PipeDevice: A Hardware-Software Co-Design Approach to Intra-Host Container Communication

Qiang Su
City University of Hong Kong
Hong Kong SAR, China

Chuanwen Wang
CUHK
Hong Kong SAR, China

Zhixiong Niu
Microsoft Research
Beijing, China

Ran Shu
Microsoft Research
Beijing, China

Peng Cheng
Microsoft Research
Beijing, China

Yongqiang Xiong
Microsoft Research
Beijing, China

Dongsu Han
KAIST
Daejeon, South Korea

Chun Jason Xue
City University of Hong Kong
Hong Kong SAR, China

Hong Xu
CUHK
Hong Kong SAR, China

ABSTRACT

Containers are prevalently adopted due to the deployment and performance advantages over virtual machines. For many containerized data-intensive applications, however, bulky data transfers may pose 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.

CCS CONCEPTS

• **Networks → Network architectures**.

KEYWORDS

Container Communication, Hardware-Software Co-Design

ACM Reference Format:
Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Jason Xue, and Hong Xu. 2022. PipeDevice: A Hardware-Software Co-Design Approach to Intra-Host Container Communication. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3555050.3569118

1 INTRODUCTION

Containers have become prevalent in public clouds due to the performance, portability, and deployment benefits compared to virtual machines [2, 29]. 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 [33, 44] and the model update process with parameter server and allreduce in distributed machine learning [10, 38].

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 [11, 25, 26]. 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 [35]. 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 [8, 37, 46, 82] 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 [64, 80]. Due to the memory isolation requirement in public clouds and the high memory overhead for a multitude of connections (§2.3), directly sharing memory between two containers is considered infeasible. A better way is to share memory between a container and the hypervisor [80]. The copy between user and kernel spaces is removed, and isolation is preserved since data is still copied across container boundaries (by hypervisor). The downside is that 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 poor scalability [43, 45, 56–59, 77, 78]. The root cause is the contention of limited on-board resources for connection state management [58, 78]. In fact, we show that the performance of commodity RDMA NICs (RNICs) drops by ~50% with 4096 connections (§2.4). As the number of containers and connections increases, cache miss becomes unavoidable and performance degrades. Although many prior arts [43, 45, 56, 57, 77, 77] 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 [17, 40, 49, 68, 75].

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. Then for scalability, we keep the connection states in host DRAM and manage them entirely in software so there is no contention of the limited hardware resources. The cost is prolonged latency in connection management at the hypervisor 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 a hugepage region in the hypervisor, 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 hugepage region is managed solely by the hypervisor who maintains the socket buffer states. The hypervisor also manages the connection states, so that data copy is streamlined and performed in a stateless manner by hardware. PipeDevice also features a set of BSD socket-like APIs for memory and communication to port applications easily.

PipeDevice is a hardware-software co-design approach for low-overhead intra-host container communication. This co-design approach 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.

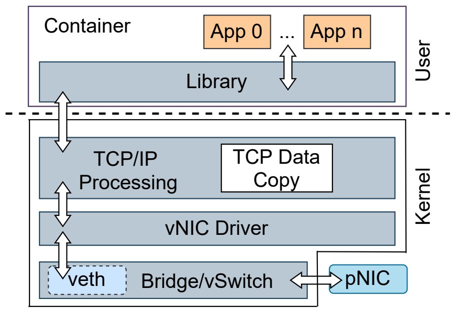


Figure 1: Current container networking architecture in the bridge mode. TCP Data Copy denotes the data copy between the user and kernel spaces.

Note PipeDevice aims to exploit hardware that has already been deployed in data centers, such as FPGA accelerators in Azure [42] and SoC SmartNICs [27], 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 (§2.2). Prior work focuses on the overheads of short connections in host networking [54, 64, 66, 70, 79] or overlay processing [84]. We also investigate RNIC's scalability limitation in intra-host communication (§2.4), while existing work focuses on inter-host networking [43, 45, 56–59, 77, 78].
- We build PipeDevice for low-overhead intra-host container communication (§3-§4). 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 [19] 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 (§5). 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 use-cases, we port baidu-allreduce [4] 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.

2 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.

2.1 Container Communication

Containers are essentially processes with namespace isolation; their applications use standard network libraries such as BSD sockets to

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%

Table 1: 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_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%

Table 2: CPU time breakdown in intra-host container communication at the receiver side. The throughput shown by iperf is 37.66 Gbps.

invoke the default TCP stack, as shown in Figure 1. Modern container frameworks, such as Docker, containerd [9], 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 [61, 84]. 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 and veth), and packets are eventually 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 [84]. 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 [47, 64, 73]. In the following, we quantitatively characterize these overheads.

2.2 Breakdown of Communication Overhead for Bulky Transfers

Bulky transfers are common in typical container workloads, such as data analytics and machine learning for intermediate data exchange, and service chaining in network function virtualization [53, 62, 81] for forwarding packets between network functions. Thus 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` [5]. 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 (see §6) because although overlay processing at the software router also incurs overhead, it can be offloaded to programmable hardware with high efficiency [48].

Tables 1 and 2 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 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 2.

These overheads directly impact application performance. To demonstrate this, we profile allreduce, a communication primitive commonly used in distributed training for deep learning models [55, 74]. 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 [31] 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.

