Host Efficient Networking Stack Utilizing NIC DRAM

Byeongkeon Lee
KAIST
byeongkeonlee@kaist.ac.kr

Donghyeon Lee
KAIST
ldh1083@kaist.ac.kr

Jisu Ok
KAIST
jisu.ok@kaist.ac.kr

Wonsup Yoon
KAIST
wsyoon@kaist.ac.kr

Sue Moon
KAIST
sbmoon@kaist.ac.kr

ABSTRACT

The growth in host resource and network speed is not synchronized, and the status quo of this imbalance from the network speed of 100~ Gbps makes the host resource the bottleneck. We categorize existing body of work to reduce the host burden into the following three approaches: (1) to eliminate payload copy (zero-copy), (2) to utilize special-purpose hardware for payload copy, and (3) to offload protocol to NIC. Each approach, however, has drawbacks. (1) Most zero-copy methods require application modification. Furthermore, the application must ensure its buffer is not modified until network I/O is complete. (2) Copy elimination through special-purpose hardware still uses host memory, consuming considerable memory bandwidth. (3) The protocol offloaded to NIC has limited flexibility.

We redesign the networking stack placing only the payload in the NIC DRAM and executing protocol processing in the host to overcome the above limitations. Our work (1) makes the application reuse its own buffer as soon as the payload is transferred data in the NIC DRAM and does not require application modification, (2) saves host memory bandwidth by putting packet payload in NIC and eliminating payload copying on the host, and (3) maintains flexibility by keeping protocol processing on the host. Compared to the networking stack with CPU-based copy, our work saves 38.6% of CPU usage and 54.0% of memory bandwidth.

CCS CONCEPTS

• **Networks** → **Transport protocols**; • **Hardware** → **Networking hardware**.

KEYWORDS

NIC, NIC DRAM, networking stack, zero-copy, host resources

ACM Reference Format:
Byeongkeon Lee, Donghyeon Lee, Jisu Ok, Wonsup Yoon, and Sue Moon. 2023. Host Efficient Networking Stack Utilizing NIC DRAM. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600070

APNET 2023, June 29–30, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600070

1 INTRODUCTION

The end of Dennard scaling and Moore's law dictates that the CPU speed has stalled. In contrast, the networking speed continues to grow. Data centers have started to deploy 40 GbE NICs by default [42]. Furthermore, Ethernet Technology Consortium has pushed out 800 GbE specifications [7]. There is a road map forecast that 800 GbE development will be completed in the 2020s [3].

Here we focus on the trend that the CPU speed remains stagnant, but the networking speed is steadily rising. Because the packet size has stayed put, the network processing overhead grows linearly with the networking speed and places a burden on the host resources. Furthermore, data copy between the application, the socket buffer, and the NIC has become a costly operation best to be avoided [6, 12, 26]. This imbalance between networking speed and CPU has shifted the bottleneck from network to host computers [6, 27].

The significant CPU usage for network processing has been an ongoing problem. There have been a variety of approaches to mitigate it over the decades: (i) eliminating payload copy by avoiding separate buffers [1, 11, 12, 15, 21–23, 29, 34, 37, 50], (ii) exploiting special-purpose hardware for the copy operation [40], and (iii) protocol offloading to NIC devices [2, 4, 14, 16, 30–32]. However, each solution has its own limitations. (i) Avoiding separate buffers makes the networking stack share the payload buffer of the application. This approach requires application modification, and the application buffer must be kept unmodified and not deallocated for a long time. (ii) Since the socket buffer is present in the host memory, delegating copy to the special hardware still consumes a significant portion of memory bandwidth. (iii) The turnaround time in NIC programming is slow, and protocol offloading suffers from limited flexibility [4].

These observations have led us to the following insights. For API compatibility, we should keep the application buffer and networking buffer separate, but for memory bandwidth reduction, we should keep the networking buffer not on the host but in NIC. For programming flexibility, the protocol processing should remain on the host. The current hardware trend of NICs and SmartNICs beefing up on NIC DRAM opens up an opportunity to implement the above insights. The NIC DRAM is closer to the network than the host DRAM and thus provides an opportune location for the networking buffer.

Based on the above insights, we propose a new I/O architecture for host efficient networking stack: exploiting plentiful NIC DRAM as a buffer only for the payload. Our design transfers[1] data

[1]We denote Xilinx DMA-based copy as "transfer" according to the Xilinx documentations [43].

