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

Qiang Su
City University of Hong Kong
Microsoft Research

Chuanwen Wang
CUHK

Zhixiong Niu
Microsoft Research

Ran Shu
Microsoft Research

Peng Cheng
Microsoft Research

Yongqiang Xiong
Microsoft Research

Dongsu Han
KAIST

Chun Jason Xue
City University of Hong Kong

Hong Xu
CUHK

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 *ACM SIGCOMM 2022 Conference (SIGCOMM '22 Demos and Posters), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3546037.3546054

1 INTRODUCTION

Containers have become prevalent in public clouds due to the performance, portability, and deployment benefits compared to virtual machines [1, 8, 14]. They support a wide variety of workloads, from microservices 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 [3, 15, 16, 23] and the model update process with parameter server and allreduce in distributed machine learning [5, 19, 32, 35].

With these applications on the rise, it is increasingly common for bulky transfers to occur in the intra-host scenario, usually through IPC channels, *e.g.*, shared-memory and network stacks, and we expect the trend to continue for a number of reasons. First, in constructing a so-called service mesh, a sidecar proxy container is deployed in each (micro-)service (with one or more containers) instance to route all traffic from this instance to other services via shared-memory, and the service containers and its sidecar proxy are always co-located on the same machine [6, 11, 12]. 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/IP and RDMA [17]. These introduce massive intra-host traffic among co-located containers. Second, cloud operators and container orchestrators often consolidate a tenant's application containers onto as few servers as possible [18, 25, 41] to improve efficiency. Moreover, with the server becoming physically more resourceful in terms of CPU cores and memory [2], hundreds of containers can reside on the same server.

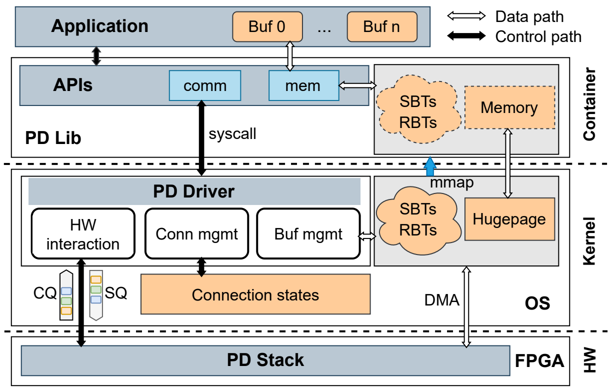

Figure 1: PipeDevice system architecture.

Extensive prior work exists on reducing the overhead of bulky transfers among co-located containers. 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 [31, 40]. However, it may cause high memory overhead for a large of connections because a dedicated shared memory region must be created between any pair of communicating containers at launch time no matter how many active flows there actually are. It also breaks the memory isolation requirement in public clouds. A better way is to share memory between a container and the hypervisor [40]. The copy between user and kernel spaces is removed, and isolation is preserved since data is still copied across container boundaries. 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 [22, 24, 27–30, 34, 37–39]. The root cause is the contention of limited on-board

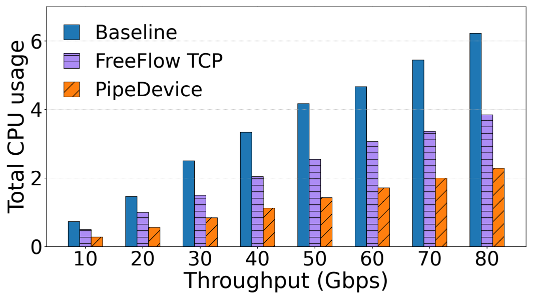

Figure 2: Total CPU usage for throughput targets (the core for FreeFlow router is not counted). It is also expected PipeDevice can achieve greater CPU saving with future hardware (more PCIe lanes or gen4/gen5 lanes).