2.3 Why Not Shared Memory?

Many systems have exploited shared-memory networking for intra-host communication [47, 64, 73]. Though it has low CPU overhead, it cannot guarantee memory isolation which is critical in public clouds [53, 83]. The typical approach, such as in SocksDirect [64], is to have a unique shared memory region between two co-located processes so both can directly access the data. This means that a dedicated shared memory region must be created between any pair of communicating containers, which inevitably breaks the isolation requirement of public clouds. In addition, the dedicated memory regions have to be allocated at container launch time, usually leading to linear increases in the memory overhead when the number of container pairs grows. Therefore, the shared-memory approach has high memory overhead. For example, to support 32 single-core containers where each has 512 sockets (each socket ring buffer is 4 MB), a 2 GB memory chunk needs to be allocated

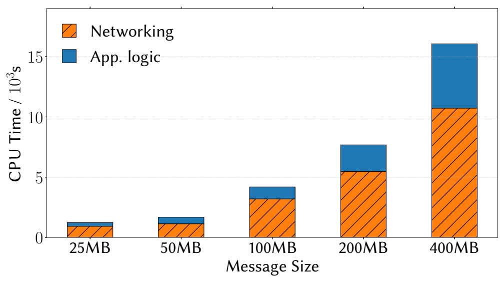


Figure 2: CPU usage of MPI_Allreduce application.

for each pair. Thus 32 GB memory has to be set aside for 16 pairs of containers no matter how many active flows there actually are (more in §5.1). Even worse, communication may occur between any two containers: For the 32-container case, 992 GB may be necessary in the worst case.

Therefore, copying is necessary in a public cloud in order to guarantee isolation and avoid memory overhead. FreeFlow [61, 80], 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 hypervisor. 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. 3.3 in [61]). This suggests the need for a hardware solution that offloads memory copy.

2.4 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 [51, 61]. 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 [56, 78]. Hence commodity RNICs suffer from poor connection scalability as a result of cache contention, a phenomenon widely reported in the community [43, 45, 56–59, 77, 78] 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. Yet 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 §2.2. We exploit `perftest` [32] and `rdma_bench` [58] to measure the throughput of 10-second flows with 1 KB-sized packets in packets-per-second (pps).

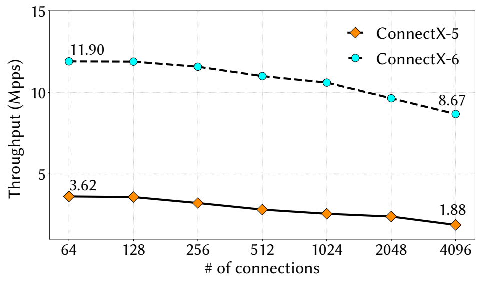


Figure 3: Total throughput (packets-per-second, pps) of intra-host RDMA READ when the number of concurrent connections grows on Mellanox ConnectX-5 25 Gbps and ConnectX-6 100 Gbps RNICs.

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 48.07% and 27.14%, 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 [43, 45, 56, 57, 77] 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 [17, 40, 49, 68, 75]. 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.

2.5 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 hypervisor now, which is negligible for bulky transfers as will be shown in our evaluation.

As explained in §2.1, 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 [18] 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 coined PipeDevice. Because the only functionality on hardware is DMA copy, PipeDevice can be implemented upon various hardware, such as FPGA, Intel DSA [21], RNICs and SmartNICs. We currently choose FPGA for PipeDevice as it has already been massively deployed in data centers to accelerate networking workloads[42, 48]; 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.

3 DESIGN

We now present the design details of PipeDevice.

3.1 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 hugepage region.[1] that can be accessed by 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?

In answering the questions, PipeDevice's design has the following highlights.

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 [16, 36, 39].[2] 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 hugepages for applications.

Memory management. Memory isolation and efficiency are required to make PipeDevice practical in public clouds. Thus PipeDevice organizes the hugepages 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.

Decoupled forwarding plane. Scalability is one of the primary goals in PipeDevice. As explained in §2, 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 [48]. 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 our hardware-software co-design philosophy that works with simple hardware.

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.

[1]PipeDevice uses hugepages because they provide memory with visible physical addresses 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.
[2]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[36].

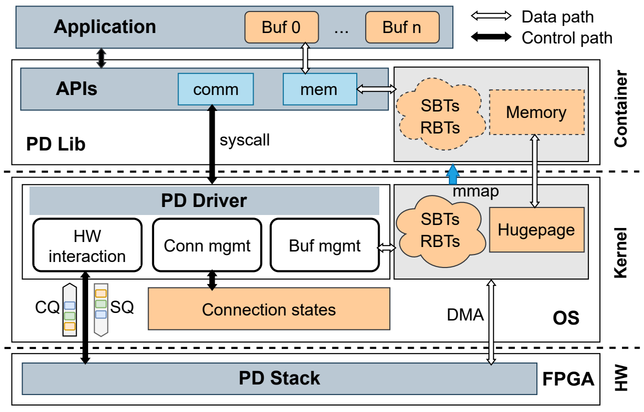


Figure 4: PipeDevice system architecture.