Buffer size per connection	1.0 KB	64.0 KB	1.0 MB
system call overhead	1.33%	3.37%	2.02%
_copy_from_iter_full	**1.24%**	**27.46%**	**61.31%**
tcp timer handling	1.66%	2.41%	0.95%
tcp_sendmsg	4.06%	10.54%	7.62%
L3 ~ L2 processing	6.29%	10.49%	6.43%
skb related processing	3.21%	12.01%	14.20%
etc.	2.33%	0.84%	0.00%
`write()` CPU in the app.	20.12%	67.12%	92.53%

Table 1: CPU usage breakdown of `write()`.

in the application buffer to FPGA NIC DRAM through the NIC DMA engine while the host CPU handles the network protocol processing. Since most networking stacks do not modify the payload [8, 37], we choose to run protocol processing and payload transfer independently.

In §2 we describe existing approaches to reduce the host burden and analyze pros and cons. In §3 we present our new networking architecture, and in §4 describe implementation details. In order to demonstrate the advantages of our new architecture, we compare our approach to three existing I/O schemes: CPU-based copy, I/OAT-based copy, and zero-copy in §5. Ours reduces CPU usage by 38.6% compared to the CPU-based approach. Ours also saves 17.3 GB/s memory bandwidth than I/OAT-based copy. Lastly, our work reduces the application buffer holding time by 52.9-67.9% depending on the buffer size with socket API intact compared to the zero-copy.

2 BACKGROUND

We begin this section with a breakdown analysis of the TCP `write()` system call and discuss the impact of the copy operation on the CPU within the TCP networking stack. Next, we review the following three approaches to mitigate the host burden: memory pool-based I/O, application buffer direct I/O, and I/OAT engine copy offload.

2.1 TCP `write()` System Call Breakdown

Significant CPU consumption for copy in the networking stack has been a known problem for decades [6]. When the application is I/O intensive, CPU-based copy dominates CPU usage. To analyze the CPU usage in detail, we break down the `write()` system call of a simple socket application. Both the sender and the receiver used Intel E810-CQDA2 100 Gbps NICs and Linux 4.15.0. The measurement tool was Linux perf [24]. Table 1 shows that the 1 KB buffer copy per connection uses 1.24% of CPU cycles. As the buffer size grows, so does CPU usage: the 64 KB buffer copy alone occupies 27.46% of the application CPU cycles, and a 1 MB buffer copy uses more than half. Clearly, data copy still remains a challenge in this era of high-speed networking.

2.2 Memory Pool-based I/O

In memory pool-based I/O approaches, a framework pre-allocates the memory pool and provides the APIs for applications to access the pool. To eliminate copy, the application fills data into buffers from the memory pool, which will be sent out eventually. Since the frameworks designate the memory pool as an I/O region, it charges minimal overhead at runtime.

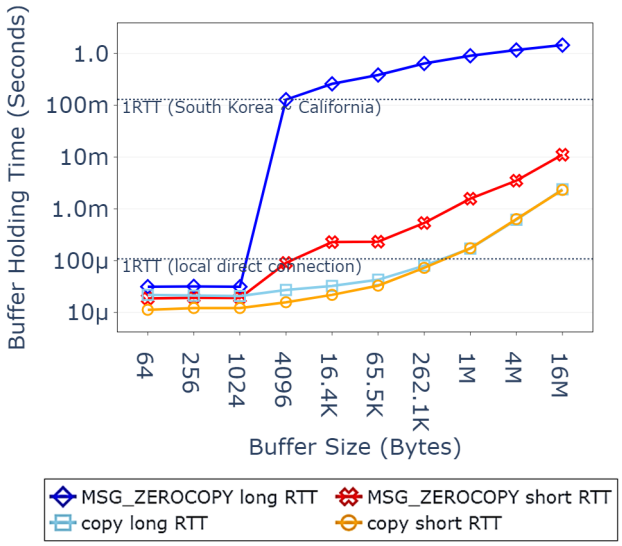

Figure 1: Buffer holding time for copy/zero-copy with different RTTs. Lower is better. (log10 scale)

There are many examples achieving payload zero-copy: memory pool-based I/O engine frameworks (e.g., Netmap [29] and DPDK [15]) and a networking stack on top of memory pool-based I/O (e.g., MAIO [1] and AF_XDP [9]).

The limitations of these approaches are as follows. First, each framework provides its own APIs to manage the buffer. It thus forces application modification. Second, the framework, not the application, manages the buffer. It makes buffers difficult for application developers to manage, track, and reuse.

2.3 Application Buffer Direct I/O

DMA from the application buffer is another approach to achieving zero-copy I/O. By having the NIC hardware fetch the data directly, it eliminates the need for an intermediate socket buffer and CPU-based copy. This approach, however, must pin the physical pages that include the application buffer via kernel API (e.g., `get_user_page_fast()` [10]). Only with pinning, the virtual-to-physical page mapping remains constant during DMA, and the kernel does not swap out the pages to disk or move them for memory compaction [35]. This pinning overhead is not negligible, but the I/O-intensive applications benefit from low CPU usage through copy avoidance.

