Fast and Compatible User-Space Container Networking with Programmable NIC

Bojie Li

USTC and Microsoft Research

1. Motivation

Containers are an emerging cloud service for developers to wrap up applications into isolated boxes. To simplify deployment, developers break large monolithic services into small self-contained microservice containers, interconnected via *container network*. For example, a Web application is typically built with a load balancer, Web application containers, cache, search index, database and background data analytic services. Each component is wrapped into a container and they communicate via a virtual network, *e.g.*, Linux bridge.

Due to extensive communication among containers, currently the OS kernel is the performance bottleneck for most containerized applications. Our stress tests on NSC DNS server, Redis, Nginx and lighttpd show that 78% to 92% of CPU time is spent in container networking. The CPU cost of container networking attributes to two factors, connection setup and data transmission. With Linux network stack, a CPU core can create 50K new connections per second, mainly attributed to file descriptor and socket initialization. With 24 cores, the connection setup throughput only increases to 6x [13]. This sub-linear scaling is due to lock contention in the file system and network stack. For data transmission, a context switch is required per *send* and *recv* system call, and the network stack is involved even if the sender and receiver are in a same server. As a result, a single-core application can only process 300K network messages per second [17]. These overheads kept many developers away from containerizing performance-critical applications.

2. Background and Related Work

Recent years we see a trend of specializing operating system for high performance. The network stack in an operating system can be roughly divided into two layers: network packet processing and application interface (*e.g.* socket, RDMA).

To accelerate network packet processing, FlexNIC [11] redesigns NIC architecture to support flexible and high-performance processing. E2 [15] and NetBricks [16] are high performance packet processing systems using a pipeline of dedicated cores. Corey [6] and Multikernel [4] achieve multicore scalability by replacing shared memory coordination with explicit message passing among cores. FreeFlow [3] achieves high performance container networking for long-lived connections using user-space communication. Inter-core message passing has limited efficiency. To receive a message from a lockless shared-memory queue, the receiver core needs to read non-cached data, because the cache has been invalidated by the message sender. There is no instruction to directly copy data from one core's private cache to another core's.

To translate application interface to network packets efficiently, one line of work develops lightweight TCP/IP stacks to provide socket-compatible APIs with high performance. Netmap [18] and VALE [19] are high-performance user-space network stacks with BSD-like API, but do not integrate in the Linux file descriptor namespace. Fastsocket [13] and F-stack [1] improves Linux socket performance and scalability while keeping the most extent compatibility. However, many Linux APIs are rarely used [20] and they constitute a large portion of processing delay on the data path [17]. TCP Chimney [14] offloads a part of network stack to hardware.

The other line of work uses hardware (the NIC) to achieve kernel-bypass networking between VMs or processes on the same host. Arrakis [17] and IX [5] leverage hardware-assisted virtualization and SR-IOV NICs to bypass kernel coordination. Mellanox VMA [2] translates socket APIs into RDMA verbs via a light-weight socket and network stack in user space. When applying these designs to container networking, there are two challenges: (1) All packets traverse through the NIC, while the NIC has limited *hairpin* processing capacity and PCIe bandwidth. (2) The number of NIC virtual functions (VF) is not enough to support each container with a VF. This paper basically falls in this category and solves the challenges using careful hardware-software co-design.

After years of broken promises, the increasing gap between network speed and CPU processing capacity finally makes *programmable NICs* deployed in data centers [12], opening up new possibilities for network stack design.

3. Design

Our goal is to leverage hardware-software co-design to develop a container networking system with both high performance and compatibility with existing software.

Due to limited PCIe throughput, we leave the data plane on the CPU and only process control messages via programmable NIC (FPGA). We use NIC as the center of coordination instead of a dedicated core, because the NIC has

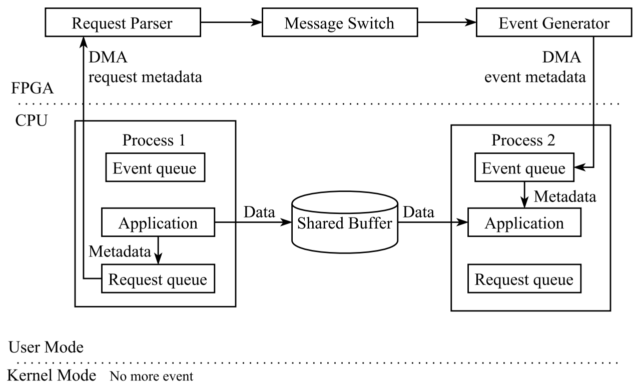

Figure 1: Container Networking with Programmable NIC.

much more processing capacity than a CPU core [12]. We design the NIC to support three sets of functions:

1. A most frequently used subset of BSD socket semantics;
2. A pair of request and event queues per container process;
3. Epoll-compatible event dispatch.

Control plane on programmable NIC. As shown in Figure 1, during container process initialization, it creates a request queue and an event queue to the programmable NIC (FPGA). The container sends requests to the NIC via request queue, and receives events from the NIC via event queue. Each queue resides in a pinned huge-page memory shared with the NIC. When an application calls a control-plane socket API (*e.g.* socket, listen, connect), it is sent to the NIC via its request queue. Because a *doorbell* from host to a PCIe device takes about 1 μs, we employ doorbell batching [10] to amortize doorbell overhead.