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 hugepages (§3.2). 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 hugepages and tracks their states using a send buffer table (SBT) and a receive buffer table (RBT) (§3.3 and §3.4). 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 correspondingly (§3.5).

3.2 Communication and Memory APIs

We first introduce the new APIs provided by PD Lib.

Communication APIs. PipeDevice exposes zero-copy BSD-like socket interfaces that largely follow the non-blocking semantics — it has a corresponding function call for each socket function (*e.g.,* `socket()` becomes `pd_socket()`). Table 3 details a partial list of the APIs. Unlike the BSD counterpart, PipeDevice supports zero-copy in both send and receive. Specifically, `pd_send()` directly uses the mmaped buffers from the hugepages; 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()` to release the slots in the receive buffer (see §3.5). 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).

Memory API. PD Lib provides pd_malloc() and pd_free() for applications to dynamically create and release buffers in the hugepages.

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.

```
    /* 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.

3.3 Memory Management

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

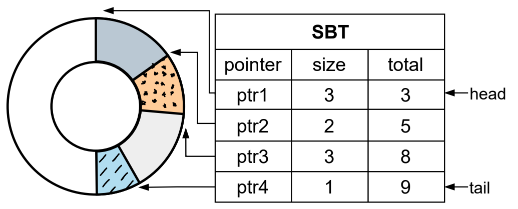
Per-socket lock-free ring buffers. PD Driver maintains the hugepages which can be accessed by PD Stack through the DMA bus. The hugepages are organized as a ring buffer pool, and PD Driver allocates *per-socket* ring buffers to avoid locking overhead. Upon receiving the pd_socket() call, PD Driver applies 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. Figure 5 depicts a send ring buffer and its 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.

3.4 Connection Management

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

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



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

Figure 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.

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

Figure 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.

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.

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 sids, and deletes the corresponding connection table entry.

3.5 Data Transmission

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

Per-core command queues. PD Driver interacts with PD Stack in 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

Function	Parameters	Description
`pd_init()`	`chardev, sqthresh, cqthresh`	Initialize PipeDevice, setup the batch sizes for FPGA to update SQ and CQ tails.
`pd_release()`		Release PipeDevice.
`pd_socket()`	`domain, type, protocol`	Create an endpoint for communication.
`pd_bind()`	`sid, pd_sock_addr, addrlen`	Bind a name to a PipeDevice socket.
`pd_listen()`	`sid, backlog`	Listen for connections on a PipeDevice socket.
`pd_accept()`	`sid, pd_sock_addr, addrlen`	Accept a connection on a PipeDevice socket.
`pd_connect()`	`sid, pd_sock_addr, addrlen`	Initialize a connection on a PipeDevice socket.
`pd_send()`	`sid, sendbuf, buflen`	Send a message on a PipeDevice socket.
`pd_buf_refresh()`	`sid, ptrs, ptrlen`	Check if the send buffers are reusable and return their pointers.
`pd_recv()`	`sid, buflen, recvbuf`	Receive data from sender and prepare the receive buffer.
`pd_recv_done()`	`sid, buflen`	Release the receive buffer (Check recv status).
`pd_close()`	`sid`	Close the PipeDevice socket.
`pd_malloc()`	`size`	Allocate a memory buffer with given size.
`pd_free()`	`buf`	Free the memory buffer.

Table 3: A partial list of PipeDevice APIs. PipeDevice also have `pd_getsockopt` and `pd_setsockopt`. `chardev` is the FPGA device, `sqthresh` and `cqthresh` are the depths of SQ and CQ, respectively; `sid` is PipeDevice socket id; `pd_sock_addr` is a structure that contains the IP address and port; `addrlen` denotes the length of `pd_sock_addr`; `backlog` is the queue depth of the backlog queue; `ptrs` is the pointer list of the sender buffers to be checked, and `ptrlen` denotes the number of checked send buffers.

possible head-of-line blocking and out-of-order execution on the hardware queues for co-located connections or applications because 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 hugepages 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.

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, `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

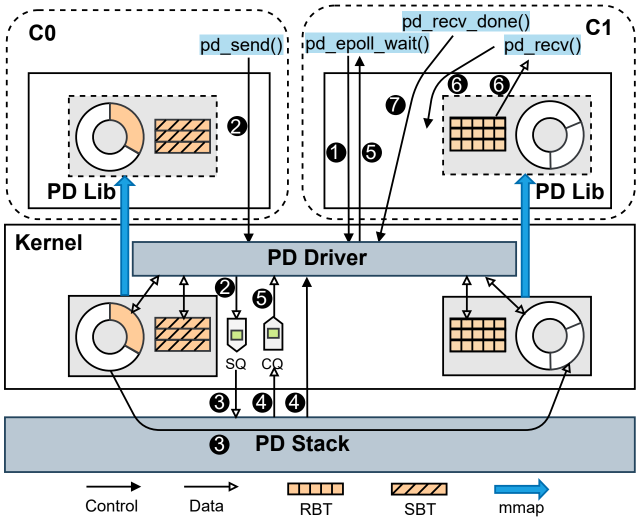


Figure 7: An example of the data transmission process from `C0` to `C1`. The double-ended data arrows show that PD Driver maintains RBTs, SBTs, and data buffers. The ring buffers in `C0`'s and `C1`'s contexts are socket `S0`'s send buffer and `S1`'s receive buffer, respectively. Only relevant buffers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