This approach, however, must protect the application buffer from modification and deallocation until the data transmission is complete. Otherwise, the corrupt data will be retransmitted to the receiver, if a packet drop occurs after the data has already been modified by the application.

There are two types of mechanisms to prevent data corruption: copy-on-write and completion notifications.

Copy-on-write. Copy-on-write makes the physical pages read-only, preventing data from being written. Once the write to that page takes place, the page fault handler copies them to another page. This mechanism eventually preserves the original buffer from modification.

ZCopy [34] and zIO [37] are examples of application buffer direct I/O with copy-on-write protection. ZCopy [34] has risen from the observation that web-caching applications rarely modify application buffers. It removes most CPU-based copies by sharing the copy-on-write protected application buffer with the network stack. zIO [37] provides copy-on-write `memcpy()`. In zIO, the destination buffer shares copy-on-write protected physical pages with the source buffer to avoid unnecessary copy.

Shortcomings of the copy-on-write mechanisms are: (i) It operates in the unit of pages. (ii) It involves additional expensive operations such as write protection and page faults. Their costs are bearable only for applications with few writes.

Completion Notification. MSG_ZEROCOPY [12, 13] and IO_uring [11] are the completion notification mechanisms that let the application know when the application buffer is safe for reuse. MSG_ZERO COPY uses a socket error queue, and IO_uring uses the existing completion queue. Applications are modified to use these queues and decide when to modify or deallocate the buffer. In case the underlying networking protocol is TCP, the application buffer is reusable only after the data is reliably transmitted. The buffer holding time is at least 1 RTT.

The buffer holding time is the amount of time an application should not modify or deallocate its own buffer. We have conducted a simple experiment to measure the buffer holding time of host CPU-based copy and zero-copy (MSG_ZEROCOPY). The buffer holding time for the copy is the `write()` system call return time, while that for the zero-copy is the time until the completion notification arrives. When two servers are close by, the RTT is very short. On the other hand, a long-haul connection, for example, between our lab in South Korea and the Amazon EC2 California region, incurs long RTT. As shown in Figure 1, the buffer holding time of the zero-copy is significantly larger than the copy, as the zero-copy should wait for at least 1 RTT. Note that the kernel performs copy for zero-copy API when a buffer is smaller than a specific size [8].

The long buffer holding time makes application memory scarce when the application runs in a memory-constrained environment. It also adds complexity to the application design. The application needs to track whether the application buffer is free for reuse for a long time.

2.4 Intel I/OAT

Intel I/O Acceleration Technology (Intel I/OAT) [17, 40] is a set of technologies that improve throughput, efficiency, and scalability for data flow.

Intel QuickData Technology "enables data copy by the chipset instead of the CPU, to move data more efficiently" [17]. This technology is applicable to the existing networking stack; replacing the subject of data copy from the CPU to I/OAT engine saves CPU cycles. However, it supports only a limited number of channels at any time, and its use is won over competition with other processes. Despite the CPU cycle reduction, it still consumes memory bandwidth if the socket buffer remains in host memory. Compared to zero-copy mechanisms in § 2.2 and § 2.3, it requires double the memory bandwidth.

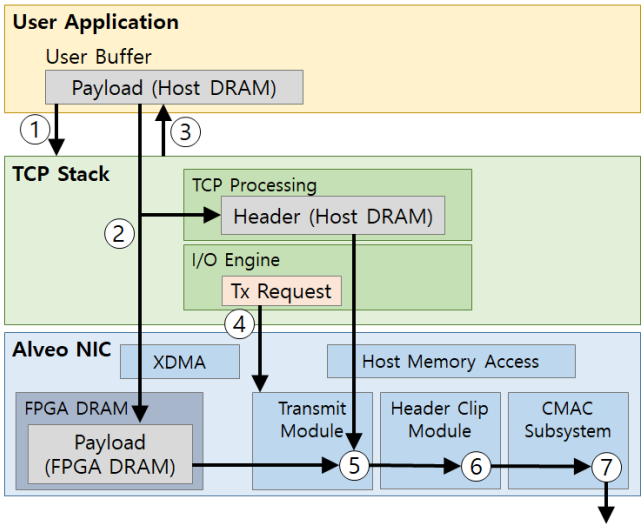

Figure 2: Architecture overview

3 ARCHITECTURE

All the existing solutions from §2 point at the importance of limiting CPU usage while using as little memory bandwidth as possible for network processing. We propose a new I/O architecture design that takes full advantage of modern FPGA NICs with significantly large DRAM. Our new I/O architecture places the payload buffer in the NIC DRAM and directly forwards the application buffer to the payload buffer without an intermediate on-host socket buffer. Except for the payload transfer, other parts of network stack processing remain on the host, allowing high flexibility in protocol updates.