Data plane on user-space CPU. During connection setup, a region of shared huge-page memory is created between the sender and receiver containers. A lockless shared memory queue is initialized in the shared memory space. Because Intel x86 architecture provides ordering guarantees on write-after-write and read-after-write hazards [8], atomic instructions or memory barriers are not required for the single-reader, single-writer queue. We only need to insert compiler barriers to prevent the compiler from reordering critical instructions. Data-plane socket API calls (*e.g.* send, recv) are processed in user space, bypassing the NIC. The data is first copied from the sender buffer to the shared ring buffer, then copied to the receiver buffer, which is optimal considering semantics of BSD *send* and *recv*.

Compatible with existing software. In addition to high performance, our solution is compatible with existing software, does not require any modification to the Linux kernel, and preserves isolation among containers. We modify the Docker daemon to set a *LD_PRELOAD* environment variable to hook the standard C library (*glibc*) and POSIX thread library (*libpthread*) of container processes.

The system calls regarding to pipe, socket, epoll and process creation are redirected to our user-mode library. We follow the design of LOS [9] and split the file descriptor (FD) space to two parts, the lower half for kernel FDs, and the higher half for user-mode FDs. System calls to kernel FDs are passed through to the kernel, so the application can still read/write files and communicate with processes and hosts outside our system. System calls to user-mode FDs are replaced by a request to the monitor process, sent via the request queue. If the system call is blocking, it polls the event queue for 1μs. The monitor process polls the request queues of all containers in an infinite loop, therefore in most cases, the monitor responds an event within 1μs, and the requester proceeds without context switch. If the monitor is too busy to respond during requester polling, call *sched_yield* to put the process to sleep.

Process and thread creation are monitored by the user-mode library such that resources are duplicated for newly-*fork*ed processes or *clone*d threads.

4. Preliminary Evaluation

Due to complexity in hardware development, we first built a software prototype using a dedicated CPU core to process control-plane messages. The core is assigned to a *monitor container*. Request queues and event queues are created between monitor container and other containers.

We evaluate our software prototype on a Dell R720 server with two Xeon E5-2650 v2 CPUs, 128 GiB DDR3 memory and Archlinux kernel 4.11.9. Between two processes on two cores in a same NUMA node, the lockless shared-memory queue can transfer up to 27 M 64-byte messages per second, 2.5x faster than a shared-memory queue using memory barriers [7] and 8.4x faster than Linux socket. The round-trip ping-pong delay on the queue is 0.25 μs, 50x faster than Linux *pipe*. Furthermore, all these speed up comes without any overall limitation. The throughput and latency does not degrade if there are other queues on other cores.

Our prototype monitor container can process 12 M requests per second. Given that each connection setup needs 3 requests (each *socket*, *connect* or *accept* call needs 1 request), up to 4 M local socket connections can be created per second, 13x of Linux kernel and 6x of Fastsocket [13]. This throughput can be further scaled if we assign more CPU cores to the monitor container. After connection setup, packets can be transfered via lockless shared memory queue. The shared memory queue has 9x throughput and 60x lower latency compared to Linux socket.

We are still working on the implementation with programmable NICs. NIC hardware has higher processing capacity than dedicated CPU cores. A NIC with PCIe Gen3 x16 can perform up to 160 M DMA requests per second, 6x compared with a CPU core which can only receive 26 M non-batched messages per second. Therefore, with programmable NICs, the throughput of our design still has much room for improvement.

5. Conclusion

There has been a long debate on where to implement network stacks: hardware, kernel or user-space. With programmable NIC, hardware and software can work together by separation of coordination-intensive control plane and communication-intensive data plane. By offloading some kernel functionalities to hardware as well as user-space, the throughput of short-lived connections and the network delay among containers have an order of magnitude improvement.

A key challenge in FPGA-based NIC design is PCIe latency. With the advent of Xeon+FPGA platform, we expect higher throughput and lower latency communication between CPU and FPGA, to enable more fine-grained hardware-software co-design.

References

[1] High-performance network framework based on dpdk. URL http://f-stack.org/.

[2] Mellanox vma. URL http://www.mellanox.com/page/software_vma.

[3] *FreeFlow: High Performance Container Networking*, November 2016. ACM. ISBN 978-1-4503-4661-0/16/11.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI)*, number EPFL-CONF-201671. USENIX, 2014.

[6] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[7] U. o. C. Computer Laboratory. Ipc benchmark. URL https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/.

[8] I. Corporation. Intel 64 and ia-32 architectures software developer manual, volume 3.

[9] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li. Los: A high performance and compatible user-level network operating system. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 50–56. ACM, 2017.

[10] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[11] A. Kaufmann, S. Peter, T. E. Anderson, and A. Krishnamurthy. Flexnic: Rethinking network dma. In *HotOS*, 2015.

[12] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 1–14. ACM, 2016.

[13] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. In *ACM SIGPLAN Notices*, volume 51, pages 339–352. ACM, 2016.

[14] S. Networking. Network protocol offloadintroducing tcp chimney, 2004.

[15] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[16] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *OSDI*, pages 203–216, 2016.

[17] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[18] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[19] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 61–72. ACM, 2012.

[20] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern linux api usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 16. ACM, 2016.