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.

4 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

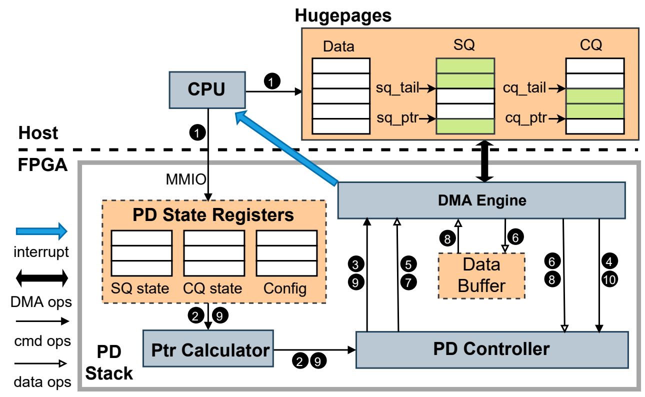


Figure 8: The FPGA PD Stack implementation. `cmd ops` **and** `data ops` **are for command queue and payload operations, respectively.** `DMA ops` **includes DMA write and read. PD Stack sets three registers to maintain the command queue states (e.g., pointers) and the FPGA configurations. The data queues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

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

Table 4: PipeDevice's resource usage on FPGA.

PD Driver as a kernel module. We use 4 GB hugepages with 2 MB pages, and each ring buffer is 4 MB. SQ and CQ are FIFO queues with a depth of 1K.

PD Stack. We implement PD Stack on Intel Arria 10 FPGA [19] (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. In addition, PD Stack processes the elements of a command queue in a FIFO order, and scans all command queues in a round-robin fasion. 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 gets the payload, writes it 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.

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.

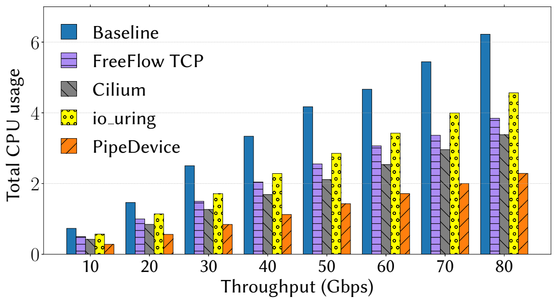


Figure 9: Total CPU usage for throughput targets (Not count the core for FreeFlow router).

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.

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 4 presents the detailed on-board resource usage. Note that 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).

How to implement PipeDevice over other hardware? PipeDevice's PD Stack can be implemented on various hardware with DMA engines, including SmartNIC [27], Intel IOAT [13], *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 [27]) and the Intel IOAT [13, 22] (*e.g.,* Intel C610 series chipset [20]).

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 [28] 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.

5 EVALUATION

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

(1) How much resource saving does PipeDevice offer? (§5.1)
(2) Does PipeDevice provide high throughput and good connection scalability? (§5.2 and §5.3)
(3) What about other microbenmarks, such as latency and fairness? (§5.4)
(4) Is it easy to port applications to PipeDevice, and how much performance gain can be achieved at the application level? (§5.5)

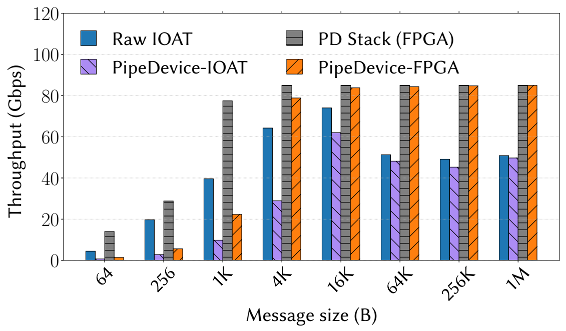


Figure 10: Comparison of PipeDevice raw throughput with varying message sizes.

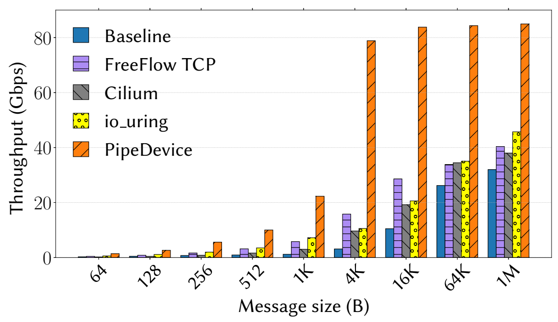


Figure 11: The throughput comparison with varing message sizes.

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 [13] which is a memory copy engine on Intel CPU chipset, and software approaches including Cilium [6], io_uring [24], and state-of-the-art FreeFlow [61] which has both RDMA- and TCP-based implementations [14], 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 [23]. 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.

5.1 Resource Savings

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

CPU Usage. We quantify PipeDevice's CPU savings over Baseline, Cilium, io_uring, and FreeFlow TCP. We obtain the CPU usage as

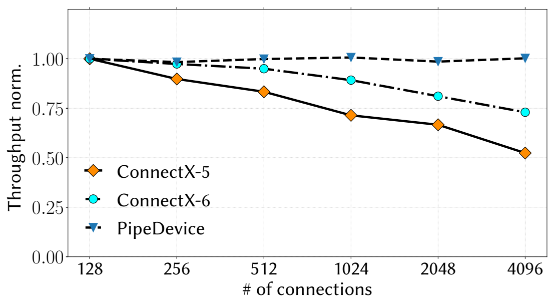


Figure 12: Connection scalability. Throughput is normalized by that of 128 connections.

the total number of consumed CPU cores for both the sender and receiver. Note that FreeFlow uses an extra core dedicated for the centralized controller (FreeFlow router). Baseline, Cilium, io_uring and FreeFlow TCP use 16 TCP streams to saturate the cores, while PipeDevice uses a single stream.[3]

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 and 2.28 cores compared to Baseline, FreeFlow TCP, Cilium and io_uring. This confirms PipeDevice's benefit in removing the overheads of memory copy and TCP/IP stack processing. Note that since Cilium bypasses the kernel TCP/IP stack and io_uring reduces the syscall overhead, they also save CPU over Baseline. We expect PipeDevice to achieve greater CPU saving with newer hardware (more in §6).

Memory Consumption. PipeDevice dynamically manages memory for containers (§3.3). On our server PipeDevice can easily support 32 single-core containers using 4 GB hugepages in total, and each container has 512 active sockets (each socket ring buffer is 4 MB). On the contrary, as explained in §2.3, 32 GB memory is necessary for 16 pairs of containers, i.e. 8x the footprint of PipeDevice.

5.2 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 §4), its raw throughput saturates with 4 KB messages. This demonstrates that our co-design approach with PD Driver and PD