3.1 Architecture Overview

Figure 2 illustrates our architecture overview. It consists of the FPGA NIC layer and the host TCP stack layer, upon which the user application layer sits. In our architecture, the socket API to the application remains intact and the application requires no modification. It is the role of the TCP stack to transfer the payload to the NIC DRAM but maintain the header in the host memory. When the host TCP stack requests, the NIC generates and transmits packets by concatenating the header on the host and the payload in FPGA NIC DRAM. The host TCP stack manages headers and processes TCP algorithms, such as reliable data transfer through loss detection and retransmission. It also exposes socket APIs intact to the application.

Here we illustrate how our new I/O architecture works step by step. First, ① the application makes a blocking `write(fd, buf, len)` system call to send the data. Then, ② the host TCP stack starts to transfer the payload from the application buffer to FPGA NIC DRAM via an FPGA hardware module, XDMA (Xilinx DMA) [43]. At the same time, the TCP stack starts to prepare packet headers and fills them into the host memory accessible by FPGA. After ②, ③ the application is unblocked and free to modify or deallocate its own buffer. Then, ④ the TCP stack sends a transmission request to FPGA NIC with the address of the header and the payload. ⑤ Upon receiving the transmission request, the transmit module in the FPGA NIC creates packets by concatenating the header from

the host memory and the payload from FPGA NIC DRAM. ⑥ Since the data transmission unit between FPGA modules is 64 bytes, this header clip module trims redundant data that exists at the end of the header if the header length is less than 64 bytes. ⑦ The Xilinx CMAC subsystem [46] finally sends out the complete packets to the network through the transceiver.

3.2 FPGA Module Design

In this section, we describe the modules we have designed for our new I/O architecture. The first three modules, Host Memory Access, XDMA, and CMAC are modules from Xilinx. The other two modules, Transmit and Header Clip modules are of our making.

Host Memory Access allows the FPGA module to read and write directly from the host memory region, bypassing the FPGA NIC DRAM [44]. We use it to load the packet headers to the FPGA transmit module efficiently in ⑤.

XDMA is the main FPGA module Xilinx provides to transfer data between host memory and FPGA NIC DRAM via PCIe [43]. We use 2 DMA channels to utilize multiple DRAM modules in parallel, maximizing the transfer rate of payload.

CMAC. (C Media Access Controller) stands for the 100 Gbps (the Roman letter C for 100) ethernet transmit system on the Alveo NIC.

Transmit Module. The host stack triggers the transmit module by submitting a request with the packet header address, header length in the host, the payload address, and payload length in the FPGA NIC DRAM. This module collects and concatenates the data from the respective addresses.

Header Clip Module. The preceding transmit module sends and receives data in units of 64 bytes according to the AXI protocol [45]. This enforces the packet header to move from the host DRAM to the transmit module in a 64-byte-padded form. The packet header, however, is often less than 64 bytes (typically 54 bytes), so the padded bits should be removed before being put onto the wire. The header clip module removes those bits (10 bytes) from the header chunk and fills this hole by shifting the payload bytes. It then sends the series of rearranged 64-byte chunks to CMAC module [46], which generates a complete packet from the chunks and transmits the packet to the wire.

3.3 Host TCP Stack Design

We separate host operations into two layers: the TCP processing layer focuses on TCP-related tasks, and the I/O engine handles sending and receiving packets.

The TCP processing layer is responsible for header buffer management, FPGA NIC control, and TCP algorithms such as connection setup and packet loss detection. It exposes blocking socket APIs to the application. After the application establishes connections through `connect()` and then calls `write()`, the TCP processing layer prepares the headers in the host memory and moves the payload to FPGA NIC DRAM simultaneously. The payload bypasses the TCP stack and avoids copies in the host. In this work, we do not consider cases of payload modification, such as IPsec, and leave them for future work.

I/O Engine. The TCP processing delivers the address of headers and payloads to the I/O engine, and the I/O engine sends Tx requests containing the addresses to the FPGA NIC. The I/O engine also detects packet arrivals from the network. It delivers the packet to the TCP processing.

4 IMPLEMENTATION

4.1 FPGA Implementation

We use the 100 Gbps Ethernet Alveo U200 data center accelerator card with 4 x 16 GB DDR4 2400 MT/s memory. The FPGA modules originated from the Vitis Network Example project of the Xilinx University Program [49]. We have removed all networking stack-related modules and implemented modules through Vitis High-Level Synthesis (HLS) [47]. The platform is `xilinx_u200_gen3x16_xdma_1_202110_1`. The Vitis and v++ compiler versions are 2021.2.

