASTE: A network programming interface for non-volatile main memory," in *Proc. USENIX NSDI*, Jan. 2018, pp. 17–33.
[50] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. USENIX NSDI*, Apr. 2014, pp. 445–458.
[51] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *Proc. USENIX OSDI*, Jan. 2020, pp. 463–479.
[52] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be general and fast," in *Proc. USENIX NSDI*, Feb. 2019, pp. 1–16.
[53] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX ATC*, Jun. 2016, pp. 437–450.
[54] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with FlexNIC," in *Proc. ACM ASPLOS*, Mar. 2016, pp. 67–81.
[55] J. Khalid et al., "Iron: Isolating network-based CPU in container environments," in *Proc. USENIX NSDI*, Jan. 2018, pp. 313–328.
[56] D. Kim et al., "FreeFlow: Software-based virtual RDMA networking for containerized clouds," in *Proc. USENIX NSDI*, Feb. 2019, pp. 113–126.
[57] S. G. Kulkarni et al., "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 71–84.
[58] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proc. ACM ASPLOS*, Apr. 2021, pp. 36–51.
[59] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *Proc. ACM EuroSys*, Apr. 2021, pp. 261–276.
[60] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *Proc. ACM SIGCOMM*, Aug. 2019, pp. 90–103.
[61] D. Li, X. Chen, M. Becchi, and Z. Zong, "Evaluating the energy efficiency of deep convolutional neural networks on CPUs and GPUs," in *Proc. IEEE BDCloud-SocialCom-SustainCom*, Oct. 2016, pp. 477–484.
[62] J. Li, S. Xue, W. Zhang, R. Ma, Z. Qi, and H. Guan, "When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization," *IEEE Trans. Cloud Comput.*, vol. 7, no. 4, pp. 1183–1196, Oct. 2019.
[63] X. Lin et al., "Scalable kernel TCP design and implementation for short-lived connections," in *Proc. ASPLOS*, Mar. 2016, pp. 339–352.
[64] G. K. Lockwood, M. Tatineni, and R. Wagner, "SR-IOV: Performance benefits for virtualized interconnects," in *Proc. ACM XSEDE*, Jul. 2014, pp. 1–7.
[65] P. MacArthur and R. D. Russell, "An efficient method for stream semantics over RDMA," in *Proc. IEEE IPDPS*, May 2014, pp. 841–851.
[66] I. Marinos, R. N. M. Watson, and M. Handley, "Network stack specialization for performance," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 175–186.
[67] Y. S. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *Proc. USENIX NSDI*, Jan. 2020, pp. 77–92.
[68] J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, "Secure inter-container communications using XDP/eBPF," *IEEE/ACM Trans. Netw.*, vol. 31, no. 2, pp. 934–947, Sep. 2022.
[69] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *Proc. USENIX ATC*, Jan. 2020, pp. 81–95.
[70] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proc. ACM SIGCOMM*, Aug. 2018, pp. 327–341.
[71] Z. Niu et al., "NetKernel: Making network stack part of the virtualized infrastructure," in *Proc. USENIX ATC*, vol. 30, Dec. 2021, pp. 999–1013.
[72] Y. Peng et al., "A generic communication scheduler for distributed DNN training acceleration," in *Proc. ACM SOSP*, Oct. 2019, pp. 16–29.
[73] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous NIC offloads," in *Proc. ACM ASPLOS*, Apr. 2021, pp. 18–35.
[74] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter, "FlexTOE: Flexible TCP offload with fine-grained parallelism," in *Proc. USENIX NSDI*, Jan. 2021, pp. 87–102.
[75] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015. [Online]. Available: https://dblp.org/rec/journals/corr/SimonyanZ14a.html?view=bibtex
[76] A. Singhvi et al., "1RMA: Re-envisioning remote memory access for multi-tenant datacenters," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 708–721.
[77] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. USENIX OSDI*, Oct. 2010, pp. 33–46.

[78] Q. Su et al., "PipeDevice: A hardware–software co-design approach to intra-host container communication," in *Proc. ACM CoNEXT*, Aug. 2022, pp. 28–30.
[79] S.-Y. Tsai and Y. Zhang, "LITE kernel RDMA support for datacenter applications," in *Proc. ACM SOSP*, Oct. 2017, pp. 306–324.
[80] J. Yang, J. Izraelevitz, and S. Swanson, "FileMR: Rethinking RDMA networking for scalable persistent memory," in *Proc. USENIX NSDI*, Jan. 2020, pp. 111–125.
[81] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "FreeFlow: High performance container networking," in *Proc. ACM HotNets*, Nov. 2016, pp. 43–49.
[82] W. Zhang et al., "OpenNetVM: A platform for high performance network service chains," in *Proc. ACM HotMiddlebox*, Aug. 2016, pp. 26–31.
[83] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Comput.*, vol. 8, no. 2, pp. 635–646, Apr. 2020.
[84] C. Zheng, Q. Lu, J. Li, Q. Liu, and B. Fang, "A flexible and efficient container-based NFV platform for middlebox networking," in *Proc. ACM SAC*, Apr. 2018, pp. 989–995.
[85] D. Zhuo et al., "Slim: OS kernel support for a low-overhead container overlay network," in *Proc. USENIX NSDI*, Feb. 2019, pp. 331–344.

Qiang Su received the B.S. degree from Northeastern University in 2018 and the Ph.D. degree from the City University of Hong Kong in 2024. His research interests include computer networking and systems.

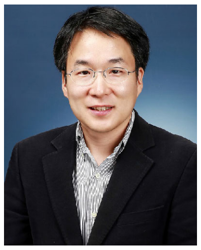
Zhixiong Niu received the B.E. degree in network engineering from Dalian Maritime University (DMU) in 2012, the M.Sc. degree in computer science from The University of Hong Kong (HKU) in 2014, and the Ph.D. degree from the Department of Computer Science, City University of Hong Kong, in 2019. He is currently a Senior Researcher with Microsoft Research Asia.



Ran Shu received the B.E. and Ph.D. degrees in computer science and technology from Tsinghua University in 2011 and 2018, respectively. He is currently a Senior Researcher with Microsoft Research Asia. His research interests include data center networks and networked systems.

Peng Cheng received the B.S. degree in software engineering from Beihang University in 2010 and the Ph.D. degree in computer science and technology from Tsinghua University in 2015. He was a Visiting Ph.D. Student with UCLA from 2013 to 2014. He is currently a Principal Researcher with Microsoft Research Asia. His research interests include computer networking and networked systems.

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



Dongsu Han (Member, IEEE) received the B.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2003 and the Ph.D. degree in computer science from Carnegie Mellon University in 2012. He is currently a Professor with the School of Electrical Engineering and the Graduate School of Information Security, KAIST. His research interests include networking, distributed systems, and network/system security. More details about his research can be found at http://ina.kaist.ac.kr.

Chun Jason Xue (Senior Member, IEEE) received the B.S. degree in computer science and engineering from The University of Texas at Arlington in 1997 and the M.S. and Ph.D. degrees in computer science from The University of Texas at Dallas in 2002 and 2007, respectively. He is currently a Professor with the Department of Computer Science, Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi. His research interests include storage and systems, and high-performance system designs for AI.

Hong Xu (Senior Member, IEEE) received the B.Eng. degree from The Chinese University of Hong Kong in 2007 and the M.A.Sc. and Ph.D. degrees from the University of Toronto in 2009 and 2013, respectively. From 2013 to 2020, he was with the City University of Hong Kong. He is currently an Associate Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include computer networking and systems, particularly big data systems and data center networks.