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

	Average	Median	75%ile	90%ile	99%ile
Baseline	12.12	12.71	12.87	14.08	16.46
FreeFlow TCP	14.25	14.37	15.06	15.89	18.19
FreeFlow RDMA	1.65	1.74	1.75	1.77	1.82
Cilium	7.49	5.66	10.06	10.26	11.14
io_uring	3.07	2.44	2.47	4.10	20.98
Raw IOAT	0.03	0.02	0.02	0.02	1.14
PipeDevice-IOAT	0.96	0.94	0.95	1.02	1.26
PipeDevice-FPGA	8.27	7.39	8.11	13.82	23.13

Table 5: Comparison of latency (us) distribution at 64 B.

	Average	Median	75%ile	90%ile	99%ile
Baseline	16.82	16.77	16.99	17.35	22.19
FreeFlow TCP	18.88	19.56	20.32	20.91	24.29
FreeFlow RDMA	3.13	3.21	3.36	3.42	10.13
Cilium	9.15	7.29	13.09	13.24	13.75
io_uring	12.06	10.02	14.41	19.31	38.47
Raw IOAT	0.19	0.02	0.02	0.95	1.54
PipeDevice-IOAT	3.42	1.06	3.60	9.85	22.13
PipeDevice-FPGA	9.85	9.54	9.78	11.68	18.84

Table 6: Comparison of latency (us) distribution at 8 KB.

Lib does not incur much performance loss for bulky data transfer. Further, considering that 2×8 PCIe lanes promise ~110 Gbps theoretical bandwidth in gen3 [72], there is room to optimize the PD Stack implementation on FPGA for better performance (more in §6).

The throughput of raw IOAT and PipeDevice-IOAT starts to drop when message size is larger than 16 KB,[4] and PipeDevice-IOAT may achieve ~62 Gbps at 16 KB. We can 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 §4). 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 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.

5.3 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

[4]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.

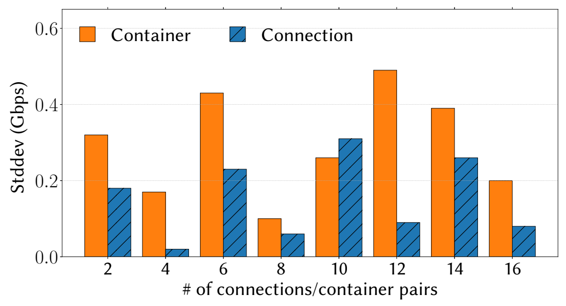


Figure 13: Stddev of throughput with varying numbers of container pairs or connections.

increases to 4096, PipeDevice's throughput remains stable at peak while RDMA's decreases sharply due to cache contention on RNIC as discussed in §2.4.

5.4 Latency and Fairness

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 present various statistics of latency over 5,000 runs. Tables 5 and 6 show the latency statistics for 64 B and 8 KB messages, respectively. We observe that PipeDevice-FPGA and PipeDevice-IOAT achieves lower latency than Baseline 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. At 64B, we can see that PipeDevice-FPGA is worse than Cilium and io_uring, the reason is that PipeDevice-FPGA introduces extra delay on PCIe transmission and kernel processing (*e.g.,* connection management). At 8KB, io_using has higher 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 latency than RDMA which is not surprising. PipeDevice relies on the hypervisor kernel to carry out control actions (memory management, connection state management) and thus bears the context switching and syscall overheads (recall §3.1), while RDMA offloads the entire stack to hardware and enjoys hardware-level latency. Note that the latency is a small price we consciously choose to pay for a more practical design for bulky transfer.

Fairness. We evaluate PipeDevice's fairness in terms of per-core bandwidth sharing. Specifically, we run 16 pairs of 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 13. Observe that the standard deviation for PipeDevice stands at a low value (< 0.5 Gbps).

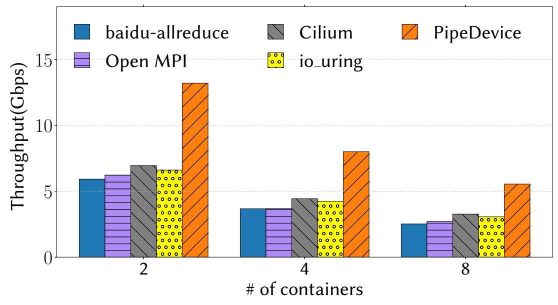


Figure 14: Comparison of allreduce's end-to-end throughput.

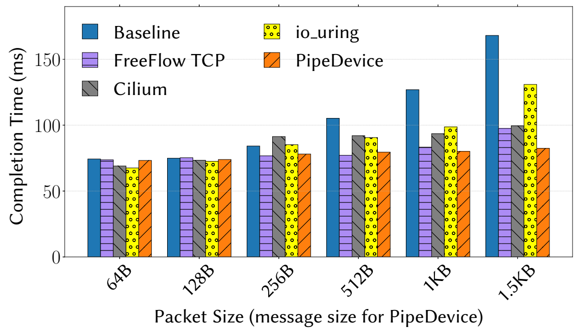


Figure 15: Comparison of the completion time of the network service chain.

This is because PipeDevice interacts with FPGA through per-core command queues, which ensures fair sharing of FPGA's performance across multiple cores.

5.5 Application Usecases

In this section, we quantify the application-level performance gain of PipeDevice.

Ring allreduce. Allreduce is widely used in distributed training of deep learning models to aggregate gradients from different workers [15, 30]. We implement a ring allreduce library using PipeDevice by porting baidu-allreduce [4]. 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 computation is done on CPU. We also port baidu-allreduce using Cilium and io_uring for comparison, by adding 199 and modifing 48 lines, respectively, and also compare to `MPI_Allreduce` in Open MPI v4.1 [31]. 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 14 shows the results averaged over 100 runs. PipeDevice significantly improves the allreduce throughput by ~2.2×, ~2.1×, ~1.9× and ~2.0× over vanilla baidu-allreduce, Open MPI, Cilium, and io_uring versions. Considering that communication takes ~60% CPU cycles for allreduce (recall §2.2), this proves the performance benefits by using the saved CPU cycles to accelerate computation. We also observe that the Cilium 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 bypasses the TCP/IP stack processing and io_uring reduces the syscall overhead.

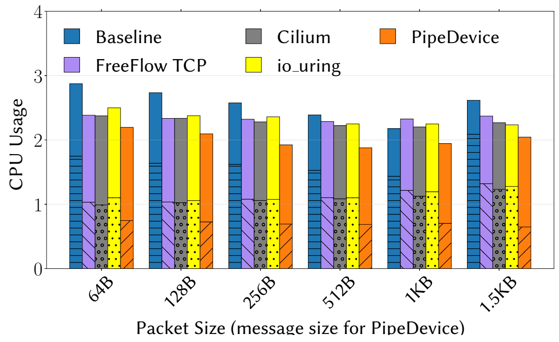