The current implementation of the receiver side transfers the payload from the network to the host memory bypassing FPGA NIC DRAM. Then, the host stack parses it immediately.

4.2 Host Implementation

Using Xilinx RunTime Library (XRT) [48], the host CPU controls the Alveo U200 device easily. The library also provides a buffer object abstraction for FPGA NIC DRAM. We implement a packet I/O engine and host TCP stack as a user library on top of XRT. We use the XRT library version 2.12.427. The entire host stack is about 2,500 lines of C++ code.

4.3 Limitations

We present the current implementation as a proof of concept for our design. Hence, minimum TCP features have been implemented. To evaluate the performance of TCP send, we focus only on TCP active open. The TCP passive open APIs are not implemented as of now.

TCP flow and congestion controls are out of scope. Since the FPGA NIC DRAM is byte-addressable by the Transmit Module, we expect their incorporation to be straightforward.

Many host protocol stacks treat checksum computation as a function offloaded to NICs. The current implementation assumes that the checksum computation is done on FPGA NIC [39] and omitted.

5 EVALUATION

The goal of this section is to demonstrate that our new I/O architecture delivers improved performance over existing solutions. We compare ours against the following three schemes: baseline, I/OAT, and zero-copy.

Baseline. Host CPU copies data from the application buffer to the socket buffer. Both buffers reside in the host memory. This is the typical performance we expect from most applications on Linux.

I/OAT. Both buffers are the same as the baseline, but the I/OAT hardware executes copy instead of the host CPU. Using I/OAT for application buffers requires the following constraints. First, it should pin the application buffer to keep the physical pages of the application buffer constant [35]. Second, the virtual-to-physical address translation is necessary since I/OAT hardware needs physical addresses for application pages rather than virtual addresses. Last, I/OAT copy granularity should be up to the page size of 4 KB. We

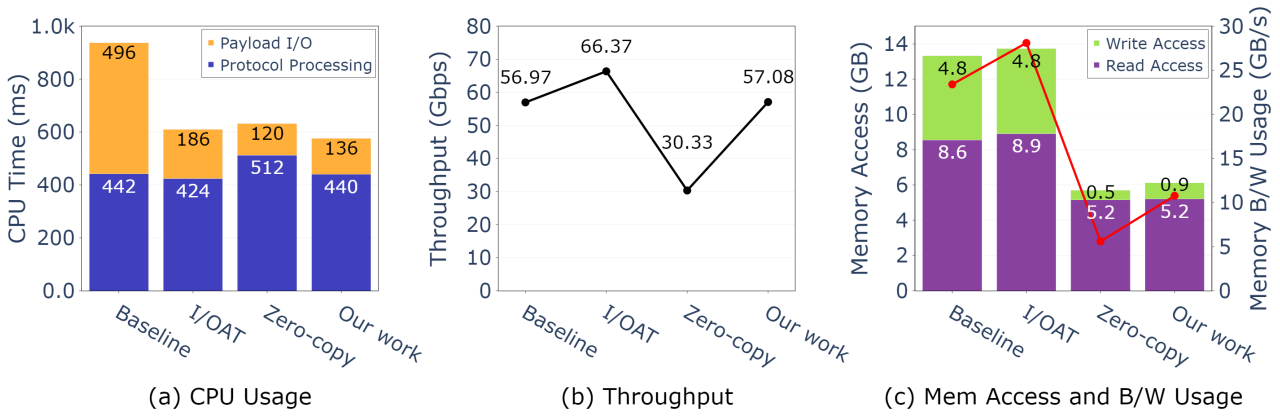

Figure 3: (a) CPU usage, (b) throughput, and (c) memory access/bandwidth usage of four I/O schemes. For (a) CPU usage and (c) memory access/bandwidth usage, lower is better.

have applied these constraints to SPDK open source that provides an interface to I/OAT [36]. Note that we translate virtual addresses to physical addresses via the `proc` file system and exclude this overhead in the evaluation.

Zero-copy. Similar to Linux MSG_ZEROCOPY [13], data is directly moved from the application buffer to the NIC transceiver via the XDMA engine without going through the FPGA NIC DRAM. This also requires application buffer pinning.

Our evaluation has focused on answering the following questions: Does our work show (i) less CPU usage than the baseline? (ii) Comparable throughput with other I/O schemes? (iii) Less memory bandwidth usage than the baseline and the I/OAT? (iv) Shorter buffer holding time than the zero-copy?