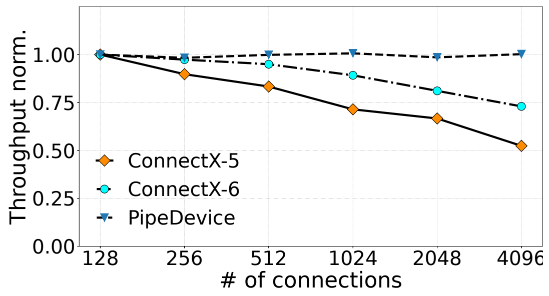

Figure 3: Connection scalability. Throughput is normalized by that of 128 connections. At 4096 connections, the throughput of ConnectX-5 and ConnectX-5 drops by by 47.62% and 27.03%.

resources for connection state management [29, 39]. In fact, we show in Figure 3 that the performance of commodity RDMA NICs (RNICs) declines sharply for co-located containers. As the number of containers and connections increases, cache miss becomes unavoidable and performance degrades. Although many prior arts [22, 24, 27, 28, 37, 37, 38] try to improve scalability by balancing the tradeoff between efficiency and on-board state, they do not eliminate the main culprit. Meanwhile, these works are all optimizations for message-based RDMA semantics, which still suffer from inefficiency on stream-based applications [9, 20, 26, 33]. Some recent work [36] mandates a clean-slate approach to fully resolve such limitations at the expense of high deployment barriers.

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 occurring in 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 [4, 13] and Intel IOAT [7]) 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 guarantee 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. For deployability, PipeDevice currently exploits FPGA that has already been deployed for accelerating various workloads [21]. PipeDevice also features a set of BSD socket-like APIs for memory and communication to allow applications to be easily ported. Note that raw performance is not our primary goal due to the heterogeneity of underlying hardware.

2 DESIGN

Figure 1 depicts the architecture of PipeDevice. A PD Lib runs inside each container to serve the applications. It has a corresponding function call for each socket function (*e.g.,* `socket()` becomes `pd_socket()`). It handles all communication requests and exposes to applications the send and receive buffers that are mapped from the hugepages. 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). 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). To transmit the data, PD Stack on FPGA polls each SQ, parses the entry, and executes the command. 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.

3 EVALUATION

We implement a prototype of PipeDevice based on Linux kernel 4.9 and Intel Arria 10 FPGA [10] (2×8 PCIe gen3 lanes). We use 4GB hugepages with 2MB pages, and each ring buffer is 4MB.

CPU savings. We quantify PipeDevice's CPU savings over kernel TCP/IP (denote as Baseline) and FreeFlow TCP. We obtain the CPU usage as the total number of consumed CPU cores for both the sender and receiver. Figure 2 shows PipeDevice significantly reduces the CPU usage while achieving the same throughput as other schemes. In achieving 80 Gbps, it saves 3.93 and 1.56 cores compared to Baseline and FreeFlow TCP. This confirms PipeDevice's benefit in removing the overheads of memory copy and TCP/IP stack processing.

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 1KB. We compare it to the READ throughput of the Mellanox ConnectX-5 25GbE RNIC and ConnectX-6 100GbE RNICs, and normalize the result by that of 128 connections. Figure 3 shows the normalized results. 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.

REFERENCES

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

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

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] Broadcom Stingray SmartNIC. https://docs.broadcom.com/doc/PS225-PB.

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

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

[7] 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.

[8] Google cloud. https://cloud.google.com/.

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

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

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

[12] kube-proxy. https://kubernetes.io/docs/concepts/overview/components/#kube-proxy.

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

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

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

[16] Running Spark on Kubernetes. https://spark.apache.org/docs/latest/running-on-kubernetes.html.

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. USENIX OSDI*, 2014.

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

[34] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proc. ACM SYSTOR*, 2019.

[35] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799, 2018.

[36] 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.

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

[38] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. StaR: Breaking the Scalability Limit for RDMA. In *Proc. IEEE ICNP*, 2021.

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

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

[41] 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.