Figure 16: Comparison of the overall CPU usage of the network service chain. For each bar, the bottom and top parts show the communication and computation cpu usage, respectively.

Network service chain. Virtual network functions are increasingly deployed using containers in practice for improved performance and efficiency [7, 81], 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 and io_uring with adding 199 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 15 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 and io_uring. This is 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 and io_uring also increases in this case. Figure 16 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. FreeFlow TCP achieves comparable performance to PipeDevice at the cost of an extra core for FreeFlow router. It consumes 3.37 cores to achieve 97.51 ms completion time, while PipeDevice only takes 2.04 cores to reach 82.42 ms (15.57% reduction).

6 DISCUSSION

This section discusses some immediate concerns one may have about PipeDevice.

Can PipeDevice support overlay networking? PipeDevice works with overlay networking which does not critically impact its design. In case a software overlay router is used, PD Driver can interact with it to ensure the necessary overlay processing such as IP address mapping. 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 already deployed [48], 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.

Does PipeDevice support various network policies? The current design of PipeDevice can support the 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 inspects the metadata in each request and only processes 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 [48] to facilitate the control plane policy enforcement as discussed before.

Can we acheive higher throughput? As explained before in §5.2, 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 [72] and beyond which is sufficient in public clouds.

Can PipeDevice be extended to support inter-host networking? Although PipeDevice is designed for intra-host traffic, it can bring benefits to inter-host container communication as well. For example, in service mesh, PipeDevice mitigates the communication overhead between containers and its co-located sidecar proxy which in turn benefits traffic across different hosts. In addition, PipeDevice can co-exist with the TCP stack gracefully if one needs to handle traffic in the wide Internet. This can be done at the application level by simply using the corresponding interfaces to invoke the inter-host stacks (*e.g.,* BSD socket for kernel TCP), or at PipeDevice by having additional logic in PD Lib to check the destination address and invoke the relevant functions of the inter-host stacks. Meanwhile, frameworks like NetKernel [73] can also be used to support concurrent use of multiple network stacks in a virtualized environment. We leave this as future work.

7 RELATED WORK

We discuss related work to PipeDevice in this section.

Container networking. Containerization in clouds drives innovations in container networking. Other than [61, 80] which we discussed in detail throughout this work, there are relatively fewer efforts. Iron [60] provides strong performance isolation for containers; Slim [84] and Falcon [63] aim to reduce the overlay processing overhead; BASTION [71] offers a container-aware communication sandbox for secure container networking. Cilium [6] supports kernel network stack optimization by eBPF for containers. They do not consider overheads of bulky transfer in intra-host scenarios.

SR-IOV. SR-IOV compliant hardware [65, 67] 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.

Host network stacks. High-performance network stacks are also beneficial to container networking, including kernel optimizations [50, 66, 76, 79] and user-space stacks [12, 34, 41, 52, 54, 69]. These arts 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.

Hardware offloading. Our community has also devoted efforts to hardware-assisted low overhead networking. Other than RDMA [43, 45, 56–59, 78] we discussed in §2.4, many work offloads various network workloads onto so-called SmartNICs, including TCP connection processing [18], stateful operations of short connections [70], and SDN policies [48]. Though they do not consider the data path of long connections in the container context, they do inspire PipeDevice's design: offloading greatly helps to reduce the CPU overhead by moving the expensive components of the current architecture away from general-purpose CPU.

8 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.