Setup. Two machines, a sender and a receiver, are directly connected with a 100 Gbps link. Each machine has a single NUMA of Intel Xeon Gold 6226R CPU@2.90 GHz with Hyper-Threading disabled and 384 GiB memory. The sender has Alveo FPGA NIC. The receiver has Intel E810-CQDA2 100 Gbps NIC and runs a simple DPDK-based TCP stack. Both NICs are attached to hosts through PCIe 3.0 x 16.

We use Linux kernel 4.15.0. We use Linux `pidstat` [25] to measure CPU usage and Intel PCM [18] to measure memory bandwidth usage. We have iterated the 3.6 MB buffers 1,125 times, 4.06 GB in total.

5.1 CPU Usage

Figure 3(a) shows the CPU usage of the `write()` system call in the four I/O schemes. It counts packet header preparation and payload transfer. As expected, the baseline shows the highest CPU usage because of its CPU-based payload copy. Compared to the baseline, our work saves CPU cycles by 38.6%.

Others have similar CPU usage. The zero-copy consumes 9.7% more CPU usage compared to our work. Waiting for completion notifications and handling them in the application increases CPU usage, while our work does not need to. Our work, however, has one more step than zero-copy. Payload transfer to FPGA NIC DRAM increases payload I/O by 13.3%.

Our work has a slightly lower CPU usage than I/OAT. The additional payload I/O cost for I/OAT is caused by checking the I/OAT task completion in the host TCP stack.

5.2 Throughput

Generally, CPU-based memory copy is expected to have higher speed than DMA transfer since memory copy is simply a set of memory read and write while DMA transfer should pull the data over PCIe. However, in the case of I/O-intensive applications, the heavy use of CPU-based memory copy exceeds the cache size [33, 37, 41] and thus aggravates the overall throughput. This counterbalance resulted in a similar throughput for the baseline and our work, as shown in Figure 3(b).

Zero-copy has the lowest throughput because the completion notification mechanism pauses the application until ACKs arrive from the receiver. I/OAT has the highest throughput, but note that address translation costs are excluded from our measurement.

5.3 Memory Access and Bandwidth Usage

As shown in the bar of Figure 3(c), the baseline and the I/OAT access memory 2.18x and 2.24x more than ours, respectively. Considering only host memory, the payload of both baseline and I/OAT should be read from the application buffer, written to the payload buffer, and read again by the NIC. In contrast, zero-copy and our work only need to read the application buffer by the NIC. This difference results in 7.6 GB lower memory access for our work than for I/OAT.

Taking the elapsed time into account, memory bandwidth also shows similar trends, as described in the line of Figure 3(c). Our work uses 54.0% and 61.7% lower memory bandwidth than baseline and I/OAT, respectively. Compared to I/OAT, it saves 17.3 GB/s. Considering our system-wide theoretical maximum bandwidth (140.8 GB/s), this is a considerable saving of 12.3%. The lowest zero-copy memory bandwidth usage is caused by the longest elapsed time due to the lowest throughput.

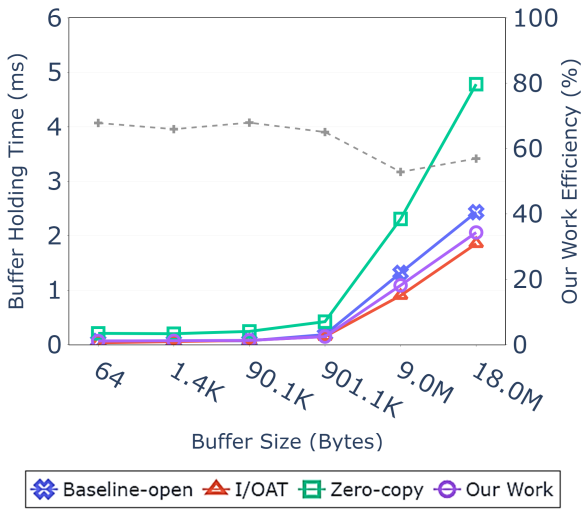

Figure 4: TCP buffer holding time of four I/O schemes. Lower is better.

5.4 Buffer Holding Time

Figure 4 shows the buffer holding time of the four I/O schemes. The application layer measures the buffer holding time, from the time it calls `write()` to the time it is free to safely reuse the buffer. As expected, zero-copy shows the longest buffer holding time due to completion notification handling. Other schemes, however, do not have the completion notification mechanism. The application on top of those schemes is free to reuse the buffer immediately after the `write()` system call returns. Finally, I/O schemes other than zero-copy show similar buffer holding times.

The grey dashed line represents the reduced buffer holding time of our work compared to zero-copy. As shown, our work has a 52.9-67.9% shorter buffer holding time than zero-copy.