We are considering several directions of future work. First, zero-copy socket interfaces. 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. Second, short connection optimization. Container networking with short connections is also a common scenario in data centers as many applications such as nginx generate latency-sensitive request and response traffic. A hardware approach may also be exploited here to reduce the context switching and connection setup overheads of TCP without RDMA's scalability issues.

ACKNOWLEDGMENT

We thank the anonymous CoNEXT reviewers and our shepherd Gianni Antichi for their valuable comments. This work is supported in part by funding from the Research Grants Council of Hong Kong (11209520) and from CUHK (4937007, 4937008, 5501329, 5501517), and a gift fund from Microsoft (6906276).

REFERENCES

[1] Achieving Fast, Scalable I/O for Virtualized Servers. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/scalable-i-o-virtualized-servers-paper.pdf.

[2] Amazon web service. https://aws.amazon.com/.

[3] AMD Zen 4 Epyc CPU. https://www.techradar.com/news/amd-zen-4-epyc-cpu-could-be-an-epic-128-core-256-thread-monster.

[4] Baidu-allreduce. https://github.com/baidu-research/baidu-allreduce.

[5] bpftrace: High-level tracing language for linux systems. https://bpftrace.org/.

[6] Cilium. https://github.com/cilium/cilium.

[7] Cloud-Native Network Functions. https://www.cisco.com/c/en/us/solutions/service-provider/industry/cable/cloud-native-network-functions.html.

[8] Container management in 2021: In-depth guide. https://research.aimultiple.com/container-management/.

[9] containerd: an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[10] Deep learning containers in Google Cloud. https://cloud.google.com/deep-learning-containers.

[11] Enable Istio proxy sidecar injection in Oracle cloud native environment. https://docs.oracle.com/en/learn/ocne-sidecars/index.html#introduction.

[12] F-Stack: A high performance userspace stack based on FreeBSD 11.0 stable. http://www.f-stack.org/.

[13] Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html.

[14] FreeFlow TCP. https://github.com/microsoft/Freeflow/tree/tcp.

[15] Gloo. https://github.com/facebookincubator/gloo.

[16] Implement `mmap()` for zero copy receive. https://lwn.net/Articles/752207/.

[17] Implementing TCP Sockets over RDMA. https://www.openfabrics.org/images/eventpresos/workshops2014/IBUG/presos/Thursday/PDF/09_Sockets-over-rdma.pdf.

[18] Information about the TCP chimney offload, receive side scaling, and network direct memory access features in Windows server 2008. https://support.microsoft.com/en-us/help/951037/information-about-the-tcp-chimney-offload-receive-side-scaling-and-net.

[19] Intel Arria 10 product table. https://www.intel.co.id/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf.

[20] Intel C610 Series Chipset Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/x99-chipset-pch-datasheet.pdf.

[21] Intel DSA specification. https://www.intel.com/content/www/us/en/develop/articles/intel-data-streaming-accelerator-architecture-specification.html.

[22] Intel QuickData Technology Software Guide. https://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf.

[23] IOAT benchmark. https://github.com/spdk/spdk/tree/master/examples/ioat/perf.

[24] io_uring. https://man.archlinux.org/man/io_uring.7.en.

[25] Istio. https://istio.io/latest/about/service-mesh/.

[26] Linkerd architecture. https://linkerd.io/2.11/reference/architecture/.

[27] Mellanox BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[28] Mellanox BlueField DPU DMA Guide. https://docs.nvidia.com/doca/sdk/dma-samples/index.html.

[29] Microsoft Azure. https://azure.microsoft.com/.

[30] NCCL. https://github.com/NVIDIA/nccl.

[31] Open MPI: Open source high performance computing. https://www.open-mpi.org/.

[32] Perftest. https://github.com/linux-rdma/perftest.

[33] Run Spark applications with Docker using Amazon EMR 6.x. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-docker.html.

[34] Seastar. http://www.seastar-project.org/.

[35] Spark and Docker: Your Spark development cycle just got 10x faster! https://towardsdatascience.com/spark-and-docker-your-spark-development-cycle-just-got-10x-faster-f41ed50c67fd.

[36] TCP `mmap()` program. https://lwn.net/Articles/752197/.

[37] What is container management and why is it important. https://searchitoperations.techtarget.com/definition/container-management-software.

[38] Why use Docker containers for machine learning development? https://aws.amazon.com/cn/blogs/opensource/why-use-docker-containers-for-machine-learning-development/.

[39] Zero-copy TCP receive. https://lwn.net/Articles/752188/.

[40] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D.K. Panda. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *Proc. IEEE ISPASS*, 2004.

[41] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. USENIX OSDI*, 2014.

[42] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proc. IEEE/ACM MICRO*, 2016.