Note that the sender and receiver are directly connected, which is the optimal environment. The buffer holding time of zero-copy is dependent on the RTT of the path between the sender and receiver. Thus it increases when packets pass through multiple hops.

5.5 Socket API Intact

Many existing zero-copy mechanisms force the use of specific APIs or completion notification handling, causing application modification. Our work, however, does not require any of them. Applications on top of our TCP stack avoids CPU-based copy without code-level modification. All the application has to do is to load the shared object via `LD_PRELOAD=./libtcp.so`.

6 RELATED WORK

Modern NICs are constantly adding features to save host resources.

Exploiting NIC Memory. Recent work demonstrates many use cases of NIC memory. Nicmem [28] aims to reduce CPU, memory, and PCIe usage for Network Functions (NFs) with an insight that most NFs do not touch payload [38]. Contents cache [5, 19, 20, 51] in the NIC memory shares the same goals as our work. But we focus on transferring the application buffer rather than caching responses.

Exploiting the NIC Processor. Protocol processing offloading to NICs is an active field of research. AccelTCP proposes offloading TCP for short-lived connections [26]. Limago implements a 100 Gbps TCP Offload Engine on the FPGA boards [30]. Since protocol offloading suffers from the flexibility issue, TONIC proposes a flexible hardware design by modularizing TCP [4]. In contrast, we aim to provide high performance while retaining high flexibility by processing protocol on the host without expensive CPU-based copy operations.

7 FUTURE WORK

Reusing NIC DRAM Buffer. Storing payload in the NIC memory is advantages in packet retransmission. Our design reuses the payload in the NIC DRAM without fetching data from the host again, saving both host memory access and PCIe usage. For lossy network connections, our approach should deliver comparable throughput with less CPU and memory bandwidth consumption.

Receiver Side CPU-based Copy Elimination. Our current implementation has focused on the sender side. Eliminating CPU-based copy on the receiver side requires careful coordination. For efficient DMA from NIC DRAM to the application buffer without involving the intermediate host buffer, the payload must be as contiguous as possible on the FPGA NIC DRAM. We plan to find a suitable receiver-side design that does not compromise flexibility.

Other Protocol Support. Although our current design and implementation is centered around TCP, it is not limited to TCP. We expect other protocols, such as DCTCP and QUIC, could apply our architecture to their code easily.

8 CONCLUSION

Host resources, more specifically, CPU cycles and memory bandwidth are a performance constraining factor as the network bandwidth increases. We have proposed a new I/O architecture utilizing the FPGA NIC DRAM. Our design aims to reduce both CPU and memory bandwidth usage, shorten the buffer holding time, and preserve the protocol processing flexibility without application modification.

We have compared four I/O schemes: baseline, I/OAT, zero-copy, and our work. Our work shows (i) 38.6% lower CPU usage than baseline, (ii) 7.6 GB less memory access and 17.3 GB/s lower memory bandwidth usage compared to I/OAT, (iii) up to 67.9% shorter buffer holding time than zero-copy.

Our design retains the protocol processing in the host CPU, and thus the flexibility is not compromised. Lastly, the application benefits from CPU-based copy elimination by loading the shared object without application modification. The current implementation is limited to the bare skeletal of the architecture and is to be expanded to include the complete TCP protocol.

ACKNOWLEDGMENTS

We sincerely thank anonymous reviewers for their helpful comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2022R1A2C2009062).

REFERENCES

[1] Alex Markuze, Igor Golikov, and Chen Dar. 2021. Rethinking Zero-Copy Networking with MAIO. The Technical Conference on Linux networking 0x15, Virtual.

[2] Alexforencich. Verilog Ethernet Components Introduction. http://alexforencich.com/wiki/en/verilog/ethernet/start

[3] Ether Alliance. Ethernet Roadmap 2023. https://ethernetalliance.org/technology/ethernet-roadmap/

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109.

[5] Herbert Bos and Kaiming Huang. 2009. CacheCard: Caching Static and Dynamic Content on the NIC *(ANCS '09)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[6] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77.

[7] Ethernet Technology Consortium. 25 Gigabit Ethernet Consortium Rebrands to Ethernet Technology Consortium; Announces 800 Gigabit Ethernet (GbE) Specification. https://ethernettechnologyconsortium.org/press-room/press-releases/25-gigabit-ethernet-consortium-rebrands-to-ethernet-technology-consortium-announces-800-gigabit-ethernet-gbe-specification-152/

[8] Jonathan Corbet. Zero-copy networking. https://lwn.net/Articles/726917/

[9] Jonathan Corbet. Accelerating networking with AF_XDP. https://lwn.net/Articles/750845/

[10] Jonathan Corbet. Explicit pinning of user-space pages. https://lwn.net/Articles/807108/

[11] Jonathan Corbet. Zero-copy network transmission with io_uring. https://lwn.net/Articles/879724/

[12] Willem de Bruijn and Eric Dumazet. 2017. sendmsg copy avoidance with MSG_ZEROCOPY. The Technical Conference on Linux networking 2.1, Le Westin Montréal, Montreal, Canada.

[13] Linux Kernel Document. MSG_ZEROCOPY. https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html

[14] Z. He, D. Korolija, and G. Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA, 197–203.

[15] Intel. Intel Data Plane Development Kit. https://www.dpdk.org/

[16] Intel. Intel Ethernet Network Adapter E810-CQDA2. https://ark.intel.com/content/www/us/en/ark/products/192558/intel-ethernet-network-adapter-e810cqda2.html

[17] Intel. Intel® I/O Acceleration Technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html

[18] Intel. Intel® Performance Counter Monitor (Intel® PCM). https://github.com/intel/pcm

[19] H. Kim, S. Rixner, and V.S. Pai. 2005. Network interface data caching. *IEEE Trans. Comput.* 54, 11 (2005), 1394–1408.

[20] Hyong-youb Kim, Vijay S. Pai, and Scott Rixner. 2002. Increasing Web Server Throughput with Network Interface Data Caching. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 239–250.

[21] Linux. sendfile — Linux manual page. https://man7.org/linux/man-pages/man2/sendfile.2.html

[22] Linux. splice — Linux manual page. https://man7.org/linux/man-pages/man2/splice.2.html

[23] Linux. vmsplice — Linux manual page. https://man7.org/linux/man-pages/man2/vmsplice.2.html

[24] Linux. perf — Linux manual page. https://man7.org/linux/man-pages/man1/perf.1.html

[25] Linux. pidstat — Linux manual page. https://man7.org/linux/man-pages/man1/pidstat.1.html

[26] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and Kyoung-Soo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92.

[27] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341.

[28] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The Benefits of General-Purpose on-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1130–1147.

[29] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112.

[30] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 286–292.

[31] D. Sidler, G. Alonso, M. Blott, K. Karras, et al. 2015. Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *FCCM'15*.

[32] D. Sidler, Z. Istvan, and G. Alonso. 2016. Low-Latency TCP/IP Stack for Data Center Applications. In *FPL'16*.

[33] Richard L. Sites. Fast memcpy, A System Design. https://www.sigarch.org/fast-memcpy-a-system-design/

[34] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2012. Revisiting Software Zero-Copy for Web-caching Applications with Twin Memory Allocation. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 355–360.

[35] SPDK. Direct Memory Access (DMA) From User Space. https://spdk.io/doc/memory.html

[36] SPDK. Storage Performance Development Kit. https://github.com/spdk/spdk

[37] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 431–445.

[38] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/3098822.3098826

[39] Gustavo Sutter, Mario Ruiz, Sergio Lopez-Buedo, and Gustavo Alonso. 2018. FPGA-based TCP/IP Checksum Offloading Engine for 100 Gbps Networks. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[40] Jonathan Stern Thai Le, Steven Briscoe. 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

[41] Karthikeyan Vaidyanathan and Dhabaleswar K. Panda. 2007. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. 220–229.

[42] Wenfeng Xia, Peng Zhao, Yonggang Wen, and Haiyong Xie. 2017. A Survey on Data Center Networking (DCN): Infrastructure and Operations. *IEEE Communications Surveys & Tutorials* 19 (2017), 640–656.

[43] Xilinx. DMA/Bridge Subsystem for PCI Express Product Guide (PG195). https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Introduction

[44] Xilinx. Host Memory Access. https://xilinx.github.io/XRT/master/html/hm.html

[45] Xilinx. Interfaces for Vitis Kernel Flow. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Interfaces-for-Vitis-Kernel-Flow

[46] Xilinx. UltraScale+ Devices Integrated 100G Ethernet Subsystem v3.1 LogiCORE IP Product Guide. https://docs.xilinx.com/r/en-US/pg203-cmac-usplus

[47] Xilinx. Vitis HLS. https://xilinx.github.io/Vitis-Tutorials/2021-2/build/html/docs/Getting_Started/Vitis_HLS/Getting_Started_Vitis_HLS.html

[48] Xilinx. Xilinx Runtime Library (XRT). https://www.xilinx.com/products/design-tools/vitis/xrt.html

[49] Xilinx. XUP Vitis Network Example (VNx). https://github.com/Xilinx/xup_vitis_network_example

[50] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56.

[51] Kenneth Yocum and Jeffrey S. Chase. 2001. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. USENIX Association, Boston, MA.