[43] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proc. ACM EuroSys*, 2019.

[44] Yuchen Cheng, Chunghsuan Wu, Yanqiang Liu, Rui Ren, Hong Xu, Bin Yang, and Zhengwei Qi. OPS: Optimized shuffle management system for Apache Spark. In *Proc. ACM ICPP*, 2020.

[45] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proc. USENIX NSDI*, 2014.

[46] Weibei Fan, Jing He, Zhijie Han, Peng Li, and Ruchuan Wang. Intelligent resource scheduling based on locality principle in data center networks. *IEEE Communications Magazine*, 58(10):94–100, 2020.

[47] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *Proc. IEEE ICDE*, 2020.

[48] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the public cloud. In *Proc. USENIX NSDI*, 2018.

[49] D. Goldenberg, M. Kagan, R. Ravid, and M.S. Tsirkin. Sockets Direct Protocol over InfiniBand in clusters: is it beneficial? In *Proc. IEEE HOTI*, 2005.

[50] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proc. USENIX OSDI*, 2012.

[51] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *Proc. ACM SIGCOMM*, 2020.

[52] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *Proc. USENIX NSDI*, 2018.

[53] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX NSDI*, 2014.

[54] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. USENIX NSDI*, 2014.

[55] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proc. USENIX OSDI*, 2020.

[56] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proc. USENIX NSDI*, 2019.

[57] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, 2014.

[58] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. USENIX ATC*, 2016.

[59] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. USENIX OSDI*, 2016.

[60] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *Proc. USENIX NSDI*, 2018.

[61] Daehyeok Kim, Tianlong Yu, Hongqiang Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based virtual RDMA networking for containerized clouds. In *Proc. USENIX NSDI*, 2019.

[62] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic backpressure and scheduling for NFV service chains. In *Proc. ACM SIGCOMM*, 2017.

[63] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. Parallelizing packet processing in container overlay networks. In *Proc. ACM EuroSys*, 2021.

[64] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter sockets can be fast and compatible. In *Proc. ACM SIGCOMM*, 2020.

[65] Jian Li, Shuai Xue, Wang Zhang, Ruhui Ma, Zhengwei Qi, and Haibing Guan. When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization. *IEEE Transactions on Cloud Computing*, 7(4):1183–1196, 2019.

[66] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel TCP design and implementation for short-lived connections. In *Proc. ASPLOS*, 2016.

[67] Glenn K. Lockwood, Mahidhar Tatineni, and Rick Wagner. SR-IOV: Performance benefits for virtualized interconnects. In *Proc. ACM XSEDE*, 2014.

[68] Patrick MacArthur and Robert D. Russell. An Efficient Method for Stream Semantics over RDMA. In *Proc. IEEE IPDPS*, 2014.

[69] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. In *Proc. ACM SIGCOMM*, 2014.

[70] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proc. USENIX NSDI*, 2020.

[71] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A security enforcement network stack for container networks. In *Proc. USENIX ATC*, 2020.

[72] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proc. ACM SIGCOMM*, 2018.

[73] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *Proc. USENIX ATC*, 2020.

[74] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proc. ACM SOSP*, 2019.

[75] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proc. ACM SIGCOMM*, 2020.

[76] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. USENIX OSDI*, 2010.

[77] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proc. ACM SOSP*, 2017.

[78] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA networking for scalable persistent memory. In *Proc. USENIX NSDI*, 2020.

[79] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In *Proc. USENIX ATC*, 2016.

[80] Tianlong Yu, Shadi Abdollahian Noghabi, Shachar Raindel, Hongqiang Liu, Jitu Padhye, and Vyas Sekar. FreeFlow: High performance container networking. In *Proc. ACM HotNets*, 2016.

[81] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phil Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *Proc. ACM HotMiddlebox*, 2015.

[82] Dongfang Zhao, Mohamed Mohamed, and Heiko Ludwig. Locality-aware scheduling for containers in cloud computing. *IEEE Transactions on Cloud Computing*, 8(2):635–646, 2020.

[83] Chao Zheng, Qiuwen Lu, Jia Li, Qinyun Liu, and Binxing Fang. A flexible and efficient container-based NFV platform for middlebox networking. In *Proc. ACM SAC*, 2018.

[84] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *Proc. USENIX NSDI*, 2019.